DOCA Documentation v2.8.0

Table of Contents

1 DOCA Documentation v2.8.0

1.1 DOCA Overview

This page provides an overview of the structure of NVIDIA DOCA documentation.

1.2 Release Notes

This page contains information on new features, bug fixes, and known issues.

1.3 User Types

This page provides a quick introduction to the NVIDIA® BlueField® family of networking platforms (i.e., DPUs and SuperNICs), its DOCA software components, and BlueField user types.

1.4 NVIDIA DOCA EULA

This page provides the NVIDIA DOCA SDK end-user license agreement.

2 Quick Start

2.1 Developer Quick Start Guide

This page details the basic steps to bring up the NVIDIA DOCA development environment and to build and run the DOCA reference applications provided along with the DOCA software framework package.

3 Installation and Setup

3.1 Profiles

This page provides an introduction to the various supported DOCA profiles.

3.1.1 NVIDIA MLNX_OFED Transition Guide

This page covers what users must know about the DOCA-Host unified software stack for NVIDIA networking products.

3.2 Installation Guide for Linux

This page details the necessary steps to set up NVIDIA DOCA in your Linux environment.

3.3 Developer Guide

This page details the recommended steps to set up an NVIDIA DOCA development environment.

4 DOCA Programming Guides

These pages are intended for developers wishing to utilize DOCA SDK to develop application on top of NVIDIA® BlueField® networking platforms.

5 Applications

This page provides an overview of the example DOCA applications implemented on top of NVIDIA® BlueField®.

5.1 App Shield Agent

This page provides process introspection system implementation on top of NVIDIA® BlueField®.

5.2 DMA Copy

This page provides an example of a DMA Copy implementation on top of NVIDIA® BlueField®.

5.3 DPA All-to-all

This page explains the all-to-all collective operation example when accelerated using the DPA in NVIDIA® BlueField®-3.

5.4 DPA L2 Reflector

This page provides an L2 reflector implementation on top of the NVIDIA® BlueField®-3.

5.5 East-west Overlay Encryption

This page describes IPsec based strongSwan solution on top of NVIDIA® BlueField®.

5.6 Ethernet L2 Forwarding

This page provides an Ethernet L2 Forwarding implementation on top of the NVIDIA® BlueField® DPU.

5.7 File Compression

This page provides a file compression implementation on top of the NVIDIA® BlueField®.

5.8 File Integrity

This page provides a file integrity implementation on top of NVIDIA® BlueField®.

5.9 GPU Packet Processing

This page provides a description of the GPU packet processing application to demonstrate using the DOCA GPUNetIO, DOCA Ethernet, and DOCA Flow libraries to implement a GPU traffic analyzer.

5.10 IPsec Security Gateway

This page provides an IPsec security gateway implementation on top of NVIDIA® BlueField®.

5.11 PCC

This page provides a DOCA PCC implementation on top of NVIDIA® BlueField®.

5.12 PSP Gateway

This page describes the usage of the NVIDIA DOCA PSP Gateway sample application on top of an NVIDIA® BlueField® networking platform or NVIDIA® ConnectX® SmartNIC.

5.13 Secure Channel

This page provides a secure channel implementation on top of NVIDIA® BlueField®.

5.14 Simple Forward VNF

This page provides a Simple Forward implementation on top of NVIDIA® BlueField®.

5.15 Switch

This page provides an example of switch implementation on top of NVIDIA® BlueField®.

5.16 UROM RDMO

This page provides a DOCA Remote Direct Memory Operation implementation on top of NVIDIA® BlueField® using Unified Communication X (UCX)..

5.17 YARA Inspection

This page provides YARA inspection implementation on top of NVIDIA® BlueField®.

6 Tools

This page provides an overview of the set of tools provided by DOCA and their purpose.

6.1 DOCA Bench

This page describes a tool which allows users to evaluate the performance of DOCA applications, with reasonable accuracy for real-world applications.

6.2 Capabilities Print Tool

This page provides instruction on the usage of the DOCA Capabilities Print Tool.

6.3 Comm Channel Admin Tool

This page provides instructions on the usage of the DOCA Comm Channel Admin Tool.

6.4 DPA Tools

This page lists a set of executables that enable the DPA application developer and the system administrator to manage and monitor DPA resources and to debug DPA applications.

6.5 PCC Counter Tool

This page provides instruction on the usage of the PCC Counter tool.

6.6 Socket Relay

This page describes DOCA Socket Relay architecture, usage, etc.

7 DOCA Services

This page provides an overview of the set of services provided by DOCA and their purpose.

7.1 Container Deployment

This page provides an overview and deployment configuration of DOCA containers for NVIDIA® BlueField®.

7.2 DOCA BlueMan Service

This page provides instructions on how to use the DOCA BlueMan service on top of NVIDIA® BlueField®.

7.3 DOCA Firefly Service

This page provides instructions on how to use the DOCA Firefly service container on top of NVIDIA® BlueField®.

7.4 DOCA Flow Inspector Service

This page provides instructions on how to use the DOCA Flow Inspector service container on top of NVIDIA® BlueField®.

7.5 DOCA HBN Service

This page provides instructions on how to use the DOCA HBN Service container on top of NVIDIA® BlueField®.

7.6 DOCA Management Service

This page provides instructions on how to use the DOCA Management Service on top of NVIDIA® BlueField® Networking Platform or ConnectX® Network Adapters.

7.7 OpenvSwitch Acceleration (OVS in DOCA)

These pages describe OVS within DOCA, particularly OVS-DOCA, a virtual switch service tailored for NVIDIA NICs and DPUs. It leverages ASAP^2 technology for accelerated data-path processing, ensuring optimal performance and features through its architecture and integration with DOCA libraries.

7.8 DOCA Telemetry Service

This page provides instructions on how to use the DOCA Telemetry Service (DTS) container on top of NVIDIA® BlueField®.

7.9 DOCA UROM Service

This page provides instructions on how to use the DOCA Telemetry Service (DTS) container on top of NVIDIA® BlueField®.

8 API References

8.1 DOCA Driver APIs

This page contains DOCA driver APIs.

8.2 DOCA Libraries APIs

This page contains DOCA libraries APIs.

9 Miscellaneous

9.1 Glossary

This page provides a list of terms and acronyms and in the DOCA documentation.

9.2 Crypto Acceleration

This page shows the ability of NVIDIA® BlueField® to accelerate crypto operations.

9.3 DOCA Services Fluent Logger

This page provides instructions on how to use the logging infrastructure for DOCA services on top of NVIDIA® BlueField®.

9.4 DPU CLI

This page provides quick access to a useful set of CLI commands and utilities on the NVIDIA® BlueField® environment.

9.5 Emulated Devices

For information on virtio-net device emulation, please refer to the NVIDIA BlueField Virtio-net documentation.

9.6 Modes of Operation

This page describes the modes of operation available for NVIDIA® BlueField®.

9.7 Switching

These pages describe the extensive switching capabilities enabled by DOCA libraries and services on these platforms.

9.8 OpenSSL

This page provides instructions on using DOCA SHA for OpenSSL implementations.

9.9 Scalable Functions (SFs)

This page provides an overview and configuration of scalable functions (sub-functions, or SFs) for NVIDIA® BlueField®.

9.10 TLS Offload

This page provides an overview and configuration steps of TLS hardware offloading via kernel-TLS, using hardware capabilities of NVIDIA® BlueField®.

9.11 Troubleshooting

This page provides troubleshooting information for common issues and misconfigurations encountered when using DOCA for NVIDIA® BlueField®.

9.12 Virtual Functions (VFs)

This page provides an overview and configuration of virtual functions for NVIDIA® BlueField® and demonstrates a use case for running the DOCA applications over x86 host.

10 Archive

10.1 LTS Versions

This page provides pointers to the DOCA long term support (LTS) releases.

10.2 Documentation Archives

This page provides pointers to archived documentation of previous DOCA software releases.

ⓘ For questions, comments, and feedback, please contact us at DOCA-Feedback@exchange.nvidia.com.

11 DOCA SDK v2.8.0

This section contains the following pages:

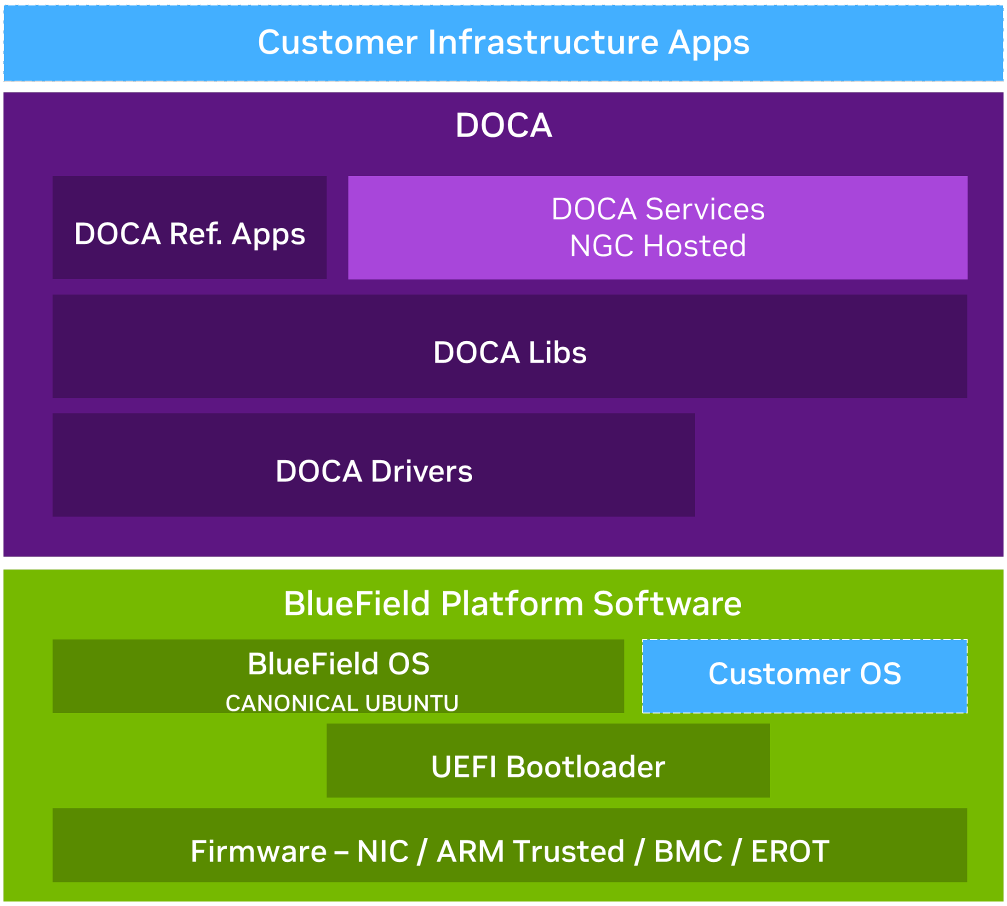
- NVIDIA DOCA Overview
- NVIDIA DOCA Release Notes
- BlueField and DOCA User Types
- NVIDIA DOCA EULA

11.1 NVIDIA DOCA Overview

This is an overview of the structure of NVIDIA DOCA documentation. It walks you through DOCA's developer zone portal which contains all the information about the DOCA toolkit from NVIDIA, providing all you need to develop NVIDIA® BlueField®-accelerated applications and the drivers for the host.

11.1.1 Introduction

The NVIDIA DOCA™ Framework enables rapidly creating and managing applications and services on top of the BlueField networking platform, leveraging industry-standard APIs. With DOCA, developers can deliver breakthrough networking, security, and storage performance by harnessing the power of NVIDIA's BlueField data-processing units (DPUs) and SuperNICs. Installing DOCA on your host provides all the necessary drivers and tools to manage NVIDIA® BlueField® and NVIDIA® ConnectX® devices.



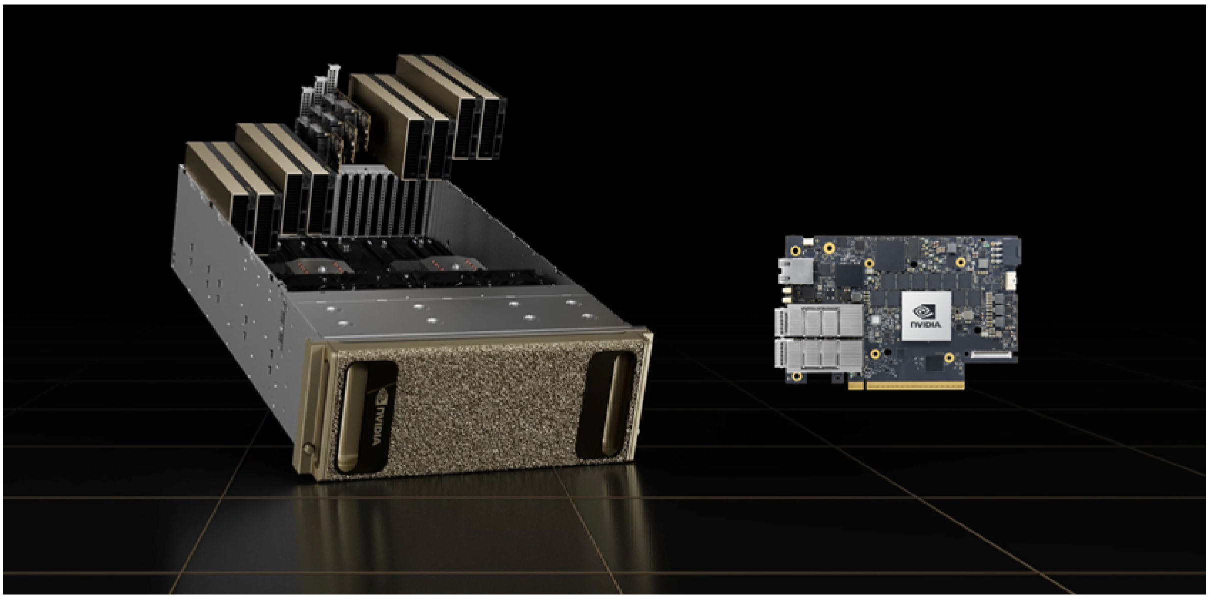
DOCA Framework includes the DOCA-Host package and the BlueField Software Bundle for BlueField Arm:

- BlueField Software Bundle (BF-Bundle) is the software package installed on the BlueField Arm cores
- DOCA-Host is the software package installed on the host server which includes different DOCA installation profiles

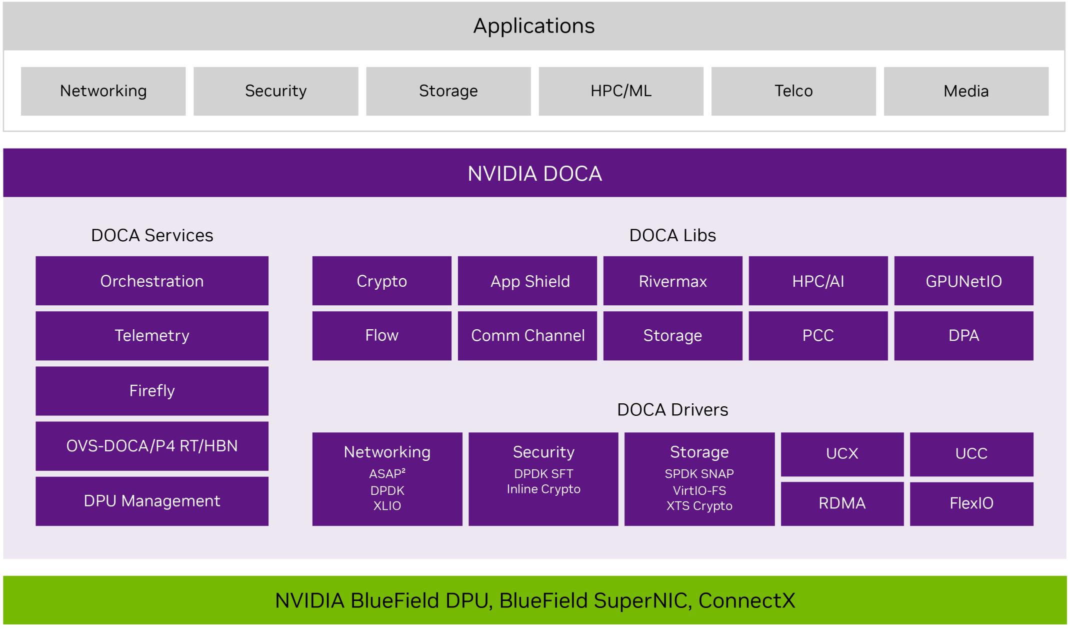
The BlueField Software Bundle includes:

- The DOCA runtime drivers and libs installed on top of the BlueField Platform
- The OS installed on the BlueField Platform
- The BlueField Platform Software (i.e., firmware and UEFI bootloader)

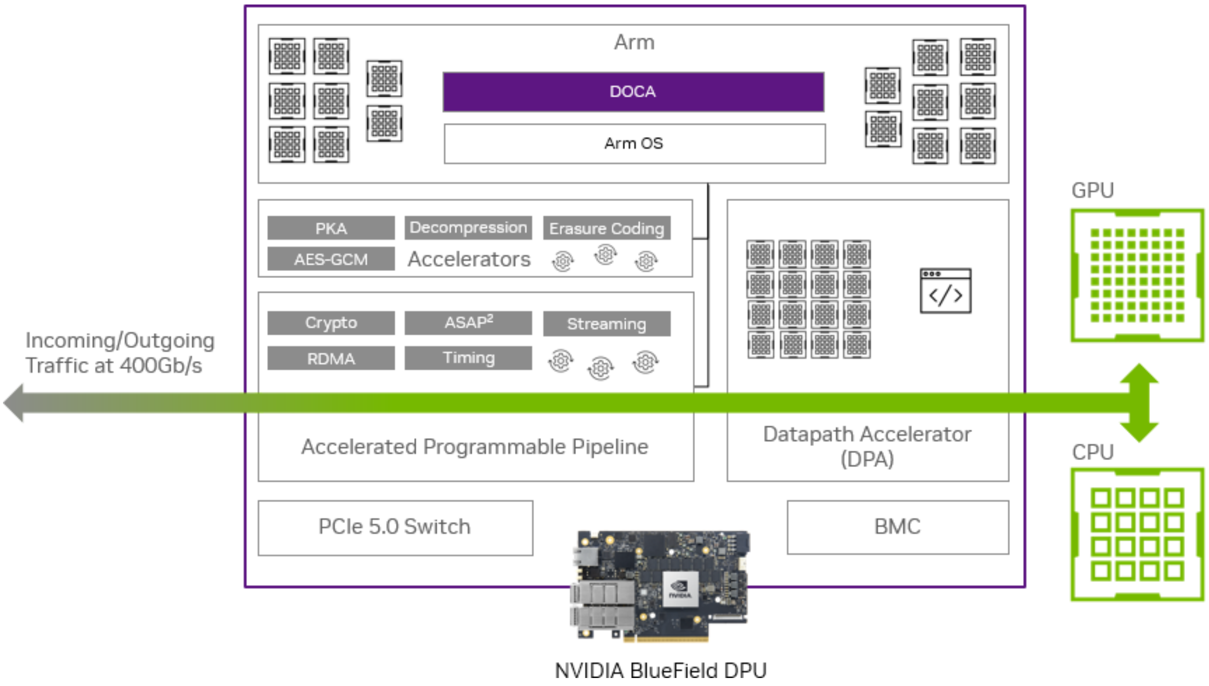
DOCA provides all the required libraries and drivers for hosts that include NVIDIA Networking platforms (i.e., BlueField and ConnectX) with a dedicated DOCA-Host package installation.



DOCA contains a runtime and development environment, including libraries and drivers for device management and programmability, for the host and as part of a BlueField Platform Software.



DOCA is the software infrastructure for BlueField's main hardware entities:



11.1.2 Installation

Installation instructions for both host and BlueField image can be found in the NVIDIA DOCA Installation Guide for Linux.

Whether DOCA has been installed on the host or on the BlueField networking platform, one can find the different DOCA components under the `/opt/mellanox/doca` directory. These include the traditional SDK-related components (libraries, header files, etc.) as well as the DOCA samples, applications, tools and more, as described in this document.

11.1.3 API

The DOCA SDK is built around the different DOCA libraries designed to leverage the capabilities of BlueField. Under the Programming Guide section, one can find a detailed description of each DOCA library, its goals, and API. These guides document DOCA's API, aiming to help develop DOCA-based programs.

The API References section holds the Doxygen-generated documentation of DOCA's official API.

11.1.4 Programming Guides

DOCA programming guides provide the full picture of DOCA libraries and their APIs. Each guide includes an introduction, architecture, API overview, and other library-specific information.

Each library's programming guide includes code snippets for achieving basic DOCA-based tasks. It is recommended to review these samples while going over the programming guide of the relevant DOCA library to learn about its API. The samples provide an implementation example of a single feature of a given DOCA library.

For a more detailed reference of full DOCA-based programs that make use of multiple DOCA libraries, please refer to the Reference Applications.

11.1.5 Applications

Applications are a higher-level reference code than the samples and demonstrate how a full DOCA-based program can be built. In addition to the supplied source code and compilation definitions, the applications are also shipped in their compiled binary form. This is to allow users an out-of-the-box interaction with DOCA-based programs without the hassle of a developer-oriented compilation process.

Many DOCA applications combine the functionality of more than one DOCA library and offer an example implementation for common scenarios of interest to users such as application recognition according to incoming/outgoing traffic, scanning files using the hardware RegEx acceleration, and much more.

For more information about DOCA applications, refer to DOCA Applications.

11.1.6 Tools

Some of the DOCA libraries are shipped alongside helper tools for both runtime and development. These tools are often an extension to the library's own API and bridge the gap between the library's expected input format and the input available to the users.

For more information about DOCA tools, refer to DOCA Tools.

11.1.7 Services

DOCA services are containerized DOCA-based programs that provide an end-to-end solution for a given use case. DOCA services are accessible as part of NVIDIA's container catalog (NGC) from which they can be easily deployed directly to BlueField, and sometimes also to the host.

For more information about container-based deployment to the BlueField Platform, refer to the NVIDIA BlueField Container Deployment Guide.

For more information about DOCA services, refer to the DOCA Services.

ⓘ For questions, comments, and feedback, please contact us at DOCA-Feedback@exchange.nvidia.com.

11.2 NVIDIA DOCA Release Notes

NVIDIA DOCA SDK release notes containing information on new features, software interoperability, and known issues.

11.2.1 Introduction

DOCA 2.8.0 introduces NVIDIA® BlueField® networking platform (DPU or SuperNIC) enhancements for high-performance and secure AI bare-metal cloud and DOCA-Host updates for supported BlueField and NVIDIA® ConnectX® devices. With programmable congestion control (PCC) and data-path acceleration (DPA). DOCA SDK provides an extensive framework for developers.

The DOCA release notes contain the following subpages:

- General Support
- Changes and New Features
- Bug Fixes in This Version
- Known Issues

11.2.2 Installation Notes

⚠ BlueField-3 devices are not supported with MLNX_OFED as the host driver and are required to use DOCA-Host.

BlueField DPUs with the following SKUs require an 8-pin ATX power supply cable connection when powering up. Without this connection to the power supply cable, the device will not complete the power-on procedure and will not function properly.

- *B3220 DPUs - 900-9D3B6-00CV-AA0 and 900-9D3B6-00SV-AA0
- *B3240 DPUs - 900-9D3B6-00CN-AB0 and 900-9D3B6-00SN-AB0
- *B3210 DPUs - 900-9D3B6-00CC-AA0 and 900-9D3B6-00SC-AA0
- *B3210E DPUs - 900-9D3B6-00CC-EA0 and 900-9D3B6-00SC-EA0

Refer to the NVIDIA DOCA Installation Guide for Linux for information on:

- Setting up DOCA SDK on your BlueField networking platform or SmartNIC
- Supported BlueField platforms

DOCA Runtime and DOCA Devel (SDK)

By default, installing DOCA profiles with standard Linux tools (yum, apt) installs both `doca-runtime` and `doca-devel` (previously `doca-sdk`).

- `doca-runtime` includes all the components, libs, drivers, and tools used in the production environment by the DOCA admin
- `doca-devel` includes all the components, libs, drivers, and tools used for development, including reference applications, compilers, etc.

Starting with DOCA 2.8.0, the default installation of BlueField-Bundle and DOCA-Host profiles will only include DOCA runtime. `doca-devel` can be installed manually as needed.

11.2.3 Supported Device Speeds

Uplink/Adapter Card	Driver Name	Uplink Speed
BlueField-2	mlx5	• InfiniBand: SDR, FDR, EDR, HDR • Ethernet: 1GbE, 10GbE, 25GbE, 40GbE, 50GbE [1] , 100GbE [1]
BlueField		• InfiniBand: SDR, QDR, FDR, FDR10, EDR • Ethernet: 1GbE, 10GbE, 25GbE, 40GbE, 50GbE, 100GbE
ConnectX-7		• InfiniBand: EDR, HDR100, HDR, NDR200, NDR • Ethernet: 1GbE, 10GbE, 25GbE, 40GbE, 50GbE [1] , 100GbE [1] , 200GbE [2] , 400GbE
ConnectX-6 Lx		• Ethernet: 1GbE, 10GbE, 25GbE, 40GbE, 50GbE [1]
ConnectX-6 Dx		• Ethernet: 10GbE, 25GbE, 40GbE, 50GbE [1] , 100GbE [1] , 200GbE [1]
ConnectX-6		• InfiniBand: SDR, FDR, EDR, HDR • Ethernet: 10GbE, 25GbE, 40GbE, 50GbE [1] , 100GbE [1] , 200GbE [1]
ConnectX-5/ConnectX-5 Ex		• InfiniBand: SDR, QDR, FDR, FDR10, EDR • Ethernet: 1GbE, 10GbE, 25GbE, 40GbE, 50GbE, 100GbE
ConnectX-4 Lx		• Ethernet: 1GbE, 10GbE, 25GbE, 40GbE, 50GbE

Uplink/Adapter Card	Driver Name	Uplink Speed
ConnectX-4		- InfiniBand: SDR, QDR, FDR, FDR10, EDR - Ethernet: 1GbE, 10GbE, 25GbE, 40GbE, 50GbE, 56GbE [3] , 100GbE

1. Speed that supports both NRZ and PAM4 modes in Force mode and Auto-Negotiation mode. ↩ ↩ ↩ ↩ ↩ ↩ ↩ ↩ ↩ ↩ ↩
2. Speed that supports PAM4 mode only. ↩
3. 56GbE is an NVIDIA proprietary link speed and can be achieved while connecting an NVIDIA adapter card to NVIDIA SX10XX switch series or when connecting an NVIDIA adapter card to another NVIDIA adapter card. ↩

11.2.4 Technical Support

Customers who purchased NVIDIA products directly from NVIDIA are invited to contact us through the following methods:

- E-mail: enterprisesupport@nvidia.com
- Enterprise Support page: https://www.nvidia.com/en-us/support/enterprise

Customers who purchased NVIDIA M-1 Global Support Services, please see your contract for details regarding Technical Support.

Customers who purchased NVIDIA products through an NVIDIA-approved reseller should first seek assistance through their reseller.

ⓘ For questions, comments, and feedback, please contact us at DOCA-Feedback@exchange.nvidia.com.

11.2.5 General Support

11.2.5.1 Embedded DOCA Firmware Components

Component	Version	Description
ATF	v2.2(release):4.8.0-41-gf0ff3a4	Arm-trusted firmware is a reference implementation of secure world software for Arm architectures
UEFI	4.8.0-36-gf01f42f	UEFI is a specification that defines the architecture of the platform firmware used for booting and its interface for interaction with the operating system
BlueField-3 NIC firmware	32.42.1000	Firmware is used to run user programs on the BlueField-3 which allow hardware to run
BlueField-2 NIC firmware	24.42.1000	Firmware is used to run user programs on the BlueField-2 which allow hardware to run
BMC firmware	24.07	BlueField BMC firmware

Component	Version	Description
BlueField-3 eROT (Glacier)	00.02.0182.0000	BlueField-3 eROT firmware
BlueField-2 eROT (CEC)	cec_ota_BMGP-04.0f	BlueField-2 eROT firmware

11.2.5.2 Supported NIC Firmware Versions

> ⚠ DOCA 2.9.0 will be the last DOCA release to support ConnectX-4. DOCA 2.9.0 will be an LTS version and will be supported for 3 years for bug fixes and CVE updates.

Adapter Card	Bundled Firmware Version
BlueField-2	24.42.1xxx
ConnectX-7	28.42.1xxx
ConnectX-6 Lx	26.42.1xxx
ConnectX-6 Dx	22.42.1xxx
ConnectX-6	20.42.1xxx
ConnectX-5/ConnectX-5 Ex	16.35.4030
BlueField	18.33.1048
ConnectX-4 Lx	14.32.1010
ConnectX-4	12.28.2006

To obtain the official firmware versions, refer to the NVIDIA Firmware Download page.

11.2.5.3 Embedded DOCA Drivers

Component	Version	Description	Licenses
clusterkit	1.14.462-1.2407052	A multifaceted node assessment tool for high-performance clusters	BSD
collectx-clxapi	1.18.2-17111037	A library which exposes the CollectX API, which allows any 3rd party to easily use CollectX functionality in their own programs	Proprietary
dpacc	1.8.0	DPACC is a high-level compiler for the DPA processor which compiles code targeted for the data-path accelerator (DPA) processor into a device executable and generates a DPA program	Proprietary
dpcp	1.1.49-1.2407052	DPCP provides a unified flexible interface for programming IB devices using DevX	Proprietary
flexio	24.04.2148-0	FlexIO SDK exposes an API for managing the device and executing native code over the DPA processor	Proprietary

Component	Version	Description	Licenses
fwctl	24.07-OFED.24.07.0.5.1.1	Subsystem designed to standardize the secure firmware interface for userspace, focusing on debugging, configuration, and provisioning	GPLv2
hcoll	4.8.3228-1.2407052	HCOLL contains support for building runtime configurable hierarchical collectives	Proprietary
ibarr	0.1.3-1.2407052	ip2gid address resolution and gid2lid path record resolution	GPL-2.0 with Linux-syscall-note or BSD-2-Clause
ibdump	6.0.0-1.2407052	Dump of InfiniBand traffic; diagnostic tool	BSD2+GPL2
ibsim	0.12-1.2407052	Open-source InfiniBand fabric simulator	GPLv2 or BSD
ibutils	2.1.1	ibdiagnet scans the fabric using directed route packets and extracts all the available information regarding its connectivity and devices.	Proprietary
ibutils2	2.1.1-0.21800.MLNX20240801.ga4352587.2407052	Utilities for InfiniBand	Proprietary
iser	24.07-OFED.24.07.0.5.2.1	Storage related drivers	GPLv2
isert	24.07-OFED.24.07.0.5.2.1	Storage related drivers	GPLv2
kernel-mft	4.29.0-127	Kernel part of MFT tools (for firmware burning, etc.)	Dual BSD/GPL
knem	1.1.4.90mlnx3-OFED.23.10.0.2.1.1	Open-source kernel module that enables high-perf intra-node MPI communication	BSD and GPLv2
libvma	9.8.60-1	The NVIDIA® Messaging Accelerator (VMA) library accelerates latency-sensitive and throughput-demanding TCP and UDP socket-based applications by offloading traffic from the user-space directly to the NIC, without going through the kernel and the standard IP stack (kernel-bypass)	GPLv2 or BSD
libxlio	3.31.2-1	The NVIDIA® XLIO software library boosts the performance of TCP/IP applications based on NGINX (CDN, DoH, etc.) and storage solutions as part of the SPDK	GPLv2 or BSD
mft	4.29.0-131	NVIDIA® MFT is a set of firmware management and debug tools for NVIDIA devices	Proprietary

Component	Version	Description	Licenses
mlnx-dpdk	22.11.0-2404	Equivalent to DPDK upstream. The versioning of MLNX_DPDK indicates which upstream DPDK it is compatible with it (e.g., 22.11 is compatible with upstream DPDK 2022.11).	BSD, LGPLv2, and GPLv2
mlnx-en	24.07-0.5.2.0.ge08362d	Kernel drivers part for Ethernet-only package	GPLv2
mlnx-ethtool	24.07-0.5.2.0.ge08362d	Ethtool with optional MLNX adaptation	GPL
mlnx-iproute2	6.9.0-1.2407052	IPRoute with optional MLNX adaptation	GPL
mlnx-libsnap	1.6.0-1	Libsnap is a common library designed to assist common tasks for applications wishing to interact with emulated hardware over BlueField and take the most advantage from hardware capabilities	Proprietary
mlnx-nfsrdma	24.07-OFED.24.07.0.5.2.1	Storage related driver for NFS over RDMA	GPLv2
mlnx-nvme	24.07-OFED.24.07.0.5.2.1	Storage related driver for NVMe	GPLv2
mlnx-ofa_kernel	24.07-OFED.24.07.0.5.2.1	Kernel drivers for Ethernet InfiniBand together	GPLv2
mlnx-snap	3.8.0-3	BlueField SNAP for NVMe and virtio-blk enables hardware-accelerated virtualization of local storage	Proprietary
mlnx-tools	24.07-0.2407052	Tools for loading modules, configurations, scripts, etc.	GPLv2 or BSD
mlx-regex	1.2-ubuntu1	RegEx is a library that provides RegEx pattern matching to DOCA applications using the regular expression processor (RXP) or software-based engines when required	Proprietary
mlx-steering-dump	1.0.0-0.2407052	Hardware/software steering dump parsing tools	GPLv2
mpitests	3.2.24-2ffc2d6.2407052	Test suite for benchmarking the MPI	BSD
mstflint	4.26.0-1	User space part of our MFT tools	GPL/BSD
multiperf	3.0-3.0.2407052	Linux tool for perf testing	BSD 3-Clause, GPL v2 or later
ofed-scripts	24.07-OFED.24.07.0.5.2	Scripts used to build OFED	GPL/BSD
openmpi	4.1.7a1-1.2407052	MPI implementation (for RDMA/RoCE) with some improvements done by the HPC team	BSD

Component	Version	Description	Licenses
opensm	5.20.0.MLNX20240801.ef1f438a-0.1.2407052	InfiniBand Subnet Manager and Subnet Administrator based on OpenSM	GPLv2 or BSD
openvswitch	2.17.8-1.2407052	OVS (virtual switch), DPDK based	ASL 2.0, LGPLv2+, and SISSL
perftest	24.07.0-0.44.g57725f2.2407052	Test suite for performance	BSD 3-Clause, GPL v2, or later
rdma-core	2407mlnx52-1.2407052	Implementation of the RDMA verbs	GPLv2 or BSD
rivermax	1.51.4	NVIDIA® Rivermax® is an optimized networking SDK for media and data streaming applications	Proprietary
rshim	2.0.38-0.gc0f82f3	The user-space driver to access the BlueField SoC via the RShim interface, providing ways to push boot stream, debug the target, or login via the virtual console or network interface	GPLv2
sharp	3.8.0.MLNX20240801.618ff287-1.2407052	Improves the performance of MPI and Machine Learning collective operation by offloading from CPUs and GPUs to the network and eliminating the need to send data multiple times between endpoints	Proprietary
sockperf	3.10-0.git5ebd327da983.2407052	Network benchmarking utility over socket API UDP/TCP designed for testing network performance (latency and throughput)	BSD
spdk	23.01.5-21	SPDK provides a set of tools and libraries for writing high performance, scalable, user-mode storage applications	Proprietary
srp	24.07-OFED.24.07.0.5.2.1	Storage-related driver for SCSI RDMA Protocol initiator	GPLv2
ucx	1.17.0-1.2407052	High-level application-oriented API for high-performance communication over RDMA networks	BSD
virtio-net-controller	24.07.11-1	Virtio-net-controller is a systemd service running on BlueField, with a user interface front-end to manage the emulated virtio-net devices	Proprietary
vma	9.8.60-1	Accelerates latency-sensitive and throughput-demanding TCP and UDP socket-based applications by offloading traffic from the user-space directly to the network interface card (NIC) or Host Channel Adapter (HCA)	GPLv2 or BSD
xlio	3.31.2-1	Boosts the performance of TCP/IP applications based on NGINX (CDN, DoH, etc.) and storage solutions as part of the SPDK	GPLv2 or BSD

Component	Version	Description	Licenses
xpmem	2.7.3-1.2407052	Kernel module to enable inter-process mapping for memory copy in user space	GPLv2 and LGPLv2.1
xpmem-lib	2.7-0.2310055	High-performance inter-process memory sharing	LGPLv2.1

11.2.5.4 DOCA Packages

Device	Component	Version	Description
Host	DOCA Devel	2.8.0	Software development kit package and tools for developing host software
	DOCA Runtime	2.8.0	Runtime libraries and tools required to run DOCA-based software applications on host
	DOCA Extra	2.8.0	Contains helper scripts (doca-info, doca-kernel-support)
	DOCA OFED	2.8.0	Software stack which operates across all NVIDIA network adapter solutions
	Arm emulated (QEMU) development container	4.8.0	Linux-based BlueField Arm emulated container for developers
Target BlueField DPU (Arm)	BlueField BSP	4.8.0	BlueField image and firmware
	DOCA SDK	2.8.0	Software development kit packages and tools for developing Arm software
	DOCA Runtime	2.8.0	Runtime libraries and tools required to run DOCA-based software applications on Arm

11.2.5.5 Supported Host OS and Features per DOCA-Host Installation Profile

The default operating system included with the BlueField Bundle (for DPU and SuperNIC) is Ubuntu 22.04.

The supported operating systems on the host machine per DOCA-Host installation profile are the following:

> ⚠ Only the following generic kernel versions are supported for DOCA local repo package for host installation.

Operating System	Architecture	Default Kernel Version (Primary)/ Tested with Kernel Version (Community)	Supported DOCA Profile			OS Support Model	ASAP^2 OVS-Kernel SR-IOV	ASAP^2 OVS-DPDK SR-IOV	NFS-over-RDMA	NVMe	GPUDirect Storage (GDS)	UCX-CUDA Version
			doca-all	doca-networking	doca-ofed							
Alinux 3.2	x86	5.10.134-13.al8.x86_64	✓	✓	✓	Primary	✗	✗	✗	✗	✗	✗
Alma 8.5	x86	4.18.0-348.12.2.EL8_5.X86_64	✗	✗	✓	Community	✗	e ✗	✗	✗	✗	✗
Anolis OS 8.4	aarch64	4.18.0-348.2.1.AN8_4.aarch64	✗	✗	✓	Community	✗	✗	✗	✗	✗	✗
	x86	4.18.0-305.AN8.X86_64	✗	✗	✓	Community	✗	✗	✗	✗	✗	✗
Anolis OS 8.6	aarch64	5.10.134+	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
	x86	5.10.134+	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
BCLinux 21.10SP2	aarch64	4.19.90-2107.6.0.0098.oe1.bclinux.aarch64	✗	✗	✓	Primary	✗	✗	✗	✓	✗	✗
	x86	4.19.90-2107.6.0.0100.oe1.bclinux.x86_64	✗	✗	✓	Primary	✗	✗	✓	✓	✗	✗
BCLinux 22.10	aarch64	5.10.0-153.24.0.100.6.oe2203sp2.bclinux.aarch64	✗	✗	✓	Primary	✗	✗	✗	✓	✗	✗

Operating System	Architecture	Default Kernel Version (Primary)/ Tested with Kernel Version (Community)	Supported DOCA Profile			OS Support Model	ASAP^2 OVS-Kernel SR-IOV	ASAP^2 OVS-DPDK SR-IOV	NFS-over-RDMA	NVMe	GPUDirect Storage (GDS)	UCX-CUDA Version
			doca-all	doca-networking	doca-ofed							
	x86	5.10.0-153.24.0.100.6.oe2203sp2.bclinux.x86_64	✗	✗	✓	Primary	✗	✗	✓	✓	✗	✗
CentOS Stream 8	aarch64	4.18.0-535.el8.aarch64	✗	✗	✓	Community	✗	✗	✗	✗	✗	✗
	x86	4.18.0-535.el8.x86_64	✗	✗	✓	Community	✗	✗	✗	✗	✗	✗
CentOS Stream 9	aarch64	5.14.0-407.el9.x86_64	✗	✗	✓	Community	✗	✗	✗	✗	✗	✗
	x86	5.14.0-407.el9.aarch64	✗	✗	✓	Community	✗	✗	✗	✗	✗	✗
CTyunOS 2.0	aarch64	4.19.90-2102.2.0.0062.ctl2.aarch64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
	x86	4.19.90-2102.2.0.0062.ctl2.x86_64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
CTyunOS 23.01	aarch64	5.10.0-136.12.0.86.ctl3.aarch64	✓	✓	✓	Primary	✗	✗	✗	✗	✗	✗
	x86	5.10.0-136.12.0.86.ctl3.x86_64	✓	✓	✓	Primary	✗	✗	✗	✗	✗	✗

Operating System	Architecture	Default Kernel Version (Primary)/ Tested with Kernel Version (Community)	Supported DOCA Profile			OS Support Model	ASAP^2 OVS-Kernel SR-IOV	ASAP^2 OVS-DPDK SR-IOV	NFS-over-RDMA	NVMe	GPUDirect Storage (GDS)	UCX-CUDA Version
			doca-all	doca-networking	doca-ofed							
Debian 10.8	aarch64	4.19.0-14-arm64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
	x86	4.19.0-14-amd64	✓	✓	✓	Primary	✗	✗	✗	✗	✗	✗
Debian 10.9	x86	4.19.0-14-amd64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
	x86	4.19.0-16-amd64	✗	✗	✓	Primary	✓	✗	✗	✓	✗	✗
Debian 10.13	aarch64	4.19.0-21-arm64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
	x86	4.19.0-21-amd64	✓	✓	✓	Primary	✗	✗	✗	✗	✗	✗
Debian 11.3	aarch64	5.10.0-13-arm64	✗	✗	✓	Primary	✓	✗	✗	✓	✗	✗
	x86	5.10.0-13-amd64	✗	✗	✓	Primary	✓	✗	✗	✓	✗	✗
Debian 12.1	aarch64	6.1.0-10-arm64	✗	✗	✓	Primary	✓	✗	✗	✓	✗	✗
	x86	6.1.0-10-amd64	✓	✓	✓	Primary	✓	✗	✗	✓	✗	✗
Debian 12.5	aarch64	6.1.0-18-arm64	✓	✗	✓	Primary	✓	✗	✗	✓	✗	✗
	x86	6.1.0-18-amd64	✓	✗	✓	Primary	✓	✗	✗	✓	✗	✗
EulerOS 2.0 SP9	aarch64	4.19.90-vhulk2006.2.0.h171.eulerosv2r9.aarch64	✗	✗	✓	Community	✗	✗	✗	✗	✗	✗

Operating System	Architecture	Default Kernel Version (Primary)/ Tested with Kernel Version (Community)	Supported DOCA Profile			OS Support Model	ASAP^2 OVS-Kernel SR-IOV	ASAP^2 OVS-DPDK SR-IOV	NFS-over-RDMA	NVMe	GPUDirect Storage (GDS)	UCX-CUDA Version
			doca-all	doca-networking	doca-ofed							
	x86	4.18.0-147.5.1.0.h269.eulerosv2r9.x86_64	✗	✗	✓	Community	✗	✗	✗	✗	✗	✗
EulerOS 2.0 SP10	aarch64	4.19.90-vhulk2110.1.0.h860.eulerosv2r10.aarch64	✗	✗	✓	Community	✗	✗	✗	✗	✗	✗
	x86	4.18.0-147.5.2.4.h694.eulerosv2r10.x86_64	✗	✗	✓	Community	✗	✗	✗	✗	✗	✗
EulerOS 2.0 SP11	aarch64	5.10.0-60.18.0.50.h323.eulerosv2r11.aarch64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
	x86	5.10.0-60.18.0.50.h323.eulerosv2r11.x86_64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
EulerOS 2.0 SP12	aarch64	5.10.0-136.12.0.86.h1032.eulerosv2r12.aarch64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗

Operating System	Architecture	Default Kernel Version (Primary)/ Tested with Kernel Version (Community)	Supported DOCA Profile			OS Support Model	ASAP^2 OVS-Kernel SR-IOV	ASAP^2 OVS-DPDK SR-IOV	NFS-over-RDMA	NVMe	GPUDirectStorage (GDS)	UCX-CUDA Version
			doca-all	doca-networking	doca-ofed							
	x86	5.10.0-136.12.0.86.h1032.eulerosv2r12.x86_64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
Kylin 1.0 SP2	aarch64	4.19.90-24.4.v2101.ky10.aarch64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
	x86	4.19.90-24.4.v2101.ky10.x86_64	✗	✗	✓	Primary	✓	✗	✗	✗	✗	✗
Kylin 1.0 SP3	aarch64	4.19.90-52.22.v2207.ky10.aarch64	✗	✗	✓	Primary	✓	✗	✗	✗	✗	✗
	x86	4.19.90-52.22.v2207.ky10.x86_64	✗	✗	✓	Primary	✓	✗	✗	✗	✗	✗
Linux Kernel 6.10	aarch64	6.10	✗	✗	✓	Primary	✓	✗	✗	✓	✗	✗
	x86		✗	✗	✓	Primary	✓	✗	✓	✓	✗	✗
Mariner 2.0	x86	5.15.148.2-2.cm2	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
Oracle Linux 7.9	x86	5.4.17-2011.6.2.el7uek.x86_64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
Oracle Linux 8.4	x86	5.4.17-2102.201.3.el8uek.x86_64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗

Operating System	Architecture	Default Kernel Version (Primary)/ Tested with Kernel Version (Community)	Supported DOCA Profile			OS Support Model	ASAP^2 OVS-Kernel SR-IOV	ASAP^2 OVS-DPDK SR-IOV	NFS-over-RDMA	NVMe	GPUDirect Storage (GDS)	UCX-CUDA Version
			doca-all	doca-networking	doca-ofed							
Oracle Linux 8.6	x86	5.4.17-2136.307.3.1.el8uek.x86_64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
Oracle Linux 8.7	x86	5.15.0-3.60.5.1.el8uek.x86_64	✓	✓	✓	Primary	✗	✗	✗	✗	✗	✗
Oracle Linux 8.8	x86	5.15.0-101.103.2.1.el8uek.x86_64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
Oracle Linux 9.0	x86	5.15.0-0.30.19.el9uek.x86_64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
Oracle Linux 9.1	x86	5.15.0-3.60.5.1.el9uek.x86_64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
Oracle Linux 9.2	x86	5.15.0-101.103.2.1.el9uek.x86_64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
OpenSUSE 15.3	aarch64	-	✗	✗	✓	Community	✗	✗	✗	✗	✗	✗
	x86	5.3.18-150300.59.43-DEFAULT	✗	✗	✓	Community	✗	✗	✗	✗	✗	✗
openEuler 20.03 SP1	aarch64	4.19.90-2012.4.0.0053.OE1.aarch64	✗	✗	✓	Community	✗	✗	✗	✗	✗	✗

Operating System	Architecture	Default Kernel Version (Primary)/ Tested with Kernel Version (Community)	Supported DOCA Profile			OS Support Model	ASAP^2 OVS-Kernel SR-IOV	ASAP^2 OVS-DPDK SR-IOV	NFS-over-RDMA	NVMe	GPUDirect Storage (GDS)	UCX-CUDA Version
			doca-all	doca-networking	doca-ofed							
	x86	4.19.90-2110.8.0.0119.OE1.X86_64	✗	✗	✓	Community	✗	✗	✗	✗	✗	✗
openEuler 20.03 SP3	aarch64	4.19.90-2112.8.0.0131.oe1.aarch64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
	x86	4.19.90-2112.8.0.0131.oe1.x86_64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
openEuler 22.03	aarch64	5.10.0-60.18.0.50.oe2203.aarch64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
	x86	5.10.0-60.18.0.50.oe2203.x86_64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
openEuler 22.03 SP1	x86	5.10.0-136.12.0.86.oe2203sp1.x86_64	✓	✗	✓	Primary	✗	✗	✗	✗	✗	✗
Photon OS 3.0	x86	4.19.225-3.ph3	✗	✗	✓	Community	✗	✗	✗	✗	✗	✗
RHEL/ CentOS 8.0	aarch64	4.18.0-80.el8.aarch64	✗	✗	✓	Primary	✓	✓	✗	✗	✓	12.5
	x86	4.18.0-80.el8.x86_64	✗	✗	✓	Primary	✓	✓	✗	✗	✓	12.5

Operating System	Architecture	Default Kernel Version (Primary)/ Tested with Kernel Version (Community)	Supported DOCA Profile			OS Support Model	ASAP^2 OVS-Kernel SR-IOV	ASAP^2 OVS-DPDK SR-IOV	NFS-over-RDMA	NVMe	GPUDirect Storage (GDS)	UCX-CUDA Version
			doca-all	doca-networking	doca-ofed							
RHEL/ CentOS 8.1	aarch64	4.18.0-147.el8.aarch64	✗	✗	✓	Primary	✓	✓	✗	✗	✓	12.5
	x86	4.18.0-147.el8.x86_64	✗	✗	✓	Primary	✓	✓	✗	✗	✓	12.5
RHEL/ CentOS 8.2	aarch64	4.18.0-193.el8.aarch64	✗	✗	✓	Primary	✓	✓	✗	✓	✓	12.5
	x86	4.18.0-193.el8.x86_64	✓	✓	✓	Primary	✓	✓	✓	✓	✓	12.5
RHEL/ CentOS 8.3	aarch64	4.18.0-240.el8.aarch64	✗	✗	✓	Primary	✓	✓	✗	✗	✓	12.5
	x86	4.18.0-240.el8.x86_64	✗	✗	✓	Primary	✓	✓	✗	✗	✓	12.5
RHEL/ CentOS 8.4	aarch64	4.18.0-305.el8.aarch64	✗	✗	✓	Primary	✓	✓	✗	✓	✓	12.5
	x86	4.18.0-305.el8.x86_64	✗	✗	✓	Primary	✓	✓	✓	✓	✓	12.5
RHEL/ CentOS 8.5	aarch64	4.18.0-348.el8.aarch64	✗	✗	✓	Primary	✓	✓	✗	✓	✓	12.5
	x86	4.18.0-348.el8.x86_64	✗	✗	✓	Primary	✓	✓	✓	✓	✓	12.5
RHEL/ Rocky 8.6	aarch64	aarch644.18.0-372.41.1.el8_6.aarch64	✗	✗	✓	Primary	✓	✓	✗	✓	✓	12.5

Operating System	Architecture	Default Kernel Version (Primary)/ Tested with Kernel Version (Community)	Supported DOCA Profile			OS Support Model	ASAP^2 OVS-Kernel SR-IOV	ASAP^2 OVS-DPDK SR-IOV	NFS-over-RDMA	NVMe	GPUDirect Storage (GDS)	UCX-CUDA Version
			doca-all	doca-networking	doca-ofed							
	x86	4.18.0-372.41.1.el8_6.x86_64	✓	✓	✓	Primary	✓	✓	✓	✓	✓	12.5
RHEL/Rocky 8.7	aarch64	4.18.0-425.14.1.el8_7.aarch64	✗	✗	✓	Primary	✓	✗	✗	✓	✓	12.5
	x86	4.18.0-425.14.1.el8_7.x86_64	✗	✗	✓	Primary	✓	✗	✓	✓	✓	12.5
RHEL/Rocky 8.8	aarch64	4.18.0-477.10.1.el8_8.aarch64	✓	✓	✓	Primary	✓	✗	✗	✓	✓	12.5
	x86	4.18.0-477.10.1.el8_8.x86_64	✓	✓	✓	Primary	✓	✗	✓	✓	✓	12.5
RHEL/Rocky 8.9	aarch64	4.18.0-513.5.1.el8_9.aarch64	✓	✓	✓	Primary	✓	✗	✗	✓	✓	12.5
	x86	4.18.0-513.5.1.el8_9.x86_64	✓	✓	✓	Primary	✓	✗	✓	✓	✓	12.5
RHEL/Rocky 8.10	aarch64	4.18.0-553.el8_10.aarch64	✓	✓	✓	Primary	✓	✗	✗	✓	✓	12.5
	x86	4.18.0-553.el8_10.x86_64	✓	✓	✓	Primary	✓	✗	✓	✓	✓	12.5

Operating System	Architecture	Default Kernel Version (Primary)/ Tested with Kernel Version (Community)	Supported DOCA Profile			OS Support Model	ASAP^2 OVS-Kernel SR-IOV	ASAP^2 OVS-DPDK SR-IOV	NFS-over-RDMA	NVMe	GPUDirect Storage (GDS)	UCX-CUDA Version
			doca-all	doca-networking	doca-ofed							
RHEL/ Rocky 9.0	aarch64	5.14.0-70.46.1.el9_0.aarch64	✗	✗	✓	Primary	✓	✗	✗	✓	✓	12.5
	x86	5.14.0-70.46.1.el9_0.x86_64	✗	✗	✓	Primary	✓	✗	✓	✓	✓	12.5
RHEL/ Rocky 9.1	aarch64	5.14.0-162.19.1.el9_1.aarch64	✗	✗	✓	Primary	✓	✗	✗	✓	✓	12.5
	x86	5.14.0-162.19.1.el9_1.x86_64	✓	✓	✓	Primary	✓	✗	✓	✓	✓	12.5
RHEL/ Rocky 9.2	aarch64	5.14.0-284.11.1.el9_2.aarch64	✗	✗	✓	Primary	✓	✗	✗	✓	✓	12.5
	x86	5.14.0-284.11.1.el9_2.x86_64	✗	✗	✓	Primary	✓	✗	✓	✓	✓	12.5
RHEL/ Rocky 9.3	aarch64	5.14.0-362.8.1.el9_3.aarch64	✗	✗	✓	Primary	✓	✗	✓	✓	✓	12.5
	x86	5.14.0-362.8.1.el9_3.x86_64	✗	✗	✓	Primary	✓	✗	✓	✓	✓	12.5
RHEL/ Rocky 9.4	aarch64	5.14.0-427.13.1.el9_4.aarch64	✗	✓	✓	Primary	✓	✗	✓	✓	✓	12.5

Operating System	Architecture	Default Kernel Version (Primary)/ Tested with Kernel Version (Community)	Supported DOCA Profile			OS Support Model	ASAP^2 OVS-Kernel SR-IOV	ASAP^2 OVS-DPDK SR-IOV	NFS-over-RDMA	NVMe	GPUDirect Storage (GDS)	UCX-CUDA Version
			doca-all	doca-networking	doca-ofed							
	x86	5.14.0-427.13.1.el9_4.x86_64	✗	✓	✓	Primary	✓	✗	✓	✓	✓	12.5
SLES 15 SP2	aarch64	5.3.18-22-default	✗	✗	✓	Primary	✓	✓	✗	✓	✗	✗
	x86	5.3.18-22-default	✗	✗	✓	Primary	✓	✓	✓	✓	✗	✗
SLES 15 SP3	aarch64	5.3.18-57-default	✗	✗	✓	Primary	✓	✗	✗	✓	✗	✗
	x86	5.3.18-57-default	✗	✗	✓	Primary	✓	✗	✓	✓	✗	✗
SLES 15 SP4	aarch64	5.14.21-150400.22-default	✗	✗	✓	Primary	✓	✗	✗	✓	✗	✗
	x86	5.14.21-150400.22-default	✗	✗	✓	Primary	✓	✗	✓	✓	✗	✗
SLES 15 SP5	aarch64	5.14.21-150500.53-default	✗	✗	✓	Primary	✓	✗	✗	✓	✗	✗
	x86	5.14.21-150500.53-default	✗	✗	✓	Primary	✓	✗	✓	✓	✗	✗
SLES 15 SP6	x86	6.4.0-150600.21-default	✗	✗	✓	Primary	✓	✗	✓	✓	✗	✗
Tencent OS 3.3	aarch64	5.4.119-19.0009.39	✗	✗	✓	Primary	✗	✗	✗	✓	✗	✗
	x86	5.4.119-19.0009.39	✗	✗	✓	Primary	✗	✗	✗	✓	✗	✗

Operating System	Architecture	Default Kernel Version (Primary)/ Tested with Kernel Version (Community)	Supported DOCA Profile			OS Support Model	ASAP^2 OVS-Kernel SR-IOV	ASAP^2 OVS-DPDK SR-IOV	NFS-over-RDMA	NVMe	GPUDirect Storage (GDS)	UCX-CUDA Version
			doca-all	doca-networking	doca-ofed							
Ubuntu 20.04	aarch64	5.4.0-26-generic	✗	✗	✓	Primary	✓	✓	✗	✓	✓	12.5
	x86	5.4.0-26-generic	✓	✓	✓	Primary	✓	✓	✓	✓	✓	12.5
Ubuntu 22.04	aarch64	5.15.0-25-generic	✓	✓	✓	Primary	✓	✓	✓	✓	✓	12.5
	x86	5.15.0-25-generic	✓	✓	✓	Primary	✓	✓	✓	✓	✓	12.5
Ubuntu 24.04	aarch64	6.6.0-14-generic	✓	✓	✓	Primary	✓	✓	✓	✓	✓	12.5
	x86	6.6.0-14-generic	✓	✓	✓	Primary	✓	✓	✓	✓	✓	12.5
UOS 20.1060	aarch64	5.10.0-46.uel20.aarch64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
	x86	5.10.0-46.uel20.x86_64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
UOS 20.1060a	aarch64	5.10.0-46.uelc20.aarch64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗
	x86	5.10.0-46.uelc20.x86_64	✗	✗	✓	Primary	✗	✗	✗	✗	✗	✗

11.2.5.6 DOCA-OFED Version Interoperability

This section reflects which versions were tested and verified for multi-version environments (i.e., environments with more than one doca-ofed version on host servers).

Target Version	Versions Verified for Interoperability
24.07-1.x.x.x July 2024	24.04-0.7.0.0 - DOCA-OFED Profile

Target Version	Versions Verified for Interoperability
	5.8-5.1.1.2 LTS

11.2.5.7 BF-Bundle (BFB) Version Upgrade/Downgrade

The following table provides a matrix for the supported upgrade/downgrade of BFBs across different versions.

Version	Upgrade to	Downgrade to
1.5.0	2.0.2; 2.2.0; 1.5.1; 1.5.2; 1.5.3	1.4.0; 1.3.0
1.5.1	1.5.2	1.5.0
1.5.2	1.5.3	1.5.1; 1.5.0
1.5.3	N/A	1.5.2; 1.5.0
2.0.2	2.2.0; 2.5.0	1.5.0; 1.4.0
2.2.0	2.5.0; 2.6.0	N/A
2.2.1	2.5.0; 2.6.0	N/A
2.5.0	2.5.1; 2.6.0	2.2.1 for BlueField-3; 2.2.0 for BlueField-2
2.5.1	2.5.2	2.5.0
2.5.2	N/A	2.5.1; 2.5.0
2.6.0	2.7.0	2.5.0; 2.2.1 for BlueField-3; 2.2.0 for BlueField-2
2.7.0	2.8.0	2.6.0; 2.5.0; 2.2.1 for BlueField-3; 2.2.0 for BlueField-2
2.8.0	N/A	2.7.0; 2.6.0; 2.5.0

11.2.5.8 Supported DOCA Version Upgrade Using Standard Linux Tools on BlueField

Version	Upgrade to
2.5.0	2.5.1; 2.6.0; 2.7.0; 2.8.0
2.5.1	2.5.2
2.5.2	N/A
2.6.0	2.7.0; 2.8.0
2.7.0	2.8.0

11.2.5.9 API Changes

> ⚠ The old DOCA Comm Channel API will be deprecated in DOCA 2.9.0.

Library	Change Description
doca_comch	• Changed features • API function name changes • API function parameter and return value changes
doca_dma	• Added features • Enable exporting DMA to GPU
doca_dpa	• Added features • Add multi-GVMI support (i.e., run DOCA DPA RDMA on VF while DOCA DPA created on PF)
doca_common	• Added features • Bitfield support • Expandable `doca_buf_inventory` • Batching support (group tasks and flash explicitly to hardware) • Set `doca_pe` (progress-engine) affinity • Changed features • Imported `doca_mmap` (to DPU) can be exported to (remote) RDMA
doca_compress	• Removed features • Decompress LZ4
doca_eth	• Added features • Ability to extend (i.e., increase) number of allocated tasks • Control notification moderation (once in `n` events or `time`) • Changed features • Parameter order in: `doca_eth_rxq_task_recv_allocate_init`
doca_gpunetio	• Added features • Support `doca_buf` on GPU (`doca_gpu_buf`) • Support dma `operations` GPU ↔ DPU/host • Changed Features • RDMA API changes
doca_pcc	• Added features • More debug/dump APIs • Performance enhancements (e.g., inline functions) • Changed features • Structure of `cc_event` - Added support for future hardware (placeholder)

11.2.5.10 Device Definition

The supported adapter cards are specified as follows:

Supported Cards	Description
All HCAs	Supported in the following adapter cards unless specifically stated otherwise: ConnectX-4/ConnectX-4 Lx/ConnectX-5/ConnectX-6/ConnectX-6 Dx/ConnectX-6 Lx/ConnectX-7/BlueField-2/BlueField-3

Supported Cards	Description
ConnectX-6 Dx and above	Supported in the following adapter cards unless specifically stated otherwise: ConnectX-6 Dx/ConnectX-6 Lx/ConnectX-7/BlueField-2/BlueField-3
ConnectX-6 and above	Supported in the following adapter cards unless specifically stated otherwise: ConnectX-6/ConnectX-6 Dx/ConnectX-6 Lx/ConnectX-7/BlueField-2/BlueField-3
ConnectX-5 and above	Supported in the following adapter cards unless specifically stated otherwise: ConnectX-5/ConnectX-6/ConnectX-6 Dx/ConnectX-6 Lx/ConnectX-7/BlueField-2/BlueField-3
ConnectX-4 and above	Supported in the following adapter cards unless specifically stated otherwise: ConnectX-4/ConnectX-4 Lx/ConnectX-5/ConnectX-6/ConnectX-6 Dx/ConnectX-6 Lx/ConnectX-7/BlueField-2/BlueField-3

11.2.5.11 Unsupported Functionalities/Features/NICs

The following are the unsupported functionalities/features/NICs in the current version:

- RDMA experimental verbs library (mlnx_lib)
- CIFS (Common Internet File System) module installation
- Relational Database Service (RDS)
- mthca InfiniBand driver
- Ethernet IPoIB (eIPoIB)
- InfiniBand Connected transport service
- IPSec over bond for crypto offload

11.2.6 Changes and New Features

11.2.6.1 New Features and Updates

> ⚠ DOCA 2.8.0 is a mandatory update release for all customers and projects with BlueField-3 DPU or SuperNIC when used in NIC mode with Arm cores disabled. This version fixes an eMMC clock toggling loop issue after boot is completed.

> ⚠ The October '24 LTS release will be the final software version to support ConnectX-4 device. Starting January '25, Connect-X 4 will no longer be supported by future DOCA-Host releases.

> ⚠ BlueField-3 networking platforms are required to use DOCA-Host as the host driver. MLNX_OFED does not support BlueField-3 devices.

- Spectrum-X 1.1 - SuperNIC Enhancements and Host Telemetry - OTLP streaming protocol

- DOCA-Flow and OVS-DOCA enhancements – Hitless upgrade/restart, micro-segmentation, "send-to-kernel" switch mode, "Basic pipe" resize, sFlow support (monitoring, debugging)
- Added alpha support for SNAP Virtio-FS file system emulation to the early access NGC service container
- Added beta support for DOCA Management Service (DMS) – systemd service in DOCA for Host package
- DOCA DPA resource allocation optimization – Allocating DPA compute resources to multiple apps
- DOCA Core – Added support for L3 cache invalidation and task batching submission/completion
- DOCA reference applications code and DOCA libs sample code is now provided under BSD-3 open-source license
- New DOCA Comch library (Comm Channel) GA, will replace the previous DOCA Comm Channel library which is scheduled to be deprecated in the next release (Oct '24). See DOCA Comch for details.
- BFB update – Added support for setting BMC password
- Added new BF-Bundle package format, `.iso`, in addition to `.bfb`

11.2.7 Bug Fixes in This Version

11.2.7.1 DOCA Bug Fixes

Ref #	Issue
3928479	Description: Users may encounter an error in "dmesg" when unplugging an emulated PCIe device.
	Keyword: DevEmu
	Reported in version: 2.7.0
3881941	Description: When working with RShim 2.0.28, PCIe host crash may rarely occur at the beginning of BFB push after the Arm reset.
	Keyword: RShim; driver
	Reported in version: 2.7.0
3882794	Description: When working with `doca_pcc_np` context, the return value from the API `doca_pcc_get_max_num_threads()` is incorrect. The function has an output parameter that indicates the maximum number of threads allowed for a `doca_pcc_np` context. The correct value that the library expects is 16 instead of the returned 64.
	Keyword: PCC; threads
	Reported in version: 2.7,0
3840230	Description: Order of cores specified in `--core-list` is not respected. Cores are picked in ascending order instead.
	Keyword: DOCA Bench
	Reported in version: 2.7,0
3849701	Description: DOCA Comch tests cannot be launched from BlueField side.

Ref #	Issue
	Keyword: DOCA Bench; DOCA Comch
	Reported in version: 2.7,0
3857097	Description: DOCA RDMA tests cannot be launched from BlueField side.
	Keyword: DOCA Bench; DOCA RDMA
	Reported in version: 2.7,0
3857095	Description: Send tasks on DOCA RDMA may fail.
	Keyword: DOCA Bench; DOCA RDMA; send
	Reported in version: 2.7,0
3859823	Description: Multi-threaded tests using DOCA Comch may hang or emit an infinite amount of log messages. Single-threaded tests are less likely to cause this issue.
	Keyword: DOCA Bench; DOCA Comch
	Reported in version: 2.7.0
3872654	Description: And issue occurs when submitting tasks with DOCA SHA with the following error. `[DOCA][ERR][doca_pe.cpp:177][task_submit] Task 0xaaaaf4865bf0: Failed to submit task: task is already submitted`
	Keyword: DOCA Bench
	Reported in version: 2.7,0
3869639	Description: Users cannot use `--job-output-buffer-size` 0 when using remote output memory (`--use-remote-output-buffers`).
	Keywords: DOCA Bench
	Reported in version: 2.7,0
3886315	Description: To reset or shut down the BlueField Arm, it is mandatory to specify the `--sync 0` argument with reset level 1 and reset type 3 or 4. For example: `mlxfwreset -d <device> -l 1 -t 4 --sync 0 r`
	Keyword: Arm; shutdown
	Reported in version: 2.7.0
3957990	Description: Sending a malformed UDP packet with VXLAN configuration causes OVS-DOCA to crash.
	Keyword: OVS-DOCA; encap; crash
	Reported in version: 2.7.0
3994490	Description: Malformed packets cause OVS to crash when performing encapsulation.
	Keyword: Openvswitch
	Reported in version: 2.7.0
3949342	Description: NVQual fails due to low line rate.

Ref #	Issue
	Keyword: NVQual
	Reported in version: 2.7.0
3960883	Description: If working with 2 different NICs with the same app, encryption can occur on the wrong port.
	Keyword: PSP gateway
	Reported in version: 2.6.0
3546202	Description: After rebooting a BlueField-3 DPU running Rocky Linux 8.6 BFB, the kernel log shows the following error: `[    3.787135] mlxbf_gige MLNXBF17:00: Error getting PHY irq. Use polling instead` This message indicates that the Ethernet driver will function normally in all aspects, except that PHY polling is enabled.
	Keywords: Linux; PHY; kernel
	Reported in version: 2.2.0

11.2.7.2 BSP Bug Fixes

Unable to render include or excerpt-include. Could not retrieve page.

11.2.7.3 BMC Bug Fixes

Unable to render include or excerpt-include. Could not retrieve page.

11.2.8 Known Issues

The following table lists the known issues and limitations for this release of DOCA SDK.

Reference	Description
4032924	Description: When upgrading to DOCA 2.8.0 on RPM-based OSes, a conflict between `strongswan-bf` or `libreswan` and strongSwan may occur.
	Workaround: Before upgrading, delete `strongswan-bf` and `libreswan`: `yum remove strongswan-bf strongswan-swanctl libreswan`
	Keyword: strongSwan; upgrade
	Reported in version: 2.8.0
4035553	Description: `oper_sample_period` does not always reflect the correct sample period. In some cases, it will reflect the `admin_sample_period` instead.

Reference	Description
	Workaround: N/A
	Keyword: Core
	Reported in version: 2.8.0
4023257	Description: If RDMA samples are compiled with memory sanitizer enabled, "read memory leak" errors are printed when running the samples with the RDMA CM flag and when running the client before the server.
	Workaround: Make sure to start the RDMA Server before RDMA Client.
	Keyword: DOCA RDMA; samples
	Reported in version: 2.8.0
4021752 4021748	Description: In all RDMA samples, if an error occurs in any of the following functions: • Exporting RDMA/MMAP/Sync event • Connecting RDMA • Writing or reading the descriptors An error is printed but the sample resumes and might: 1. Fail later, or be in busy-wait state indefinitely; and/or 2. Result in access to an unknown address, causing an address sanitizer violation.
	Workaround for 1: Either: • Follow the error logs to verify no errors occurred in the relevant function. And if it did, stop the sample. • Fix the issue locally. Workaround for 2: The mentioned address sanitizer violation shall be ignored in case of an error in a relevant function.
	Keyword: DOCA RDMA; samples
	Reported in version: 2.8.0
3961940	Description: OVS-DOCA connection tracking with E2E enabled is not supported.
	Workaround: N/A
	Keyword: OVS-DPDK; connection tracking; E2E
	Reported in version: 2.8.0
3989851	Description: A DOCA Flow pipe has multiple actions. When the action `idx` is not 0 and it has a shared endecap action, a crash occurs when attempting to create an entry.
	Workaround: N/A
	Keyword: DOCA Flow
	Reported in version: 2.8.0
3988904	Description: Failure to create a control entry with shared endecap action.
	Workaround: N/A
	Keyword: DOCA Flow
	Reported in version: 2.8.0
3886674	Description: Installing doca-all and other DOCA metapackages does not install the `mlnx-nvme` driver.

Reference	Description

	Workaround: `mlnx-nvme` is only needed for NVMe-over-RDMA remote storage support. If you wish to install it, add the `mlnx-nvme` package to the install command.
	• On RHEL:
	`apt install doca-all mlnx-nvme-modules`
	• On Ubuntu:
	`dnf install doca-all-kmod-mlnx-nvme`

	Keyword: NVMe; DOCA profile
	Reported in version: 2.7.0
3885930	Description: When installing DOCA-Host on a system using NVMe storage (typically local NVMe disk), and the script `doca-kernel-support` is used to rebuild and install kernel modules, unloading the `mlx5` drivers is only possible after also unmounting the NVMe storage, which would typically necessitate a reboot.
	Workaround: N/A
	Keyword: NVMe; doca-kernel-support; DOCA for host
	Reported in version: 2.7.0
3837255	Description: When running Arm shutdown from the host OS it is expected to get the message `-E- Failed to send Register MRSI`. This message should be ignored.
	Workaround: Wait 2 more minutes before rebooting the host. Before proceeding with host OS reboot, it is recommended to query the operational state of the BlueField Arm cores from the BlueField BMC to verify that shutdown state has been reached. Run the following command: `ipmitool -C 17 -I lanplus -H <bmc_ip> -U root -P <password> raw 0x32 0xA3` Expected output is `"06"`.
	Keyword: Host OS; reboot; error
	Reported in version: 2.7.0
3844705	Description: In OpenEuler 20.03, the Linux Kernel version 4.19.90 is affected by an issue that impacts the discard/trim functionality for the BlueField eMMC device which may cause degraded performance of the BlueField eMMC over time.
	Workaround: Upgrade to Linux Kernel version 5.10 or later.
	Keyword: eMMC discard; trim functionality
	Reported in version: 2.7.0

Reference	Description

3877725	Description: During BFB installation in NIC mode on BlueField-3, too much information is added into RShim log which fills it, causing the Linux installation progress log to not appear in the RShim log.
	`echo "DISPLAY_LEVEL 2" > /dev/rshim0/misc`
	`cat /dev/rshim0/misc`

	Workaround: Monitor the BlueField-3 Arm's UART console to check whether BFB installation has completed or not for NIC mode.
	`[13:58:39] INFO: Installation finished`
	`...`
	`[14:01:53] INFO: Rebooting...`

	Keyword: NIC mode; BFB install
	Reported in version: 2.7.0
3855702	Description: Trying to jump from a steering level in the hardware to a lower level using software steering is not supported on `rdma-core` lower than 48.x.
	Workaround: N/A
	Keyword: RDMA; SWS
	Reported in version: 2.7.0

3855485	Description: When enabling the `PCI_SWITCH_EMULATION_ENABLE` NVconfig, the mlx devices, and potentially the RShim devices disappear. Also, looking at the kernel logs using `dmesg` shows the following messages:
	`pci 0000:29:00.0: BAR 0: no space for [mem size 0x0200 0000 64bit pref]`
	`pci 0000:29:00.0: BAR 2: no space for [mem size 0x0080 0000 64bit pref]`
	`...`

	Workaround: N/A
	Keyword: NVconfig; RShim; dmsg
	Reported in version: 2.7.0
3831230	Description: In OpenEuler 20.03, the Linux Kernel version 4.19.90 is affected by an issue that impacts the discard/trim functionality for BlueField eMMC device which may cause degraded performance of BlueField eMMC over time.
	Workaround: Upgrade to Linux Kernel version 5.10 or later.
	Keyword: eMMC discard; trim functionality
	Reported in version: 2.7.0
3743879	Description: `mlxfwreset` could timeout on servers where the RShim driver is running and INTx is not supported. The following error message is printed: `BF reset flow encountered a failure due to a reset state error of negotiation timeout`.

Reference	Description

	Workaround: Set `PCIE_HAS_VFIO=0` and `PCIE_HAS_UIO=0` in `/etc/rshim.conf` and restart the RShim driver. Then re-run the `mlxfwreset` command.
	If host Linux kernel lockdown is enabled, then manually unbind the RShim driver before `mlxfwreset` and bind it back after `mlxfwreset`:
	`echo "DROP_MODE 1" > /dev/rshim0/misc`
	`mlxfwreset <arguments>`
	`echo "DROP_MODE 0" > /dev/rshim0/misc`

	Keyword: Timeout; mlxfwreset; INTx
	Reported in version: 2.7.0
3665070	Description: Virtio-net controller fails to load if `DPA_AUTHENTICATION` is enabled.
	Workaround: N/A
	Keyword: Virtio-net; DPA
	Reported in version: 2.5.0
3678069	Description: If using BlueField with NVMe and mmcbld and configured to boot from mmcblk, users must create `bf.cfg` file with `device=/dev/mmcblk0`, then install the `*.bfb` as normal.
	Workaround: N/A
	Keyword: NVMe
	Reported in version: 2.5.0
3680538	Description: When using strongSwan or OVS-IPsec as explained in the NVIDIA BlueField DPU BSP, the IPSec Rx data path is not offloaded to hardware and occurs in software running on the Arm cores. As a result, bandwidth performance is substantially low.
	Workaround: N/A
	Keyword: IPsec
	Reported in version: 2.5.0
N/A	Description: Execution unit partitions are still not implemented and would be added in a future release.
	Workaround: N/A
	Keyword: EU tool
	Reported in version: 2.5.0
3666160	Description: Installing BFB using `bfb-install` when `mlxconfig PF_TOTAL_SF` >1700, triggers server reboot immediately.
	Workaround: Change `PF_TOTAL_SF` to 0, perform a graceful shutdown, power cycle, then installing BFB.
	Keyword: SF; `PF_TOTAL_SF`; BFB installation
	Reported in version: 2.2.1

Reference	Description
3594836	Description: When enabling Flex IO SDK tracer at high rates, a slow-down in processing may occur and/or some traces may be lost.
	Workaround: Keep tracing limited to ~1M traces per second to avoid a significant processing slow-down. Use tracer for debug purposes and consider disabling it by default.
	Keyword: Tracer FlexIO
	Reported in version: 2.2.1
3592080	Description: When using UEK8 on the host in DPU mode, creating a VF on the host consumes about 100MB memory on BlueField
	Workaround: N/A
	Keyword: UEK; VF
	Reported in version: 2.2.1
3546202	Description: After rebooting a BlueField-3 DPU running Rocky Linux 8.6 BFB, the kernel log shows the following error: `[    3.787135] mlxbf_gige MLNXBF17:00: Error getting PHY irq. Use polling instead` This message indicates that the Ethernet driver will function normally in all aspects, except that PHY polling is enabled.
	Workaround: N/A
	Keyword: Linux; PHY; kernel
	Reported in version: 2.2.0
3566042	Description: Virtio hotplug is not supported in GPU-HOST mode on the NVIDIA Converged Accelerator.
	Workaround: N/A
	Keyword: Virtio; Converged Accelerator
	Reported in version: 2.2.0
3546474	Description: PXE boot over ConnectX interface might not work due to an invalid MAC address in the UEFI boot entry.
	Workaround: On BlueField, create `/etc/bf.cfg` file with the relevant PXE boot entries, then run the command `bfcfg`.
	Keyword: PXE; boot; MAC
	Reported in version: 2.2.0
3561723	Description: Running `mlxfwreset sync 1` on NVIDIA Converged Accelerators may be reported as supported although it is not. Executing the reset will fail.
	Workaround: N/A
	Keywords: mlxfwreset
	Reported in version: 2.2.0
3306489	Description: When performing longevity tests (e.g., mlxfwreset, DPU reboot, burning of new BFBs), a host running an Intel CPU may observer errors related to "CPU 0: Machine Check Exception".

Reference	Description
	Workaround: Add `intel_idle.max_cstate=1` entry to the kernel command line.
	Keywords: Longevity; mlxfwreset; DPU reboot
	Reported in version: 2.2.0
3538486	Description: When removing LAG configuration from BlueField, a kernel warning for `uverbs_destroy_ufile_hw` is observed if virtio-net-controller is still running.
	Workaround: Stop virtio-net-controller service before cleaning up bond configuration.
	Keywords: Virtio-net; LAG
	Reported in version: 2.2.0
3534219	Description: On BlueField-3 devices, from DOCA 2.2.0 to 32.37.1306 (or lower), the host crashes when executing partial Arm reset (e.g., Arm reboot; BFB push; mlxfwreset).
	Workaround: Before downgrading the firmware: 1. Run: `echo 0 > /sys/bus/platform/drivers/mlxbf-bootctl/large_icm` 2. Reboot Arm.
	Keyword: BlueField-3; downgrade
	Reported in version: 2.2.0

3462630	When trying to perform a PXE installation when UEFI Secure Boot is enabled, the following error messages may be observed:
	`error: shim_lock protocol not found.`
	`error: you need to load the kernel first.`

	Workaround: Download a Grub EFI binary from the Ubuntu website. For further information on Ubuntu UEFI Secure Boot PXE Boot, please visit Ubuntu's official website.
	Keyword: PXE; UEFI Secure Boot
	Reported in version: 2.0.2
3448841	Description: While running CentOS 8.2, switchdev Ethernet BlueField runs in "shared" RDMA net namespace mode instead of "exclusive".
	Workaround: Use `ib_core` module parameter `netns_mode=0`. For example: `echo "options ib_core netns_mode=0" >> /etc/modprobe.d/mlnx-bf.conf`
	Keyword: RDMA; isolation; Net NS
	Reported in version: 2.0.2
2706803	Description: When an NVMe controller, SoC management controller, and DMA controller are configured, the maximum number of VFs is limited to 124.
	Workaround: N/A

Reference	Description
	Keyword: VF; limitation
	Reported in version: 2.0.2
3273435	Description: Changing the mode of operation between NIC and DPU modes results in different capabilities for the host driver which might cause unexpected behavior.
	Workaround: Reload the host driver or reboot the host.
	Keyword: Modes of operation; driver
	Reported in version: 2.0.2

3264749	Description: In Rocky and CentOS 8.2 inbox-kernel BFBs, RegEx requires the following extra huge page configuration for it to function properly:
	`sudo hugeadm --pool-pages-min DEFAULT:2048M`
	`sudo systemctl start mlx-regex.service`
	`systemctl status mlx-regex.service`
	If these commands have executed successfully you should see `active (running)` in the last line of the output.

	Workaround: N/A
	Keyword: RegEx; hugepages
	Reported in version: 1.5.1
3240153	Description: DOCA kernel support only works on a non-default kernel.
	Workaround: N/A
	Keyword: Kernel
	Reported in version: 1.5.0
3217627	Description: The `doca_devinfo_rep_list_create` API returns success on the host instead of `Operation not supported`.
	Workaround: N/A
	Keyword: DOCA core; InfiniBand
	Reported in version: 1.5.0

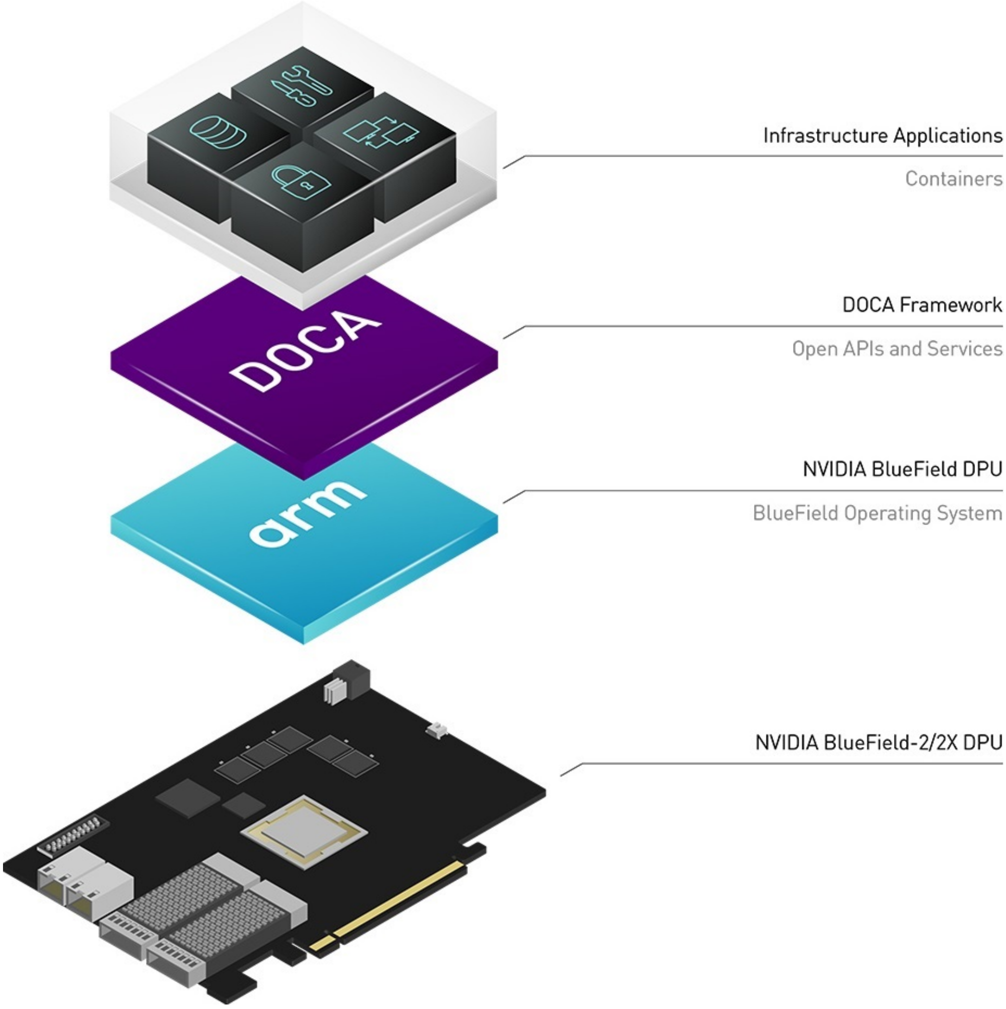
11.3 BlueField and DOCA User Types

This guide provides a quick introduction to the NVIDIA® BlueField® networking platform, its DOCA software components, and BlueField user types.

11.3.1 Introduction

The BlueField family of networking platforms includes data processing units (DPUs) and SuperNICs, and is optimized for traditional enterprise, high-performance computing (HPC), and modern cloud workloads, delivering a broad set of accelerated software-defined networking, storage, security, and management services. BlueField enables organizations to transform their IT infrastructures into state-of-the-art data centers that are accelerated, fully programmable, and armed with zero-trust security to prevent data breaches and cyber-attacks.

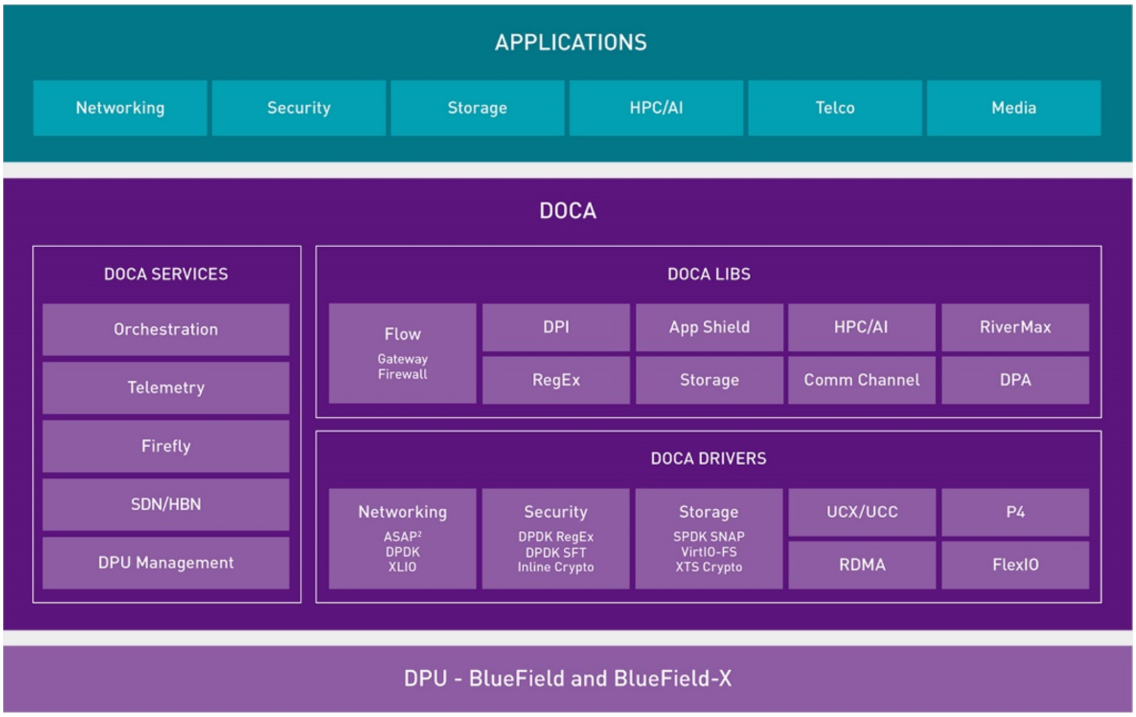
NVIDIA DOCA™ brings together a wide range of powerful APIs, libraries, and frameworks for programming and acceleration of the modern data center infrastructure. Like NVIDIA® CUDA® for GPUs, DOCA is a consistent and essential resource across all existing and future generations of BlueField products.


Infrastructure Applications
Containers
DOCA
DOCA Framework
Open APIs and Services
arm
NVIDIA BlueField DPU
BlueField Operating System
NVIDIA BlueField-2/2X DPU

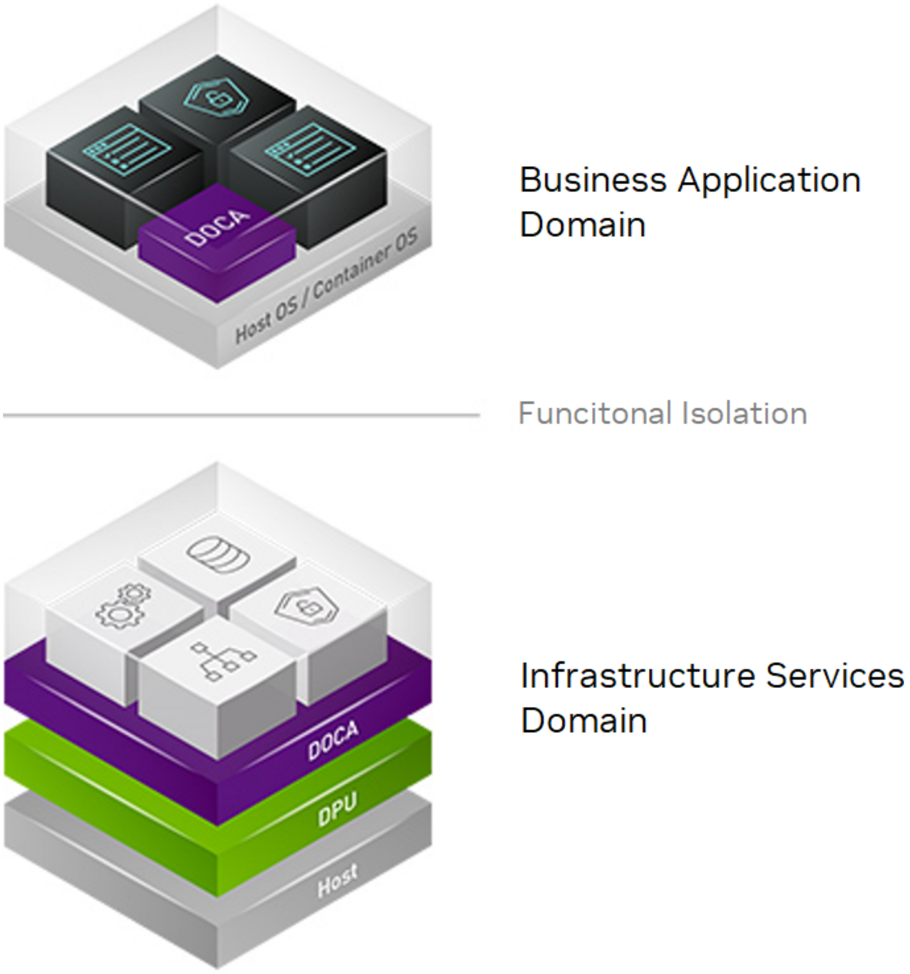
11.3.2 DOCA Components

DOCA software consists of a development and a runtime environment.

- DOCA-Devel provides industry-standard open APIs and frameworks, including Data Plane Development Kit (DPDK) and P4 for networking and security, and the Storage Performance Development Kit (SPDK) for storage. The frameworks simplify application offload with integrated NVIDIA acceleration packages. The Devel environment supports a range of operating systems and distributions and includes drivers, libraries, tools, documentation, and reference applications.



- DOCA runtime includes tools for provisioning, deploying, and orchestrating containerized services on BlueField Platforms in bulk across the data center.



11.3.3 BlueField Networking Platform User Types

11.3.3.1 BlueField Administrator

A BlueField administrator can be a system admin, an IT specialist, a security operations specialist, or anyone managing data center servers and their functionality. The admin would usually be interfacing with BlueField configuration and DOCA services and applications running on the BlueField Platform.

Common operations performed by the BlueField admin:

- Updating the BlueField image
- Running reference applications on the BlueField Platform
- Running DOCA services on the BlueField Platform

For more information, please visit BlueField Administrator Quick Start Guide.

11.3.3.2 DOCA Developer

A DOCA developer creates the services and applications that run on top of the BlueField Platform and usually interfaces with DOCA libraries and drivers to create the necessary workflow and functionality.

Common operations performed by the DOCA developer:

- Developing DOCA applications using DOCA libraries and drivers
- Compiling DOCA reference applications
- Using DOCA sample code to create a new workflow

For more information, please refer to the NVIDIA DOCA Developer Quick Start Guide.

11.4 NVIDIA DOCA EULA

NVIDIA DOCA SDK end-user license agreement.

11.4.1 End-User License Agreement

This license is a legal agreement between you and Mellanox Technologies, Ltd. ("NVIDIA Mellanox") and governs the use of the NVIDIA DOCA software and materials provided hereunder ("SOFTWARE").

This license can be accepted only by an adult of legal age of majority in the country in which the SOFTWARE is used. If you are under the legal age of majority, you must ask your parent or legal guardian to consent to this license. If you are entering this license on behalf of a company or other legal entity, you represent that you have legal authority and "you" will mean the entity you represent.

By using the SOFTWARE, you affirm that you have reached the legal age of majority, you accept the terms of this license, and you take legal and financial responsibility for the actions of your permitted users.

You agree to use the SOFTWARE only for purposes that are permitted by (a) this license, and (b) any applicable law, regulation or generally accepted practices or guidelines in the relevant jurisdictions.

1. LICENSE. Subject to the terms of this license, NVIDIA Mellanox hereby grants you a non-exclusive, non-transferable license, without the right to sublicense (except as expressly provided in this license) to:
    a. Install and use the SOFTWARE,
    b. Modify and create derivative works of sample or reference source code delivered in the SOFTWARE, and
    c. Distribute the following portions of the SOFTWARE as incorporated in object code format into a software application, subject to the distribution requirements indicated in this license: API headers, drivers, libraries and sample applications.

    BlueField SNAP software and materials, if delivered to you under this license, are licensed only for use in BlueField DPUs and subject to license fees Per DPU. "Per DPU" license means a license that allows concurrent authorized users to use the SOFTWARE in a single DPU under the license, and in some cases the SKU or documentation will indicate the maximum number of concurrent authorized users or virtual machines per DPU. Notwithstanding contrary terms in Section 1 above, you may not use or copy BlueField SNAP software without the necessary licenses.

2. DISTRIBUTION REQUIREMENTS. These are the distribution requirements for you to exercise the grants above:
    a. An application must have material additional functionality, beyond the included portions of the SOFTWARE.
    b. The following notice shall be included in modifications and derivative works of source code distributed: "This software contains source code provided by Mellanox Technologies Ltd."
    c. You agree to distribute the SOFTWARE subject to the terms at least as protective as the terms of this license, including (without limitation) terms relating to the license grant, license restrictions and protection of NVIDIA Mellanox's intellectual property rights. Additionally, you agree that you will protect the privacy, security and legal rights of your application users.
    d. You agree to notify NVIDIA Mellanox in writing of any known or suspected distribution or use of the SOFTWARE not in compliance with the requirements of this license, and to enforce the terms of your agreements with respect to the distributed portions of the SOFTWARE.
3. AUTHORIZED USERS. You may allow employees and contractors of your entity or of your subsidiary(ies) to access and use the SOFTWARE from your secure network to perform work on your behalf. If you are an academic institution you may allow users enrolled or employed by the academic institution to access and use the SOFTWARE from your secure network. You are responsible for the compliance with the terms of this license by your authorized users.
4. LIMITATIONS. Your license to use the SOFTWARE is restricted as follows:
    a. The SOFTWARE is licensed for you to develop applications only for their use in systems with NVIDIA DPUs or adapter products or related adapter products.
    b. Except as provided in this Agreement, you may not modify, reverse engineer, decompile or disassemble, or remove copyright or other proprietary notices from any portion of the SOFTWARE or copies of the SOFTWARE.
    c. You may not disclose the results of benchmarking, competitive analysis, regression or performance data relating to the SOFTWARE without the prior written permission from NVIDIA Mellanox.
    d. Except as expressly provided in this license, you may not copy, sell, rent, sublicense, transfer, distribute, modify, or create derivative works of any portion of the SOFTWARE. For clarity, unless you have an agreement with NVIDIA Mellanox for this purpose you may not distribute or sublicense the SOFTWARE as a stand-alone product.
    e. Unless you have an agreement with NVIDIA Mellanox for this purpose, you may not indicate that an application created with the SOFTWARE is sponsored or endorsed by NVIDIA Mellanox.
    f. You may not bypass, disable, or circumvent any technical limitation, encryption, security, digital rights management or authentication mechanism in the SOFTWARE.
    g. You may not replace any NVIDIA Mellanox software components in the SOFTWARE that are governed by this license with other software that implements NVIDIA Mellanox APIs.
    h. You may not use the SOFTWARE in any manner that would cause it to become subject to an open-source software license. As examples, licenses that require as a condition of use, modification, and/or distribution that the SOFTWARE be: (i) disclosed or

distributed in source code form; (ii) licensed for the purpose of making derivative works; or (iii) redistributable at no charge.

   i. Unless you have an agreement with NVIDIA Mellanox for this purpose, you may not use the SOFTWARE with any system or application where the use or failure of the system or application can reasonably be expected to threaten or result in personal injury, death, or catastrophic loss. Examples include use in avionics, navigation, military, medical, life support or other life critical applications. NVIDIA Mellanox does not design, test or manufacture the SOFTWARE for these critical uses and NVIDIA Mellanox shall not be liable to you or any third party, in whole or in part, for any claims or damages arising from such uses.
   j. You agree to defend, indemnify and hold harmless NVIDIA Mellanox and its affiliates, and their respective employees, contractors, agents, officers and directors, from and against any and all claims, damages, obligations, losses, liabilities, costs or debt, fines, restitutions and expenses (including but not limited to attorney's fees and costs incident to establishing the right of indemnification) arising out of or related to your use of the SOFTWARE outside of the scope of this license, or not in compliance with its terms.

5. UPDATES. NVIDIA Mellanox may, at its option, make available patches, workarounds or other updates to this SOFTWARE. Unless the updates are provided with their separate governing terms, they are deemed part of the SOFTWARE licensed to you as provided in this license. You agree that the form and content of the SOFTWARE that NVIDIA Mellanox provides may change without prior notice to you. While NVIDIA Mellanox generally maintains compatibility between versions, NVIDIA Mellanox may in some cases make changes that introduce incompatibilities in future versions of the SOFTWARE.
6. PRE-RELEASE VERSIONS. SOFTWARE versions identified as alpha, beta, preview, early access or otherwise as pre-release may not be fully functional, may contain errors or design flaws, and may have reduced or different security, privacy, availability, and reliability standards relative to commercial versions of NVIDIA Mellanox software and materials. You may use a pre-release SOFTWARE version at your own risk, understanding that these versions are not intended for use in production or business-critical systems. NVIDIA Mellanox may choose not to make available a commercial version of any pre-release SOFTWARE. NVIDIA Mellanox may also choose to abandon development and terminate the availability of a pre-release SOFTWARE at any time without liability.
7. COMPONENTS UNDER OTHER LICENSES. The SOFTWARE may include NVIDIA Mellanox or third-party components with separate legal notices or terms as may be described in proprietary notices accompanying the SOFTWARE, such as components governed by open source software licenses. If and to the extent there is a conflict between the terms in this license and the license terms associated with a component, the license terms associated with the components control only to the extent necessary to resolve the conflict.
8. OWNERSHIP
   a. NVIDIA Mellanox reserves all rights, title and interest in and to the SOFTWARE not expressly granted to you under this license. NVIDIA Mellanox and its suppliers hold all rights, title and interest in and to the SOFTWARE, including their respective intellectual property rights. The SOFTWARE is copyrighted and protected by the laws of the United States and other countries, and international treaty provisions.
   b. Subject to the rights of NVIDIA Mellanox and its suppliers in the SOFTWARE, you hold all rights, title and interest in and to your applications and your derivative works of the

sample source code delivered in the SOFTWARE including their respective intellectual property rights.

9. FEEDBACK. You may, but are not obligated to, provide to NVIDIA Mellanox Feedback. "Feedback" means suggestions, fixes, modifications, feature requests or other feedback regarding the SOFTWARE. Feedback, even if designated as confidential by you, shall not create any confidentiality obligation for NVIDIA Mellanox. NVIDIA Mellanox and its designees have a perpetual, non-exclusive, worldwide, irrevocable license to use, reproduce, publicly display, modify, create derivative works of, license, sublicense, and otherwise distribute and exploit Feedback as NVIDIA Mellanox sees fit without payment and without obligation or restriction of any kind on account of intellectual property rights or otherwise.
10. NO WARRANTIES. THE SOFTWARE IS PROVIDED AS-IS. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW NVIDIA MELLANOX AND ITS AFFILIATES EXPRESSLY DISCLAIM ALL WARRANTIES OF ANY KIND OR NATURE, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE. NVIDIA MELLANOX DOES NOT WARRANT THAT THE SOFTWARE WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION THEREOF WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT ALL ERRORS WILL BE CORRECTED.
11. LIMITATIONS OF LIABILITY. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW NVIDIA MELLANOX AND ITS AFFILIATES SHALL NOT BE LIABLE FOR ANY SPECIAL, INCIDENTAL, PUNITIVE OR CONSEQUENTIAL DAMAGES, OR FOR ANY LOST PROFITS, PROJECT DELAYS, LOSS OF USE, LOSS OF DATA ORLOSS OF GOODWILL, OR THE COSTS OF PROCURING SUBSTITUTE PRODUCTS, ARISING OUT OF OR IN CONNECTION WITH THIS LICENSE OR THE USE OR PERFORMANCE OF THE SOFTWARE, WHETHER SUCH LIABILITY ARISES FROM ANY CLAIM BASED UPON BREACH OF CONTRACT, BREACH OF WARRANTY, TORT (INCLUDING NEGLIGENCE), PRODUCT LIABILITY OR ANY OTHER CAUSE OF ACTION OR THEORY OF LIABILITY, EVEN IF NVIDIA MELLANOX HAS PREVIOUSLY BEEN ADVISED OF, OR COULD REASONABLY HAVE FORESEEN, THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT WILL NVIDIA MELLANOX'S AND ITS AFFILIATES TOTAL CUMULATIVE LIABILITY UNDER OR ARISING OUT OF THIS LICENSE EXCEED US$10.00. THE NATURE OF THE LIABILITY OR THE NUMBER OF CLAIMS OR SUITS SHALL NOT ENLARGE OR EXTEND THIS LIMIT.
12. TERMINATION. Your rights under this license will terminate automatically without notice from NVIDIA Mellanox if you fail to comply with any term and condition of this license or if you commence or participate in any legal proceeding against NVIDIA Mellanox with respect to the SOFTWARE. NVIDIA Mellanox may terminate this license with advance written notice to you, if NVIDIA Mellanox decides to no longer provide the SOFTWARE in a country or, in NVIDIA Mellanox's sole discretion, the continued use of it is no longer commercially viable. Upon any termination of this license, you agree to promptly discontinue use of the SOFTWARE and destroy all copies in your possession or control. Your prior distributions in accordance with this license are not affected by the termination of this license. All provisions of this license will survive termination, except for the license granted to you.
13. APPLICABLE LAW. This license will be governed in all respects by the laws of the United States and of the State of Delaware, without regard to the conflicts of laws principles. The United Nations Convention on Contracts for the International Sale of Goods is specifically disclaimed. You agree to all terms of this license in the English language. The state or federal courts residing in Santa Clara County, California shall have exclusive jurisdiction over any dispute or claim arising out of this license. Notwithstanding this, you agree that NVIDIA Mellanox shall still be allowed to apply for injunctive remedies or urgent legal relief in any jurisdiction.

14. NO ASSIGNMENT. This license and your rights and obligations thereunder may not be assigned by you by any means or operation of law without NVIDIA Mellanox's permission. Any attempted assignment not approved by NVIDIA Mellanox in writing shall be void and of no effect. NVIDIA Mellanox may assign, delegate or transfer this license and its rights and obligations, and if to a non-affiliate you will be notified.
15. EXPORT. The SOFTWARE is subject to United States export laws and regulations. You agree to comply with all applicable U.S. and international export laws, including the Export Administration Regulations (EAR) administered by the U.S. Department of Commerce and economic sanctions administered by the U.S. Department of Treasury's Office of Foreign Assets Control (OFAC). These laws include restrictions on destinations, end-users and end-use. By accepting this license, you confirm that you are not currently residing in a country or region currently embargoed by the U.S. and that you are not otherwise prohibited from receiving the SOFTWARE.
16. GOVERNMENT USE. The SOFTWARE is, and shall be treated as being, "Commercial Items" as that term is defined at 48 CFR § 2.101, consisting of "commercial computer software" and "commercial computer software documentation", respectively, as such terms are used in, respectively, 48 CFR § 12.212 and 48 CFR §§ 227.7202 & 252.227-7014(a)(1). Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions in this license pursuant to 48 CFR § 12.212 or 48 CFR § 227.7202. In no event shall the US Government user acquire rights in the SOFTWARE beyond those specified in 48 C.F.R. 52.227-19(b)(1)-(2).
17. NOTICES. Please direct your legal notices or other correspondence to NVIDIA Corporation, 2788 San Tomas Expressway, Santa Clara, California 95051, United States of America, Attention: Legal Department and NBU legal_notices@exchange.nvidia.com.
18. ENTIRE AGREEMENT. This license is the final, complete and exclusive agreement between the parties relating to the subject matter of this license and supersedes all prior or contemporaneous understandings and agreements relating to this subject matter, whether oral or written. If any court of competent jurisdiction determines that any provision of this license is illegal, invalid or unenforceable, the remaining provisions will remain in full force and effect. Any amendment or waiver under this license shall be in writing and signed by representatives of both parties.
19. LICENSING. If the distribution terms in this license are not suitable for your organization, or for any questions regarding this license, please contact NVIDIA Mellanox at doca_license@nvidia.com.

Last updated: May 10, 2022

12 Quick Start for BlueField Developers

This section contains the following pages:

- NVIDIA DOCA Developer Quick Start Guide

12.1 NVIDIA DOCA Developer Quick Start Guide

This guide details the basic steps to bring up the NVIDIA DOCA development environment and to build and run the DOCA reference applications provided along with the DOCA software framework package.

12.1.1 Introduction

NVIDIA DOCA brings together a wide range of powerful APIs, libraries, and frameworks for programming and accelerating modern data center infrastructures. Like NVIDIA® CUDA® for GPUs, DOCA is a consistent and essential resource across all existing and future generations of BlueField DPU and SuperNIC products.

This document is intended for those wishing to develop applications using the DOCA framework.

> ⚠ Not sure which installation type to use? To expand on different DOCA user types and the relevant installation for each, see BlueField and DOCA User Types.

12.1.2 Install BlueField Networking Platform

Install the BlueField networking platform into your host according to the installation instructions in the BlueField's hardware user guide. The steps include installing BlueField into the PCIe slot and properly securing it in the chassis. Make sure your host OS is listed under the supported operating systems section.

12.1.3 Install DOCA Software Package

A detailed step-by-step process for downloading and installing the required development software on both the host and BlueField can be found in the NVIDIA DOCA Installation Guide for Linux.

During installation, you must change the default password, `ubuntu` , to access the NVIDIA® BlueField® networking platform.

12.1.4 Access BlueField

After a successful installation, on the host, the RShim driver exposes a virtual Ethernet device called `tmfifo_net0` .

1. Configure the host side of the `tmfifo_net0` with a static IP to enable IPv4-based communication to the BlueField OS according to the instructions on "Host-side Interface Configuration" in the *NVIDIA BlueField DPU BSP* document.
2. Log into BlueField's Ubuntu-based OS by running the following command from the host:

```
host# ssh ubuntu@192.168.100.2
```

Use the BlueField networking platform password you defined during the installation process.

At this stage DOCA is installed on BlueField and the host server.

12.1.5 Run Reference DOCA Application

DOCA package assets (e.g., references, tools) are located on Bluefield and on the host under `/opt/mellanox/doca/`.

The DOCA package includes a set of reference applications to facilitate developer on-boarding. Please refer to the DOCA Reference Applications and DOCA Programming Guide for more information.

To run the DOCA Secure Channel reference application which demonstrates accelerated and secure message transmission between the host and BlueField over the Comm Channel interface:

1. Run the application as server on the BlueField networking platform using the following command (all parameters are available in the secure channel application guide):

```
# /opt/mellanox/doca/applications/secure_channel/bin/doca_secure_channel -s 256 -n 10 -p 03:00.0 -r 3b:00.0
```

2. Run the application as client on the host using the following command (all parameters are available in the secure channel application guide):

```
# /opt/mellanox/doca/applications/secure_channel/bin/doca_secure_channel -s 256 -n 10 -p 3b:00.0
```

12.1.6 More Information

To learn more about NVIDIA BlueField networking platforms, refer to the NVIDIA BlueField Hardware Manuals.

For questions, comments, and feedback, please contact us at DOCA-Feedback@exchange.nvidia.com.

13 Installation and Setup

This section contains the following page:

- NVIDIA DOCA Profiles
- NVIDIA DOCA Installation Guide for Linux
- NVIDIA DOCA Developer Guide

13.1 NVIDIA DOCA Profiles

The following document provides an introduction to the various supported DOCA-Host profiles.

13.1.1 Introduction

NVIDIA DOCA™ can be installed on the host and used by a variety of customers who have different workloads and requirements. The DOCA-Host package includes drivers, libraries, and tools to support NVIDIA® BlueField® Networking Platform and NVIDIA® ConnectX® SmartNIC, Ethernet and InfiniBand, with both kernel and user-space components. Depending on their specific needs, customers may choose not to install the full DOCA-Host package on their host server but only the subset of components and tools relevant for their use case (whether to have a smaller installation size, lower integration/validation effort, etc).

To support the different use cases, DOCA includes DOCA-Host Installation Profiles, which are a subset of the full DOCA installation. DOCA-Host profiles are validated and tested installation packages. The following are the available DOCA profiles:

- doca-all
- doca-networking
- doca-ofed

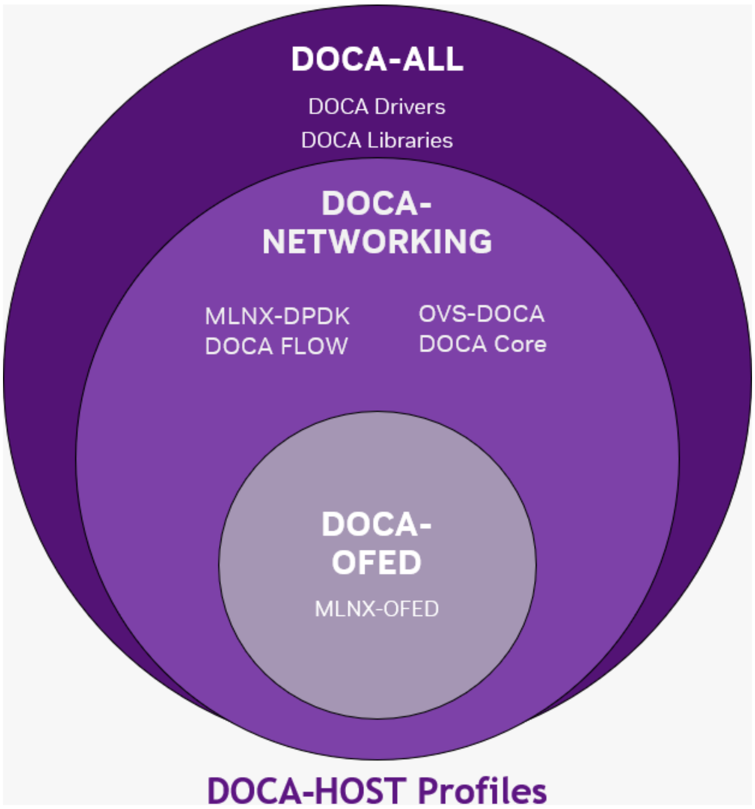
DOCA-Host supports the following NVIDIA devices:

- BlueField-3
- BlueField-2
- ConnectX-7
- ConnectX-6 DX
- ConnectX-6 LX
- ConnectX-6
- ConnectX-5
- ConnectX-4 LX
- ConnectX-4

For hardware details on these devices, refer to the following pages:

- BlueField devices
- ConnectX devices

DOCA functionality is limited by the specific device capabilities.



13.1.2 doca-all

The full DOCA-Host installation is intended for users who wish to utilize the full extent of DOCA libs and drivers.

This profile is the super-set of components, which also includes the content of doca-ofed and doca-networking.

All DOCA libraries, drivers and tools are included in doca-all.

> When installing doca-all on host, BlueField Platforms can utilize all DOCA libs and drivers whereas ConnectX devices can utilize only doca-ofed and doca-networking subset of functions from within the super-set of doca-all, depending on the device's capabilities.

13.1.3 doca-networking

The doca-networking profile is intended for users who wish to benefit only from the networking functionality of DOCA.

The content of the doca-networking package is the following:

- MLNX_OFED
- DOCA Core
- MLNX-DPDK
- OVS-DOCA
- DOCA Flow

ⓘ BlueField DPUs, BlueField SuperNICs, and ConnectX devices can utilize all included libs and drivers in the doca-networking profile, based on the device's capabilities.

13.1.4 doca-ofed

This profile is intended for users who wish to have the same user experience and content as MLNX_OFED but with DOCA package. doca-ofed installs the MLNX_OFED drivers and tools and does not include any other DOCA components.

The content of the doca-ofed package is:

- MLNX_OFED drivers and tools

ⓘ BlueField Platforms and ConnectX devices can utilize only the drivers in doca-ofed, based on the device's capabilities. No added DOCA libs are supported with any of the devices with doca-ofed profile installation.

13.1.5 Which Profile to Install?

Selecting the right DOCA-Host installation profile is important to fully utilize the capabilities of your BlueField Platforms or ConnectX.

The functionality of DOCA-Host is limited by the device capabilities (e.g., ConnectX devices cannot utilize DOCA libs such as DPA, even if doca-all is installed on the host).

For BlueField devices:

- It is recommended to use doca-all
- If you require the smallest installation package for networking-only purposes, use doca-networking
- For MLNX_OFED-like installation, use doca-ofed (no additional DOCA functionality)

For ConnectX devices:

- It is recommended to use doca-networking
- For future-proof and mixed BlueField/ConnectX deployments, use doca-all
- For MLNX_OFED-like installation use doca-ofed (no additional DOCA functionality)

13.1.6 DOCA-Host Profile Installation

DOCA-Host can be installed on specific host OSs. Each of the Host Installation Profiles has specific OSs on which is can be installed as specified in section "Supported Host OS per DOCA-Host Installation Profile".

Follow the instructions under section "Installing Software on Host" in the *NVIDIA DOCA Installation Guide for Linux*.

13.1.7 Supported Host OS per DOCA-Host Installation Profile

Unable to render include or excerpt-include. Could not retrieve page.

13.1.8 NVIDIA MLNX_OFED to DOCA-OFED Transition Guide

This guide covers what users must know about the DOCA-Host unified software stack for NVIDIA networking products.

13.1.8.1 Introduction

MLNX_OFED is a software stack that provides kernel drivers, user space libraries, and management tools for NVIDIA networking products, including ConnectX and BlueField adapters. MLNX_OFED has been the standard software stack for NVIDIA networking products for many years, providing high performance, scalability, and compatibility with various operating systems and applications.

With the introduction of NVIDIA BlueField Networking Platform and DOCA as the software framework to support it, there are 2 host-server software packages dedicated for different devices.

DOCA-Host is the unified software package for your host-server, supporting both BlueField and ConnectX. Customers may choose to use the Inbox drivers of the operating system vendor. The drivers with the latest features are included as part of NVIDIA software packages, and specifically DOCA-OFED.

13.1.8.2 What is DOCA-Host?

DOCA-Host can be installed on the host-server and used by customers with different workloads and requirements. The DOCA-Host package includes drivers, libraries, and tools to support the NVIDIA® BlueField® networking platform (DPU or SuperNIC) and NVIDIA® ConnectX® SmartNIC, Ethernet and InfiniBand, with both kernel and user-space components. Depending on their needs, customers may choose not to install the full DOCA-Host package on their host server but only the subset of components and tools relevant for their use case.

To support the different use cases, DOCA includes DOCA-Host Installation Profiles, which are a subset of the full DOCA installation.

> ⓘ DOCA-Host profiles are validated and tested installation packages.

The following are the available DOCA profiles:

- doca-all – intended for users who wish to utilize the full extent of DOCA libraries and drivers
- doca-networking – intended for users who wish to benefit only from the networking functionality of DOCA
- doca-ofed – intended for users who wish to have the same user experience and content as MLNX_OFED. Doca-ofed installs the MLNX_OFED drivers and tools and does not include any other DOCA components.

13.1.8.3 What is DOCA-OFED?

DOCA-OFED is an equivalent package of MLNX_OFED, providing the same functionality as MLNX_OFED and including the same kernel drivers, user space libraries, and management tools for NVIDIA networking products. DOCA-OFED supports the same OSs and applications as MLNX_OFED.

13.1.8.4 Why Switch to DOCA-OFED and DOCA-Host?

DOCA-OFED is a 1-to-1 substitute for MLNX_OFED. All customers using MLNX_OFED on their host-server should install DOCA-OFED instead.

Following the last release of MLNX_OFED, no new features will be added to MLNX_OFED. All new features will only be included as part of DOCA-OFED.

13.1.8.5 Switching to DOCA-OFED and DOCA-Host

Switching to DOCA-Host with any of the installation profiles, and specifically DOCA-OFED, is a straightforward process. You just need to follow these steps:

1. Download the latest DOCA-Host package from the NVIDIA website or public repo.
2. Uninstall the existing MLNX_OFED package from your system.
3. Install the DOCA-OFED package on your host server using standard Linux package manager.
4. Reboot your system and verify that the DOCA-OFED components are working properly.

13.1.8.5.1 Installation Example of DOCA-OFED from Online Repo

```
# echo "[doca]
name=DOCA Online Repo
baseurl=https://linux.mellanox.com/public/repo/doca/2.7.0/rhel9.4/x86_64/
enabled=1
# gpgcheck=0" > /etc/yum.repos.d/doca.repo
# sudo dnf clean all
# sudo dnf -y install doca-ofed
```

13.1.8.5.2 Installation Example of DOCA-OFED Offline Repo

```
# wget https://www.mellanox.com/downloads/DOCA/DOCA_v2.7.0/host/doca-host-2.7.0-209000_24.04_rhel94.x86_64.rpm
# sudo rpm -i doca-host-2.7.0-209000_24.04_rhel94.x86_64.rpm
# sudo dnf clean all
# sudo dnf -y install doca-ofed
```

13.1.8.6 MLNX_EN Transition

With the transition from MLNX_OFED, the MLNX_EN lite weight software package will also no longer be supported. Customers who wish to get the smaller package of drivers available via MLNX_EN thus far, are advised to use Inbox drivers, providing the same components.

DOCA-Host will also support a new installation profile, DOCA-RoCE, which is a subset of DOCA_OFED and includes only Ethernet and RoCE drivers, without IB specific components. So, customers can also use this profile which includes more content than MLNX_EN.

13.1.8.7 Transition Timeline

The transition timeline from MLNX_OFED to DOCA-OFED gives users enough time to switch to the new software stack. The timeline for the transition is as follows:

- October 2024 – The last standalone release of MLNX_OFED. Following this release, MLNX_OFED will no longer receive support for new features or enhancements.

> ✓ Customers are encouraged to switch to DOCA-OFED as soon as possible to stay up-to-date on new features and enhancements for NVIDIA networking products.

- October 2024-October 2027 – The last standalone MLNX_OFED release will receive critical bug fixes and security updates for MLNX_OFED users as part of its long-term support (LTS) plan
- October 2027 – MLNX_OFED will no longer receive support or updates by NVIDIA (MLNX_OFED end of life)

> Users are strongly advised to switch to DOCA-OFED before this date, to avoid any compatibility or security issues.

13.1.8.8 Summary

DOCA-OFED is the new software stack for NVIDIA networking products, with the exact same user experience as MLNX_OFED. Users are encouraged to switch to DOCA-OFED as soon as possible to enjoy the full potential of NVIDIA Networking products. Users can download the latest DOCA-OFED package from the NVIDIA website or directly from DOCA public repo, and follow the simple installation steps.

The last standalone release of MLNX_OFED will be October 2024. Afterwards, MLNX_OFED enters the LTS period and will only receive critical bug fixes and security updates for 3 years.

> In October 2027, MLNX_OFED will no longer be supported or updated by NVIDIA.

13.2 NVIDIA DOCA Installation Guide for Linux

This guide details the necessary steps to set up NVIDIA DOCA in your Linux environment.

13.2.1 Introduction

Installation of the NVIDIA® BlueField® networking platform (DPU or SuperNIC) software requires following the following step-by-step procedure.

13.2.1.1 Supported Platforms

13.2.1.1.1 Supported BlueField Platforms

The following NVIDIA® BlueField® platforms are supported with DOCA:

NVIDIA SKU	Legacy OPN	PSID	Description
900-9D3B6-00CV-AA0	N/A	MT_0000000884	BlueField-3 B3220 P-Series FHHL DPU; 200GbE (default mode) / NDR200 IB; Dual-port QSFP112; PCIe Gen5.0 x16 with x16 PCIe extension option; 16 Arm cores; 32GB on-board DDR; integrated BMC; Crypto Enabled

NVIDIA SKU	Legacy OPN	PSID	Description
900-9D3B6-00SV-AA0	N/A	MT_0000000965	BlueField-3 B3220 P-Series FHHL DPU; 200GbE (default mode) / NDR200 IB; Dual-port QSFP112; PCIe Gen5.0 x16 with x16 PCIe extension option; 16 Arm cores; 32GB on-board DDR; integrated BMC; Crypto Disabled
900-9D3B6-00CC-AA0	N/A	MT_0000001024	BlueField-3 B3210 P-Series FHHL DPU; 100GbE (default mode) / HDR100 IB; Dual-port QSFP112; PCIe Gen5.0 x16 with x16 PCIe extension option; 16 Arm cores; 32GB on-board DDR; integrated BMC; Crypto Enabled
900-9D3B6-00SC-AA0	N/A	MT_0000001025	BlueField-3 B3210 P-Series FHHL DPU; 100GbE (default mode) / HDR100 IB; Dual-port QSFP112; PCIe Gen5.0 x16 with x16 PCIe extension option; 16 Arm cores; 32GB on-board DDR; integrated BMC; Crypto Disabled
900-9D219-0086-ST1	MBF2M516A-CECOT	MT_0000000375	BlueField-2 E-Series DPU 100GbE Dual-Port QSFP56; PCIe Gen4 x16; Crypto and Secure Boot Enabled; 16GB on-board DDR; 1GbE OOB management; FHHL
900-9D219-0086-ST0	MBF2M516A-EECOT	MT_0000000376	BlueField-2 E-Series DPU 100GbE/EDR/HDR100 VPI Dual-Port QSFP56; PCIe Gen4 x16; Crypto and Secure Boot Enabled; 16GB on-board DDR; 1GbE OOB management; FHHL
900-9D219-0056-ST1	MBF2M516A-EENOT	MT_0000000377	BlueField-2 E-Series DPU 100GbE/EDR/HDR100 VPI Dual-Port QSFP56; PCIe Gen4 x16; Crypto Disabled; 16GB on-board DDR; 1GbE OOB management; FHHL
900-9D206-0053-SQ0	MBF2H332A-AENOT	MT_0000000539	BlueField-2 P-Series DPU 25GbE Dual-Port SFP56; PCIe Gen4 x8; Crypto Disabled; 16GB on-board DDR; 1GbE OOB management; HHHL
900-9D206-0063-ST2	MBF2H332A-AEEOT	MT_0000000540	BlueField-2 P-Series DPU 25GbE Dual-Port SFP56; PCIe Gen4 x8; Crypto Enabled; 16GB on-board DDR; 1GbE OOB management; HHHL
900-9D206-0083-ST3	MBF2H332A-AECOT	MT_0000000541	BlueField-2 P-Series DPU 25GbE Dual-Port SFP56; PCIe Gen4 x8; Crypto and Secure Boot Enabled; 16GB on-board DDR; 1GbE OOB management; HHHL
900-9D206-0083-ST1	MBF2H322A-AECOT	MT_0000000542	BlueField-2 P-Series DPU 25GbE Dual-Port SFP56; PCIe Gen4 x8; Crypto and Secure Boot Enabled; 8GB on-board DDR; 1GbE OOB management; HHHL
900-9D206-0063-ST1	MBF2H322A-AEEOT	MT_0000000543	BlueField-2 P-Series DPU 25GbE Dual-Port SFP56; PCIe Gen4 x8; Crypto Enabled; 8GB on-board DDR; 1GbE OOB management; HHHL

NVIDIA SKU	Legacy OPN	PSID	Description
900-9D219-0066-ST0	MBF2M516A-EEEOT	MT_0000000559	BlueField-2 E-Series DPU 100GbE/EDR/HDR100 VPI Dual-Port QSFP56; PCIe Gen4 x16; Crypto Enabled; 16GB on-board DDR; 1GbE OOB management; FHHL
900-9D219-0056-SN1	MBF2M516A-CENOT	MT_0000000560	BlueField-2 E-Series DPU 100GbE Dual-Port QSFP56; PCIe Gen4 x16; Crypto Disabled; 16GB on-board DDR; 1GbE OOB management; FHHL
900-9D219-0066-ST2	MBF2M516A-CEEOT	MT_0000000561	BlueField-2 E-Series DPU 100GbE Dual-Port QSFP56; PCIe Gen4 x16; Crypto Enabled; 16GB on-board DDR; 1GbE OOB management; FHHL
900-9D219-0006-ST0	MBF2H516A-CEEOT	MT_0000000702	BlueField-2 DPU 100GbE Dual-Port QSFP56; PCIe Gen4 x16; Crypto; 16GB on-board DDR; 1GbE OOB management; FHHL
900-9D219-0056-ST2	MBF2H516A-CENOT	MT_0000000703	BlueField-2 DPU 100GbE Dual-Port QSFP56; PCIe Gen4 x16; Crypto Disabled; 16GB on-board DDR; 1GbE OOB management; FHHL
900-9D219-0066-ST3	MBF2H516A-EEEOT	MT_0000000704	BlueField-2 DPU 100GbE/EDR/HDR100 VPI Dual-Port QSFP56; PCIe Gen4 x16; Crypto Enabled; 16GB on-board DDR; 1GbE OOB management; FHHL
900-9D219-0056-SQ0	MBF2H516A-EENOT	MT_0000000705	BlueField-2 DPU 100GbE/EDR/HDR100 VPI Dual-Port QSFP56; PCIe Gen4 x16; Crypto Disabled; 16GB on-board DDR; 1GbE OOB management; FHHL
900-9D250-0038-ST1	MBF2M345A-HESOT	MT_0000000715	BlueField-2 E-Series DPU; 200GbE/HDR single-port QSFP56; PCIe Gen4 x16; Secure Boot Enabled; Crypto Disabled; 16GB on-board DDR; 1GbE OOB management; HHHL
900-9D250-0048-ST1	MBF2M345A-HECOT	MT_0000000716	BlueField-2 E-Series DPU; 200GbE/HDR single-port QSFP56; PCIe Gen4 x16; Secure Boot Enabled; Crypto Enabled; 16GB on-board DDR; 1GbE OOB management; HHHL
900-9D218-0073-ST1	MBF2H512C-AESOT	MT_0000000723	BlueField-2 P-Series DPU 25GbE Dual-Port SFP56; integrated BMC; PCIe Gen4 x8; Secure Boot Enabled; Crypto Disabled; 16GB on-board DDR; 1GbE OOB management; FHHL
900-9D218-0083-ST2	MBF2H512C-AECOT	MT_0000000724	BlueField-2 P-Series DPU 25GbE Dual-Port SFP56; integrated BMC; PCIe Gen4 x8; Secure Boot Enabled; Crypto Enabled; 16GB on-board DDR; 1GbE OOB management; FHHL

NVIDIA SKU	Legacy OPN	PSID	Description
900-9D208-0086-ST4	MBF2M516C-EECOT	MT_0000000728	BlueField-2 E-Series DPU 100GbE/EDR/HDR100 VPI Dual-Port QSFP56; integrated BMC; PCIe Gen4 x16; Secure Boot Enabled; Crypto Enabled; 16GB on-board DDR; 1GbE OOB management; Tall Bracket; FHHL
900-9D208-0086-SQ0	MBF2H516C-CECOT	MT_0000000729	BlueField-2 P-Series DPU 100GbE Dual-Port QSFP56; integrated BMC; PCIe Gen4 x16; Secure Boot Enabled; Crypto Enabled; 16GB on-board DDR; 1GbE OOB management; Tall Bracket; FHHL
900-9D208-0076-ST5	MBF2M516C-CESOT	MT_0000000731	BlueField-2 E-Series DPU 100GbE Dual-Port QSFP56; integrated BMC; PCIe Gen4 x16; Secure Boot Enabled; Crypto Disabled; 16GB on-board DDR; 1GbE OOB management; Tall Bracket; FHHL
900-9D208-0076-ST6	MBF2M516C-EESOT	MT_0000000732	BlueField-2 E-Series DPU 100GbE/EDR/HDR100 VPI Dual-Port QSFP56; integrated BMC; PCIe Gen4 x16; Secure Boot Enabled; Crypto Disabled; 16GB on-board DDR; 1GbE OOB management; Tall Bracket; FHHL
900-9D208-0086-ST3	MBF2M516C-CECOT	MT_0000000733	BlueField-2 E-Series DPU 100GbE Dual-Port QSFP56; integrated BMC; PCIe Gen4 x16; Secure Boot Enabled; Crypto Enabled; 16GB on-board DDR; 1GbE OOB management; Tall Bracket; FHHL
900-9D208-0076-ST2	MBF2H516C-EESOT	MT_0000000737	BlueField-2 P-Series DPU 100GbE/EDR/HDR100 VPI Dual-Port QSFP56; integrated BMC; PCIe Gen4 x16; Secure Boot Enabled; Crypto Disabled; 16GB on-board DDR; 1GbE OOB management; Tall Bracket; FHHL
900-9D208-0076-ST1	MBF2H516C-CESOT	MT_0000000738	BlueField-2 P-Series DPU 100GbE Dual-Port QSFP56; integrated BMC; PCIe Gen4 x16; Secure Boot Enabled; Crypto Disabled; 16GB on-board DDR; 1GbE OOB management; Tall Bracket; FHHL
900-9D218-0083-ST4	MBF2H532C-AECOT	MT_0000000765	BlueField-2 P-Series DPU 25GbE Dual-Port SFP56; integrated BMC; PCIe Gen4 x8; Secure Boot Enabled; Crypto Enabled; 32GB on-board DDR; 1GbE OOB management; FHHL
900-9D218-0073-ST0	MBF2H532C-AESOT	MT_0000000766	BlueField-2 P-Series DPU 25GbE Dual-Port SFP56; integrated BMC; PCIe Gen4 x8; Secure Boot Enabled; Crypto Disabled; 32GB on-board DDR; 1GbE OOB management; FHHL
900-9D208-0076-ST3	MBF2H536C-CESOT	MT_0000000767	BlueField-2 P-Series DPU 100GbE Dual-Port QSFP56; integrated BMC; PCIe Gen4 x16; Secure Boot Enabled; Crypto Disabled; 32GB on-board DDR; 1GbE OOB management; FHHL

NVIDIA SKU	Legacy OPN	PSID	Description
900-9D208-0086-ST2	MBF2H536C-CECOT	MT_0000000768	BlueField-2 P-Series DPU 100GbE Dual-Port QSFP56; integrated BMC; PCIe Gen4 x16; Secure Boot Enabled; Crypto Enabled; 32GB on-board DDR; 1GbE OOB management; FHHL
900-9D218-0073-ST4	MBF2H512C-AEUOT	MT_0000000972	BlueField-2 P-Series DPU 25GbE Dual-Port SFP56; integrated BMC; PCIe Gen4 x8; Secure Boot Enabled with UEFI disabled; Crypto Disabled; 16GB on-board DDR; 1GbE OOB management
900-9D208-0076-STA	MBF2H516C-CEUOT	MT_0000000973	BlueField-2 P-Series DPU 100GbE Dual-Port QSFP56; integrated BMC; PCIe Gen4 x16; Secure Boot Enabled with UEFI disabled; Crypto Disabled; 16GB on-board DDR; 1GbE OOB management
900-9D208-0076-STB	MBF2H536C-CEUOT	MT_0000001008	BlueField-2 P-Series DPU 100GbE Dual-Port QSFP56, integrated BMC, PCIe Gen4 x16, Secure Boot Enabled with UEFI Disabled, Crypto Disabled, 32GB on-board DDR, 1GbE OOB management, Tall Bracket, FHHL
P1004/699210040230	N/A	NVD0000000015	BlueField-2 A30X, P1004 SKU 205, Generic, GA100, 24GB HBM2e, PCIe passive Dual Slot 230W GEN4, DPU Crypto ON W/ Bkt, 1 Dongle, Black, HF, VCPD
P4028/699140280000	N/A	NVD0000000020	ZAM / NAS

13.2.1.1.2 Supported ConnectX NICs

The NVIDIA® ConnectX® NICs supported with DOCA-Host can be found in: NVIDIA DOCA Profiles

13.2.1.2 Hardware Prerequisites

For BlueField Platform users, this guide assumes that a BlueField device has been installed in a server according to the instructions detailed in your DPU's hardware user guide.

13.2.1.3 DOCA Packages

See information in the NVIDIA DOCA Release Notes page.

13.2.1.4 Supported Host OS per DOCA-Host Installation Profile

See information in the NVIDIA DOCA Profiles page.

13.2.2 BlueField Networking Platform Image Installation

This guide provides the minimal instructions for setting up DOCA on a standard system.

Important!

Make sure to follow the instructions in this section sequentially. Make sure to update DOCA on the host side first before installing the BFB Bundle on the BlueField.

13.2.2.1 Installation Files

To download the DOCA for host packages from the links in this table, please register to the NVIDIA Developer Program. Otherwise, please use the public repo from DOCA Downloader.

Device	Component	OS	Arch	Link
Host	These files contain the following components suitable for their respective OS version. Please take a look on the NVIDIA DOCA Profiles to know which from the below profiles are supported on your desired OS. • DOCA-All v2.8.0 • DOCA-Networking v2.8.0 • DOCA-OFED v2.8.0 • DOCA-Extra v2.8.0 (included in all)	Alinux 3.2	x86	doca-host-2.8.0-204000_24.07_alinux32.x86_64.rpm
		Anolis	aarch64	doca-host-2.8.0-204000_24.07_anolis86.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_anolis86.x86_64.rpm
		BCLinux 21.10	aarch64	doca-host-2.8.0-204000_24.07_bclinux2210.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_bclinux2210.x86_64.rpm
		BCLinux 21.10 SP2	aarch64	doca-host-2.8.0-204000_24.07_bclinux2110sp2.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_bclinux2110sp2.x86_64.rpm
		CTyunOS 2.0	aarch64	doca-host-2.8.0-204000_24.07_ctyunos20.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_ctyunos20.x86_64.rpm
		CTyunOS 23.01	aarch64	doca-host-2.8.0-204000_24.07_ctyunos2301.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_ctyunos2301.x86_64.rpm

Device	Component	OS	Arch	Link
		Debian 10.13	aarch64	doca-host_2.8.0-204000-24.07-debian1013_arm64.deb
			x86	doca-host_2.8.0-204000-24.07-debian1013_amd64.deb
		Debian 10.8	aarch64	doca-host_2.8.0-204000-24.07-debian108_arm64.deb
			x86	doca-host_2.8.0-204000-24.07-debian108_amd64.deb
		Debian 10.9	x86	doca-host_2.8.0-204000-24.07-debian109_amd64.deb
		Debian 11.3	aarch64	doca-host_2.8.0-204000-24.07-debian113_arm64.deb
			x86	doca-host_2.8.0-204000-24.07-debian113_amd64.deb
		Debian 12.1	aarch64	doca-host_2.8.0-204000-24.07-debian121_arm64.deb
			x86	doca-host_2.8.0-204000-24.07-debian121_amd64.deb
		Debian 12.5	aarch64	doca-host_2.8.0-204000-24.07-debian125_arm64.deb
			x86	doca-host_2.8.0-204000-24.07-debian125_amd64.deb
		EulerOS 20 SP11	aarch64	doca-host-2.8.0-204000_24.07_euleros20sp11.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_euleros20sp11.x86_64.rpm
		EulerOS 20 SP12	aarch64	doca-host-2.8.0-204000_24.07_euleros20sp12.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_euleros20sp12.x86_64.rpm

Device	Component	OS	Arch	Link
		Fedora32	x86	doca-host-2.8.0-204000_24.07_fc32.x86_64.rpm
		Kylin 1.0 SP2	aarch64	doca-host-2.8.0-204000_24.07_kylin10sp2.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_kylin10sp2.x86_64.rpm
		Kylin 1.0 SP3	aarch64	doca-host-2.8.0-204000_24.07_kylin10sp3.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_kylin10sp3.x86_64.rpm
		Mariner 2.0	x86	doca-host-2.8.0-204000_24.07_mariner20.x86_64.rpm
		Oracle Linux 7.9	x86	doca-host-2.8.0-204000_24.07_ol79.x86_64.rpm
		Oracle Linux 8.4	x86	doca-host-2.8.0-204000_24.07_ol84.x86_64.rpm
		Oracle Linux 8.6	x86	doca-host-2.8.0-204000_24.07_ol86.x86_64.rpm
		Oracle Linux 8.7	x86	doca-host-2.8.0-204000_24.07_ol87.x86_64.rpm
		Oracle Linux 8.8	x86	doca-host-2.8.0-204000_24.07_ol88.x86_64.rpm
		Oracle Linux 9.1	x86	doca-host-2.8.0-204000_24.07_ol91.x86_64.rpm
		Oracle Linux 9.2	x86	doca-host-2.8.0-204000_24.07_ol92.x86_64.rpm
		openEuler 20.03 SP3	aarch64	doca-host-2.8.0-204000_24.07_openeuler2003sp3.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_openeuler2003sp3.x86_64.rpm

Device	Component	OS	Arch	Link
		openEuler 22.03	aarch64	doca-host-2.8.0-204000_24.07_openeuler2203.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_openeuler2203.x86_64.rpm
		openEuler 22.03 SP1	x86	doca-host-2.8.0-204000_24.07_openeuler2203sp1.x86_64.rpm
		RHEL/CentOS 8.0	aarch64	doca-host-2.8.0-204000_24.07_rhel80.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_rhel80.x86_64.rpm
		RHEL/CentOS 8.1	aarch64	doca-host-2.8.0-204000_24.07_rhel81.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_rhel81.x86_64.rpm
		RHEL/CentOS 8.2	aarch64	doca-host-2.8.0-204000_24.07_rhel82.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_rhel82.x86_64.rpm
		RHEL/CentOS 8.3	aarch64	doca-host-2.8.0-204000_24.07_rhel83.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_rhel83.x86_64.rpm
		RHEL/CentOS 8.4	aarch64	doca-host-2.8.0-204000_24.07_rhel84.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_rhel84.x86_64.rpm
		RHEL/CentOS 8.5	aarch64	doca-host-2.8.0-204000_24.07_rhel85.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_rhel85.x86_64.rpm

Device	Component	OS	Arch	Link
		RHEL/Rocky 8.6	aarch64	doca-host-2.8.0-204000_24.07_rhel86.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_rhel86.x86_64.rpm
		RHEL/Rocky 8.7	aarch64	doca-host-2.8.0-204000_24.07_rhel87.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_rhel87.x86_64.rpm
		RHEL/Rocky 8.8	aarch64	doca-host-2.8.0-204000_24.07_rhel88.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_rhel88.x86_64.rpm
		RHEL/Rocky 8.9	aarch64	doca-host-2.8.0-204000_24.07_rhel89.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_rhel89.x86_64.rpm
		RHEL/Rocky 8.10	aarch64	doca-host-2.8.0-204000_24.07_rhel810.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_rhel810.x86_64.rpm
		RHEL/Rocky 9.0	aarch64	doca-host-2.8.0-204000_24.07_rhel90.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_rhel90.x86_64.rpm
		RHEL/Rocky 9.1	aarch64	doca-host-2.8.0-204000_24.07_rhel91.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_rhel91.x86_64.rpm
		RHEL/Rocky 9.2	aarch64	doca-host-2.8.0-204000_24.07_rhel92.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_rhel92.x86_64.rpm

Device	Component	OS	Arch	Link
		RHEL/Rocky 9.3	aarch64	doca-host-2.8.0-204000_24.07_rhel93.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_rhel93.x86_64.rpm
		RHEL/Rocky 9.4	aarch64	doca-host-2.8.0-204000_24.07_rhel94.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_rhel94.x86_64.rpm
		SLES 15 SP2	aarch64	doca-host-2.8.0-204000_24.07_sles15sp2.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_sles15sp2.x86_64.rpm
		SLES 15 SP3	aarch64	doca-host-2.8.0-204000_24.07_sles15sp3.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_sles15sp3.x86_64.rpm
		SLES 15 SP4	aarch64	doca-host-2.8.0-204000_24.07_sles15sp4.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_sles15sp4.x86_64.rpm
		SLES 15 SP5	aarch64	doca-host-2.8.0-204000_24.07_sles15sp5.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_sles15sp5.x86_64.rpm
		SLES 15 SP6	x86	doca-host-2.8.0-204000_24.07_sles15sp6.x86_64.rpm
		TencentOS 3.3	aarch64	doca-host-2.8.0-204000_24.07_tencentos33.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_tencentos33.x86_64.rpm

Device	Component	OS	Arch	Link
		Ubuntu 20.04	aarch64	doca-host_2.8.0-204000-24.07-ubuntu2004_arm64.deb
			x86	doca-host_2.8.0-204000-24.07-ubuntu2004_amd64.deb
		Ubuntu 22.04	aarch64	doca-host_2.8.0-204000-24.07-ubuntu2204_arm64.deb
			x86	doca-host_2.8.0-204000-24.07-ubuntu2204_amd64.deb
		Ubuntu 24.04	aarch64	doca-host_2.8.0-204000-24.07-ubuntu2404_arm64.deb
			x86	doca-host_2.8.0-204000-24.07-ubuntu2404_amd64.deb
		UOS20.1060	aarch64	doca-host-2.8.0-204000_24.07_uos201060.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_uos201060.x86_64.rpm
		UOS20.1060A	aarch64	doca-host-2.8.0-204000_24.07_uos201060a.aarch64.rpm
			x86	doca-host-2.8.0-204000_24.07_uos201060a.x86_64.rpm
		XenServer 8.2	x86	doca-host-2.8.0-204000_24.07_xenserver82.x86_64.rpm
Target BlueField Platform (Arm)	BlueField Software v4.8.0	Ubuntu 22.04	aarch64	bf-bundle-2.8.0-98_24.07_ubuntu-22.04_prod.bfb
	DOCA SDK v2.8.0	Ubuntu 22.04	aarch64	doca-dpu-repo-ubuntu2204-local_1-2.8.0081-1.24.07.0.6.1.bf.4.8.0.13249_arm64.deb
	DOCA Runtime v2.8.0			

13.2.2.2 Uninstalling Software from Host

If an older DOCA (or MLNX_OFED) software version is installed on your host, make sure to uninstall it before proceeding with the installation of the new version:

Deb-based

```
$ for f in $( dpkg --list | grep doca | awk '{print $2}' ); do echo $f ; apt
remove --purge $f -y ; done
$ /usr/sbin/ofed_uninstall.sh --force
$ sudo apt-get autoremove
```

RPM-based

```
host# for f in $(rpm -qa | grep -i doca ) ; do yum -y remove $f; done
host# /usr/sbin/ofed_uninstall.sh --force
host# yum autoremove
host# yum makecache
```

Then perform the following steps:

> The following procedure is valid for RPM-based OS only.

1. Download NVIDIA's RPM-GPG-KEY-Mellanox-SHA256 key:

```
# wget http://www.mellanox.com/downloads/ofed/RPM-GPG-KEY-Mellanox-SHA256
--2018-01-25 13:52:30--  http://www.mellanox.com/downloads/ofed/RPM-GPG-KEY-Mellanox-SHA256
Resolving www.mellanox.com... 72.3.194.0
Connecting to www.mellanox.com|72.3.194.0|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1354 (1.3K) [text/plain]
Saving to: ?RPM-GPG-KEY-Mellanox-SHA256?

100%[=================================================>] 1,354       --.-K/s   in 0s

2018-01-25 13:52:30 (247 MB/s) - ?RPM-GPG-KEY-Mellanox-SHA256? saved [1354/1354]
```

2. Install the key:

```
# sudo rpm --import RPM-GPG-KEY-Mellanox-SHA256
warning: rpmts_HdrFromFdno: Header V3 DSA/SHA1 Signature, key ID 6224c050: NOKEY
Retrieving key from file:///repos/MLNX_OFED//RPM-GPG-KEY-Mellanox
Importing GPG key 0x6224C050:
 Userid: "Mellanox Technologies (Mellanox Technologies - Signing Key v2) "
 From  : /repos/MLNX_OFED//RPM-GPG-KEY-Mellanox-SHA256
Is this ok [y/N]:
```

3. Verify that the key was successfully imported:

```
# rpm -q gpg-pubkey --qf '%{NAME}-%{VERSION}-%{RELEASE}\t%{SUMMARY}\n' | grep Mellanox
gpg-pubkey-a9e4b643-520791ba    gpg(Mellanox Technologies )
```

13.2.2.3 Installing Prerequisites on Host for Target BlueField

Install RShim to manage and flash the BlueField Platform.

OS	Procedure

Deb-based	1. Download the DOCA host repo package from the "Installation Files" section.
	2. Unpack the deb repo. Run:
	`host# sudo dpkg -i <repo_file>`
	3. Perform apt update. Run:
	`host# sudo apt-get update`
	4. Run `apt install` for RShim:
	`host# sudo apt install rshim`

RPM-based	1. Download the DOCA host repo package from the "Installation Files" section.
	2. Unpack the RPM repo. Run:
	`host# sudo rpm -Uvh <repo_file>`
	3. Enable new dnf repos. Run:
	`host# sudo dnf makecache`
	4. Run `dnf install` to install RShim:
	`host# sudo dnf install rshim`

> Skip section "Installing Software on Host" to proceed without the DOCA local repo package for host.

13.2.2.4 Installing Software on Host

> Skip this section if you intend to update only the BlueField software (`*.bfb`). The RShim driver is sufficient for that purpose.

> Make sure to have followed the instructions under "Installing Prerequisites on Host for Target BlueField".

1. Install DOCA local repo package for host:

   > The following table provides instructions for installing the DOCA host repo on your device depending on your OS and desired profile.

OS	Profile	Instructions

Deb-based	**doca-all**	a. Download the DOCA host repo from section "Installation Files" for the host.
		b. Unpack the deb repo. Run:
		`host# dpkg -i <repo_file>`
		c. Perform apt update. Run:
		`host# apt-get update`
		d. If the kernel version on your host is not supported (not shown under "Supported Operating System Distributions"), refer to section "DOCA Extra Package".
		e. Ensure that the kernel headers installed match the version of the currently running kernel.
		If the build directory exists in under `/lib/modules/$(uname -r)/build`, then the kernel headers are installed.
		f. Run apt install for DOCA SDK and DOCA runtime:
		`host# sudo apt install -y doca-all mlnx-fw-updater`

OS	Profile	Instructions

	doca-networking	a. Download the DOCA host repo from section "Installation Files" for the host.
		b. Unpack the deb repo. Run:
		`host# dpkg -i <repo_file>`
		c. Perform apt update. Run:
		`host# apt-get update`
		d. If the kernel version on your host is not supported (not shown under "Supported Operating System Distributions"), refer to section "DOCA Extra Package".
		e. Ensure that the kernel headers installed match the version of the currently running kernel.
		If the build directory exists in under `/lib/modules/$(uname -r)/build` , then the kernel headers are installed.
		f. Run apt install for DOCA SDK and DOCA runtime:
		`host# sudo apt install -y doca-networking mlnx-fw-updater`

OS	Profile	Instructions

	doca-ofed	a. Download the DOCA host repo from section "Installation Files" for the host.
		b. Unpack the deb repo. Run:
		`host# sudo dpkg -i <repo_file>`
		c. Perform apt update. Run:
		`host# sudo apt-get update`
		d. If the kernel version on your host is not supported (not shown under "Supported Operating System Distributions"), refer to section "DOCA Extra Package".
		e. Ensure that the kernel headers installed match the version of the currently running kernel.
		ⓘ If the build directory exists in under `/lib/modules/$(uname -r)/build`, then the kernel headers are installed.
		f. Install `doca-ofed`. Run:
		`host# sudo apt install -y doca-ofed mlnx-fw-updater`

RPM-based	**doca-all**	a. Download the DOCA host repo from section "Installation Files" for the host.
		b. Unpack the rpm repo. Run:
		`host# rpm -Uvh <repo_file>.rpm`
		c. Perform yum update. Run:
		`host# sudo yum makecache`
		d. If the kernel version on your host is not supported (not shown under "Supported Operating System Distributions"), refer to section "DOCA Extra Package".
		e. Run yum install for DOCA SDK and DOCA runtime:
		`host# sudo yum install -y doca-all mlnx-fw-updater`

OS	Profile	Instructions

	doca-networking	a. Download the DOCA host repo from section "Installation Files" for the host.
		b. Unpack the rpm repo. Run:
		`host# rpm -Uvh <repo_file>.rpm`
		c. Perform yum update. Run:
		`host# sudo yum makecache`
		d. If the kernel version on your host is not supported (not shown under "Supported Operating System Distributions"), refer to section "DOCA Extra Package".
		e. Run yum install for DOCA SDK and DOCA runtime:
		`host# sudo yum install -y doca-networking mlnx-fw-updater`

	doca-ofed	a. Download the DOCA host repo from section "Installation Files" for the host.
		b. Unpack the RPM repo. Run:
		`host# sudo rpm -Uvh <repo_file>.rpm`
		c. Perform yum update. Run:
		`host# sudo yum makecache`
		d. If the kernel version on your host is not supported (not shown under "Supported Operating System Distributions"), refer to section "DOCA Extra Package".
		e. Install `doca-ofed`. Run:
		`host# sudo yum install -y doca-ofed mlnx-fw-updater`

2. Load the drivers:

```
host# sudo /etc/init.d/openibd restart
```

3. Initialize MST. Run:

```
host# sudo mst restart
```

13.2.2.4.1 DOCA Extra Package

If the kernel version on your host is not supported (not shown under "Supported Operating System Distributions"), two options are available:

- Switch to a compatible kernel.

- Install `doca-extra` package:
    a. Run:

    ```
    host# sudo apt/yum install -y doca-extra
    ```

    b. Execute the `doca-kernel-support` script which rebuilds and installs the DOCA-Host kernel modules with the running kernel:

    ```
    host# sudo /opt/mellanox/doca/tools/doca-kernel-support
    ```

    c. Install user-space packages:

    ```
    host# sudo apt/yum install -y doca-ofed-userspace
    ```

    > `doca-kernel-support` does not support customized or unofficial kernels.

    d. (Optional) Retrieve installed packages and their versions as part of DOCA Host installation:

    ```
    host# doca-info

    Versions:
    - DOCA Base MLNX_OFED_LINUX-24.07-0.5.5.0
    - MFT 4.29.0-127

    UEFI\ATF versions:
    - mst_device: mt41692_pciconf0
         UEFI Version: 4.7.0-42-g13081ae
         ATF Version: 4.7.0-25-g5569834

    Firmware (Current):
    - BlueField-3 32.41.1000

    DOCA:
    - doca-all 2.8.0-0.0.4
    - doca-apsh-config 2.8.0079-1
    - doca-bench 2.8.0079-1
    …

    DOCA Dependencies:
    …
    - flexio 24.07.2300
    - mlnx-dpdk 22.11.0-2407.0.10


    OFED:
    …
    - rdma-core 2407mlnx52-1.2407055
    …
    - ucx 1.17.0-1.2407055
    …
    ```

    > If BlueField has a BlueField Bundle version older than 2.7.0 installed on it, UEFI\ATF versions appear as N\A. If your version is 2.7.0 or higher and still see N\A, then perform driver restart on the host:
    >
    > ```
    > /etc/init.d/openibd restart
    > ```

13.2.2.5 Installing Software on BlueField

Users have two options for installing DOCA on BlueField DPU or SuperNIC:

- Upgrading the full DOCA image on BlueField (recommended) – this option overwrites the entire boot partition with an Ubuntu 22.04 installation and updates BlueField and NIC firmware.
- Upgrading DOCA local repo package on BlueField – this option upgrades DOCA components without overwriting the boot partition. Use this option to preserve configurations or files on BlueField itself.
- Upgrading DOCA online repo package on BlueField – this option upgrades DOCA components without overwriting the boot partition. Use this option to preserve configurations or files on BlueField itself.

13.2.2.5.1 Installing Full DOCA Image on DPU via Host

This step overwrites the entire boot partition.

This installation sets up the OVS bridge.

If you are installing DOCA on multiple BlueField platforms, skip to section Installing Full DOCA Image on Multiple BlueField Platforms.

13.2.2.5.1.1 Option 1 – No Pre-defined Password

To change the default Ubuntu password during the BFB bundle installation, proceed to Option 2.

BFB installation is executed as follows:

```
host# sudo bfb-install --rshim rshim<N> --bfb <image_path.bfb>
```

Where `rshim<N>` is `rshim0` if you only have one Bluefield. You may run the following command to verify:

```
host# ls -la /dev/ | grep rshim
```

13.2.2.5.1.2 Option 2 – Set Pre-defined Password

Ubuntu users can provide a unique password that will be applied at the end of the BlueField BFB bundle installation. This password needs to be defined in a `bf.cfg` configuration file.

To set the password for the "ubuntu" user:

1. Create password hash. Run:

```
host# openssl passwd -1
Password:
Verifying - Password:
$1$3B0RIrfX$TlHry93NFUJzg3Nya00rE1
```

2. Add the password hash in quotes to the `bf.cfg` file:

   ```
   host# echo ubuntu_PASSWORD='$1$3B0RIrfX$TlHry93NFUJzg3Nya00rE1' > bf.cfg
   ```

   When running the installation command, use the `--config` flag to provide the file containing the password:

   ```
   host# sudo bfb-install --rshim rshim<N> --bfb <image_path.bfb> --config bf.cfg
   ```

   > ⚠ Optionally, to upgrade the BlueField integrated BMC firmware using BFB bundle, please provide the current BMC root credentials in a `bf.cfg` file, as shown in the following:
   >
   > ```
   > BMC_PASSWORD="<root password>"
   > BMC_USER="root"
   > BMC_REBOOT="yes"
   > ```
   >
   > Unless previously changed, the default BMC root password is `0penBmc`.

   > ⚠ If `--config` is not used, then upon first login to the BlueField device, users will be prompted to update the default 'ubuntu' password.

   The following is an example of Ubuntu-22.04 BFB bundle installation (Release version may vary in the future).

   ```
   host# sudo bfb-install --rshim rshim0 --bfb bf-bundle-2.7.0_24.04_ubuntu-22.04_prod.bfb --config bf.cfg
   Pushing bfb 1.41GiB 0:02:02 [11.7MiB/s]
   [           <=>
                                        ]
   Collecting BlueField booting status. Press Ctrl+C to stop
    INFO[PSC]: PSC BL1 START
    INFO[BL2]: start
    INFO[BL2]: boot mode (rshim)
    INFO[BL2]: VDDQ: 1120 mV
    INFO[BL2]: DDR POST passed
    INFO[BL2]: UEFI loaded
    INFO[BL31]: start
    INFO[BL31]: lifecycle GA Secured
    INFO[BL31]: VDD: 850 mV
    INFO[BL31]: runtime
    INFO[BL31]: MB ping success
    INFO[UEFI]: eMMC init
    INFO[UEFI]: eMMC probed
    INFO[UEFI]: UPVS valid
    INFO[UEFI]: PMI: updates started
    INFO[UEFI]: PMI: total updates: 1
    INFO[UEFI]: PMI: updates completed, status 0
    INFO[UEFI]: PCIe enum start
    INFO[UEFI]: PCIe enum end
    INFO[UEFI]: UEFI Secure Boot
    INFO[UEFI]: PK configured
    INFO[UEFI]: Redfish enabled
    INFO[UEFI]: exit Boot Service
    INFO[MISC]: Found bf.cfg
    INFO[MISC]: Ubuntu installation started
    INFO[MISC]: Installing OS image
    INFO[MISC]: Changing the default password for user ubuntu
    INFO[MISC]: Ubuntu installation completed
    INFO[MISC]: Updating NIC firmware...
    INFO[MISC]: NIC firmware update done
    INFO[MISC]: Installation finished
   ```

   To verify the BlueField has completed booting up, allow additional 90 seconds then perform the following:

   ```
   host# sudo cat /dev/rshim<N>/misc
   ...
   ```

```
INFO[MISC]: Linux up
INFO[MISC]: DPU is ready
```

13.2.2.5.2 Installing Full DOCA Image on Multiple BlueField Platforms

On a host with multiple BlueField devices, the BFB image can be installed on all of them using the `multi-bfb-install` script.

```
host# ./multi-bfb-install --bfb <image_path.bfb> --password <password>
```

This script detects the number of RShim devices and configures them statically.

- For Ubuntu - the script creates a configuration file `/etc/netplan/20-tmfifo.yaml`
- For CentOS/RH 8.0 and 8.2 - the script installs the `bridge-utils` package to use the `brctl` command, creates the `tm-br` bridge, and connects all RShim interfaces to it

After the installation is complete, the configuration of the bridge and each RShim interface can be observed using `ifconfig`. The expected result is to see the IP on the `tm-br` bridge configured to `192.168.100.1` with subnet `255.255.255.0`.

> To log into BlueField with `rshim0`, run:
>
> ```
> ssh ubuntu@192.168.100.2
> ```
>
> For each RShim after that, add 1 to the fourth octet of the IP address (e.g., `ubuntu@192.168.100.3` for `rshim1`, `ubuntu@192.168.100.4` for `rshim2`, etc).

The script burns a new MAC address to each BlueField and configures a new IP, 192.168.100.x, as described earlier.

13.2.2.5.3 Installing DOCA Local Repo Package on BlueField

> If you have already installed BlueField image, be aware that the DOCA SDK, Runtime, and Tools are already contained in the BFB, and this installation is not mandatory. If you have not installed the BlueField image and wish to update DOCA Local Repo package, proceed with the following procedure.

> Before installing DOCA on the target BlueField, make sure the out-of-band interface (`mgmt`) is connected to the internet.

1. Download the DOCA SDK and DOCA Runtime package from section Installation Files.
2. Copy deb repo package into BlueField. Run:

   ```
   host# sudo scp -r doca-repo-aarch64-ubuntu2204-local_<version>_arm64.deb ubuntu@192.168.100.2:/tmp/
   ```

3. Unpack the deb repo. Run:

```
dpu# sudo dpkg -i doca-dpu-repo-ubuntu2204-local_<version>_arm64.deb
```

4. Run apt update.

   ```
   dpu# sudo apt-get update
   ```

5. Run `apt install` for DOCA Runtime and DOCA SDK:

   ```
   dpu# sudo apt install doca-runtime doca-sdk
   ```

13.2.2.6 Upgrading Firmware

> This operation is only required if the user skipped NIC firmware update during BFB bundle installation using the parameter `WITH_NIC_FW_UPDATE=no` in the `bf.cfg` file.

This section explains how to update the NIC firmware on a DOCA installed BlueField OS.

> If multiple BlueFields are installed, the following steps must be performed on all of them after BFB installation.

An up-to-date NIC firmware image is provided in BlueField BFB bundle and copied to the BlueField filesystem during BFB installation.

To upgrade firmware in the BlueField Arm OS:

1. SSH to your BlueField Arm OS by any means available.
   The following instructions enable to login to the BlueField Arm OS from the host OS over the RShim virtual interface, `tmfifo_net<N>` and do not require LAN connectivity with the BlueField OOB network port.

   > This operation can be performed over the host's `tmfifo_net0` IPv4, 192.168.100.1 (preconfigured) with BlueField Arm OS at 192.168.100.2 (default).
   >
   > If multiple BlueField DPUs were updated using the `multi-bfb-install` script, as explained above, then each target BlueField OS IPv4 address changes in its last octate according to the underlaying RShim interface number: 192.168.100.3 for rshim1, 192.168.100.4 for rshim2, etc.

   The default credentials for Ubuntu are as follows:

Username	Password
ubuntu	`ubuntu`

   For example, to log into BlueField Arm OS over IPv6:

   ```
   host]# systemctl restart rshim
   // Wait 10 seconds

   host]# ssh -6 fe80::21a:caff:feff:ff01%tmfifo_net<N>
   ```

```
Password: <configured-password>
```

2. Upgrade firmware in BlueField. Run:

   ```
   dpu# sudo /opt/mellanox/mlnx-fw-updater/mlnx_fw_updater.pl --force-fw-update
   ```

   Example output:

   ```
   Device #1:
   ----------

     Device Type:      BlueField-2
     [...]
     Versions:         Current        Available
        FW             <Old_FW>       <New_FW>
   ```
3. For the firmware upgrade to take effect perform a BlueField system reboot.

13.2.2.7 Post-installation Procedure

1. Restart the driver. Run:

   ```
   host# sudo /etc/init.d/openibd restart
   Unloading HCA driver:                                      [  OK  ]
   Loading HCA driver and Access Layer:                       [  OK  ]
   ```
2. Configure the physical function (PF) interfaces.

   ```
   host# sudo ifconfig <interface-1> <network-1/mask> up
   host# sudo ifconfig <interface-2> <network-2/mask> up
   ```

   For example:

   ```
   host# sudo ifconfig p2p1 192.168.200.32/24 up
   host# sudo ifconfig p2p2 192.168.201.32/24 up
   ```

   Pings between the source and destination should now be operational.

13.2.3 Upgrading BlueField Using Standard Linux Tools

Unable to render include or excerpt-include. Could not retrieve page.

13.2.4 Post-Installation Procedure

1. Restart the driver. Run:

   ```
   host# sudo /etc/init.d/openibd restart
   Unloading HCA driver:                                      [  OK  ]
   Loading HCA driver and Access Layer:                       [  OK  ]
   ```
2. Configure the physical function (PF) interfaces.

   ```
   host# sudo ifconfig <interface-1> <network-1/mask> up
   host# sudo ifconfig <interface-2> <network-2/mask> up
   ```

   For example:

```
host# sudo ifconfig p2p1 192.168.200.32/24 up
host# sudo ifconfig p2p2 192.168.201.32/24 up
```

Pings between the source and destination should now be operational.

13.2.5 Building Your Own BFB Installation Image

Users wishing to build their own customized BlueField OS image can use the BFB build environment. Please refer to the bfb-build project in this GitHub webpage for more information.

> For a customized BlueField OS image to boot on the UEFI secure-boot-enabled BlueField (default BlueField secure boot setting), the OS must be either signed with an existing key in the UEFI DB (e.g., the Microsoft key), or UEFI secure boot must be disabled. Please refer to the "Secure Boot" page under NVIDIA BlueField DPU Platform Operating System Documentation for more details.

13.2.6 Setting Up Build Environment for Developers

For full instructions about setting up a development environment, refer to the NVIDIA DOCA Developer Guide.

13.2.7 Additional SDKs for DOCA

13.2.7.1 Installing CUDA on NVIDIA Converged Accelerator

NVIDIA® CUDA® is a parallel computing platform and programming model developed by NVIDIA for general computing GPUs.

This section details the necessary steps to set up CUDA on your environment. This section assumes that a BFB image has already been installed on your environment.
To install CUDA on your converged accelerator:

1. Download and install the latest NVIDIA Data Center GPU driver.
2. Download and install CUDA

   > The CUDA version tested to work with DOCA SDK is 11.8.0.

> Downloading CUDA includes the latest NVIDIA Data Center GPU driver and CUDA toolkit. For more information about CUDA and driver compatibility, refer to the NVIDIA CUDA Toolkit Release Notes.

13.2.7.1.1 Configuring Operation Mode

There are two modes that the NVIDIA Converged Accelerator may operate in:

- Standard mode (default) – the BlueField and the GPU operate separately
- BlueField-X mode – the GPU is exposed to BlueField and is no longer visible on the host

To verify which mode the system is operating in, run:

```
host# sudo mst start
host# sudo mlxconfig -d <device-id> q PCI_DOWNSTREAM_PORT_OWNER[4]
```

> To learn your BlueField Platform's device ID, refer to section "Determining BlueField Device ID".

- Standard mode output:

  ```
  Device #1:
  […]
  Configurations:                             Next Boot
           PCI_DOWNSTREAM_PORT_OWNER[4]        DEVICE_DEFAULT(0)
  ```

- BlueField-X mode output:

  ```
  Device #1:
  […]
  Configurations:                             Next Boot
           PCI_DOWNSTREAM_PORT_OWNER[4]        EMBEDDED_CPU(15)
  ```

To configure BlueField-X mode, run:

```
host# mlxconfig -d <device-id> s PCI_DOWNSTREAM_PORT_OWNER[4]=0xF
```

To configure standard mode, run:

```
host# mlxconfig -d <device-id> s PCI_DOWNSTREAM_PORT_OWNER[4]=0x0
```

> To learn your BlueField Platform's device ID, refer to section "Determining BlueField Device ID".

Power cycle is required for configuration to take effect. For power cycle the host run:

```
host# ipmitool power cycle
```

13.2.7.1.2 Downloading and Installing CUDA Toolkit and Driver

This section details the necessary steps to set up CUDA on your environment. It assumes that a BFB image has already been installed on your environment.

1. Install CUDA by visiting the CUDA Toolkit Downloads webpage.

   > Select the Linux distribution and version relevant for your environment.

   > This section shows the native compilation option either on x86 or aarch64 hosts.

2. Test that the driver installation completed successfully. Run:

```
dpu# nvidia-smi

Tue Apr  5 13:37:59 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 510.47.03    Driver Version: 510.47.03    CUDA Version: 11.8     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA BF A10       Off  | 00000000:06:00.0 Off |                    0 |
|  0%   43C    P0    N/A / 225W |      0MiB / 23028MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

3. Verify that the installation completed successfully.
   a. Download CUDA samples repo. Run:

   ```
   dpu# git clone https://github.com/NVIDIA/cuda-samples.git
   ```

   b. Build and run vectorAdd CUDA sample. Run:

   ```
   dpu# cd cuda-samples/Samples/0_Introduction/vectorAdd
   dpu# make
   dpu# ./vectorAdd
   ```

> If the `vectorAdd` sample works as expected, it should output "`Test Passed`".

> If it seems that the GPU is slow or stuck, stop execution and run:
>
> ```
> dpu# sudo setpci -v -d ::0302 800.L=201 # CPL_VC0 = 32
> ```

13.2.7.1.3 GPUDirect RDMA

For information on GPUDirect RMDA and more, refer to DOCA GPUNetIO documentation.

13.2.7.2 Installing Rivermax on BlueField

NVIDIA Rivermax offers a unique IP-based solution for any media and data streaming use case.

This section provides the steps to install Rivermax assuming that a BFB image has already been installed on your environment.

13.2.7.2.1 Downloading Rivermax Driver

1. Navigate to the NVIDIA Rivermax SDK product page.
2. Register to be able to download the driver package using the JOIN button at the top of the page.
3. Download the appropriate driver package according to your BFB under the "Linux" subsection. For example, for Ubuntu 22.04 BFB, download `rivermax_ubuntu2204_<version>.tar.gz`.

13.2.7.2.2 Installing Rivermax Driver

1. Copy the `.tgz` file to BlueField:

   ```
   host# sudo scp -r rivermax_ubuntu2204_<version>.tar.gz ubuntu@192.168.100.2:/tmp/
   ```

2. Extract the Rivermax file:

   ```
   dpu# sudo tar xzf rivermax_ubuntu2204_<version>.tar.gz
   ```

3. Install the Rivermax driver package:

   ```
   dpu# cd <rivermax-version>/Ubuntu.22.04/deb-dist/aarch64/
   dpu# sudo dpkg -i rivermax_<version>.deb
   ```

13.2.7.2.3 Installing Rivermax Libraries from DOCA

Rivermax libraries are compatibles with DOCA components and can be found inside the `doca-dpu-repo`.

1. Unpack the doca-dpu-repo:

   ```
   dpu# sudo dpkg -i doca-dpu-repo-ubuntu2204-local_<version>_arm64.deb
   ```

2. Run apt update:

   ```
   dpu# sudo apt-get update
   ```

3. Install the Rivermax libraries:

   ```
   dpu# sudo apt install doca-rmax-libs
   dpu# sudo apt install libdoca-rmax-libs-dev
   ```

For additional details and guidelines, please visit the NVIDIA Rivermax SDK product page.

> For questions, comments, and feedback, please contact us at DOCA-Feedback@exchange.nvidia.com.

13.3 NVIDIA DOCA Developer Guide

This guide details the recommended steps to set up an NVIDIA DOCA development environment.

13.3.1 Introduction

This guide is intended for software developers aiming to modify existing NVIDIA DOCA applications or develop their own DOCA-based software.

Instructions for installing DOCA on the NVIDIA® BlueField® Networking Platform (i.e., DPU or SuperNIC) can be found in the NVIDIA DOCA Installation Guide for Linux.

This guide focuses on the recommended flow for developing DOCA-based software, and will address the following scenarios:

- BlueField is accessible and can be used during the development and testing process
  - Working within a development container
- BlueField is inaccessible, and the development happens on the host or on a different server
  - Cross-compilation from the host
  - Working within a development container on top of QEMU running on the host

It is recommended to follow the instructions for the first scenario, leveraging BlueField during the development and testing process.

This guide recommends using DOCA's development container during the development process on BlueField Platforms or on the host. Deploying development containers allows multiple developers to work simultaneously on the same device (host or BlueField Platform) in an isolated manner and even across multiple different DOCA SDK versions. This can allow multiple developers to work on the BlueField Platform itself, for example, without needing to have a dedicated BlueField per developer.

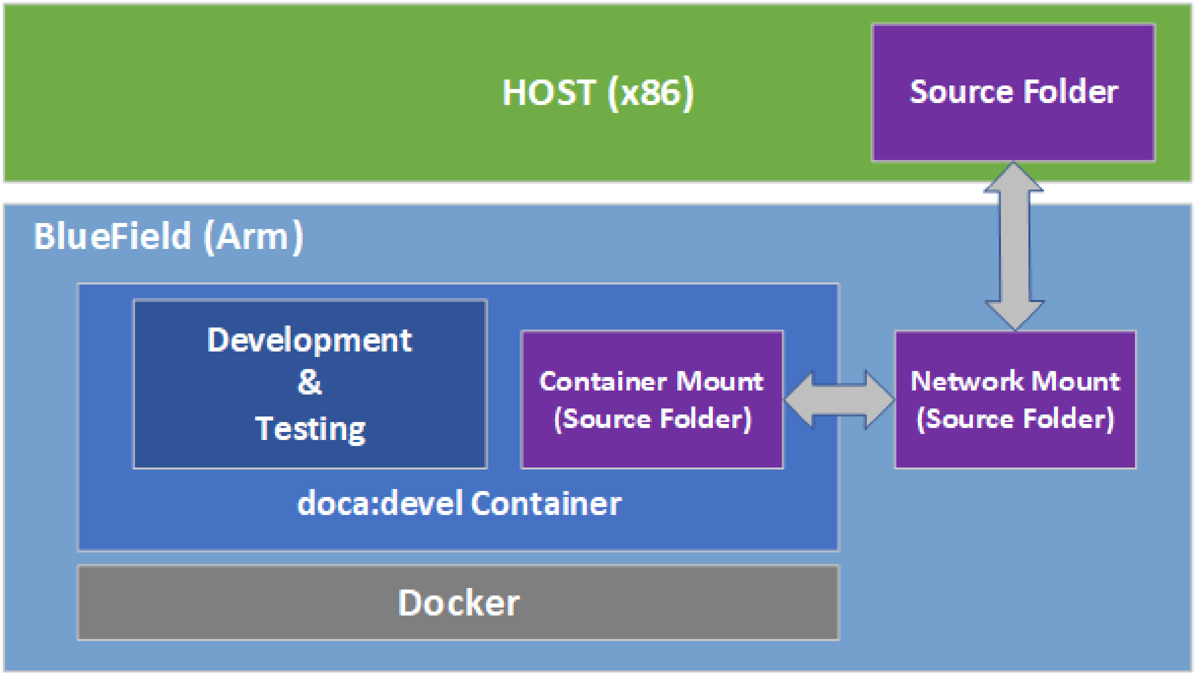
Another benefit of this container-based approach is that the development container allows developers to create and test their DOCA-based software in a user-friendly environment that comes pre-shipped with a set of handy development tools. The development container is focused on improving the development experience and is designed for that purpose, whereas the BlueField software is meant to be an efficient runtime environment for DOCA products.

ⓘ For questions, comments, and feedback, please contact us at DOCA-Feedback@exchange.nvidia.com.

13.3.2 Developing Using BlueField Networking Platform

13.3.2.1 Setup

DOCA's base image containers include a DOCA development container for the BlueField (`doca:devel`) which can be found on NGC. It is recommended to deploy this container on top of BlueField when preparing a development setup.



The recommended approach for working using DOCA's development container on top of the BlueField, is by using docker, which is already included in the supplied BFB image.

1. Make sure the docker service is started. Run:

   ```
   sudo systemctl daemon-reload
   sudo systemctl start docker
   ```

2. Pull the container image:
   a. Visit the NGC page of the DOCA base image.
   b. Under the "Tags" menu, select the desired development tag for BlueField.
   c. The container tag for the docker `pull` command is copied to your clipboard once selected. Example docker `pull` command using the selected tag:

      ```
      sudo docker pull nvcr.io/nvidia/doca/doca:1.5.1-devel
      ```

3. Once loaded locally, you may find the image's ID using the following command:

   ```
   sudo docker images
   ```

   Example output:

   ```
   REPOSITORY                  TAG            IMAGE ID       CREATED         SIZE
   nvcr.io/nvidia/doca/doca    1.5.1-devel    931bd576eb49   10 months ago   1.49GB
   ```

4. Run the docker image:

   ```
   sudo docker run -v <source-code-folder>:/doca_devel -v /dev/hugepages:/dev/hugepages --privileged --
   net=host -it <image-name/ID>
   ```

   For example, to map a source folder named `my_sources` into the same container tag from the example above, the command should look like this:

```
sudo docker run -v my_sources:/doca_devel -v /dev/hugepages:/dev/hugepages --privileged --net=host -it
nvcr.io/nvidia/doca/doca:1.5.1-devel
```

After running the command, you get a shell inside the container where you can build your project using the regular build commands:

- From the container's perspective, the mounted folder will be named `/doca_devel`

  > Make sure to map a folder with write privileges to `everyone`. Otherwise, the docker would not be able to write the output files to it.

- `--net=host` ensures the container has network access, including visibility to SFs and VFs as allocated on BlueField
- `-v /dev/hugepages:/dev/hugepages` ensures that allocated huge pages are accessible to the container

13.3.2.2 Development

It is recommended to do the development within the `doca:devel` container. That said, some developers prefer different integrated development environments (IDEs) or development tools, and sometimes prefer working using a graphical IDE until it is time to compile the code. As such, the recommendation is to mount a network share to BlueField (refer to NVIDIA DOCA DPU CLI for more information) and to the container.

> Having the same code folder accessible from the IDE and the container helps prevent edge cases where the compilation fails due to a typo in the code, but the typo is only fixed locally within the container and not propagated to the main source folder.

13.3.2.3 Testing

The container is marked as "privileged", hence it can directly access the hardware capabilities of the BlueField Platform. This means that once the tested program compiles successfully, it can be directly tested from within the container without the need to copy it to BlueField and running it there.

13.3.2.4 Publishing

Once the program passes the testing phase, it should be prepared for deployment. While some proof-of-concept (POC) programs are just copied "as-is" in their binary form, most deployments will probably be in the form of a package (`.deb` / `.rpm`) or a container.

Construction of the binary package can be done as-is inside the current `doca:devel` container, or as part of a CI pipeline that will leverage the same development container as part of it.

For the construction of a container to ship the developed software, it is recommended to use a multi-staged build that ships the software on top of the runtime-oriented DOCA base images:

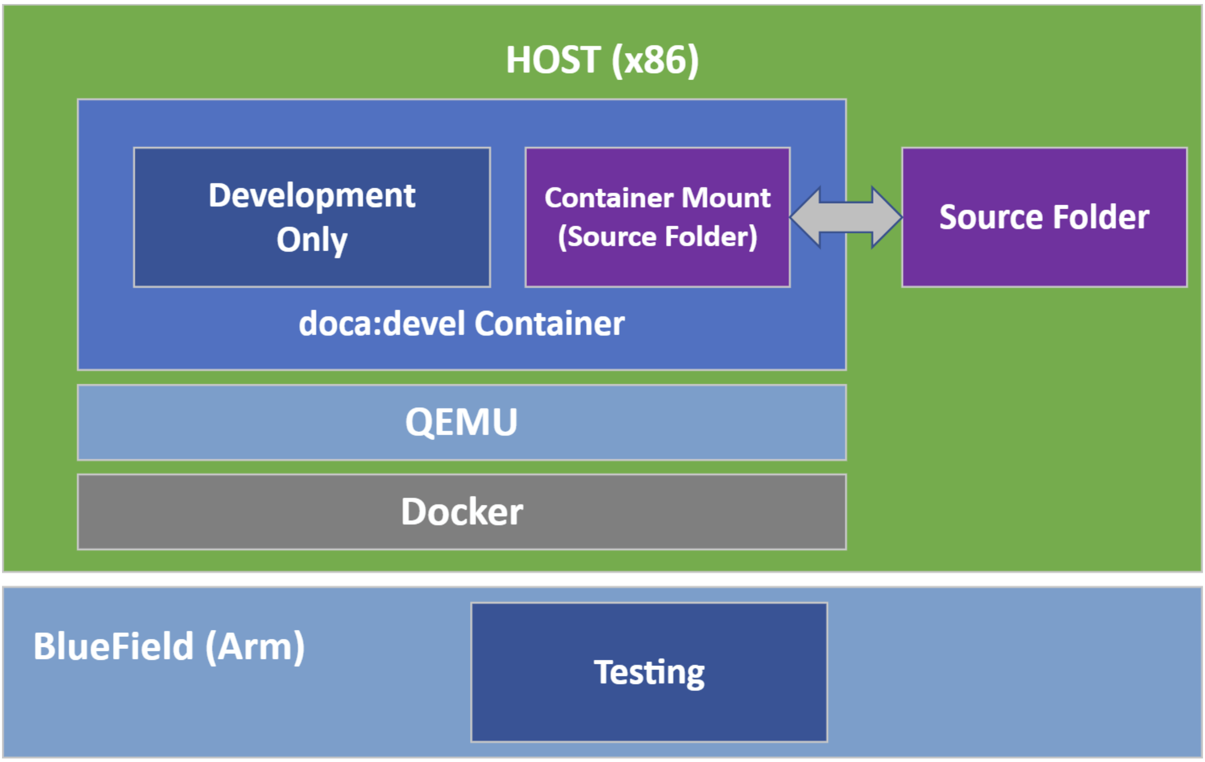
- `doca:base-rt` - slim DOCA runtime environment
- `doca:full-rt` - full DOCA runtime environment similar to the BlueField image

The runtime DOCA base images, alongside more details about their structure, can be found under the same NGC page that hosts the `doca:devel` image.

For a multi-staged build, it is recommended to compile the software inside the `doca:devel` container, and later copy it to one of the runtime container images. All relevant images must be pulled directly from NGC (using `docker pull`) to the container registry of BlueField.

13.3.3 Developing Without BlueField Networking Platform

If the development process needs to be done without access to a BlueField Platform, the recommendation is to use a QEMU-based deployment of a container on top of a regular x86 server. The development container for the host will be the same `doca:devel` image we mentioned previously.



13.3.3.1 Setup

1. Make sure Docker is installed on your host. Run:

```
docker version
```

If it is not installed, visit the official Install Docker Engine webpage for installation instructions.

2. Install QEMU on the host.

> ⚠ This step is for x86 hosts only. If you are working on an aarch64 host, move to the next step.

Host OS	Command

Ubuntu	`sudo apt-get install qemu binfmt-support qemu-user-static`
	`sudo docker run --rm --privileged multiarch/qemu-user-static --reset -p yes`

CentOS/RHEL 7.x	`sudo yum install epel-release`
	`sudo yum install qemu-system-arm`

CentOS 8.0/8.2	`sudo yum install epel-release`
	`sudo yum install qemu-kvm`

Fedora	`sudo yum install qemu-system-aarch64`

3. If you are using CentOS or Fedora on the host, verify if `qemu-aarch64.conf` Run:

   ```
   cat /etc/binfmt.d/qemu-aarch64.conf
   ```

   If it is missing, run:

   ```
   echo ":qemu-aarch64:M::
   \x7fELF\x02\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\xb7:\xff\xff\xff\xff\xff\xff\xff\xfc\xff\xf
   f\xff\xff\xff\xff\xff\xff\xfe\xff\xff:/usr/bin/qemu-aarch64-static:" > /etc/binfmt.d/qemu-aarch64.conf
   ```

4. If you are using CentOS or Fedora on the host, restart system binfmt. Run:

   ```
   $ sudo systemctl restart systemd-binfmt
   ```

5. To load and execute the development container, refer to the "Setup" section discussing the same docker-based deployment on the BlueField side.

The `doca:devel` container supports multiple architectures. Therefore, Docker by default attempts to pull the one matching that of the current machine (i.e., `amd64` for the host and `arm64` for BlueField). Pulling the `arm64` container from the x86 host can be done by adding the flag `--platform=linux/arm64` :

```
sudo docker pull --platform=linux/arm64 nvcr.io/nvidia/doca/doca:1.5.1-devel
```

13.3.3.2 Development

Much like the development phase using BlueField, it is recommended to develop within the container running on top of QEMU.

13.3.3.3 Testing

While the compilation can be performed on top of the container, testing the compiled software must be done on top of a BlueField Platform. This is because the QEMU environment emulates an aarch64 architecture, but it does not emulate the hardware devices present on the BlueField Platform. Therefore, the tested program will not be able to access the devices needed for its successful execution, thus mandating that the testing is done on top of a physical BlueField.

> ⚠ Make sure that the DOCA version used for compilation is the same as the version installed on BlueField used for testing.

13.3.3.4 Publishing

The publishing process is identical to the publishing process when using BlueField.

14 DOCA Programming Guide

The DOCA Programming Guide is intended for developers wishing to utilize DOCA SDK to develop application on top of the NVIDIA® BlueField® DPUs and SuperNICs.

- DOCA Programming Overview is important to read for new DOCA developers to understand the architecture and main building blocks most applications will rely on.
- DOCA Development Best Practices outlines common development pitfalls and capabilities to speed up application development, qualification, and productization.
- DOCA Libraries describes in details how to use each DOCA library, its APIs, and different aspects related to that library. Users may choose to only read the pages concerning DOCA libraries required for their application.
- DOCA Utils includes modules that may be used by application developers to speed up their development process (e.g., DOCA Arg Parser which simplifies the creation of a command-line interface for your application).
- DOCA Drivers describes additional frameworks used within DOCA.

For questions, comments, and feedback, please contact us at DOCA-Feedback@exchange.nvidia.com.

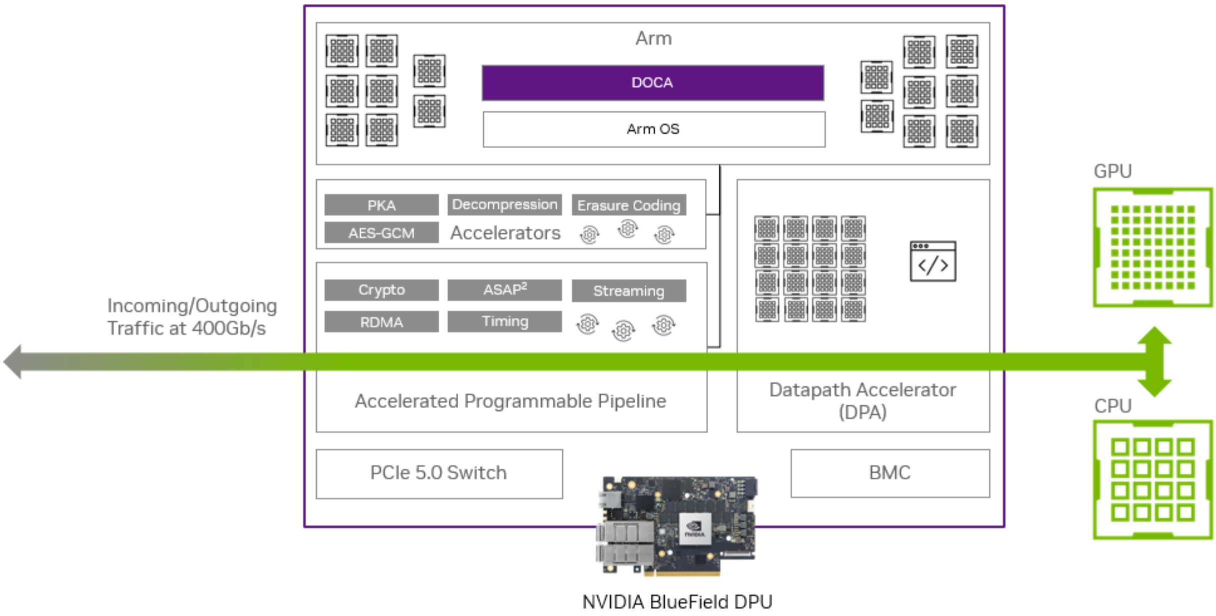
14.1 DOCA Programming Overview

This section contains the following pages:

- Hardware Overview
- DOCA SDK Architecture

14.1.1 Hardware Overview

DOCA is the software framework for BlueField's main hardware entities:



- Arm cores – optimized for control-path applications, general-purpose applications and single-flow performance
    - 16 A78 Arm cores general-purpose processor
    - Coherent Mesh architecture
    - Last level cache (LLC)
    - DDR5 memory subsystem
    - Base OS and microservices
- Accelerated programmable pipeline – optimized for high-performance packet processing applications and advanced packet handling
    - Programmable 64-128 packet processor
    - Multi-staged, highly parallelized
    - Flow-based classification and action engine
    - RDMA, crypto, time-based scheduling
- Data-path accelerator – optimized for IO-intensive applications, high insertion rate, network flow processing, device emulation, and collective and DMA operations
    - 16 hyper-threaded cores I/O and packet processor
    - Real-time OS

14.1.2 DOCA SDK Architecture

DOCA provides libraries for networking and data processing programmability that leverage NVIDIA® BlueField® networking platform (DPU or SuperNIC) and NVIDIA® ConnectX® NIC hardware accelerators.

DOCA software framework is built on top of DOCA Core, which provides a unified software framework for DOCA libraries, to form a processing pipeline or workflow build of one or many DOCA libraries.

14.1.2.1 Device Subsystem

The DOCA SDK allows applications to offload resource intensive tasks (e.g., encryption, and compression) to hardware. DOCA also allows applications to offload network related tasks (e.g., packet acquisition, RDMA send). As such, BlueField and ConnectX provide dedicated hardware processing units for executing such tasks.

The DOCA device subsystem provides an abstraction of the hardware processing units referred to as device.

DOCA Device subsystem provides means to:

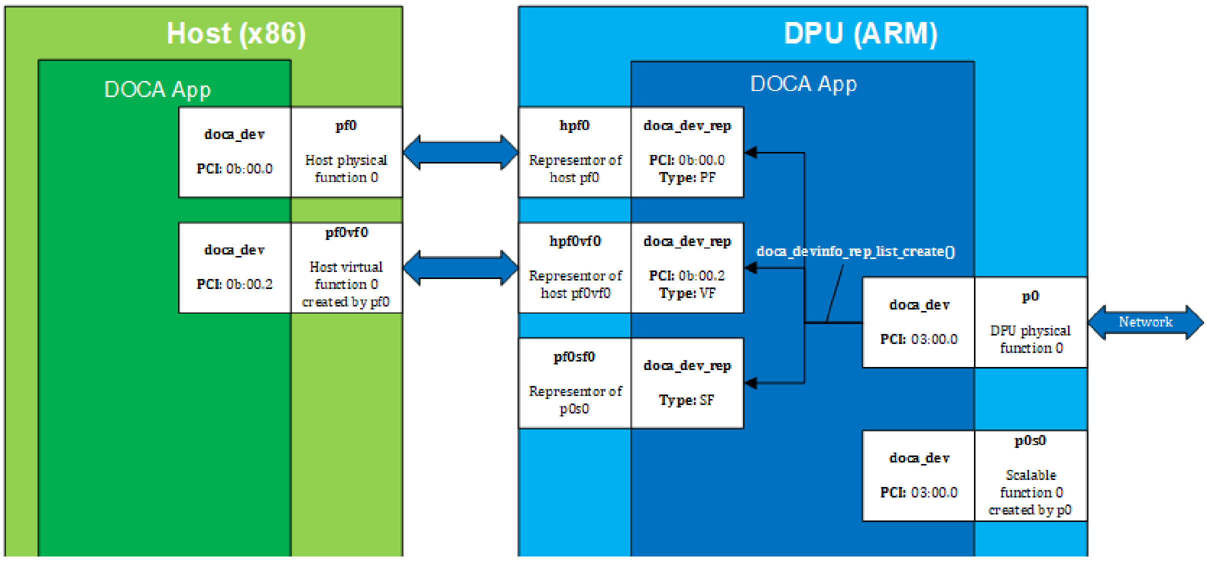
- Discover available hardware acceleration units provided by DPUs/SuperNICs/NICs
- Query capabilities and properties of available hardware acceleration units
- Open device to enable libraries to allocate and share resources necessary for hardware acceleration

On a given system, there can be multiple available devices. An application can choose a device based on the following characteristics topology (e.g., PCIe address) and/or capabilities (e.g., encryption support).

DOCA Core supports two DOCA Device types:

- Local device - this is an actual device exposed in the local system (BlueField or host) and can perform DOCA library processing jobs. This can be a PCIe physical function (PF), virtual function (VF), or scalable function (SF)
- Representor device - this is a representation of a local device. The represented local device is typically on the host (except for SFs) and the representor is always on the BlueField side (a proxy on the BlueField for the host-side device).

The following figure provides an example of host local devices with representors on BlueField:



DPU Mode

The diagram shows typical topology when using BlueField in DPU mode as described in NVIDIA BlueField DPU Modes of Operation.

The diagram shows BlueField (on the right side of the figure) connected to a host (on the left). The host has physical function PF0 with a child virtual function VF0.

The BlueField side has a representor-device per host function in a 1-to-1 ratio (e.g., `hpf0` is the representor device for the host's PF0 device, etc.) as well as a representor for each SF function, such that both the SF and its representor reside in BlueField.

For more details on the DOCA Device subsystem, see section "DOCA Device".

14.1.2.2 Memory Management Subsystem

Hardware processing tasks require data buffers as inputs and/or outputs to processing operations. The application is responsible to provide the input data and/or read the output data. To achieve maximum performance, the SDK uses zero-copy technology to pass data to hardware. To allow zero-copy, the application must register the memory that would hold data buffers beforehand. The memory management subsystem provides a means to register memory and manage allocation of data buffers on registered memory.

Memory registration:

- Defines user application memory range to use to hold data buffers
- Allows one or more devices to access the memory range
- Defines the access permission (e.g., read only)

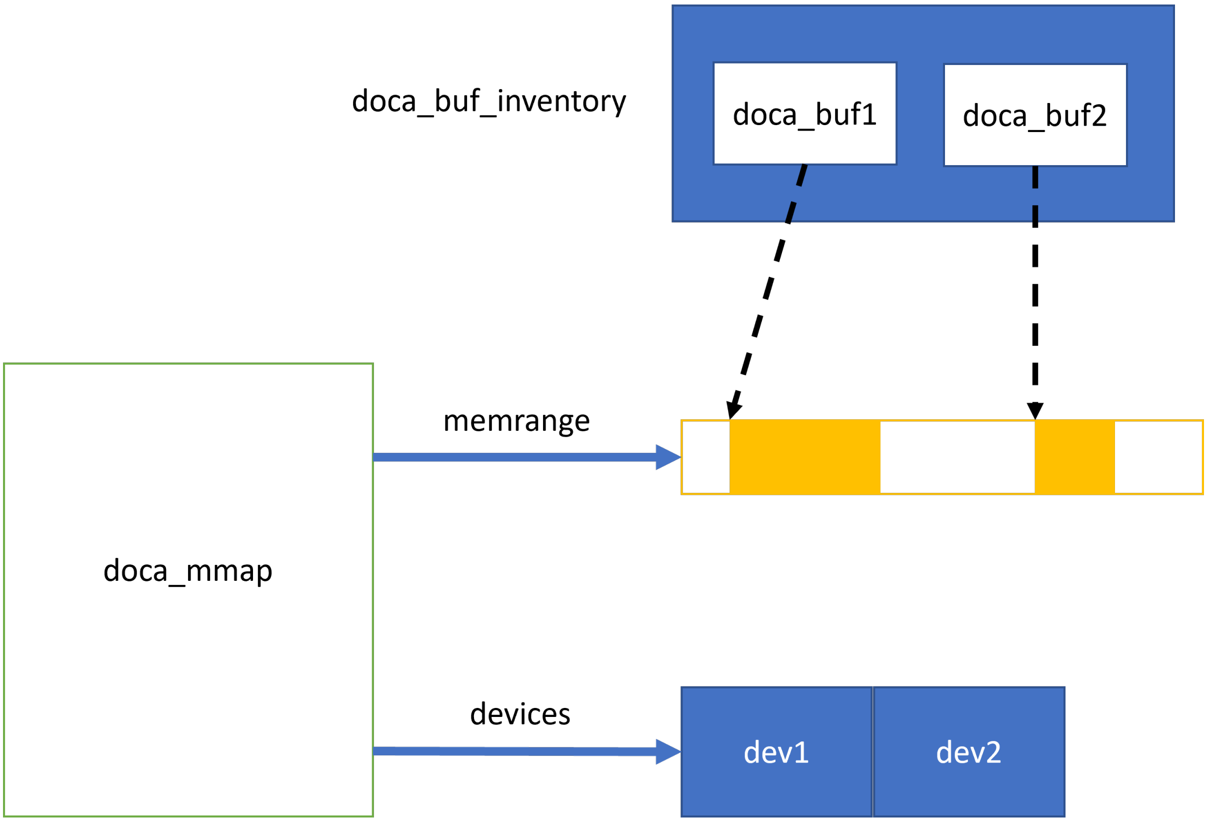
Data buffer allocation management:

- Allows allocating data buffers that cover subranges within the registered memory
- Allows memory pool semantics over registered memory

DOCA memory has the following main components:

- `doca_buf` - describes a data buffer, and is used as input/output to various hardware processing tasks within DOCA libraries
- `doca_mmap` - describes registered memory, which is accessible by devices, with a set of permissions. `doca_buf` is a segment in the memory range represented by `doca_mmap` .
- `doca_buf_inventory` - pool of `doca_buf` with the same characteristics (see more in sections "DOCA Core Buffers" and "DOCA Core Inventories")

The following diagram shows the various modules within the DOCA memory subsystem:



The diagram shows a `doca_buf_inventory` containing 2 `doca_buf` s. Each `doca_buf` points to a portion of the memory buffer which is part of a `doca_mmap` . The mmap is populated with one continuous memory range and is registered with 2 DOCA Devices, `dev1` and `dev2` .

> For more details about DOCA Memory management subsystem, see section "DOCA Memory Subsystem".

14.1.2.3 Execution Model

DOCA SDK introduces libraries that utilize hardware processing units. Each library defines dedicated APIs for achieving a specific processing task (e.g., encryption). The library abstracts all the low-level details related to operation of the hardware, allowing the application focus on what matters. This type of library is referred to as a context. Since a context utilizes a hardware processing unit, it requires a device to operate. This device also determines which buffers are accessible by that context. Contexts provide hardware processing operation APIs in the form of tasks and events.

Task:

- Application prepares the task arguments
- Application submits the task; this issues a request to the relevant hardware processing unit
- Application receives a completion in the form of a callback once hardware processing completes

Event:

- Application registers to the event. This informs hardware to report whenever the event occurs.
- Application receives a completion in the form of a callback every time hardware identifies that the event has occurred

Since hardware processing is asynchronous in nature. DOCA provides an object that allows waiting on processing operations (tasks and events). This object is referred to as a Progress Engine (PE). The PE allows waiting on completions using the following methods:

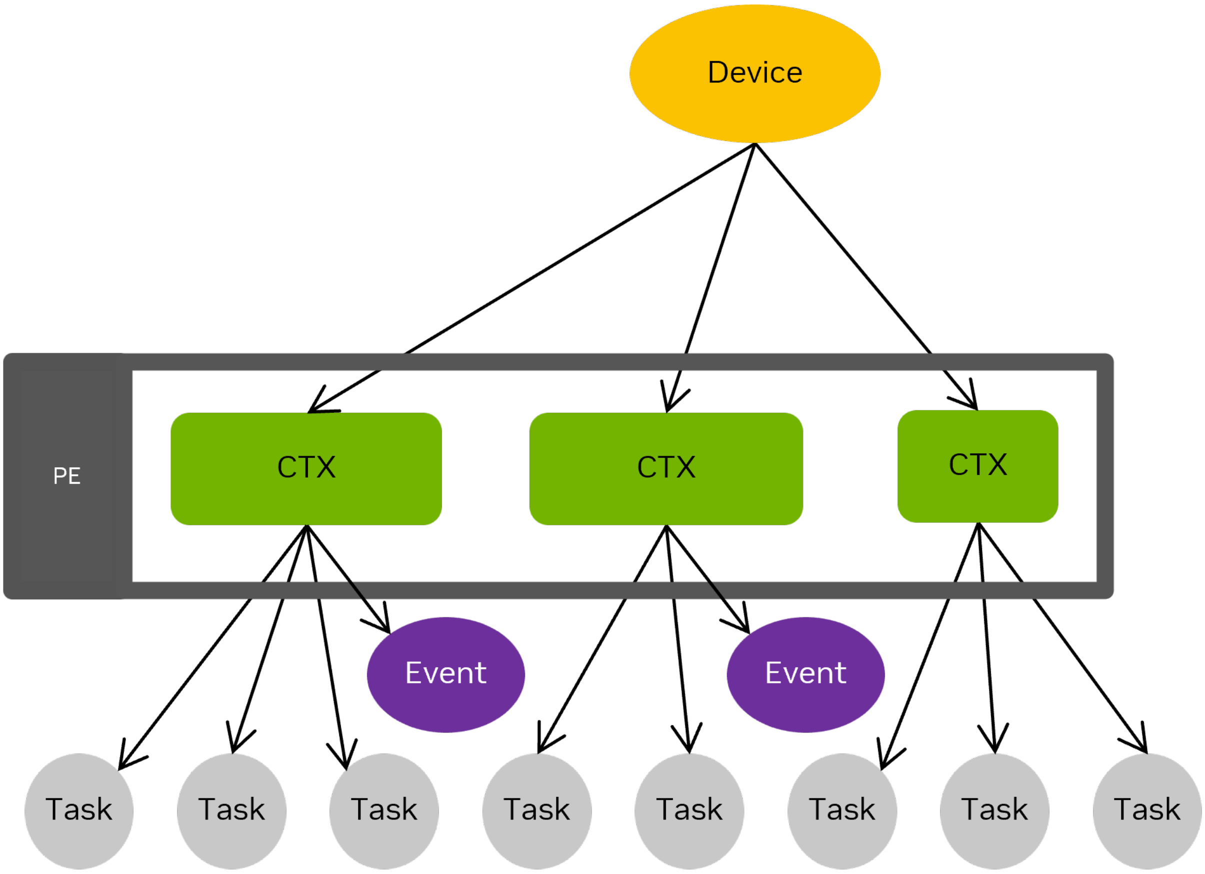
- Busy waiting/polling mode – in this case, the application repeatedly invokes a method that checks if a completion has occurred
- Notification-driven mode – in this case, the application can use OS primitives (e.g., `linux event fd`) to notify the thread whenever some completion has occurred

Once completion occurs, whether caused by a task or event, the relevant callback is invoked as part of the PE method.

A single PE instance allows waiting on multiple tasks/events from different contexts. As such, it is possible for an application to utilize a single PE per thread.

> ⓘ For more details about the DOCA Progress Engine, see section "DOCA Progress Engine".

The following diagram illustrates how a combination of various DOCA modules combine DOCA cross-library processing runtime.



The diagram shows 3 contexts utilizing the same device, each context has some tasks/events that have been submitted/registered by the application. All 3 contexts are connected to the same PE, where the application can use the same PE to wait on all completions at once.

> ⓘ For more details about DOCA Execution model see section "DOCA Execution Model".

14.2 DOCA Backward Compatibility Policy

The NVIDIA DOCA™ SDK enables developers to rapidly create applications and services on top of NVIDIA® BlueField® networking platforms.

The DOCA software package is released on a quarterly release cadence to deliver new features, performance improvements, and critical bug fixes. DOCA compatibility allows users to update the latest DOCA software package (including all libraries, drivers, and tools) without requiring updating the application.

14.2.1 DOCA SDK Versioning

DOCA versions follow the Semantic Versioning scheme. That is, the DOCA version is of the form X.Y.Z, and each part is incremented when the following applies:

- Major version - when incompatible API changes may be introduced
- Minor version - when functionality is added in a backwards compatible manner
- Patch version - when backwards compatible bug fixes are submitted

14.2.2 DOCA SDK API Backwards Compatibility

One of the key attributes of enterprise grade SDK is backward compatibility. Backward compatible APIs allows application developers using the SDK to monetize on their investment, by guaranteeing that their application will continue to operate successfully as they update to a newer SDK version.

DOCA SDK APIs may go through the following lifecycle stages:

1. Experimental – an API marked as `DOCA_EXPERIMENTAL` is an experimental API and is not guaranteed to be present across upcoming releases
2. Stable – an API marked as `DOCA_STABLE` is guaranteed to be supported throughout the lifecycle of the current major version
3. Deprecated – an API marked as `DOCA_DEPRECATED` will be removed from DOCA SDKs header files in an upcoming release. If the API was previously marked as `DOCA_STABLE`, it will only be removed in an upcoming major release.
4. Removed – an API that was present on an older major version and is now no longer supported. If this API was previously marked as `DOCA_STABLE`, the binary representation is preserved to maintain binary backwards compatibility.

The following subsections explain the different backwards compatibility types including how semantic versions are mapped to these different types.

14.2.2.1 Source Compatibility

Source compatibility guarantees that a program written and compiled using a given DOCA SDK version compiles successfully against a newer DOCA SDK version.

As described in section "DOCA SDK Versioning", DOCA SDK is source compatible across minor and patch versions. However, across major version, APIs can be changed, deprecated, or removed (see the lifecycle stages under section "DOCA SDK API Backwards Compatibility"). Therefore, an application that compiles successfully on an older major DOCA SDK version of the toolkit may require changes to compile against a newer major version.

14.2.2.2 Binary Compatibility

Binary compatibility guarantees that a program dynamically linked against a given DOCA SDK library (`*.so`) successfully links against a newer DOCA SDK library.

DOCA SDK API has a versioned C-style application binary interface (ABI) which guarantees binary compatibility across both minor and major versions. This means that upgrading the DOCA SDK package installed on a system to a newer version always supports existing applications and their functions.

14.2.2.3 Behavioral Compatibility

Behavioral compatibility (i.e., semantic compatibility) guarantees that given the same inputs, a function or component will produce the same outputs. Thus, an application developed, compiled, linked, and tested with a given DOCA SDK and relying on the SDK's behavior, can successfully run with newer version of DOCA SDK, as the behavior will be compatible (apart from fixing bugs).

14.2.3 DOCA SDK Protocol Compatibility

Some DOCA SDK components include interaction across remote entities (host-to-BlueField, BlueField-to-BlueField, or host-to-host). That is, communication channel between a process running on the host server and a process running on the BlueField networking platform Arm processors. Since applications using DOCA may be deployed in large clusters and upgraded on a different schedule, DOCA SDK guarantees maintaining different DOCA SDK versions protocol-compatible with each other. This allows the flexibility to perform a rolling upgrade to DOCA SDK applications while maintaining operations throughout the process (nodes with different SDK versions maintain communication).

14.2.4 DOCA SDK Dependencies Compatibility

DOCA is distributed in a meta-package format, either as a `*.bfb` file for installation on the BlueField networking platform Arm processor, or as a DOCA-for-host package (`*.rpm` or `*.deb`) for installation on the server hosting the BlueField networking platform. This package includes different libraries, tools, executables, firmware, and sample applications.

DOCA SDK is developed and tested to work with all components included in the meta-package. There is no guarantee that DOCA SDK would work correctly if any of these components is upgraded independently. Thus, updating DOCA to a newer version requires updating the meta-package with all its components.

14.3 DOCA Development Best Practices

The following sub-sections describe some best practices DOCA SDK users/developers should consider when using DOCA SDK.

- Capability Checking
- Debuggability

14.3.1 Capability Checking

An application that uses a DOCA Device relies on a subset of features for it to function as designed. As such, it is recommended to check whether these features exist for the selected DOCA Device. To achieve this, DOCA SDK exposes capabilities which are a set of APIs with a common look and feel, as described on this page.

The application is expected to use these capability APIs prior to any use of DOCA SDK APIs (Core, libraries) to fail as soon as possible (before initializing any resource) and to be able to implement fallback flows instead of getting error unexpectedly in the application flow.

14.3.1.1 Device Capability

An application that uses DOCA Core APIs may need to identify the specific DOCA Device to work based on specific capabilities.

For that, `doca_devinfo` and `doca_devinfo_rep` expose APIs with the prefix `doca_devinfo_cap_*` / `doca_devinfo_rep_cap_*` . For example:

```
doca_error_t doca_devinfo_cap_is_hotplug_manager_supported(const struct doca_devinfo *devinfo, uint8_t
*is_hotplug_manager);
doca_error_t doca_devinfo_rep_cap_is_filter_emulated_supported(const struct doca_devinfo *devinfo, uint8_t
*filter_emulated_supported);
```

14.3.1.2 Library Capability

Each DOCA library exposes a set of capability APIs for the following purposes:

- Querying the maximum/minimum valid values of a configuration property of the library context or a library task
- Validating whether a library task is supported for a specific DOCA Device

All library capability API starts with the prefix `doca_<library_name>_cap_*`. Moreover:

- Configuration limitation capability APIs start with the prefix `doca_<library_name>_cap_[task_<task_type>]_get_min/max_*`
- Task supported capability APIs have the naming template `doca_<library_name>_cap_task_<task_type>_is_supported`

For example, DOCA DMA exposes:

```
doca_error_t doca_dma_cap_task_memcpy_is_supported(const struct doca_devinfo *devinfo);
doca_error_t doca_dma_cap_get_max_num_tasks(struct doca_dma *dma, uint32_t *max_num_tasks);
doca_error_t doca_dma_cap_task_memcpy_get_max_buf_size(const struct doca_devinfo *devinfo, uint64_t *buf_size);
```

14.3.1.3 Core Capability

Like any other DOCA library, each DOCA Core module also exposes capability APIs.

For example:

- A hotplug (of emulated PCIe functions) oriented application can check if a specific DOCA Device information structure enables hotplugging emulated devices, by calling:

```
doca_error_t doca_devinfo_cap_is_hotplug_manager_supported(const struct doca_devinfo *devinfo, uint8_t
*is_hotplug_manager);
```

- An application that works with DOCA mmap to be shared between the host and BlueField, must export the `doca_mmap` from the host and import it from BlueField. Before starting the workflow, the application can check if those operations are supported for a given a `doca_devinfo` using the following APIs:

```
doca_error_t doca_mmap_cap_is_export_pci_supported(const struct doca_devinfo *devinfo, uint8_t
*mmap_export);
doca_error_t doca_mmap_cap_is_create_from_export_pci_supported(const struct doca_devinfo *devinfo, uint8_t
*from_export);
```

14.3.2 Debuggability

14.3.2.1 Return value

All DOCA APIs return the status in the form of doca_error_t.

The return value of every call to the DOCA API should be checked to verify that it was successful. In case of an error, one should look at the meaning of the returned value in the description of the failing function.

14.3.2.2 SDK log

DOCA SDK supports error message and debug prints.

For enabling the DOCA SDK log messages one should create a backend and set the verbosity level of that backend, if needed.

For more details about DOCA log, see section "DOCA Log".

14.4 DOCA Libraries

This section describes in details how to use each DOCA library, its APIs, and different aspects related to that library.

Users may choose to only read the pages concerning DOCA libraries required for their application.

This section contains the following pages:

- DOCA Common
- DOCA Flow
- DPA Subsystem
- DOCA DMA
- DOCA Comch
- DOCA UROM
- DOCA RDMA
- DOCA Ethernet
- DOCA GPUNetIO
- DOCA App Shield
- DOCA Compress
- DOCA SHA
- DOCA Erasure Coding
- DOCA AES-GCM
- DOCA Rivermax
- DOCA Telemetry Exporter
- DOCA Telemetry Diagnostics
- DOCA Device Emulation

ⓘ For questions, comments, and feedback, please contact us at DOCA-Feedback@exchange.nvidia.com.

14.4.1 DOCA Common

DOCA Common is comprised of the following libraries:

- DOCA Core

- DOCA Log

14.4.1.1 DOCA Core

This document provides guidelines on using DOCA Core objects as part of DOCA SDK programming.

14.4.1.1.1 Introduction

> The DOCA Core library is supported at beta level.

DOCA Core objects provide a unified and holistic interface for application developers to interact with various DOCA libraries. The DOCA Core API and objects bring a standardized flow and building blocks for applications to build upon while hiding the internal details of dealing with hardware and other software components. DOCA Core is designed to give the right level of abstraction while maintaining performance.

DOCA Core has the same API (header files) for both NVIDIA® BlueField® and CPU installations, but specific API calls may return `DOCA_ERROR_NOT_SUPPORTED` if the API is not implemented for that processor. However, this is not the case for Windows and Linux as DOCA Core does have API differences between Windows and Linux installations.

DOCA Core exposes C-language API to application writers and users must include the right header file to use according to the DOCA Core facilities needed for their application.

DOCA Core can be divided into the following software modules:

DOCA Core Module	Description
General	• DOCA Core enumerations and basic structures • Header files - `doca_error.h` , `doca_types.h`
Device handling	• Queries device information (host-side and BlueField) and device capabilities (e.g., device's PCIe BDF address) • On BlueField • Gets local BlueField devices • Gets representors list (representing host local devices) • On the host • Gets local devices • Queries device capabilities and library capabilities • Opens and uses the selected device representor • Relevant entities - `doca_devinfo` , `doca_devinfo_rep` , `doca_dev` , `doca_dev_rep` • Header files - `doca_dev.h` There is a symmetry between device entities on host and its representor (on BlueField). The convention of adding `rep` to the API or the object hints that it is representor-specific.

DOCA Core Module	Description
Memory management	• Handles optimized memory pools to be used by applications and enables sharing resources between DOCA libraries (while hiding hardware-related technicalities) • Data buffer services (e.g., linked list of buffers to support scatter-gather list) • Maps host memory to BlueField for direct access • Relevant entities - `doca_buf`, `doca_mmap`, `doca_buf_inventory`, `doca_buf_array`, `doca_bufpool` • Header files - `doca_buf.h`, `doca_buf_inventory.h`, `doca_mmap.h`, `doca_buf_array.h`, `doca_bufpool`
Progress engine and task execution	• Enables submitting tasks to DOCA libraries and track task progress (supports both polling mode and event-driven mode) • Relevant entities - `doca_ctx`, `doca_task`, `doca_event`, `doca_event_handle_t`, `doca_pe` • Header files - `doca_ctx.h`
Sync events	• Sync events are used to synchronize different processors (e.g., synchronize BlueField and host) • header files - `doca_dpa_sync_event.h`, `doca_sync_event.h`

The following sections describe DOCA Core's architecture and subsystems along with some basic flows that help users get started using DOCA Core.

14.4.1.1.2 Prerequisites

DOCA Core objects are supported on NVIDIA® BlueField® networking platforms (DPU or SuperNIC) and the host machine. Both must meet the following prerequisites:

- DOCA version 2.0.2 or greater
- NVIDIA® BlueField® software 4.0.2 or greater
- NVIDIA® BlueField®-3 firmware version 32.37.1000 and higher
- NVIDIA® BlueField®-2 firmware version 24.37.1000 and higher
- Please refer to the DOCA Backward Compatibility Policy

14.4.1.1.3 Changes From Previous Releases

14.4.1.1.3.1 Changes in 2.8.0

Added

- `doca_bitfield.h`
- `doca_error_t doca_buf_inventory_expand(struct doca_buf_inventory *inventory, uint32_t num_elements)`
- `void doca_ctx_flush_tasks(struct doca_ctx *ctx)`
- `doca_error_t doca_devinfo_cap_is_notification_moderation_supported(const struct doca_devinfo *devinfo, uint8_t *is_notification_moderation_supported)`

- New DOCA errors: `DOCA_ERROR_AUTHENTICATION`, `DOCA_ERROR_BAD_CONFIG`, `DOCA_ERROR_SKIPPED`
- `doca_error_t doca_task_submit_ex(struct doca_task *task, uint32_t flags)`
- `doca_error_t doca_pe_set_notification_affinity(struct doca_pe *pe, uint32_t core_id)`
- `doca_error_t doca_pe_is_set_notification_affinity_supported(const struct doca_devinfo *devinfo, uint8_t *is_set_notification_affinity_supported`

Changed

- `doca_error_t doca_devinfo_get_active_rate(const struct doca_devinfo *devinfo, doubleuint64_t *active_rate); // Gb/s -> bits/s`
- `doca_buf_set_data_len` is STABLE API
- Imported mmap can be exported to RDMA

14.4.1.1.4 Architecture

The following sections describe the architecture for the various DOCA Core software modules. Please refer to the NVIDIA DOCA Library APIs for DOCA header documentation.

14.4.1.1.4.1 General

All core objects adhere to same flow that later helps in doing no allocations in the fast path.

The flow is as follows:

1. Create the object instance (e.g., `doca_mmap_create`).
2. Configure the instance (e.g., `doca_mmap_set_memory_range`).
3. Start the instance (e.g., `doca_mmap_start`).

After the instance is started, it adheres to zero allocations and can be used safely in the data path. After the instance is complete, it must be stopped and destroyed (`doca_mmap_stop`, `doca_mmap_destroy`).

There are core objects that can be reconfigured and restarted again (i.e., create → configure → start → stop → configure → start). Please read the header file to see if specific objects support this option.

doca_error_t

All DOCA APIs return the status in the form of `doca_error_t`.

```
typedef enum doca_error {
    DOCA_SUCCESS,
    DOCA_ERROR_UNKNOWN,
    DOCA_ERROR_NOT_PERMITTED,           /**< Operation not permitted */
    DOCA_ERROR_IN_USE,                  /**< Resource already in use */
    DOCA_ERROR_NOT_SUPPORTED,           /**< Operation not supported */
    DOCA_ERROR_AGAIN,                   /**< Resource temporarily unavailable, try again */
    DOCA_ERROR_INVALID_VALUE,           /**< Invalid input */
    DOCA_ERROR_NO_MEMORY,               /**< Memory allocation failure */
    DOCA_ERROR_INITIALIZATION,          /**< Resource initialization failure */
    DOCA_ERROR_TIME_OUT,                /**< Timer expired waiting for resource */
    DOCA_ERROR_SHUTDOWN,                /**< Shut down in process or completed */
    DOCA_ERROR_CONNECTION_RESET,        /**< Connection reset by peer */
    DOCA_ERROR_CONNECTION_ABORTED,      /**< Connection aborted */
    DOCA_ERROR_CONNECTION_INPROGRESS,   /**< Connection in progress */
    DOCA_ERROR_NOT_CONNECTED,           /**< Not Connected */
    DOCA_ERROR_NO_LOCK,                 /**< Unable to acquire required lock */
    DOCA_ERROR_NOT_FOUND,              /**< Resource Not Found */
    DOCA_ERROR_IO_FAILED,              /**< Input/Output Operation Failed */
    DOCA_ERROR_BAD_STATE,              /**< Bad State */
    DOCA_ERROR_UNSUPPORTED_VERSION,    /**< Unsupported version */
    DOCA_ERROR_OPERATING_SYSTEM,       /**< Operating system call failure */
    DOCA_ERROR_DRIVER,                 /**< DOCA Driver call failure */
    DOCA_ERROR_UNEXPECTED,             /**< An unexpected scenario was detected */
    DOCA_ERROR_ALREADY_EXIST,          /**< Resource already exist */
    DOCA_ERROR_FULL,                   /**< No more space in resource */
    DOCA_ERROR_EMPTY,                  /**< No entry is available in resource */
    DOCA_ERROR_IN_PROGRESS,            /**< Operation is in progress */
    DOCA_ERROR_TOO_BIG,                /**< Requested operation too big to be contained */
 } doca_error_t;
```

See `doca_error.h` for more.

Generic Structures/Enum

The following types are common across all DOCA APIs.

```
union doca_data {
        void *ptr;
        uint64_t u64;
};

enum doca_access_flags {
    DOCA_ACCESS_LOCAL_READ_ONLY     = 0,
    DOCA_ACCESS_LOCAL_READ_WRITE    = (1 << 0),
    DOCA_ACCESS_RDMA_READ           = (1 << 1),
    DOCA_ACCESS_RDMA_WRITE          = (1 << 2),
    DOCA_ACCESS_RDMA_ATOMIC         = (1 << 3),
    DOCA_ACCESS_DPU_READ_ONLY       = (1 << 4),
    DOCA_ACCESS_DPU_READ_WRITE      = (1 << 5),
};

enum doca_pci_func_type {
        DOCA_PCI_FUNC_PF = 0, /* physical function */
        DOCA_PCI_FUNC_VF,     /* virtual function */
        DOCA_PCI_FUNC_SF,     /* sub function */
};
```

For more see `doca_types.h`.

14.4.1.1.4.2 DOCA Device

Local Device and Representor

Prerequisites

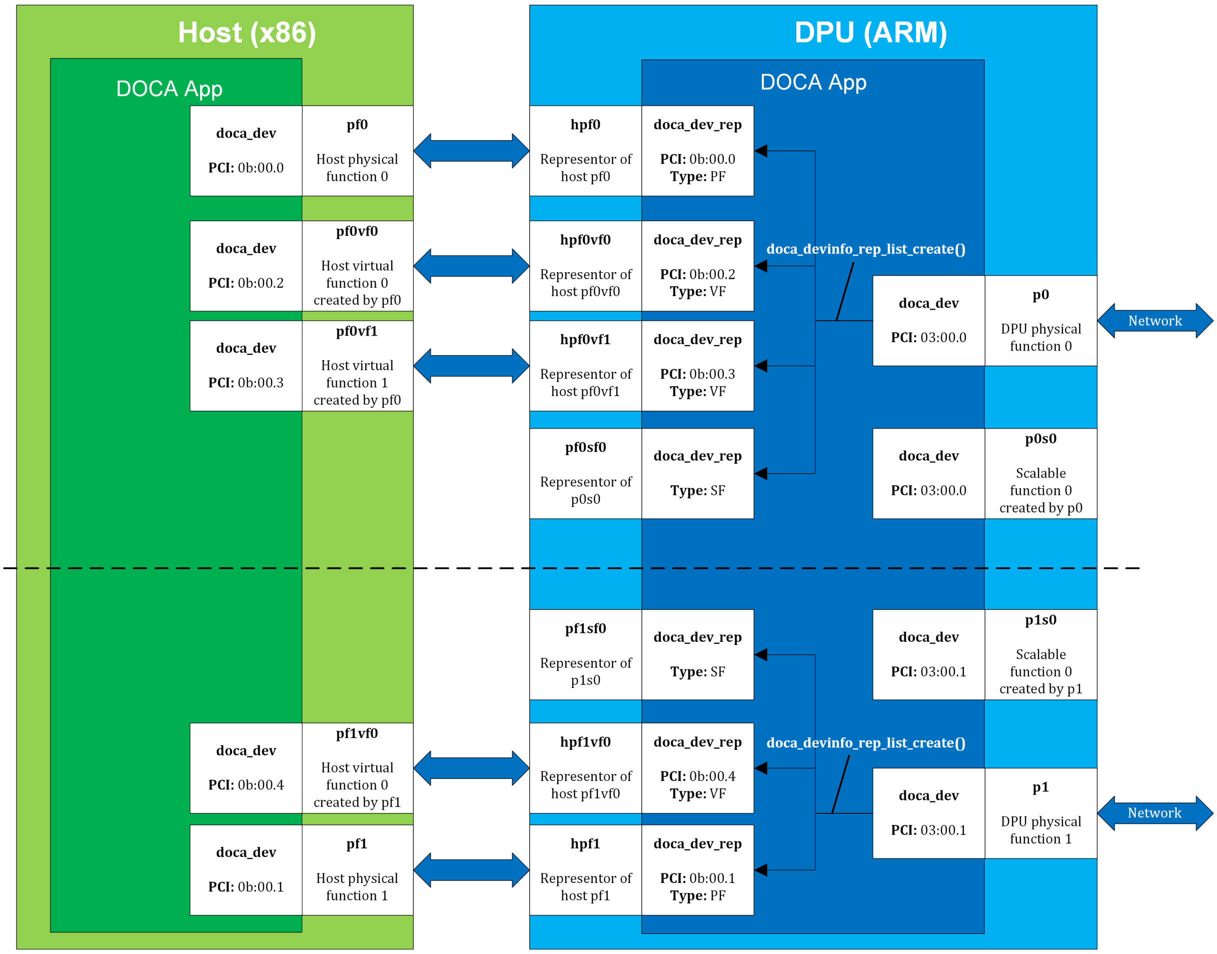
For the representors model, BlueField must be operated in DPU mode. See NVIDIA BlueField Modes of Operation.

Topology

The DOCA device represents an available processing unit backed by hardware or software implementation. The DOCA device exposes its properties to help an application in choosing the right device(s). DOCA Core supports two device types:

- Local device – this is an actual device exposed in the local system (BlueField or host) and can perform DOCA library processing tasks.
- Representor device – this is a representation of a local device. The local device is usually on the host (except for SFs) and the representor is always on BlueField side (a proxy on BlueField for the host-side device).

The following figure provides an example topology:



The diagram shows a BlueField device (on the right side of the figure) connected to a host (on the left side of the figure). The host topology consists of two physical functions (PF0 and PF1). Furthermore, PF0 has two child virtual functions, VF0 and VF1. PF1 has only one VF associated with it, VF0. Using the DOCA SDK API, the user gets these five devices as local devices on the host.

The BlueField side has a representor-device per each host function in a 1-to-1 relation (e.g., `hpf0` is the representor device for the host's PF0 device and so on) as well as a representor for each SF function, such that both the SF and its representor reside in BlueField.

If the user queries local devices on the BlueField (not representor devices), they get the two (in this example) BlueField DPU PFs, `p0` and `p1`. These two BlueField local devices are the parent devices for:

- 7 representor devices –
    - 5 representor devices shown as arrows to/from the host (devices with the prefix `hpf*`) in the diagram
    - 2 representor devices for the SF devices, `pf0sf0` and `pf1sf0`
- 2 local SF devices (not the SF representors), `p0s0` and `p1s0`

In the diagram, the topology is split into two parts (note the dotted line), each part is represented by a BlueField physical device, `p0` and `p1`, each of which is responsible for creating all other local devices (host PFs, host VFs, and BlueField SFs). As such, the BlueField physical device can be referred to as the parent device of the other devices and would have access to the representor of every other function (via `doca_devinfo_rep_list_create`).

Local Device and Representor Matching

Based on the topology diagram, the mmap export APIs can be used as follows:

Device to Select on Host When Using doca_mmap_export_dpu()	BlueField Matching Representor	Device to Select on BlueField When Using doca_mmap_create_from_export()
pf0 - 0b:00.0	hpf0 - 0b:00.0	p0 - 03:00.0
pf0vf0 - 0b:00.2	hpf0vf0 - 0b:00.2	
pf0vf1 - 0b:00.3	hpf0vf1 - 0b:00.3	
pf1 - 0b:00.1	hpf1 - 0b:00.1	p1 - 03:00.1
pf1vf0 - 0b:00.4	hpf1vf0 - 0b:00.4	

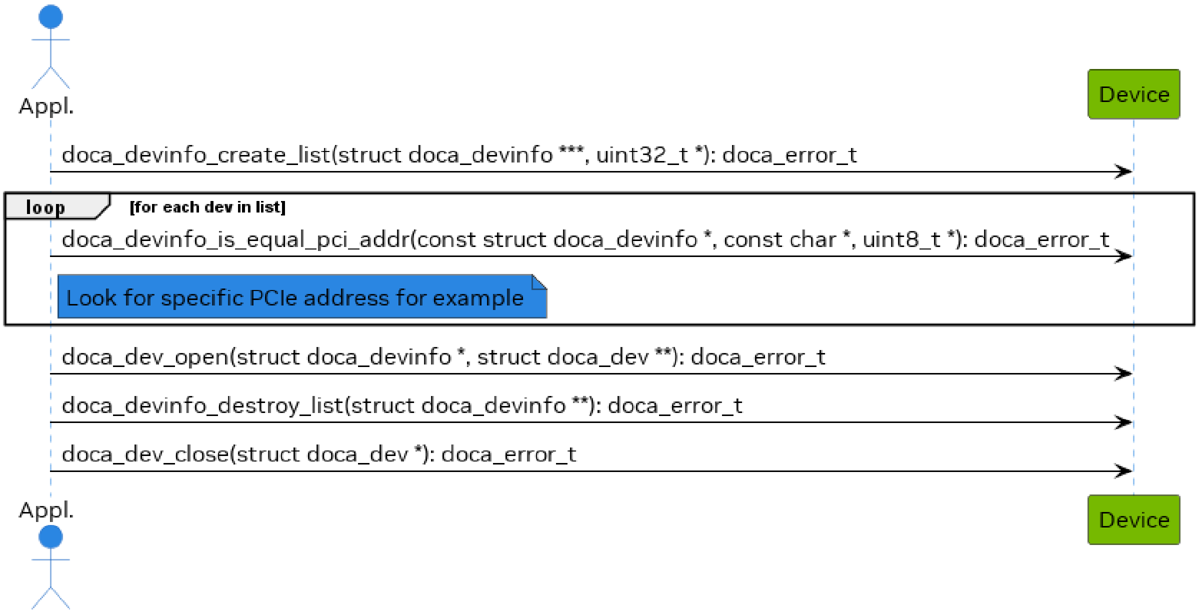
Expected Flow

Device Discovery

To work with DOCA libraries or DOCA Core objects, application must open and use a device on BlueField or host.

There are usually multiple devices available depending on the setup. See section "Topology" for more information.

An application can decide which device to select based on capabilities, the DOCA Core API, and every other library which provides a wide range of device capabilities. The flow is as follows:



```
doca_devinfo_create_list(struct doca_devinfo ***, uint32_t *): doca_error_t
doca_devinfo_is_equal_pci_addr(const struct doca_devinfo *, const char *, uint8_t *): doca_error_t
doca_dev_open(struct doca_devinfo *, struct doca_dev **): doca_error_t
doca_devinfo_destroy_list(struct doca_devinfo **): doca_error_t
doca_dev_close(struct doca_dev *): doca_error_t
```

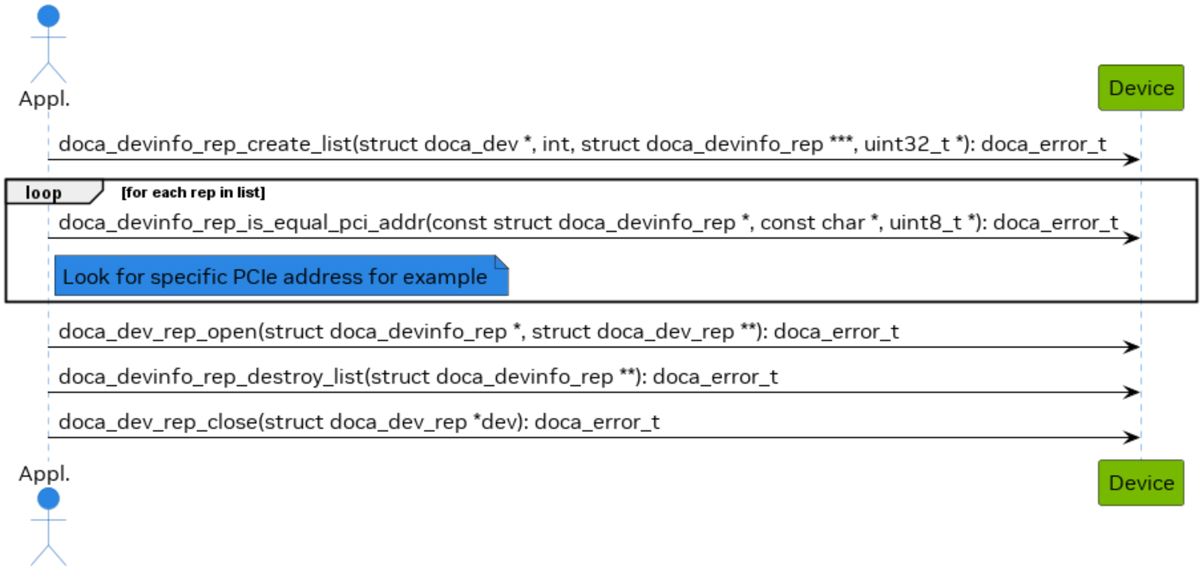
Look for specific PCIe address for example

1. The application gets a list of available devices.
2. Select a specific `doca_devinfo` to work with according to one of its properties and capabilities. This example looks for a specific PCIe address.
3. Once the `doca_devinfo` that suits the user's needs is found, open `doca_dev`.

4. After the user opens the right device, they can close the `doca_devinfo` list and continue working with `doca_dev`. The application eventually must close the `doca_dev`.

Representor Device Discovery

To work with DOCA libraries or DOCA Core objects, some applications must open and use a representor device on BlueField. Before they can open the representor device and use it, applications need tools to allow them to select the appropriate representor device with the necessary capabilities. The DOCA Core API provides a wide range of device capabilities to help the application select the right device pair (device and its BlueField representor). The flow is as follows:



Appl.
Device
doca_devinfo_rep_create_list(struct doca_dev *, int, struct doca_devinfo_rep ***, uint32_t *): doca_error_t
loop
[for each rep in list]
doca_devinfo_rep_is_equal_pci_addr(const struct doca_devinfo_rep *, const char *, uint8_t *): doca_error_t
Look for specific PCIe address for example
doca_dev_rep_open(struct doca_devinfo_rep *, struct doca_dev_rep **): doca_error_t
doca_devinfo_rep_destroy_list(struct doca_devinfo_rep **): doca_error_t
doca_dev_rep_close(struct doca_dev_rep *dev): doca_error_t
Appl.
Device

1. The application "knows" which device it wants to use (e.g., by its PCIe BDF address). On the host, it can be done using DOCA Core API or OS services.
2. On the BlueField side, the application gets a list of device representors for a specific BlueField local device.
3. Select a specific `doca_devinfo_rep` to work with according to one of its properties. This example looks for a specific PCIe address.
4. Once the `doca_devinfo_rep` that suits the user's needs is found, open `doca_dev_rep`.
5. After the user opens the right device representor, they can close the `doca_devinfo_rep` list and continue working with `doca_dev_rep`. The application eventually must close `doca_dev_rep` too.

As mentioned previously, the DOCA Core API can identify devices and their representors that have a unique property (e.g., the BDF address, the same BDF for the device, and its BlueField representor).

> Regarding representor device property caching, the function `doca_devinfo_rep_create_list` provides a snapshot of the DOCA representor device properties when it is called. If any representor's properties are changed dynamically (e.g., BDF address changes after bus reset), the device properties that the function returns would not reflect this change. One should create the list again to get the updated properties of the representors.

14.4.1.1.4.3 DOCA Memory Subsystem

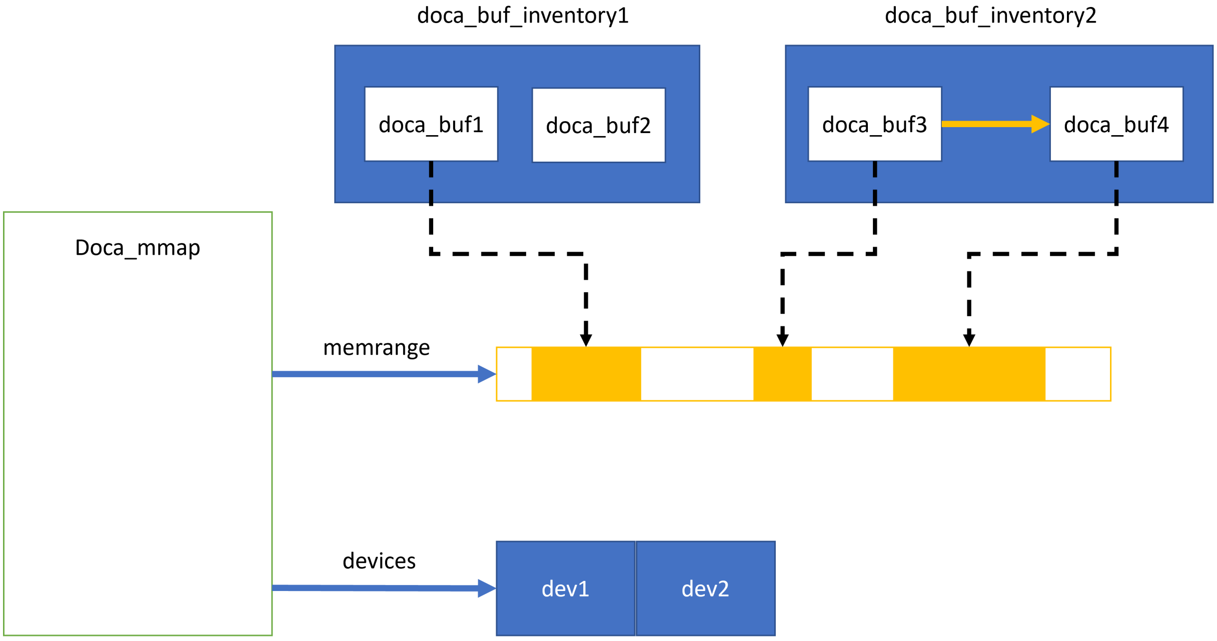
DOCA memory subsystem is designed to optimize performance while keeping a minimal memory footprint (to facilitate scalability) as main design goal.

DOCA memory has the following main components:

- `doca_buf` - this is the data buffer descriptor. This is not the actual data buffer, rather, it is a descriptor that holds metadata on the "pointed" data buffer.
- `doca_mmap` - this is the data buffers pool which `doca_buf` points at. The application provides the memory as a single memory region, as well as permissions for certain devices to access it.

As the `doca_mmap` serves as the memory pool for data buffers, there is also an entity called `doca_buf_inventory` which serves as a pool of `doca_buf` with same characteristics (see more in sections "DOCA Core Buffers" and "DOCA Core Inventories"). As all DOCA entities, memory subsystem objects are opaque and can be instantiated by DOCA SDK only.

The following diagram shows the various modules within the DOCA memory subsystem.



In the diagram, you may see two `doca_buf_inventory` s. Each `doca_buf` points to a portion of the memory buffer which is part of a `doca_mmap` . The mmap is populated with one continuous memory buffer `memrange` and is mapped to two devices, `dev1` and `dev2` .

Requirements and Considerations

- The DOCA memory subsystem mandates the usage of pools as opposed to dynamic allocation
    - Pool for `doca_buf` → `doca_buf_inventory`
    - Pool for data memory → `doca_mmap`
- The memory buffer in the mmap can be mapped to one device or more
- Devices in the mmap are restricted by access permissions defining how they can access the memory buffer

- `doca_buf` points to a specific memory buffer (or part of it) and holds the metadata for that buffer
- The internals of mapping and working with the device (e.g., memory registrations) is hidden from the application
- As best practice, the application should start the `doca_mmap` in the initialization phase as the start operation is time consuming. `doca_mmap` should not be started as part of the data path unless necessary.
- The host-mapped memory buffer can be accessed by BlueField

doca_mmap

`doca_mmap` is more than just a data buffer as it hides a lot of details (e.g., RDMA technicalities, device handling, etc.) from the application developer while giving the right level of abstraction to the software using it. `doca_mmap` is the best way to share memory between the host and BlueField so BlueField can have direct access to the host-side memory or vice versa.

DOCA SDK supports several types of mmap that help with different use cases: local mmap and mmap from export.

Local mmap

This is the basic type of mmap which maps local buffers to the local device(s).

1. The application creates the `doca_mmap`.
2. The application sets the memory range of the mmap using `doca_mmap_set_memrange`. The memory range is memory that the application allocates and manages (usually holding the pool of data sent to the device's processing units).
3. The application adds devices, granting the devices access to the memory region.
4. The application can specify the access permission for the devices to that memory range using `doca_mmap_set_permissions`.
    - If the mmap is used only locally, then `DOCA_ACCESS_LOCAL_*` must be specified
    - If the mmap is created on the host but shared with BlueField (see step 6), then `DOCA_ACCESS_PCI_*` must be specified
    - If the mmap is created on BlueField but shared with the host (see step 6), then `DOCA_ACCESS_PCI_*` must be specified
    - If the mmap is shared with a remote RDMA target, then `DOCA_ACCESS_RDMA_*` must be specified
5. The application starts the mmap.

    > ⚠ From this point no more changes can be made to the mmap.

6. To share the mmap with BlueField/host or the RDMA remote target, call `doca_mmap_export_pci` or `doca_mmap_export_rdma` respectively. If appropriate access has not been provided, the export fails.

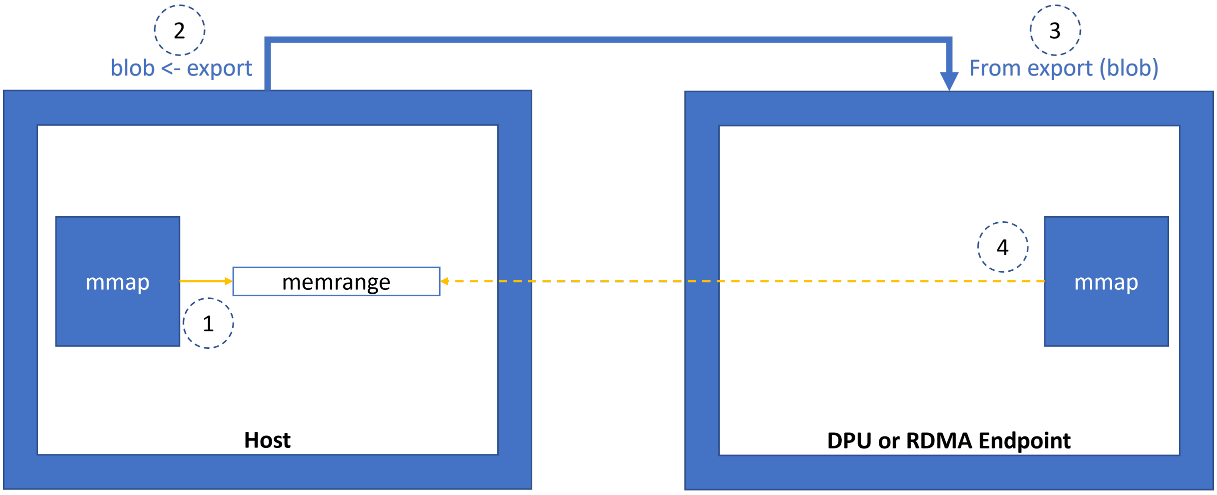
    > ❗ The exported data contains sensitive information. Make sure to pass this data through a secure channel!

7. The generated blob from the previous step can be shared out of band using a socket. If shared with a BlueField, it is recommended to use the DOCA Comm Channel instead. See the DMA Copy application for the exact flow.

mmap from Export

This mmap is used to access the host memory (from BlueField) or the remote RDMA target's memory.

1. The application receives a blob from the other side. The blob contains data returned from step 6 in the former bullet.
2. The application calls `doca_mmap_create_from_export` and receives a new mmap that represents memory defined by the other side.



1 – Create mmap on Host set local memory + access permissions + devices
2 – Export the host mmap and send to other side
3 – on DPU import (create from export) the host mmap to otherside
4 – Access the host memory directly from otherside (using DPU or RDMA)

Now the application can create `doca_buf` to point to this imported mmap and have direct access to the other machine's memory.

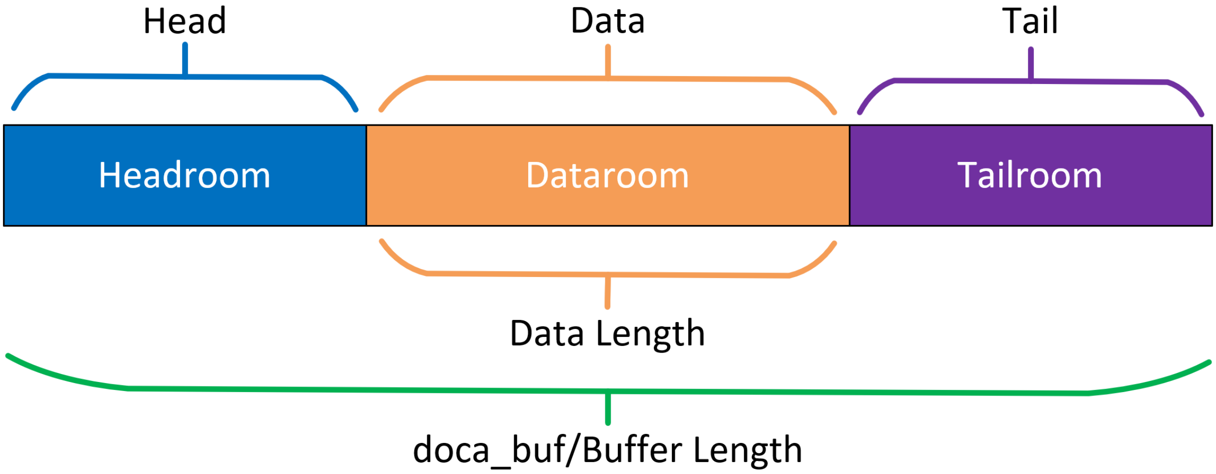
> ⚠ BlueField can access memory exported to BlueField if the exporter is a host on the same machine. Or it can access memory exported through RDMA which can be on the same machine, a remote host, or on a remote BlueField.

> ⚠ The host can only access memory exported through RDMA. This can be memory on a remote host, remote BlueField, or BlueField on same machine.

Buffers

The DOCA buffer object is used to reference memory that is accessible by BlueField hardware. The buffer can be utilized across different BlueField accelerators. The buffer may reference CPU, GPU, host, or even RDMA memory. However, this is abstracted so once a buffer is created, it can be handled in a similar way regardless of how it got created. This section covers usage of the DOCA buffer after it is allocated.

The DOCA buffer has an address and length describing a memory region. Each buffer can also point to data within the region using the data address and data length. This distinguishes three sections of the buffer: The headroom, the dataroom, and the tailroom.



- Headroom - memory region starting from the buffer's address up to the buffer's data address
- Dataroom - memory region starting from the buffer's data address with a length indicated by the buffer's data length
- Tailroom - memory region starting from the end of the dataroom to the end of the buffer
- Buffer length - the total length of the headroom, the dataroom, and the tailroom

Buffer Considerations

- There are multiple ways to create the buffer but, once created, it behaves in the same way (see section "Inventories").
- The buffer may reference memory that is not accessible by the CPU (e.g., RDMA memory)
- The buffer is a thread-unsafe object
- The buffer can be used to represent non-continuous memory regions (scatter/gather list)
- The buffer does not own nor manage the data it references. Freeing a buffer does not affect the underlying memory.

Headroom

The headroom is considered user space. For example, this can be used by the user to hold relevant information regarding the buffer or data coupled with the data in the buffer's dataroom.

This section is ignored and remains untouched by DOCA libraries in all operations.

Dataroom

The dataroom is the content of the buffer, holding either data on which the user may want to perform different operations using DOCA libraries or the result of such operations.

Tailroom

The tailroom is considered as free writing space in the buffer by DOCA libraries (i.e., a memory region that may be written over in different operations where the buffer is used as output).

Buffer as Source

When using `doca_buf` as a source buffer, the source data is considered as the data section only (the dataroom).

Buffer as Destination

When using `doca_buf` as a destination buffer, data is written to the tailroom (i.e., appended after existing data, if any).

When DOCA libraries append data to the buffer, the data length is increased accordingly.

Scatter/Gather List

To execute operations on non-continuous memory regions, it is possible to create a buffer list. The list would be represented by a single `doca_buf` which represents the head of the list.

To create a list of buffers, the user must first allocate each buffer individually and then chain them. Once they are chained, they can be unchained as well:

- The chaining operation, `doca_buf_chain_list()` , receives two lists (heads) and appends the second list to the end of the first list
- The unchaining operation, `doca_buf_unchain_list()` , receives the list (head) and an element in the list, and separates them
- Once the list is created, it can be traversed using `doca_buf_get_next_in_list()` . `NULL` is returned once the last element is reached.

Passing the list to another library is same as passing a single buffer; the application sends the head of the list. DOCA libraries that support this feature can then treat the memory regions that comprise the list as one contiguous.

When using the buffer list as a source, the data of each buffer (in the dataroom) is gathered and used as continuous data for the given operation.

When using the buffer list as destination, data is scattered in the tailroom of the buffers in the list until it is all written (some buffers may not be written to).

Buffer Use Cases

The DOCA buffer is widely used by the DOCA acceleration libraries (e.g., DMA, compress, SHA). In these instances, the buffer can be provided as a source or as a destination.

Buffer use-case considerations:

- If the application wishes to use a linked list buffer and concatenate several `doca_buf`s to a scatter/gather list, the application is expected to ensure the library indeed supports a linked list buffer. For example, to check linked-list support for DMA memcpy task, the application may call `doca_dma_cap_task_memcpy_get_max_buf_list_len()` .
- Operations made on the buffer's data are not atomic unless stated otherwise
- Once a buffer has been passed to the library as part of the task, ownership of the buffer moves to the library until that task is complete

> ⚠ When using `doca_buf` as an input to some processing library (e.g., `doca_dma`), `doca_buf` must remain valid and unmodified until processing is complete.

- Writing to an in-flight buffer may result in anomalous behavior. Similarly, there are no guarantees for data validity when reading from an in-flight buffer.

Inventories

The inventory is the object responsible for allocating DOCA buffers. The most basic inventory allows allocations to be done without having to allocate any system memory. Other inventories involve enforcing that buffer addresses do not overlap.

Inventory Considerations

- All inventories adhere to zero allocation after start.
- Allocation of a DOCA buffer requires a data source and an inventory.
  - The data source defines where the data resides, what can access it, and with what permissions.
  - The data source must be created by the application. For creation of mmaps, see `doca_mmap`.
- The inventory describes the allocation pattern of the buffers, such as, random access or pool, variable-size or fixed-size buffers, and continuous or non-continuous memory.
- Some inventories require providing the data source, `doca_mmap`, when allocating the buffers, others require it on creation of the inventory.
- All inventory types are thread-unsafe.

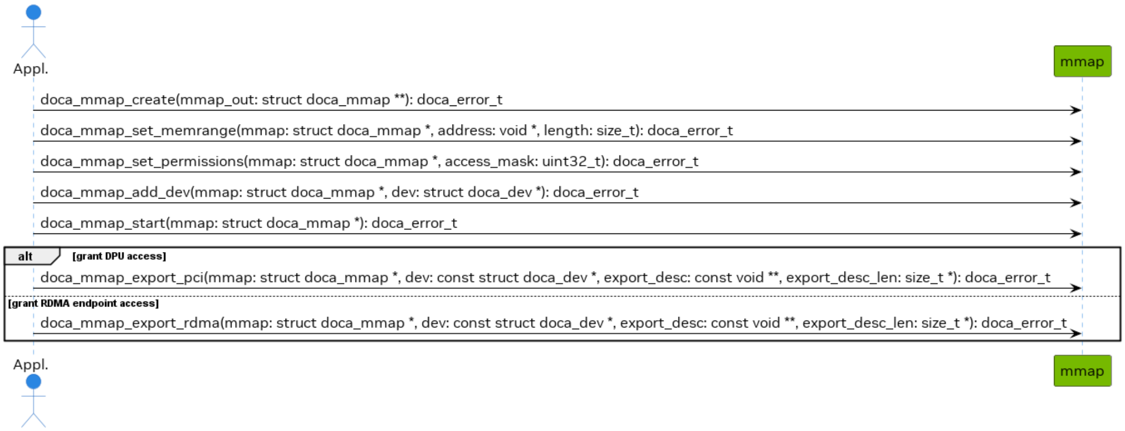
Inventory Types

Inventory Type	Characteristics	When to Use	Notes
`doca_buf_inventory`	Multiple mmaps, flexible address, flexible buffer size.	When multiple sizes or mmaps are used.	Most common use case.
`doca_buf_array`	Single mmap, fixed buffer size. User receives an array of pointers to DOCA buffers. In case of DPA, mmap and buffer size can be unconfigured and later can be set from the DPA.	Use for creating DOCA buffers on GPU or DPA.	`doca_buf_arr` can be configured on the CPU and created on the GPU or DPA
`doca_bufpool`	Single mmap, fixed buffer size, address not controlled by the user.	Use as a pool of buffers of the same characteristics when buffer address is not important.	Slightly faster than `doca_buf_inventory`.

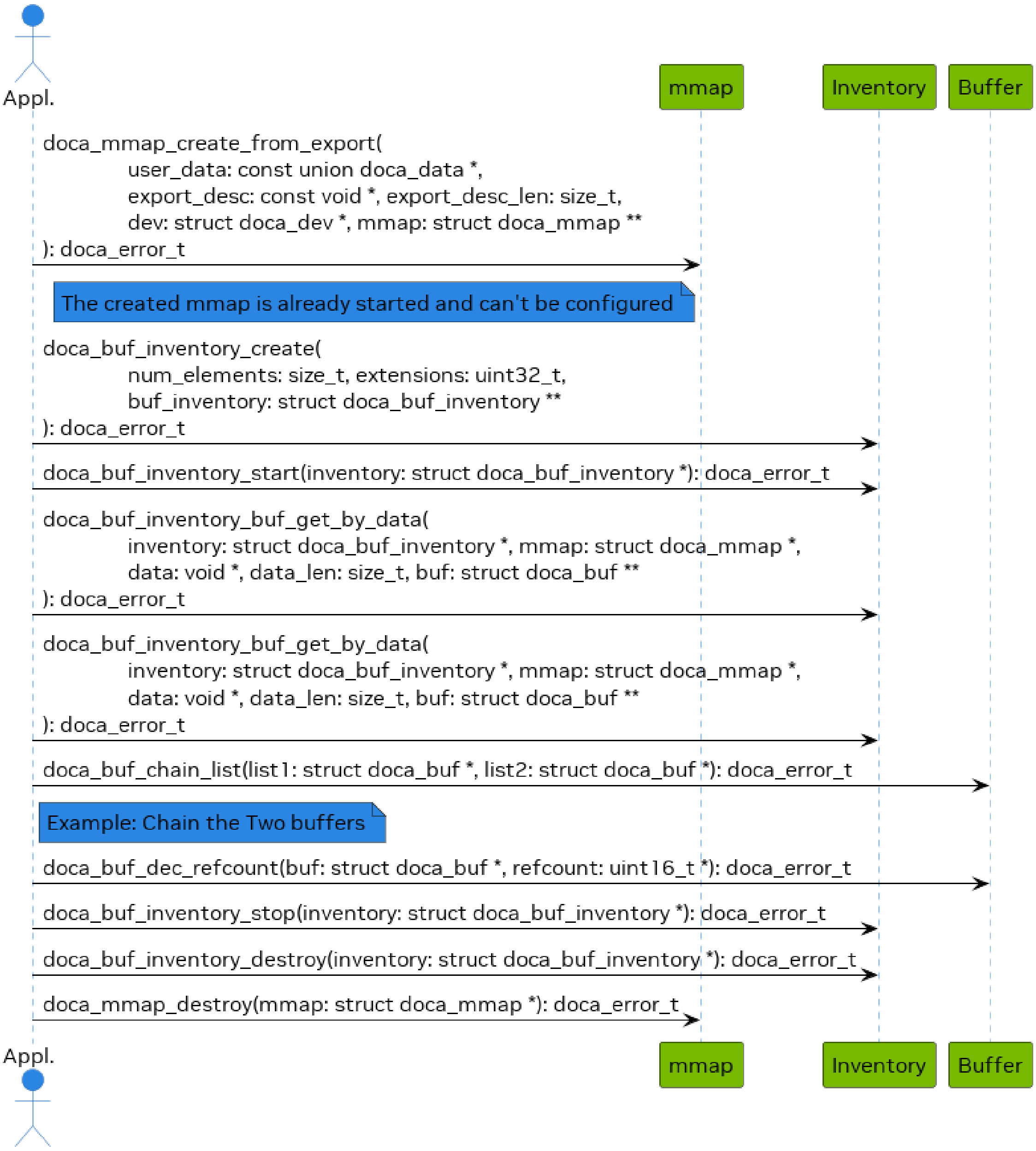
Example Flow

The following is a simplified example of the steps expected for exporting the host mmap to BlueField to be used by DOCA for direct access to the host memory (e.g., for DMA):

1. Create mmap on the host (see section "Expected Flow" for information on how to choose the `doca_dev` to add to mmap if exporting to BlueField). This example adds a single `doca_dev` to the mmap and exports it so the BlueField/RDMA endpoint can use it.


Appl.
mmap
doca_mmap_create(mmap_out: struct doca_mmap **): doca_error_t
doca_mmap_set_memrange(mmap: struct doca_mmap *, address: void *, length: size_t): doca_error_t
doca_mmap_set_permissions(mmap: struct doca_mmap *, access_mask: uint32_t): doca_error_t
doca_mmap_add_dev(mmap: struct doca_mmap *, dev: struct doca_dev *): doca_error_t
doca_mmap_start(mmap: struct doca_mmap *): doca_error_t
alt
[grant DPU access]
doca_mmap_export_pci(mmap: struct doca_mmap *, dev: const struct doca_dev *, export_desc: const void **, export_desc_len: size_t *): doca_error_t
[grant RDMA endpoint access]
doca_mmap_export_rdma(mmap: struct doca_mmap *, dev: const struct doca_dev *, export_desc: const void **, export_desc_len: size_t *): doca_error_t
Appl.
mmap

2. Import to the BlueField/RDMA endpoint (e.g., use the mmap descriptor output parameter as input to `doca_mmap_create_from_export`).



```
doca_mmap_create_from_export(
        user_data: const union doca_data *,
        export_desc: const void *, export_desc_len: size_t,
        dev: struct doca_dev *, mmap: struct doca_mmap **
): doca_error_t
```

The created mmap is already started and can't be configured

```
doca_buf_inventory_create(
        num_elements: size_t, extensions: uint32_t,
        buf_inventory: struct doca_buf_inventory **
): doca_error_t
doca_buf_inventory_start(inventory: struct doca_buf_inventory *): doca_error_t
doca_buf_inventory_buf_get_by_data(
        inventory: struct doca_buf_inventory *, mmap: struct doca_mmap *,
        data: void *, data_len: size_t, buf: struct doca_buf **
): doca_error_t
doca_buf_inventory_buf_get_by_data(
        inventory: struct doca_buf_inventory *, mmap: struct doca_mmap *,
        data: void *, data_len: size_t, buf: struct doca_buf **
): doca_error_t
doca_buf_chain_list(list1: struct doca_buf *, list2: struct doca_buf *): doca_error_t
```

Example: Chain the Two buffers

```
doca_buf_dec_refcount(buf: struct doca_buf *, refcount: uint16_t *): doca_error_t
doca_buf_inventory_stop(inventory: struct doca_buf_inventory *): doca_error_t
doca_buf_inventory_destroy(inventory: struct doca_buf_inventory *): doca_error_t
doca_mmap_destroy(mmap: struct doca_mmap *): doca_error_t
```

14.4.1.1.4.4 DOCA Execution Model

The execution model is based on hardware processing on data and application threads. DOCA does not create an internal thread for processing data.

The workload is made up of tasks and events. Some tasks transform source data to destination data. The basic transformation is a DMA operation on the data which simply copies data from one memory location to another. Other operations allow users to receive packets from the network or involve calculating the SHA value of the source data and writing it to the destination.

For instance, a transform workload can be broken into three steps:

1. Read source data (`doca_buf` see memory subsystem).
2. Apply an operation on the read data (handled by a dedicated hardware accelerator).
3. Write the result of the operation to the destination (`doca_buf` see memory subsystem).

Each such operation is referred to as a task (`doca_task`).

Tasks describe operations that an application would like to submit to DOCA (hardware or BlueField). To do so, the application requires a means of communicating with the hardware/BlueField. This is where the `doca_pe` comes into play. The progress engine (PE) is a per-thread object used to queue tasks to offload to DOCA and eventually receive their completion status.

`doca_pe` introduces three main operations:

1. Submission of tasks.
2. Checking progress/status of submitted tasks.
3. Receiving a notification on task completion (in the form of a callback).

A workload can be split into many different tasks that can be executed on different threads; each thread represented by a different PE. Each task must be associated to some context, where the context defines the type of task to be done.

A context can be obtained from some libraries within the DOCA SDK. For example, to submit DMA tasks, a DMA context can be acquired from `doca_dma.h` , whereas SHA context can be obtained using `doca_sha.h` . Each such context may allow submission of several task types.

A task is considered asynchronous in that once an application submits a task, the DOCA execution engine (hardware or BlueField) would start processing it, and the application can continue to do some other processing until the hardware finishes. To keep track of which task has finished, there are two modes of operation: polling mode and event-driven mode.

Requirements and Considerations

- The task submission/execution flow/API is optimized for performance (latency)
- DOCA does not manage internal (operating system) threads. Rather, progress is managed by application resources (calling DOCA API in polling mode or waiting on DOCA notification in event-driven mode).
- The basic object for executing the task is a `doca_task` . Each task is allocated from a specific DOCA library context.
- `doca_pe` represents a logical thread of execution for the application and tasks submitted to the progress engine (PE)

  > ⚠ PE is not thread safe and it is expected that each PE is managed by a single application thread (to submit a task and manage the PE).

- Execution-related elements (e.g., `doca_pe` , `doca_ctx` , `doca_task`) are opaque and the application performs minimal initialization/configuration before using these elements
- A task submitted to PE can fail (even after the submission succeeds). In some cases, it is possible to recover from the error. In other cases, the only option is to reinitialize the relevant objects.

- PE does not guarantee order (i.e., tasks submitted in certain order might finish out-of-order). If the application requires order, it must impose it (e.g., submit a dependent task once the previous task is done).
- A PE can either work in polling mode or event-driven mode, but not in both at same time
- All DOCA contexts support polling mode (i.e., can be added to a PE that supports polling mode)

DOCA Context

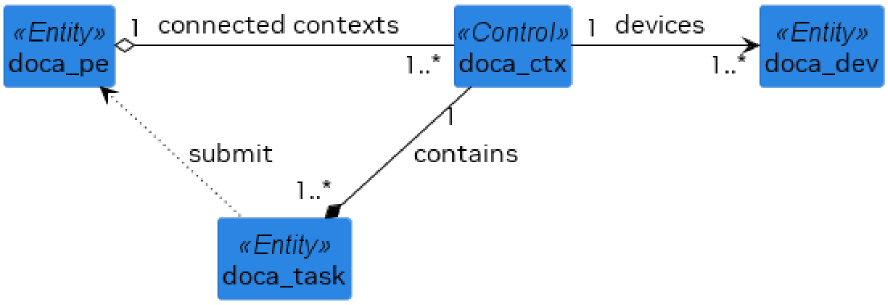
DOCA Context (`struct doca_ctx`) defines and provides (implements) task/event handling. A context is an instance of a specific DOCA library (i.e., when the library provides a DOCA Context, its functionality is defined by the list of tasks/events it can handle). When more than one type of task is supported by the context, it means that the supported task types have a certain degree of similarity to implement and utilize common functionality.

The following list defines the relationship between task contexts:

- Each context utilizes at least one DOCA Device functionality/accelerated processing capabilities
- For each task type there is one and only context type supporting it
- A context virtually contains an inventory per supported task type
- A context virtually defines all parameters of processing/execution per task type (e.g., size of inventory, device to accelerate processing)

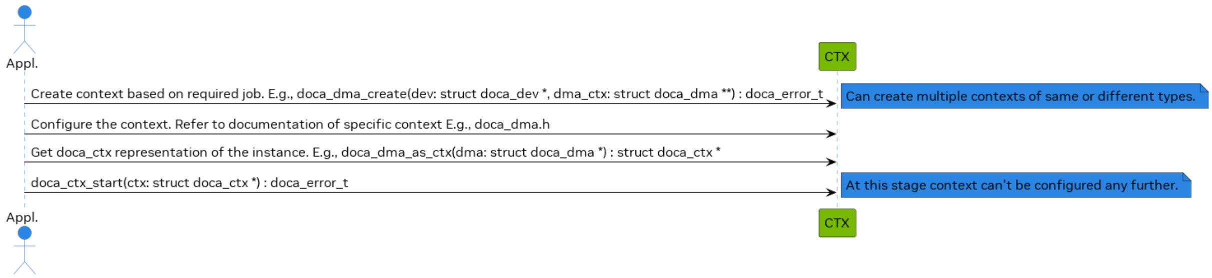
Each context needs an instance of progress engine (PE) as a runtime for its tasks (i.e., a context must be associated with a PE to execute tasks).

The following diagram shows the high-level (domain model) relations between various DOCA Core entities.



1. `doca_task` is associated to a relevant `doca_ctx` that executes the task (with the help of the relevant `doca_dev`).
2. `doca_task` , after it is initialized, is submitted to `doca_pe` for execution.
3. `doca_ctx` s are connected to the `doca_pe` . Once a `doca_task` is queued to `doca_pe` , it is executed by the `doca_ctx` that is associated with that task in this PE.

The following diagram describes the initialization sequence of a context:


Appl.
CTX
Create context based on required job. E.g., doca_dma_create(dev: struct doca_dev *, dma_ctx: struct doca_dma **) : doca_error_t
Can create multiple contexts of same or different types.
Configure the context. Refer to documentation of specific context E.g., doca_dma.h
Get doca_ctx representation of the instance. E.g., doca_dma_as_ctx(dma: struct doca_dma *) : struct doca_ctx *
doca_ctx_start(ctx: struct doca_ctx *) : doca_error_t
At this stage context can't be configured any further.
Appl.
CTX

After the context is started, it can be used to enable the submission of tasks to a PE based on the types of tasks that the context supports. See section "DOCA Progress Engine" for more information.

> Context is a thread-unsafe object which can be connected to a single PE only.

Configuration Phase

A DOCA context must be configured before attempting to start it using `doca_ctx_start()`. Some configurations are mandatory (e.g., providing `doca_dev`) while others are not.

- Configurations can be useful to allow certain tasks/events, to enable features which are disabled by default, and to optimize performance depending on a specific workload.
- Configurations are provided using setter functions. Refer to context documentation for a list of mandatory and optional configurations and their corresponding APIs.
- Configurations are provided after creating the context and before starting it. Once the context is started, it can no longer be configured unless it is stopped again.

Examples of common configurations:

- Providing a device – usually done as part of the create API
- Enabling tasks or registering to events – all tasks are disabled by default

Execution Phase

Once context configuration is complete, the context can be used to execute tasks. The context executes the tasks by offloading the workload to hardware, while software polls the tasks (i.e., waits) until they are complete.

In this phase, an application uses the context to allocate and submit asynchronous tasks, and then polls tasks (waits) until completion.

The application must build an event loop to poll the tasks (wait), utilizing one of the following modes:

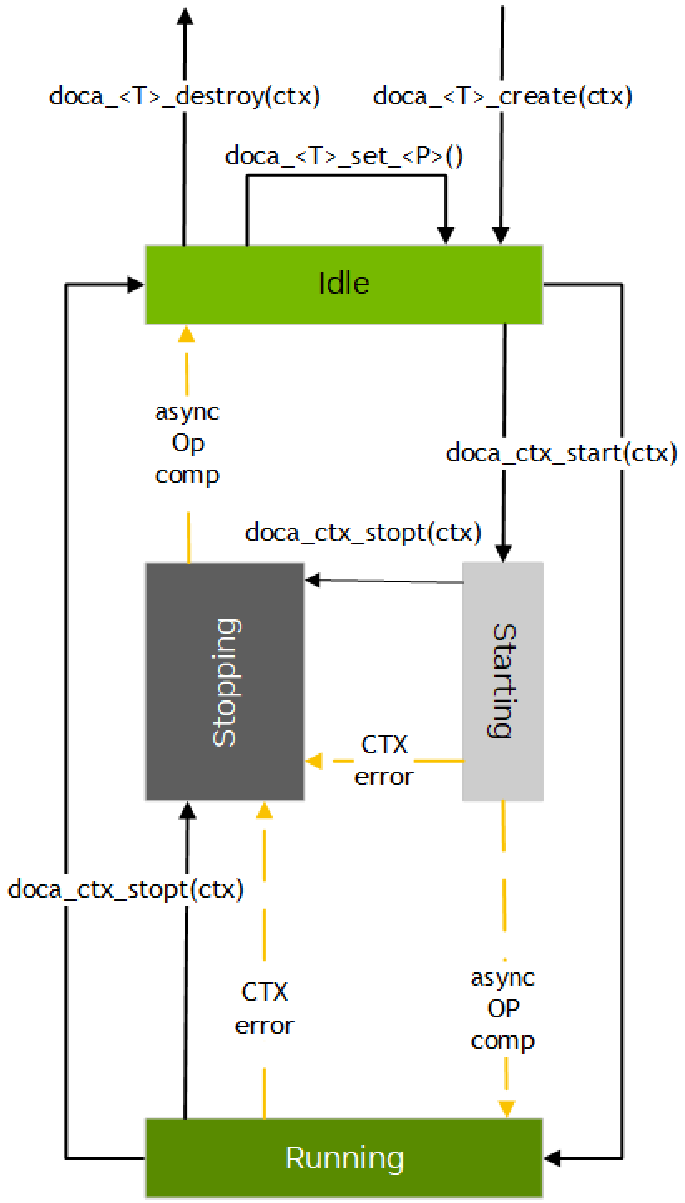
- Polling Mode
- Notification-driven Mode

In this phase, the context and all core objects perform zero allocations by utilizing memory pools. It is recommended that the application utilizes same approach for its own logic.

State Machine

State	Description
Idle	• 0 in-flight tasks • On init (right after `doca_<T>_create(ctx)`): All configuration APIs enabled • On reconf (on transition from stopping state): Some configuration APIs enabled
Starting	This state is mandatory for CTXs where transition to running state is conditioned by one or more async op completions/external events. For example, when a client connects to comm channel, it enters running state. Waiting for state change can be terminated by a voluntary (user) `doca_ctx_stop()` call or involuntary context state change due to internal error.
Running	• Task allocation/submission enabled (disabled in all other states) • All configuration APIs are disabled
Stopping	• Preparation before stopped state • Clean all in-flight tasks that may not complete in near future • Procedures relying on external entity actions should be terminated by CTX logic

The following diagram describes DOCA Context state transitions:



Internal Error

DOCA Context states can encounter internal errors at any time. If the state is starting or running, an internal error can cause an involuntary transition to stopping state.

For instance, an involuntary transition from running to stopping can happen when a task execution fails. This results in a completion with error for the failed task and all subsequent task completions.

After stopping, the state may become idle. However, `doca_ctx_start()` may fail if there is a configuration issue or if an error event prevented proper transition to starting or running state.

DOCA Task

A task is a unit of (functional/processing) workload offload-able to hardware. The majority of tasks utilize NVIDIA® BlueField® and NVIDIA® ConnectX® hardware to provide accelerated processing of the workload defined by the task. Tasks are asynchronous operations (e.g., tasks submitted for processing via non-blocking `doca_task_submit()` API).

Upon task completion, the preset completion callback is executed in context of `doca_pe_progress()` call. The completion callback is a basic/generic property of the task, similar to user data. Most tasks are IO operations executed/accelerated by NVIDIA device hardware.

Task Properties

Task properties share generic properties which are common to all task types and type-specific properties. Since task structure is opaque (i.e., its content not exposed to the user), the access to task properties provided by set/get APIs.

The following are generic task properties:

- Setting completion callback – it has separate callbacks for successful completion and completion with failure.
- Getting/setting user data – used in completion callback as some structure associated with specific task object.
- Getting task status – intended to retrieve error code on completion with failure.

For each task there is only one owner: a context object. There is a `doca_task_get_ctx()` API to get generic context object.

The following are generic task APIs:

- Allocating and freeing from CTX (internal/virtual) inventory
- Configuring via setters (or init API)
- Submit-able (i.e., implements `doca_task_submit(task)`)

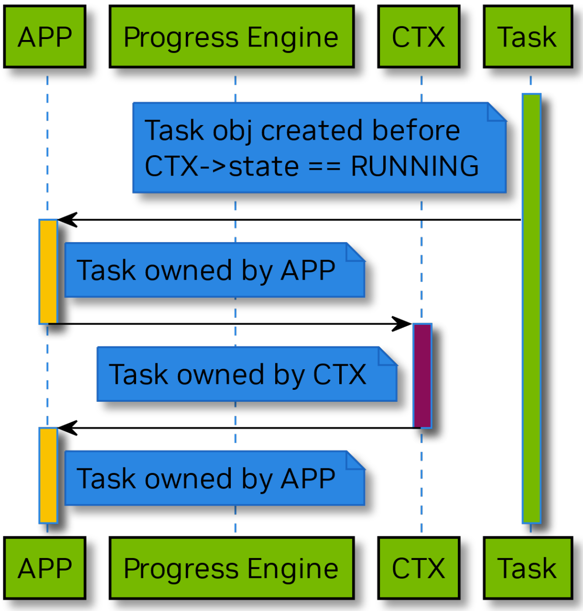
Upon completion, there is a set of getters to access the results of the task execution.

Task Lifecycle

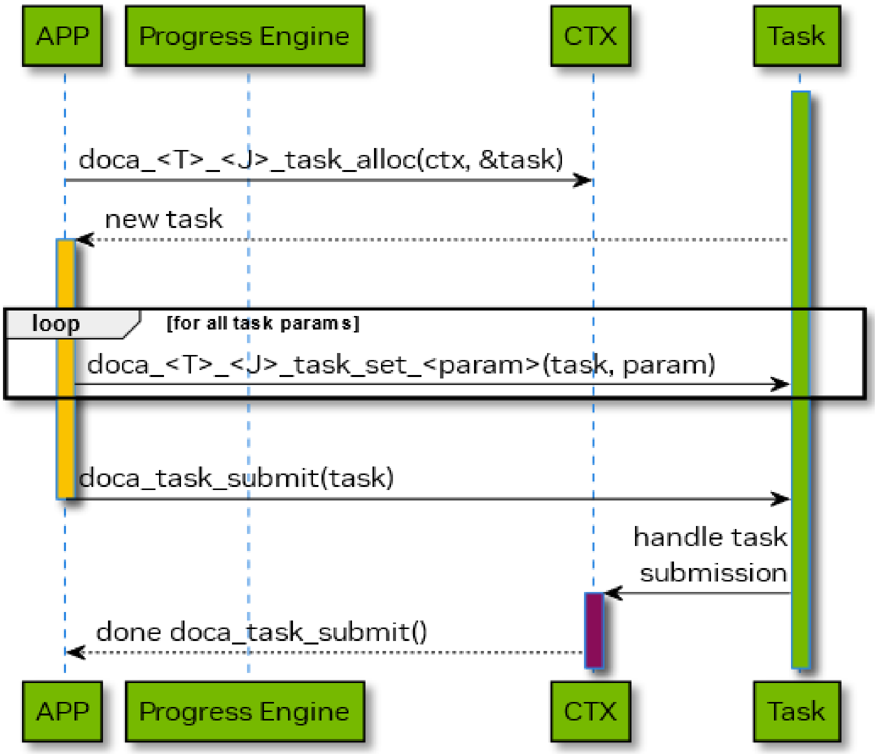
This section describes the lifecycle of DOCA Task. Each DOCA Task object lifecycle:

- starts on the event of entering *Running* state by the DOCA Context owning the task i.e., once *Running* state entered application can obtain the task from CTX by calling `doca_<CTX name>_task_<Task name>_alloc_init(ctx, ... &task)`.
- ends on the event of entering *Stopped* state by the DOCA Context owning the task i.e., application can no longer allocate tasks once the related DOCA Context left the *Running* state.

From application perspective DOCA Context provides a virtual task inventory The diagram below shows the how ownership if the DOCA Task passed from DOCA Context virtual inventory to application and than from application back to CTX, pay attention to the colors used in activation bars for application (APP) participant & DOCA Context (CTX) participant and DOCA Context Task virtual inventory (Task).



The diagram below shows the lifecycle of DOCA Task staring from its allocation to its submission.

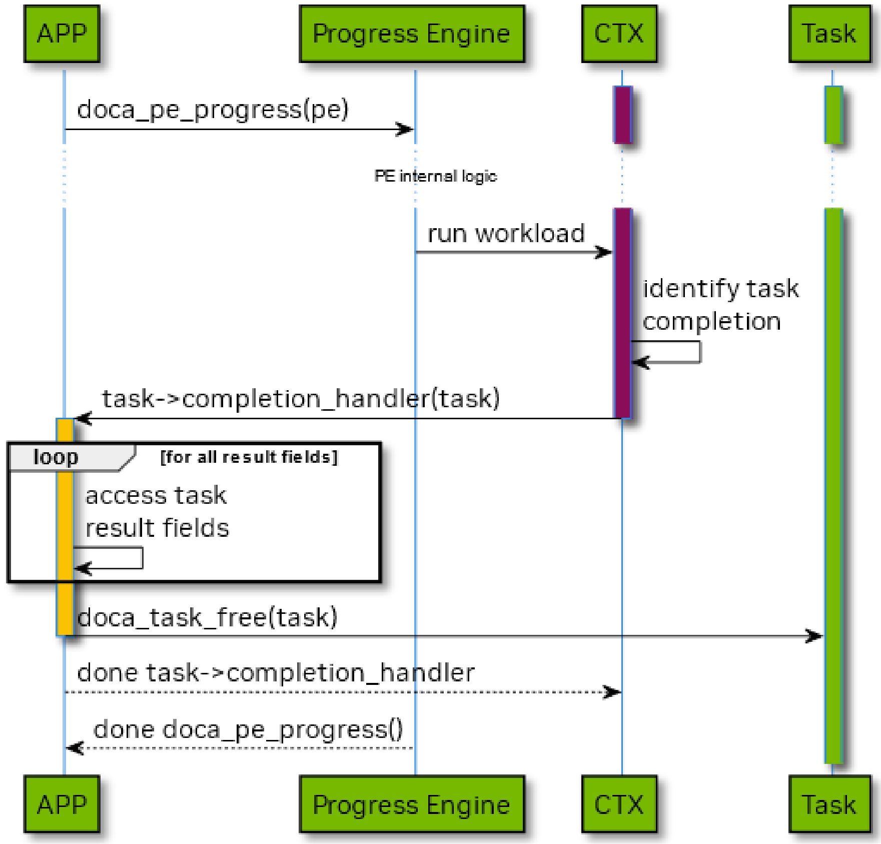


The diagram above displays following ownership transitions during DOCA Task object lifecycle:

- starting from allocation task ownership passed from context to application
- application may modify task attributes via API templated as `doca_<CTX name>_task_<Task name>_set_<Parameter name>(task, param)` ; on return from the task modification call the ownership of the task object returns to application.

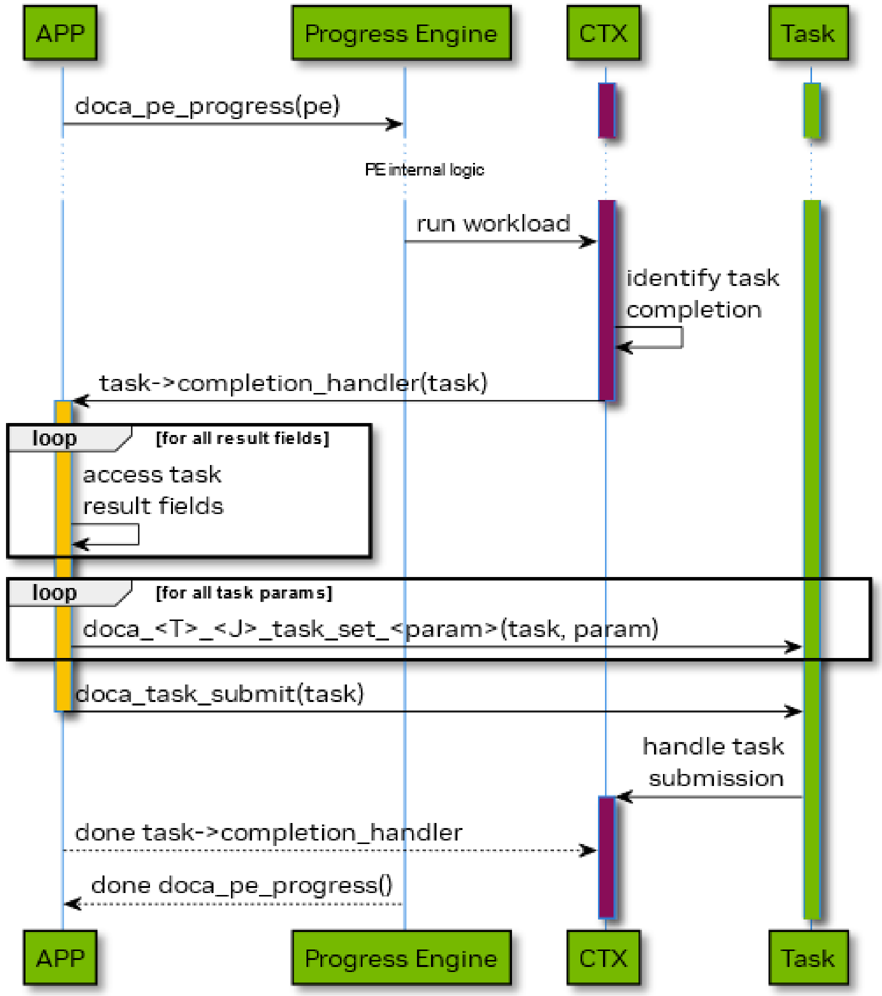
- submit the task for processing in the PE, once all required modifications/settings of the task object completed. On task submission the ownership of the object passed to the related context.

The next two diagrams below shows the lifecycle of DOCA Task on its completion.



The diagram above displays following ownership transitions during *DOCA Task* object lifecycle:

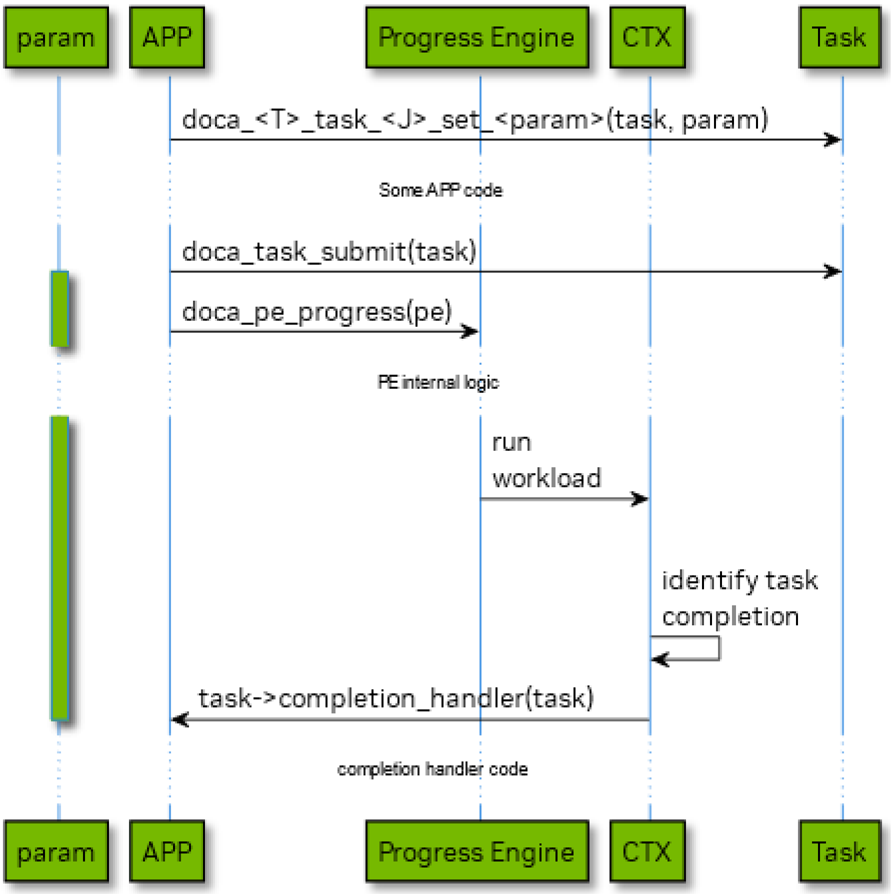
- on *DOCA Task* completion the appropriate handler provided by application invoked; on handler invocation the *DOCA Task* ownership passed to application.
- after *DOCA Task* completion application may access task attributes & result fields utilizing appropriate APIs; application remains owner of the task object.
- application may call `doca_task_free()` when task is no longer needed; on return from the call task ownership passed to *DOCA Context* while task became uninitialized & pre-allocated till the context enters Idle state.



The diagram above displays similar to the previous diagram ownership transitions during *DOCA Task* object lifecycle with the only difference that instead of `doca_task_free(task)` `doca_task_submit(task)` was called:

- *DOCA Task* result (related attributes) can be accessed right after enter successful task completion callback, similar to the previous case
- lifecycle of the *DOCA Task* results ends on exit from the task completion callback scope.
- On `doca_task_free()` or `doca_<CTX name>_task_<Task name>_set_<Parameter name>(task, param)` call all task results should be considered invalidated regardless of scope.

The diagram below shows the lifecycle of *DOCA Task* set-able parameters while API to set such a parameter templated as `doca_<CTX name>_task_<Task name>_set_<Parameter name>(task, param)` .



Green activation of param participant describes the time slice when all *DOCA Task* parameters owned by DOCA library. On `doca_task_submit()` call the ownership on all task arguments passed from application to the DOCA Context the related Task object belongs to. The ownership of task arguments passed back to application on task completion. The application should not modify and/or destroy/free Task argument related objects if it doesn't own the argument.

DOCA Progress Engine

The progress engine (PE) enables asynchronous processing and handling of multiple tasks and events of different types in a single-threaded execution environment. It is an event loop for all context-based DOCA libraries, with I/O completion being the most common event type.

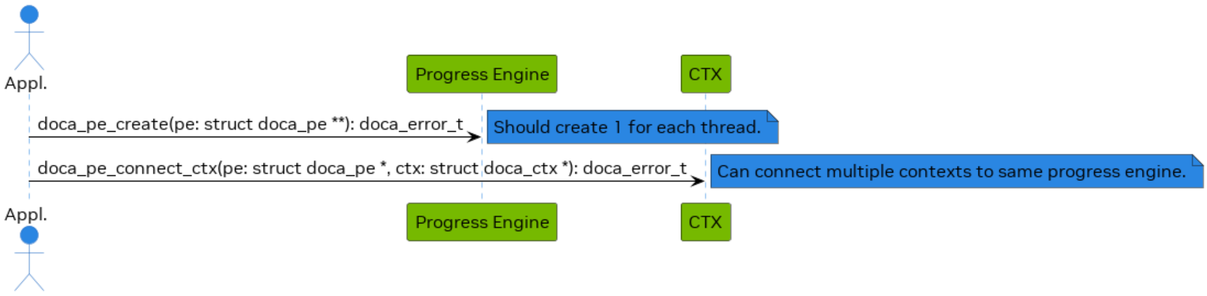
PE is designed to be thread unsafe (i.e., it can only be used in one thread at a time) but a single OS thread can use multiple PEs. The user can assign different priorities to different contexts by adding them to different PEs and adjusting the polling frequency for each PE accordingly. Another way to view the PE is as a queue of workload units that are scheduled for execution.

There are no explicit APIs to add and/or schedule a workload to/on a PE but a workload can be added by:

- Adding a DOCA context to PE
- Registering a DOCA event to probe (by the PE) and executing the associated handler if the probe is positive

PE is responsible for scheduling workloads (i.e., picking the next workload to execute). The order of workload execution is independent of task submission order, event registration order, or order of context associations with a given PE object. Multiple task completion callbacks may be executed in an order different from the order of related task submissions.

The following diagram describes the initialization flow of the PE:


Appl.
Progress Engine
CTX
doca_pe_create(pe: struct doca_pe **): doca_error_t
Should create 1 for each thread.
doca_pe_connect_ctx(pe: struct doca_pe *, ctx: struct doca_ctx *): doca_error_t
Can connect multiple contexts to same progress engine.
Appl.
Progress Engine
CTX

After a PE is created and connected to contexts, it can start progressing tasks which are submitted to the contexts. Refer to context documentation to find details such as what tasks can be submitted using the context.

Note that the PE can be connected to multiple contexts. Such contexts can be of the same type or of different types. This allows submitting different task types to the same PE and waiting for any of them to finish from the same place/thread.

After initializing the PE, an application can define an event loop using one of these modes:

- Polling mode
- Blocking (notification-driven) mode

PE as Event Loop Mode of Operation

All completion handlers for both tasks and events are executed in the context of `doca_pe_progress()` . `doca_pe_progress()` loops for every workload (i.e., for each workload unit) scheduled for execution:

Run the selected workload unit. For the following cases:

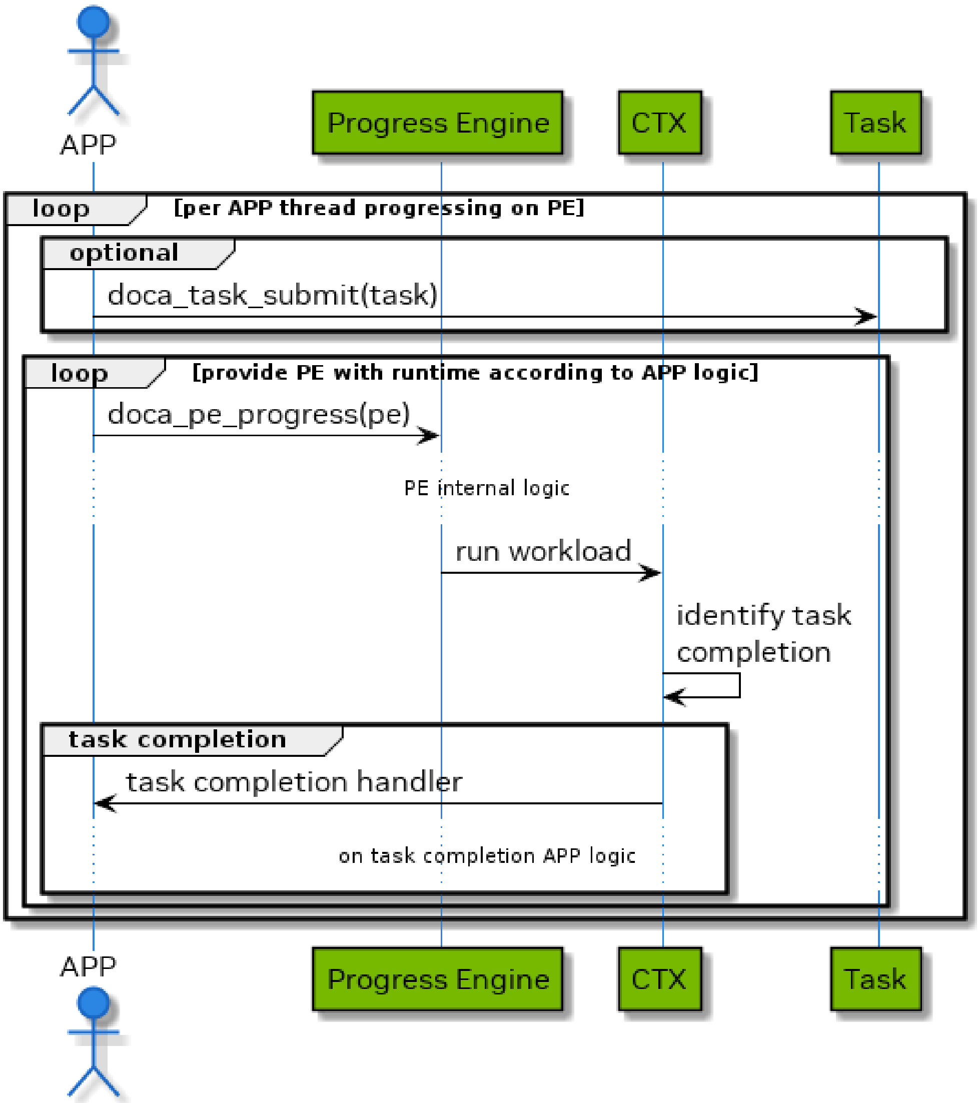
- Task completion, execute associated handler and break the loop and return status `made some progress`
- Positive probe of event, execute associated handler and break the loop and return status `made some progress`
- Considerable progress is made to contribute to future task completion or positive event probe, break the loop and return status `made some progress`

Otherwise, reach the end of the loop and return status `no progress` .

Polling Mode

In this mode, the application submits a task and then does busy-wait to find out when the task has completed.

The following diagram demonstrates this sequence:



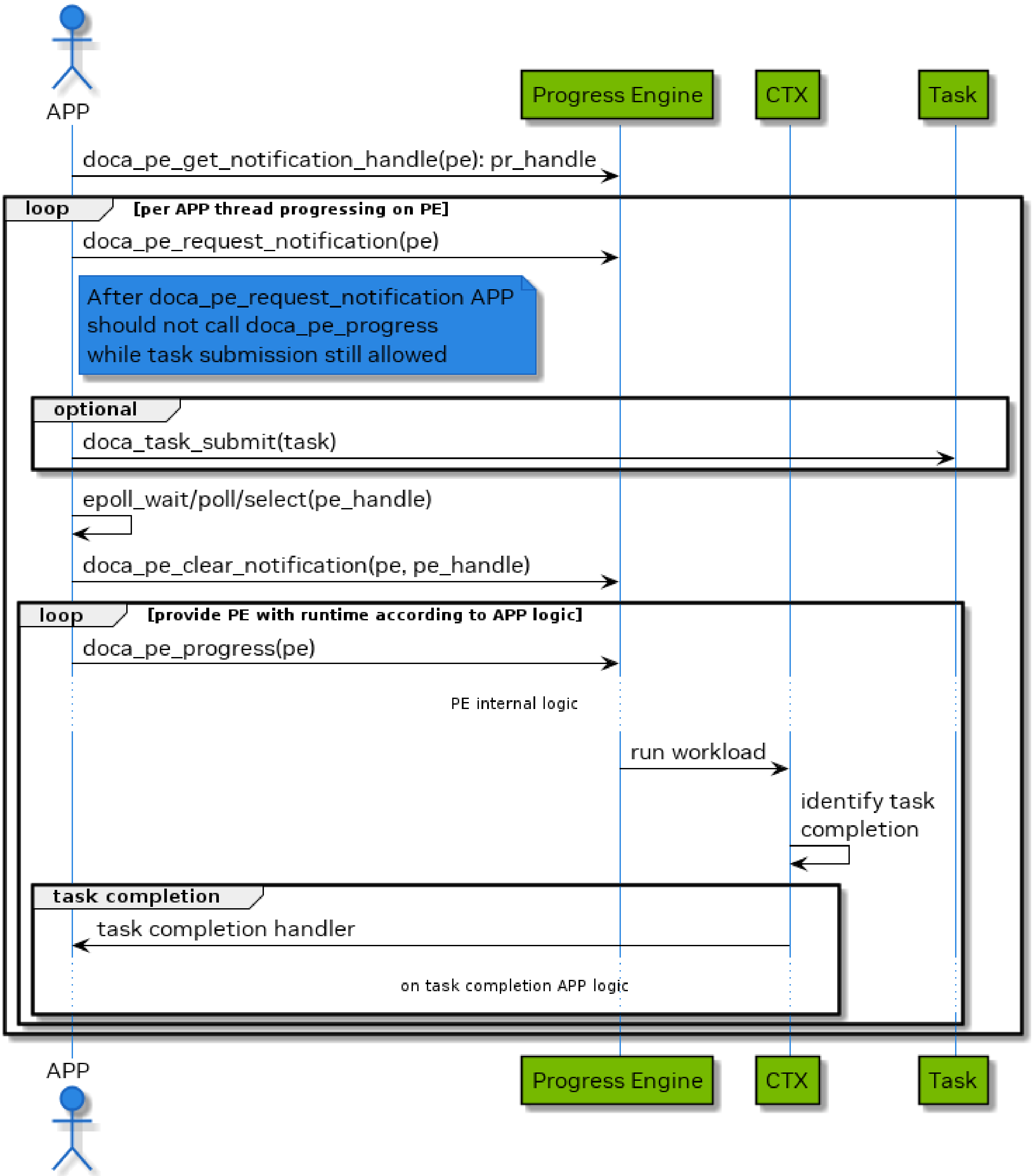
1. The application submits all tasks (one or more) and tracks the number of task completions to know if all tasks are done.
2. The application waits for a task to complete by consecutive polls on `doca_pe_progress()`.
   a. If `doca_pe_progress()` returns 1, it means progress is being made (i.e., some task completed or some event handled).
   b. Each time a task is completed or an event is handled, its preset completion or event handling callback is executed accordingly.
   c. If a task is completed with an error, preset task completion with error callback is executed (see section "Error Handling").
3. The application may add code to completion callbacks or event handlers for tracking the amount of completed and pending workloads.

⚠ In this mode, the application is always using the CPU even when it is doing nothing (busy-wait).

Blocking Mode - Notification Driven

In this mode, the application submits a task and then waits for a notification to be received before querying the status.

The following diagram demonstrates this sequence:


APP
Progress Engine
CTX
Task
doca_pe_get_notification_handle(pe): pr_handle
loop
[per APP thread progressing on PE]
doca_pe_request_notification(pe)
After doca_pe_request_notification APP should not call doca_pe_progress while task submission still allowed
optional
doca_task_submit(task)
epoll_wait/poll/select(pe_handle)
doca_pe_clear_notification(pe, pe_handle)
loop
[provide PE with runtime according to APP logic]
doca_pe_progress(pe)
PE internal logic
run workload
identify task completion
task completion
task completion handler
on task completion APP logic
APP
Progress Engine
CTX
Task

1. The application gets a notification handle from the `doca_pe` representing a Linux file descriptor which is used to signal the application that some work has finished.
2. The application then arms the PE with `doca_pe_request_notification()` .

> ⚠ This must be done every time an application is interested in receiving a notification from the PE.

> ⚠ After `doca_pe_request_notification()` , no calls to `doca_pe_progress()` are allowed. In other words, `doca_pe_request_notification()` should be followed by `doca_pe_clear_notification` before any calls to `doca_pe_progress()` .

3. The application submits a task.
4. The application waits (e.g., Linux epoll/select) for a signal to be received on the `pe-fd` .
5. The application clears the notifications received, notifying the PE that a signal has been received and allowing it to perform notification handling.
6. The application attempts to handle received notifications via (multiple) calls to `doca_pe_progress()` .

   > ⚠ There is no guarantee that the call to `doca_pe_progress()` would execute any task completion/event handler, but the PE can continue the operation.

7. The application handles its internal state changes caused by task completions and event handlers called in the previous step.
8. Repeat steps 2-7 until all tasks are completed and all expected events are handled.

Progress Engine versus Epoll

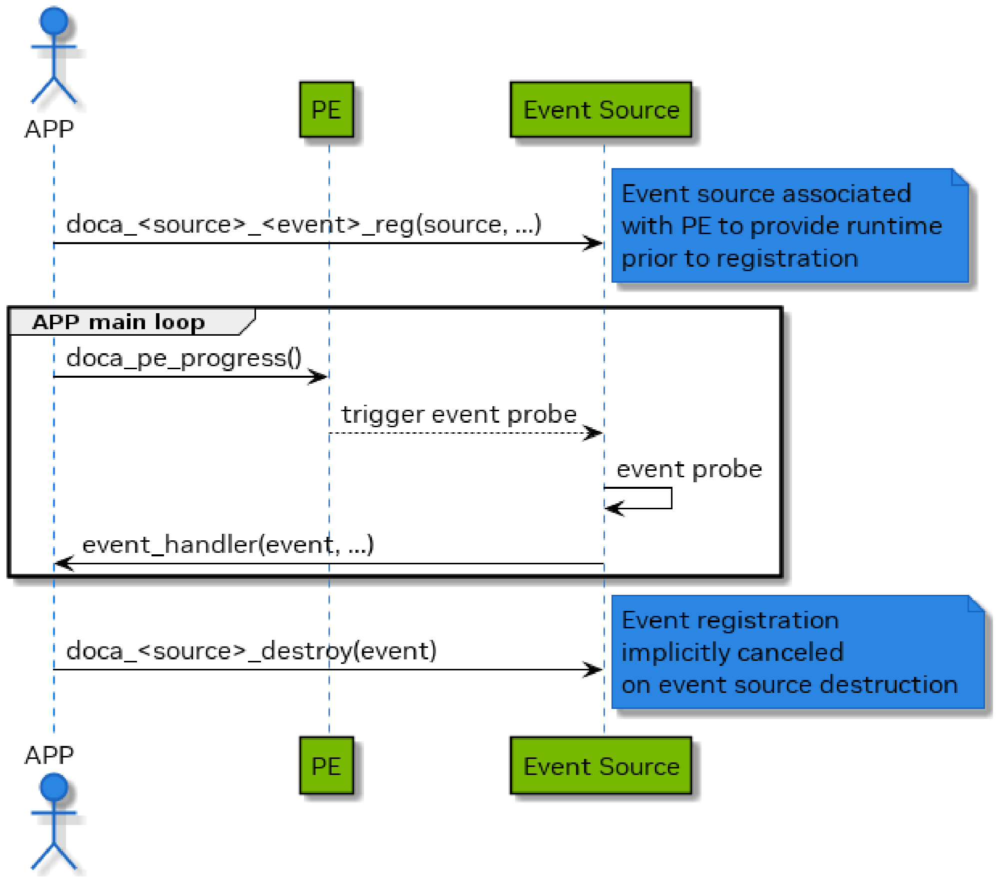
The epoll mechanism in Linux and the DOCA PE handles high concurrency in event-driven architectures. Epoll, like a post office, tracks "mailboxes" (file descriptors) and notifies the "postman" (the `epoll_wait` function) when a "letter" (event) arrives. DOCA PE, like a restaurant, uses a single "waiter" to handle "orders" (workload units) from "customers" (DOCA contexts). When an order is ready, it is placed on a "tray" (task completion handler/event handler execution) and delivered in the order received. Both systems efficiently manage resources while waiting for events or tasks to complete.

DOCA Event

An event is a type of occurrence that can be detected or verified by the DOCA software, which can then trigger a handler (a callback function) to perform an action. Events are associated with a specific source object, which is the entity whose state or attribute change defines the event's occurrence. For example, a context state change event is caused by the change of state of a context object.

To register an event, the user must call the `doca_<event_type>_reg(pe, ...)` function, passing a pointer to the user handler function and an opaque argument for the handler. The user must also associate the event handler with a PE, which is responsible for running the workloads that involve event detection and handler execution.

Once an event is registered, it is periodically checked by the `doca_pe_progress()` function, which runs in the same execution context as the PE to which the event is bound. If the event condition is met, the handler function is invoked. Events are not thread-safe objects and should only be accessed by the PE to which they are bound.

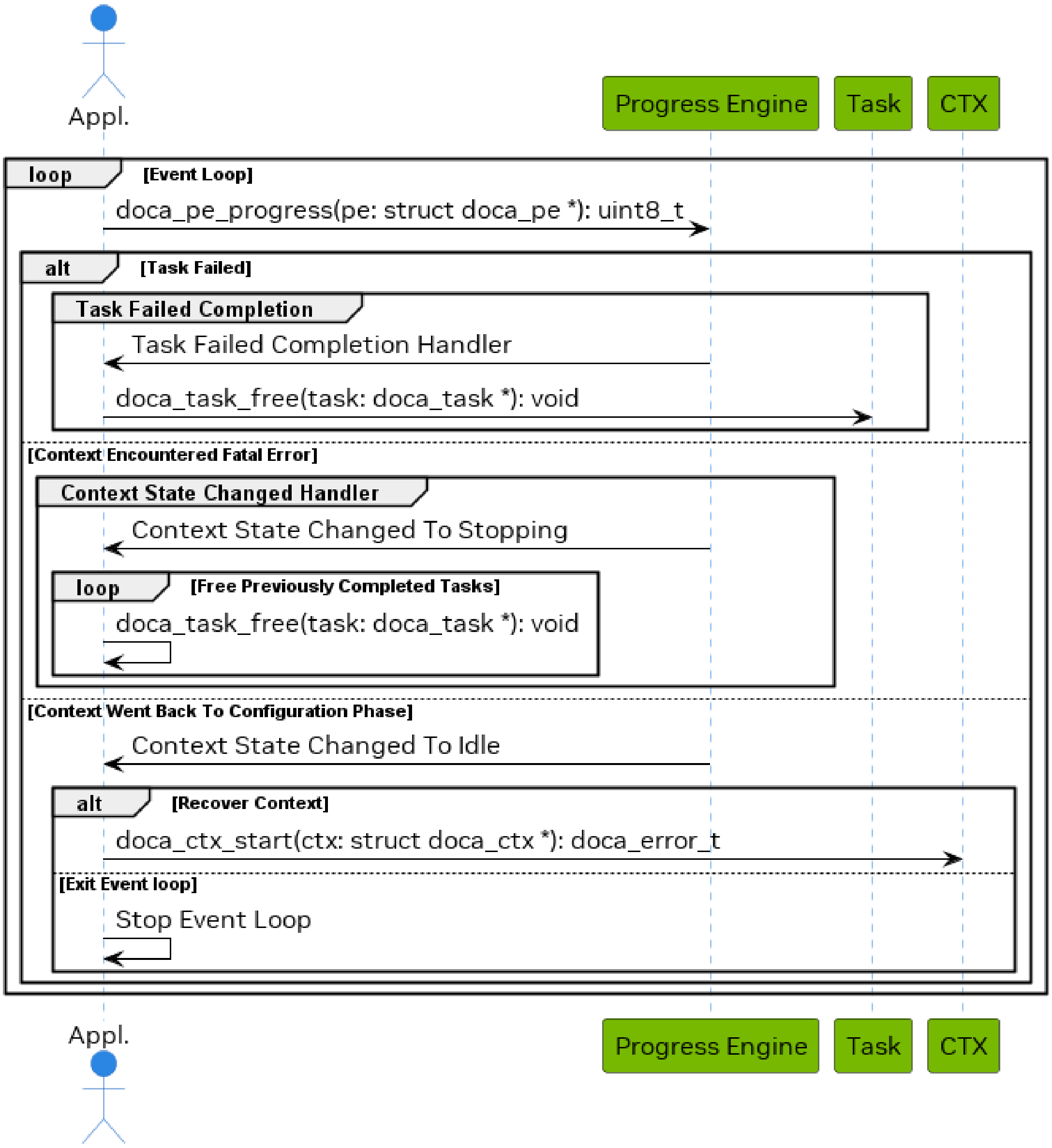

APP
PE
Event Source
doca_<source>_<event>_reg(source, ...)
Event source associated with PE to provide runtime prior to registration
APP main loop
doca_pe_progress()
trigger event probe
event probe
event_handler(event, ...)
doca_<source>_destroy(event)
Event registration implicitly canceled on event source destruction
APP
PE
Event Source

Error Handling

After a task is submitted successfully, consequent calls to `doca_pe_progress()` may fail (i.e., task failure completion callback is called).

Once a task fails, the context may transition to stopping state, in this state, the application has to progress all in-flight tasks until completion before destroying or restarting the context.

The following diagram shows how an application may handle an error from `doca_pe_progress()` :


Appl.
Progress Engine
Task
CTX
loop
[Event Loop]
doca_pe_progress(pe: struct doca_pe *): uint8_t
alt
[Task Failed]
Task Failed Completion
Task Failed Completion Handler
doca_task_free(task: doca_task *): void
[Context Encountered Fatal Error]
Context State Changed Handler
Context State Changed To Stopping
loop
[Free Previously Completed Tasks]
doca_task_free(task: doca_task *): void
[Context Went Back To Configuration Phase]
Context State Changed To Idle
alt
[Recover Context]
doca_ctx_start(ctx: struct doca_ctx *): doca_error_t
[Exit Event loop]
Stop Event Loop
Appl.
Progress Engine
Task
CTX

1. Application runs event loop.
2. Any of the following may happen:
   - [Optional] Task fails, and the task failed completion handler is called
     - This may be caused by bad task parameters or another fatal error
     - Handler releases the task and all associated resources
   - [Optional] Context transitions to stopping state, and the context state changed handler is called
     - This may be caused by failure of a task or another fatal error
     - In this state, all in-flight tasks are guaranteed to fail
     - Handler releases tasks that are not in-flight if such tasks exist
   - [Optional] Context transitions to idle state, and the context state changed handler is called
     - This may happen due to encountering an error and the context does not have any resources that must be freed by the application

- In this case, the application may decide to recover the context by calling start again or it may decide to destroy the context and possibly exit the application

Task and Event Batching

DOCA Batching is an approach for grouping multiple tasks or events of the same type and handling them as a single unit. DOCA offers two options of achieving this as described in the following subsections.

Batch Task/Event

In this batching option, a library (e.g., `doca_eth_txq`) offers a task that represents a batched operation (e.g., sending multiple packets), the task is considered a batch task and has a task type that is separate from the non-batched operation (e.g., sending a single packet).

To submit the batch task, the user is required to build the batch and then submit it at once, similar to submitting a regular task.

The completion of the batch is based on the completion of all items in the batch and is handled as the completion of a single unit. This allows for multiple DOCA Task initialization/submission and multiple DOCA Task/Event completion handling in a single API call (see DOCA Ethernet for example).

Iterative Batch

In this batching option, it is possible to utilize existing task types to build a batch operation, where each task within the batch is submitted individually and each task receives its own completion.

Furthermore, the batch is built iteratively, where the user is not required to have information for the entire batch ahead of time.

To utilize this option, the user can submit each task in the batch using an extended submit API `doca_task_submit_ex` while providing additional submit flags.

The extended submit API is similar to a regular submit API (`doca_task_submit`) but with the ability to receive submit flags. These flags are used as hints to the library that executes the tasks. They can have implications on the current task but may also have implications on previously submitted flags, as described in the following table:

Submit Flag [1]	Effect on Current Task		Effect on Previous Tasks [2]		Default Behavior of doca_task_submit	Comments
	Flag Provided	Flag not Provided	Flag Provided	Flag not Provided		
`DOCA_TASK_SUBMIT_FLAG_FLUSH`	Task is submitted for hardware execution immediately, and is considered "flushed".	Task may not be submitted for hardware execution, and is considered "unflushed".	All previous tasks which are considered unflushed become flushed.	None	Flag is provided	As long as the task is unflushed, it never completes. The flag allows batching such that multiple tasks are flushed at once, instead of individually.

Submit Flag [1]	Effect on Current Task		Effect on Previous Tasks [2]		Default Behavior of doca_task_submit	Comments
	Flag Provided	Flag not Provided	Flag Provided	Flag not Provided		
`DOCA_TASK_SUBMIT_FLAG_OPTIMIZE_REPORTS`	The user does not receive task completion after hardware has completed execution of the task, and the completion is considered "unreported".	The user receives task completion after hardware has completed execution of the task, and the completion is considered "reported".	None	Once the hardware completes execution of this task, all previous [3] unreported completions become reported.	Flag is not provided	As long as the task is unreported, the user would never know that it has been completed. The completion of a task is reported through a completion callback using the progress engine. The library does not guarantee any order of execution/completion of tasks. The flag allows batching, such that multiple task completions are reported using a single hardware completion, instead of receiving a completion for every task.

[1] Note that these flags are hints which may allow internal optimizations. However, on a task by task basis, the library may decide to ignore user flags and revert to default submit behavior. ↩

[2] "Previous tasks" only refers to tasks submitted to the same library instance (doca_ctx). The flags do not allow optimizations across different library instances. ↩

[3] "previous" refers to tasks that have been submitted before this one. ↩

DOCA Graph Execution

DOCA Graph facilitates running a set of actions (tasks, user callbacks, graphs) in a specific order and dependencies. DOCA Graph runs on a DOCA progress engine.

DOCA Graph creates graph instances that are submitted to the progress engine (`doca_graph_instance_submit`).

Nodes

DOCA Graph is comprised of context, user, and sub-graph nodes. Each of these types can be in any of the following positions in the network:

- Root nodes - a root node does not have a parent. The graph can have one or more root nodes. All roots begin running when the graph instance is submitted.
- Edge nodes - an edge node is a node that does not have child nodes connected to it. The graph instance is completed when all edge nodes are completed.
- Intermediate node - a node connected to parent and child nodes

Context Node

A context node runs a specific DOCA task and uses a specific DOCA context (`doca_ctx`). The context must be connected to the progress engine before the graph is started.

The task lifespan must be longer or equal to the life span of the graph instance.

User Node

A user node runs a user callback to facilitate performing actions during the run time of the graph instance (e.g., adjust next node task data, compare results).

Sub-graph Node

A sub-graph node runs an instance of another graph.

Using DOCA Graph

1. Create the graph using `doca_graph_create` .
2. Create the graph nodes (e.g., `doca_graph_node_create_from_ctx`).
3. Define dependencies using `doca_graph_add_dependency` .

   > ⚠ DOCA graph does not support circle dependencies (e.g., A => B => A).

4. Start the graph using `doca_graph_start` .
5. Create the graph instance using `doca_graph_instance_create` .
6. Set the nodes data (e.g., `doca_graph_instance_set_ctx_node_data`).
7. Submit the graph instance to the pe using `doca_graph_instance_submit` .
8. Call `doca_pe_progress` until the graph callback is invoked.
   - Progress engine can run graph instances and standalone tasks simultaneously.

DOCA Graph Limitations

- DOCA Graph does not support circle dependencies
- DOCA Graph must contain at least one context node. A graph containing a sub-graph with at least one context node is a valid configuration.

DOCA Graph Sample

The graph sample is based on the DOCA DMA library. The sample copies 2 buffers using DMA.

The graph ends with a user callback node that compares source and destinations.

Running DOCA Graph Sample

1. Refer to the following documents:
    - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
    - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample:

    ```
    cd /opt/mellanox/doca/samples/doca_common/graph/
    meson build
    ninja -C build
    ```
3. Sample (e.g., `doca_graph`) usage:

    ```
    ./build/doca_graph
    ```
    No parameters required.

Alternative Data Path

DOCA Progress Engine utilizes the CPU to offload data path operations to hardware. However, some libraries support utilization of DPA and/or GPU.

Considerations:

- Not all contexts support alternative datapath
- Configuration phase is always done on CPU
- Datapath operations are always offloaded to hardware. The unit that offloads the operation itself can be either CPU/DPA/GPU.
- The default mode of operation is CPU
- Each mode of operation introduces a different set of APIs to be used in execution path. The used APIs are mutually exclusive for specific context instance.

DPA

Users must first refer to the programming guide of the relevant context (e.g., DOCA RDMA) to check if datapath on DPA is supported. Additionally, the guide provides what operations can be used.

To set the datapath mode to DPA, acquire a DOCA DPA instance, then use the `doca_ctx_set_datapath_on_dpa()` API.

After the context has been started with this mode, it becomes possible to get a DPA handle, using an API defined by the relevant context (e.g., `doca_rdma_get_dpa_handle()`). This handle can then be used to access DPA data path APIs within DPA code.

GPU

Users must first refer to the programming guide of the relevant context (E.g., DOCA Ethernet) to check if datapath on GPU is supported. Additionally, the guide provides what operations can be used.

To set the data path mode to GPU, acquire a DOCA GPU instance, then use the `doca_ctx_set_datapath_on_gpu()` API.

After the context has been started with this mode, it becomes possible to get a GPU handle, using an API defined by the relevant context (e.g., `doca_eth_rxq_get_gpu_handle()`). This handle can then be used to access GPU data path APIs within GPU code.

14.4.1.1.4.5 Object Life Cycle

Most DOCA Core objects share the same handling model in which:

1. The object is allocated by DOCA so it is opaque for the application (e.g., `doca_buf_inventory_create` , `doca_mmap_create`).
2. The application initializes the object and sets the desired properties (e.g., `doca_mmap_set_memrange`).
3. The object is started, and no configuration or attribute change is allowed (e.g., `doca_buf_inventory_start` , `doca_mmap_start`).
4. The object is used.
5. The object is stopped and deleted (e.g., `doca_buf_inventory_stop` → `doca_buf_inventory_destroy` , `doca_mmap_stop` → `doca_mmap_destroy`).

The following procedure describes the mmap export mechanism between two machines (remote machines or host-BlueField):

1. Memory is allocated on Machine1.
2. Mmap is created and is provided memory from step 1.
3. Mmap is exported to the Machine2 pinning the memory.
4. On the Machine2, an imported mmap is created and holds a reference to actual memory residing on Machine1.
5. Imported mmap can be used by Machine2 to allocate buffers.
6. Imported mmap is destroyed.
7. Exported mmap is destroyed.
8. Original memory is destroyed.

14.4.1.1.4.6 RDMA Bridge

The DOCA Core library provides building blocks for applications to use while abstracting many details relying on the RDMA driver. While this takes away complexity, it adds flexibility especially for applications already based on rdma-core. The RDMA bridge allows interoperability between DOCA SDK and rdma-core such that existing applications can convert DOCA-based objects to rdma-core-based objects.

Requirements and Considerations

- This library enables applications already using rdma-core to port their existing application or extend it using DOCA SDK.
- Bridge allows converting DOCA objects to equivalent rdma-core objects.

DOCA Core Objects to RDMA Core Objects Mapping

The RDMA bridge allows translating a DOCA Core object to a matching RDMA Core object. The following table shows how the one object maps to the other.

RDMA Core Object	DOCA Equivalent	RDMA Object to DOCA Object	DOCA Object to RDMA Object
`ibv_pd`	`doca_dev`	`doca_rdma_bridge_open_dev_from_pd`	`doca_rdma_bridge_get_dev_pd`
`ibv_mr`	`doca_buf`	`-`	`doca_rdma_bridge_get_buf_mkey`

14.4.1.1.5 DOCA Core Samples

ⓘ All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

14.4.1.1.5.1 Progress Engine Samples

All progress engine (PE) samples use DOCA DMA because of its simplicity. PE samples should be used to understand the PE not DOCA DMA.

pe_common

`pe_common.c` and `pe_common.h` contain code that is used in most or all PE samples.

Users can find core code (e.g., create MMAP) and common code that uses PE (e.g., `poll_for_completion`).

Struct `pe_sample_state_base` (defined in `pe_common.h`) is the base state for all PE samples, containing common members that are used by most or all PE samples.

pe_polling

The polling sample is the most basic sample for using PE. Start with this sample to learn how to use DOCA PE.

ⓘ You can diff between `pe_polling_sample.c` and any other `pe_x_sample.c` to see the unique features that the other sample demonstrates.

The sample demonstrates the following functions:

- How to create a PE
- How to connect a context to the PE
- How to allocate tasks
- How to submit tasks
- How to run the PE
- How to cleanup (e.g., destroy context, destroy PE)

  ⚠ Pay attention to the order of destruction (e.g., all contexts must be destroyed before the PE).

The sample performs the following:

1. Uses one DMA context.
2. Allocates and submits 16 DMA tasks.

   > Task completion callback checks that the copied content is valid.

3. Polls until all tasks are completed.

pe_async_stop

A context can be stopped while it still processes tasks. This stop is asynchronous because the context must complete/abort all tasks.

The sample demonstrates the following functions:

- How to asynchronously stop a context
- How to implement a context state changed callback (with regards to context moving from stopping to idle)
- How to implement task error callback (check if this is a real error or if the task is flushed)

The sample performs the following:

1. Submits 16 tasks and stops the context after half of the tasks are completed.
2. Polls until all tasks are complete (half are completed successfully, half are flushed).

The difference between `pe_polling_sample.c` and `pe_async_stop_sample.c` is to learn how to use PE APIs for event-driven mode.

pe_event

Event-driven mode reduces CPU utilization (wait for event until a task is complete) but may increase latency or reduce performance.

The sample demonstrates the following functions:

- How to run the PE in event-driven mode

The sample performs the following:

1. Runs 16 DMA tasks.
2. Waits for event.

The difference between `pe_polling_sample.c` and `pe_event_sample.c` is to learn how to use PE APIs for event-driven mode.

pe_multi_context

A PE can host more than one instance of a specific context. This facilitates running a single PE with multiple BlueField devices.

The sample demonstrates the following functions:

- How to run a single PE with multiple instances of a specific context

The sample performs the following:

1. Connects 4 instances of DOCA DMA context to the PE.

2. Allocates and submits 4 tasks to every context instance.
3. Polls until all tasks are complete.

The difference between `pe_polling_sample.c` and `pe_multi_context_sample.c` is to learn how to use PE with multiple instances of a context.

pe_reactive

PE and contexts can be maintained in callbacks (task completion and state changed).

The sample demonstrates the following functions:

- How to maintain the context and PE in the callbacks instead of the program's main function

The user must make sure to:

- Review the task completion callback and the state changed callbacks
- Review the difference between `poll_to_completion` and the polling loop in main

The sample performs the following:

1. Runs 16 DMA tasks.
2. Stops the DMA context in the completion callback after all tasks are complete.

The difference between `pe_polling_sample.c` and `pe_reactive_sample.c` is to learn how to use PE in reactive model.

pe_single_task_cb

A DOCA task can invoke a success or error callback. Both callbacks share the same structure (same input parameters).

DOCA recommends using 2 callbacks:

- Success callback – does not need to check the task status, thereby improving performance
- Error callback – may need to run a different flow than success callback

The sample demonstrates the following functions:

- How to use a single callback instead of two callbacks

The sample performs the following:

1. Runs 16 DMA tasks.
2. Handles completion with a single callback.

The difference between `pe_polling_sample.c` and `pe_single_task_comp_cb_sample.c` is to learn how to use PE with a single completion callback.

pe_task_error

Task execution may fail causing the associated context (e.g., DMA) to move to stopping state due to this fatal error.

The sample demonstrates the following functions:

- How to mitigate a task error during runtime

The user must make sure to:

- Review the state changed callback and the error callback to see how the sample mitigates context error

The sample performs the following:

1. Submits 255 tasks.
2. Allocates the second task with invalid parameters that cause hardware to fail.
3. Mitigates the failure and polls until all submitted tasks are flushed.

The difference between `pe_polling_sample.c` and `pe_task_error_sample.c` is to learn how to mitigate context error.

pe_task_resubmit

A task can be freed or reused after it is completed:

- Task resubmit can improve performance because the program does not free and allocate the task.
- Task resubmit can reduce memory usage (using a smaller task pool).
- Task members (e.g., source or destination buffer) can be set, so resubmission can be used if the source or destination are changed every iteration.

The sample demonstrates the following functions:

- How to re-submit a task in the completion callback
- How to replace buffers in a DMA task (similar to other task types)

The sample performs the following:

1. Allocates a set of 4 tasks and 16 buffer pairs.
2. Uses the tasks to copy all sources to destinations by resubmitting the tasks.

The difference between `pe_polling_sample.c` and `pe_task_resubmit_sample.c` is to learn how to use task resubmission.

pe_task_try_submit

`doca_task_submit` does not validate task inputs (to increase performance). Developers can use `doca_task_try_submit` to validate the tasks during development.

> ⚠ Task validation impacts performance and should not be used in production.

The sample demonstrates the following functions:

- How to use `doca_task_try_submit` instead of `doca_task_submit`

The sample performs the following:

1. Allocates and tries to submit tasks using `doca_task_try_submit`.

The difference between `pe_polling_sample.c` and `pe_task_try_submit_sample.c` is to learn how to use `doca_task_try_submit`.

14.4.1.1.5.2 Graph Sample

The graph sample demonstrates how to use DOCA graph with PE. The sample can be used to learn how to build and use DOCA graph.

The sample uses two nodes of DOCA DMA and one user node.

The graph runs both DMA nodes (copying a source buffer to two destinations). Once both nodes are complete, the graph runs the user node that compares the buffers.

The sample runs 10 instances of the graph in parallel.

14.4.1.1.6 Backward Compatibility of DOCA Core doca_buf

This section lists changes to the DOCA SDK which impacts backward compatibility.

14.4.1.1.6.1 DOCA Core doca_buf

Unable to render include or excerpt-include. Could not retrieve page.

14.4.1.1.7 Sync Event

> ⚠ DOCA Sync Event API is considered thread-unsafe

> ⚠ DOCA Sync Event does not currently support GPU related features.

14.4.1.1.7.1 Introduction

DOCA Sync Event (SE) is a software synchronization mechanism for parallel execution across the CPU, DPU, DPA and remote nodes. The SE holds a 64-bit counter which can be updated, read, and waited upon from any of these units to achieve synchronization between executions on them.

To achieve the best performance, DOCA SE defines a subscriber and publisher locality, where:

- Publisher – the entity which updates (sets or increments) the event value
- Subscriber – the entity which gets and waits upon the SE

  > ⓘ Both publisher and subscriber can read (get) the actual counter's value.

Based on hints, DOCA selects memory locality of the SE counter, closer to the subscriber side. Each DOCA SE is configured with a single publisher location and a single subscriber location which can be the CPU or DPU.

The SE control path happens on the CPU (either host CPU or DPU CPU) through the DOCA SE CPU handle. It is possible to retrieve different execution-unit-specific handles (DPU/DPA/GPU/remote handles) by exporting the SE instance through the CPU handle. Each SE handle refers to the DOCA SE instance from which it is retrieved. By using the execution-unit-specific handle, the associated SE instance can be operated from that execution unit.

In a basic scenario, synchronization is achieved by updating the SE from one execution and waiting upon the SE from another execution unit.

14.4.1.1.7.2 Prerequisites

DOCA SE can be used as a context which follows the architecture of a DOCA Core Context, it is recommended to read the following sections of the DOCA Core page before proceeding:

- DOCA Execution Model
- DOCA Device
- DOCA Memory Subsystem

14.4.1.1.7.3 Environment

DOCA SE based applications can run either on the host machine or on the NVIDIA® BlueField® DPU target and can involve DPA, GPU and other remote nodes.

Using DOCA SE with DPU requires BlueField to be configured to work in DPU mode as described in NVIDIA BlueField Modes of Operation.

> ⓘ Asynchronous wait on a DOCA SE requires NVIDIA® BlueField-3® or newer.

14.4.1.1.7.4 Architecture

DOCA SE can be converted to a DOCA Context as defined by DOCA Core. See DOCA Context for more information.

As a context, DOCA SE leverages DOCA Core architecture to expose asynchronous tasks/events offloaded to hardware.

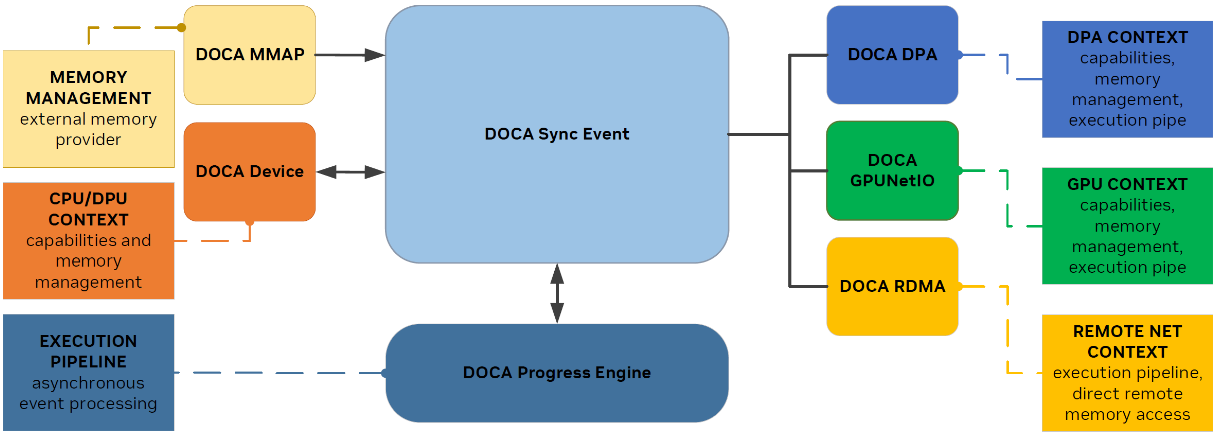
The figure that follows demonstrates components used by DOCA SE. DOCA Device provides information on the capabilities of the configured HW used by SE to control system resources.

DOCA DPA, GPUNetIO, and RDMA modules are required for cross-device synchronization (could be DPA, GPU, or remote peer respectively).

DOCA SE allows flexible memory management by its ability to specify an external buffer, where a DOCA mmap module handles memory registration for advanced synchronization scenarios.

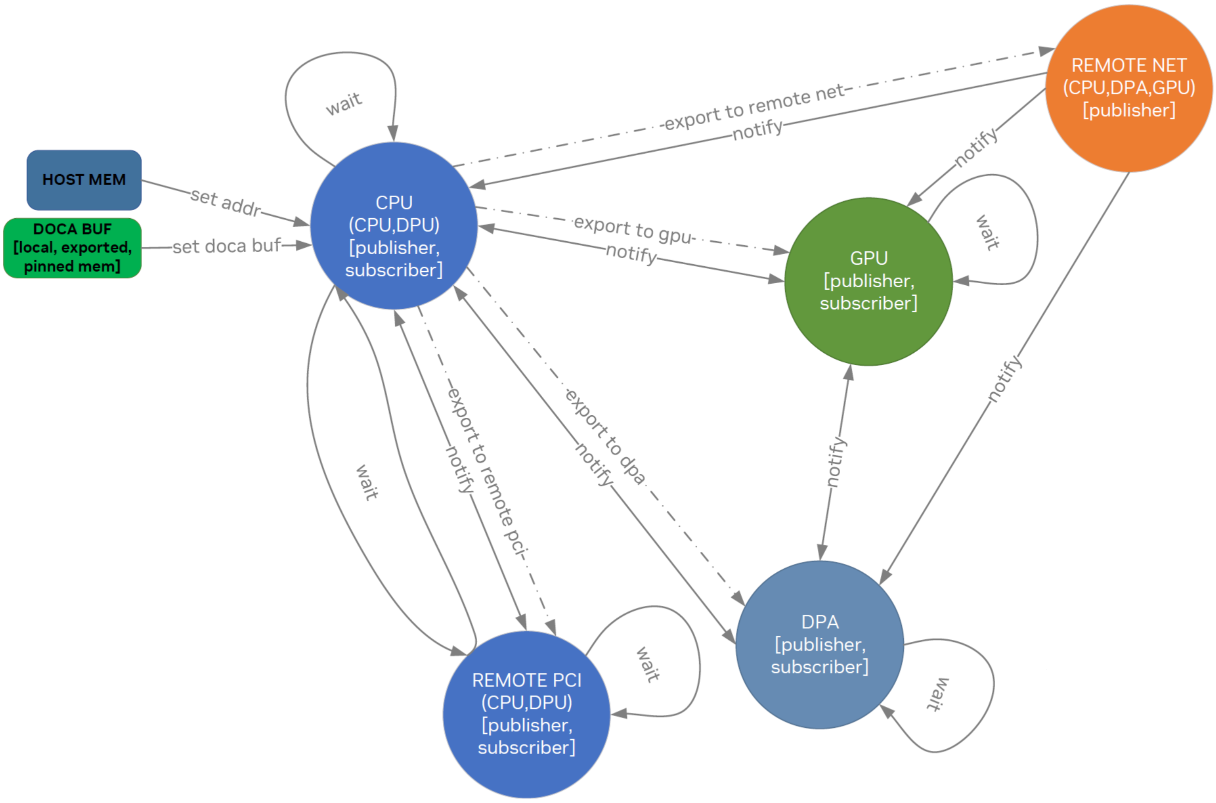
For asynchronous operation scheduling, SE uses the DOCA Progress Engine (PE) module.

DOCA Sync Event Components Diagram


MEMORY MANAGEMENT
external memory provider
DOCA MMAP
CPU/DPU CONTEXT
capabilities and memory management
DOCA Device
EXECUTION PIPELINE
asynchronous event processing
DOCA Sync Event
DOCA Progress Engine
DOCA DPA
DOCA GPUNetIO
DOCA RDMA
DPA CONTEXT
capabilities, memory management, execution pipe
GPU CONTEXT
capabilities, memory management, execution pipe
REMOTE NET CONTEXT
execution pipeline, direct remote memory access

The following diagram represents DOCA SE synchronization abilities on various devices.

DOCA Sync Event Interaction Diagram



DOCA Sync Event Objects

DOCA SE exposes different types of handles per execution unit as detailed in the following table.

Execution Unit	Type	Description
CPU (host/DPU)	`struct doca_sync_event`	Handle for interacting with the SE from the CPU
DPU	`struct doca_sync_event`	Handle for interacting with the SE from the DPU

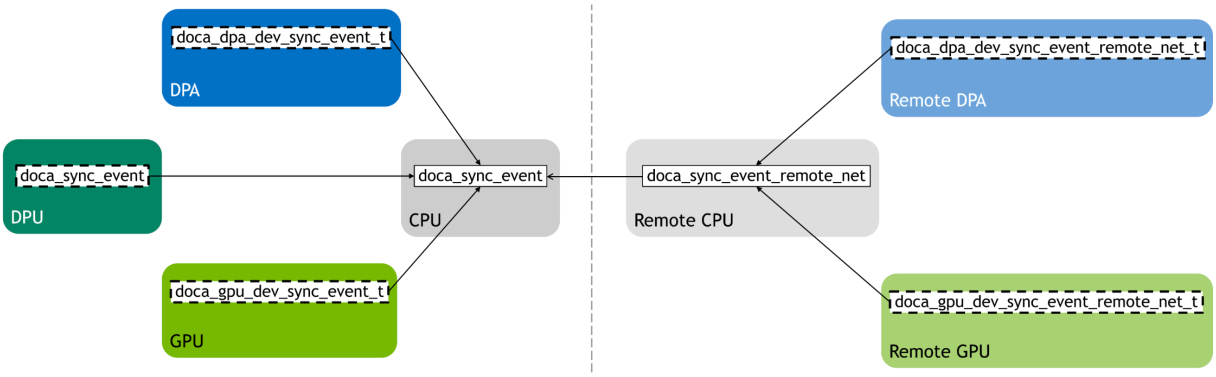
Execution Unit	Type	Description
DPA	`doca_dpa_dev_sync_event_t`	Handle for interacting with the SE from the DPA
GPU	`doca_gpu_dev_sync_event_t`	Handle for interacting with the SE from the GPU
Remote net CPU	`doca_sync_event_remote_net`	Handle for interacting with the SE from a remote CPU
Remote net DPA	`doca_dpa_dev_sync_event_remote_net_t`	Handle for interacting with the SE from a remote DPA
Remote net GPU	`doca_gpu_dev_sync_event_remote_net_t`	Handle for interacting with the SE from a remote GPU

Each one of these handle types has its own dedicated API for creating the handle and interacting with it.

14.4.1.1.7.5 Configuration Phase

Any DOCA SE creation starts with creating CPU handle by calling `doca_sync_event_create` API.

After creation, the SE entity could be shared with local PCIe, remote CPU, DPA, and GPU by a dedicated handle creation via the DOCA SE export flow, as illustrated in the following diagram:


doca_dpa_dev_sync_event_t
DPA
doca_sync_event
DPU
doca_sync_event
CPU
doca_gpu_dev_sync_event_t
GPU
doca_dpa_dev_sync_event_remote_net_t
Remote DPA
doca_sync_event_remote_net
Remote CPU
doca_gpu_dev_sync_event_remote_net_t
Remote GPU

Operation Modes

DOCA SE exposes two different APIs for starting it depending on the desired operation mode, synchronous or asynchronous.

> Once started, SE operation mode cannot be changed.

Synchronous Mode

Start the SE to operate in synchronous mode by calling `doca_sync_event_start`.

In synchronous operation mode, each data path operation (get, update, wait) blocks the calling thread from continuing until the operation is done.

⚠ An operation is considered done if the requested change fails and the exact error can be reported or if the requested change has taken effect.

Asynchronous Mode

To start the SE to operate in asynchronous mode, convert the SE instance to `doca_ctx` by calling `doca_sync_event_as_ctx`. Then use DOCA CTX API to start the SE and DOCA PE API to submit tasks on the SE (see section "DOCA Progress Engine" for more).

Configurations

Mandatory Configurations

These configurations must be set by the application before attempting to start the SE:

- DOCA SE CPU handle must be configured by providing the runtime hints on the publisher and subscriber locations. Both the subscriber and publisher locations must be configured using the following APIs:
  - `doca_sync_event_add_publisher_location_<cpu|dpa|gpu|remote_pci|remote_net>`
  - `doca_sync_event_add_subscriber_location_<cpu|dpa|gpu|remote_pci>`
- For the asynchronous use case, at least one task/event type must be configured. See configuration of tasks.

Optional Configurations

ⓘ If these configurations are not set, a default value is used.

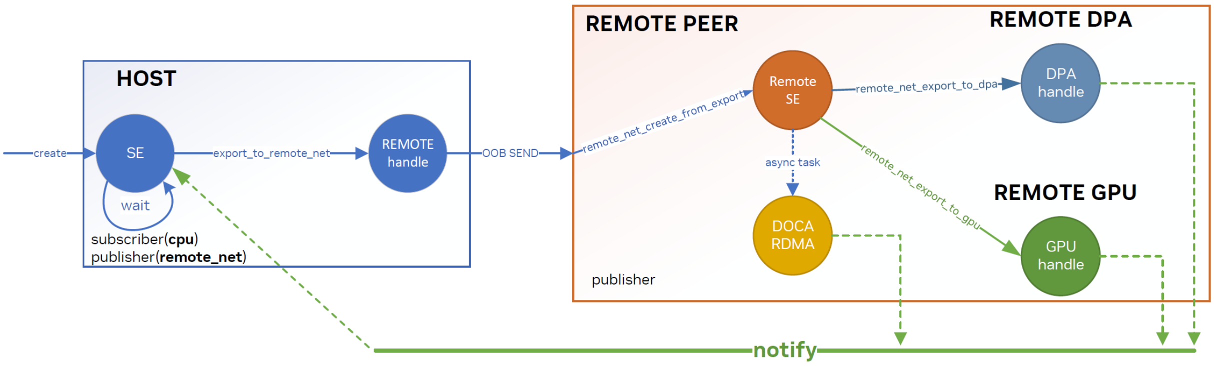
- These configurations provide an 8-byte buffer to be used as the backing memory of the SE. If set, it is user responsibility to handle the memory (i.e., preserve the memory allocated during DOCA SE lifecycle and free it after DOCA SE destruction). If not provided, the SE backing memory is allocated by the SE.
  - `doca_sync_event_set_addr`
  - `doca_sync_event_set_doca_buf`

Export DOCA Sync Event to Another Execution Unit

To use an SE from an execution unit other than the CPU, it must be exported to get a handle for the specific execution unit:

- DPA - `doca_sync_event_get_dpa_handle` returns a DOCA SE DPA handle (`doca_dpa_dev_sync_event_t`) which can be passed to the DPA SE data path APIs from the DPA kernel
- GPU - `doca_sync_event_get_gpu_handle` returns a DOCA SE GPU handle (`doca_gpu_dev_sync_event_t`) which can be passed to the GPU SE data path APIs for the CUDA kernel
- DPU - `doca_sync_event_export_to_remote_pci` returns a blob which can be used from the DPU CPU to instantiate a DOCA SE DPU handle (`struct doca_sync_event`) using the `doca_sync_event_create_from_export` function

DOCA SE allows notifications from remote peers (remote net) utilizing capabilities of the DOCA RDMA library. The following figure illustrates the remote net export flow:



- Remote net CPU - `doca_sync_event_export_to_remote_net` returns a blob which can be used from the remote net CPU to instantiate a DOCA SE remote net CPU handle (`struct doca_sync_event_remote_net`) using the `doca_sync_event_remote_net_create_from_export` function. The handle can be used directly for submitting asynchronous tasks through the `doca_rdma` library or exported to the remote DPA/GPU.
- Remote net DPA - `doca_sync_event_remote_net_get_dpa_handle` returns a DOCA SE remote net DPA handle (`doca_dpa_dev_sync_event_remote_net_t`) which can be passed to the DPA RDMA data path APIs from a DPA kernel
- Remote net GPU - `doca_sync_event_remote_net_get_gpu_handle` returns a DOCA SE remote net GPU handle (`doca_gpu_dev_sync_event_remote_net_t`) which can be passed to the GPU RDMA data path APIs from a CUDA kernel

The CPU handle (`struct doca_sync_event`) can be exported only to the location where the SE is configured.

Prior to calling any export function, users must first verify it is supported by calling the corresponding export capability getter:
`doca_sync_event_cap_is_export_to_dpa_supported` ,

`doca_sync_event_cap_is_export_to_gpu_supported`,
`doca_sync_event_cap_is_export_to_remote_pci_supported`, or
`doca_sync_event_cap_is_export_to_remote_net_supported`.

Prior to calling any `*_create_from_export` function, users must first verify it is supported by calling the corresponding create from the export capability getter: `doca_sync_event_cap_is_create_from_export_supported` or `doca_sync_event_cap_remote_net_is_create_from_export_supported`.

Once created from an export, both the SE DPU handle `struct doca_sync_event` and the SE remote net CPU handle `struct doca_sync_event_remote_net` cannot be configured, but only the SE DPU handle must be started before it is used.

Data exported in `doca_sync_event_export_to_*` functions contains sensitive information. Make sure to pass this data through a secure channel!

Device Support

DOCA SE needs a device to operate. For instructions on picking a device, see DOCA Core device discovery.

Both NVIDIA® BlueField®-2 and BlueField®-3 devices are supported as well as any `doca_dev` is supported.

Asynchronous wait (blocking/polling) is supported on NVIDIA® BlueField®-3 and NVIDIA® ConnectX®-7 and later.

As device capabilities may change in the future (see DOCA Capability Check), it is recommended to choose your device using any relevant capability method (starting with the prefix `doca_sync_event_cap_*`).

Capability APIs to query whether sync event can be constructed from export blob:

- `doca_sync_event_cap_is_create_from_export_supported`
- `doca_sync_event_cap_remote_net_is_create_from_export_supported`

Capability APIs to query whether sync event can be exported to other execution units:

- `doca_sync_event_cap_is_export_to_remote_pci_supported`
- `doca_sync_event_cap_is_export_to_dpa_supported`
- `doca_sync_event_cap_is_export_to_gpu_supported`
- `doca_sync_event_cap_is_export_to_remote_net_supported`
- `doca_sync_event_cap_remote_net_is_export_to_dpa_supported`
- `doca_sync_event_cap_remote_net_is_export_to_gpu_supported`

Capability APIs to query whether an asynchronous task is supported:

- `doca_sync_event_cap_task_get_is_supported`
- `doca_sync_event_cap_task_notify_set_is_supported`
- `doca_sync_event_cap_task_notify_add_is_supported`
- `doca_sync_event_cap_task_wait_eq_is_supported`
- `doca_sync_event_cap_task_wait_neq_is_supported`

14.4.1.1.7.6 Execution Phase

This section describes execution on CPU. For additional execution environments refer to section "Alternative Datapath Options".

DOCA Sync Event Data Path Operations

The DOCA SE synchronization mechanism is achieved using exposed datapath operations. The API exposes a function for "writing" to the SE and for "reading" the SE.

The synchronous API is a set of functions which can be called directly by the user, while the asynchronous API is exposed by defining a corresponding `doca_task` for each synchronous function to be submitted on a DOCA PE (see DOCA Progress Engine and DOCA Context for additional information).

> Remote net CPU handle (`struct doca_sync_event_remote_net`) can be used for submitting asynchronous tasks using the DOCA RDMA library.

> Prior to asynchronous task submission, users must check if the job is supported using `doca_error_t doca_sync_event_cap_task_<task_type>_is_supported`.

The following subsections describe the DOCA SE datapath operation with respect to synchronous and asynchronous operation modes.

Publishing on DOCA Sync Event

Setting DOCA Sync Event Value

Users can set DOCA SE to a 64-bit value:

- Synchronously by calling `doca_sync_event_update_set`
- Asynchronously by submitting a `doca_sync_event_task_notify_set` task

Adding to DOCA Sync Event Value

Users can atomically increment the value of a DOCA SE:

- Synchronously by calling `doca_sync_event_update_add`
- Asynchronously by submitting a `doca_sync_event_task_notify_add` task

Subscribe on DOCA Sync Event

Getting DOCA Sync Event Value

Users can get the value of a DOCA SE:

- Synchronously by calling `doca_sync_event_get`
- Asynchronously by submitting a `doca_sync_event_task_get` task

Waiting on DOCA Sync Event

Waiting for an event is the main operation for achieving synchronization between different execution units.

Users can wait until an SE reaches a specific value in a variety of ways.

Synchronously

`doca_sync_event_wait_gt` waits for the value of a DOCA SE to be greater than a specified value in a "polling busy wait" manner (100% processor utilization). This API enables users to wait for an SE in real time.

`doca_sync_event_wait_gt_yield` waits for the value of a DOCA SE to be greater than a specified value in a "periodically busy wait" manner. After each polling iteration, the calling thread relinquishes the CPU, so a new thread gets to run. This API allows a tradeoff between real-time polling to CPU starvation.

`doca_sync_event_wait_eq` waits for the value of a DOCA SE to be equal to a specified value in a "polling busy wait" manner (100% processor utilization). This API enables users to wait for an SE in real time.

`doca_sync_event_wait_eq_yield` waits for the value of a DOCA SE to be equal to a specified value in a "periodically busy wait" manner. After each polling iteration, the calling thread relinquishes the CPU so a new thread gets to run. This API allows a tradeoff between real-time polling to CPU starvation.

`doca_sync_event_wait_neq` waits for the value of a DOCA SE to not be equal to a specified value in a "polling busy wait" manner (100% processor utilization). This API enables users to wait for an SE in real time.

`doca_sync_event_wait_neq_yield` waits for the value of a DOCA SE to not be equal to a specified value in a "periodically busy wait" manner. After each polling iteration, the calling thread relinquishes the CPU so a new thread gets to run. This API allows a tradeoff between real-time polling to CPU starvation.

> ⚠ This wait method is supported only from the CPU.

Asynchronously

DOCA SE exposes an asynchronous wait method by defining a `doca_sync_event_task_wait_eq` and `doca_sync_event_task_wait_neq` tasks.

Users can wait for wait-job completion in the following methods:

- Blocking - get a `doca_event_handle_t` from the `doca_pe` to blocking-wait on
- Polling - poll the wait task by calling `doca_pe_progress`

ⓘ Asynchronous wait (blocking/polling) is supported on BlueField-3 and ConnectX-7 and later.

⚠ Users may leverage the `doca_sync_event_task_get` job to implement asynchronous wait by asynchronously submitting the task on a DOCA PE and comparing the result to some threshold.

Tasks

DOCA SE context exposes asynchronous tasks that leverage the DPU hardware according to the DOCA Core architecture. See DOCA Core Task.

Get Task

The get task retrieves the value of a DOCA SE.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_sync_event_task_get_set_conf`	`doca_sync_event_cap_task_get_is_supported`
Number of tasks	`doca_sync_event_task_get_set_conf`	-

Task Input

Common input described in DOCA Core Task.

Name	Description
Return value	8-bytes memory pointer to hold the DOCA SE value

Task Output

Common output described in DOCA Core Task.

Task Completion Success

After the task is completed successfully, the return value memory holds the DOCA SE value.

Task Completion Failure

If the task fails midway:

- The context may enter a stopping state if a fatal error occurs
- The return value memory may be modified

Task Limitations

All limitations are described in DOCA Core Task.

Notify Set Task

The notify set task allows setting the value of a DOCA SE.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_sync_event_task_notify_set_set_conf`	`doca_sync_event_cap_task_notify_set_is_supported`
Number of tasks	`doca_sync_event_task_notify_set_set_conf`	-

Task Input

Common input described in DOCA Core Task.

Name	Description
Set value	64-bit value to set the DOCA SE value to

Task Output

Common output described in DOCA Core Task.

Task Completion Success

After the task is completed successfully, the DOCA SE value is set to the given set value.

Task Completion Failure

If the task fails midway, the context may enter a stopping state if a fatal error occurs.

Task Limitations

This operation is not atomic. Other limitations are described in DOCA Core Task.

Notify Add Task

The notify add task allows atomically setting the value of a DOCA SE.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_sync_event_task_notify_add_set_conf`	`doca_sync_event_cap_task_notify_add_is_supported`
Number of tasks	`doca_sync_event_task_notify_add_set_conf`	-

Task Input

Common input described in DOCA Core Task.

Name	Description
Increment value	64-bit value to atomically increment the DOCA SE value by
Fetched value	8-bytes memory pointer to hold the DOCA SE value before the increment

Task Output

Common output described in DOCA Core Task.

Task Completion Success

After the task is completed successfully, the following occurs:

- The DOCA SE value is incremented according to the given increment value
- The fetched value memory holds the DOCA SE value before the increment

Task Completion Failure

If the task fails midway:

- The context may enter a stopping state if a fatal error occurs
- The fetched value memory may be modified.

Task Limitations

All limitations are described in DOCA Core Task.

Wait Equal-to Task

The wait-equal task allows atomically waiting for a DOCA SE value to be equal to some threshold.

Task Configuration

Description	API to set the configuration	API to query support
Enable the task	`doca_sync_event_task_wait_eq_set_conf`	`doca_sync_event_cap_task_wait_eq_is_supported`
Number of tasks	`doca_sync_event_task_wait_eq_set_conf`	-

Task Input

Common input described in DOCA Core Task.

Name	Description
Wait threshold	64-bit value to wait for the DOCA SE value to be equal to
Mask	64-bit mask to apply on the DOCA SE value before comparing with the wait threshold

Task Output

Common output described in DOCA Core Task.

Task Completion Success

After the task is completed successfully, the following occurs:

- The DOCA SE value is equal to the given wait threshold.

Task Completion Failure

If the task fails midway, the context may enter a stopping state if a fatal error occurs.

Task Limitations

Other limitations are described in DOCA Core Task.

Wait Not-equal-to Task

The wait-not-equal task allows atomically waiting for a DOCA SE value to not be equal to some threshold.

Task Configuration

Description	API to set the configuration	API to query support
Enable the task	`doca_sync_event_task_wait_neq_set_conf`	`doca_sync_event_cap_task_wait_neq_is_supported`
Number of tasks	`doca_sync_event_task_wait_neq_set_conf`	-

Task Input

Common input described in DOCA Core Task.

Name	Description
Wait threshold	64-bit value to wait for the DOCA SE value to be not equal to
Mask	64-bit mask to apply on the DOCA SE value before comparing with the wait threshold

Task Output

Common output described in DOCA Core Task.

Task Completion Success

After the task is completed successfully, the following occurs:

- The DOCA SE value is not equal to the given wait threshold.

Task Completion Failure

If the task fails midway, the context may enter a stopping state if a fatal error occurs.

Task Limitations

Limitations are described in DOCA Core Task.

Events

DOCA SE context exposes asynchronous events to notify about changes that happen unexpectedly, according to the DOCA Core architecture.

The only event DOCA SE context exposes is common events as described in DOCA Core Event.

14.4.1.1.7.7 State Machine

The DOCA SE context follows the Context state machine as described in DOCA Core Context State Machine.

The following subsection describe how to move to specific states and what is allowed in each state.

Idle

In this state, it is expected that the application will:

- Destroy the context; or
- Start the context

Allowed operations in this state:

- Configure the context according to section "Configurations"
- Start the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop after making sure all tasks have been freed
Stopping	Call progress until all tasks are completed and then freed

Starting

This state cannot be reached.

Running

In this state, it is expected that the application will:

- Allocate and submit tasks
- Call progress to complete tasks and/or receive events

Allowed operations in this state:

- Allocate previously configured task
- Submit an allocated task
- Call stop

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

Stopping

In this state, it is expected that the application will:

- Call progress to complete all inflight tasks (tasks will complete with failure)
- Free any completed tasks

Allowed operations in this state:

- Call progress

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and fatal error occurs
Running	Call stop without freeing all tasks

14.4.1.1.7.8 DOCA Sync Event Tear Down

Multiple SE handles (for different execution units) associated with the same DOCA SE instance can live simultaneously, though the teardown flow is performed only from the CPU on the CPU handle.

> ⚠ Users must validate active handles associated with the CPU handle during the teardown flow because DOCA SE does not do that.

Stopping DOCA Sync Event

To stop a DOCA SE:

- Synchronous - call `doca_sync_event_stop` on the CPU handle
- Asynchronous - stop the DOCA context associated with the DOCA SE instance

> ⚠ Stopping a DOCA SE must be followed by destruction. Refer to section "Destroying DOCA Sync Event" for details.

Destroying DOCA Sync Event

Once stopped, a DOCA SE instance can be destroyed by calling `doca_sync_event_destroy` on the CPU handle.

Remote net CPU handle instance terminates and frees by calling `doca_sync_event_remote_net_destroy` on the remote net CPU handle.

Upon destruction, all the internal resources are released, allocated memory is freed, associated `doca_ctx` (if it exists) is destroyed, and any associated exported handles (other than CPU handles) and their resources are destroyed.

14.4.1.1.7.9 Alternative Datapath Options

DOCA SE supports datapath on CPU (see section "Execution Phase") and also on DPA and GPU.

GPU Datapath

DOCA SE does not currently support GPU related features.

DPA Datapath

> ⓘ An SE with DPA-subscriber configuration currently supports synchronous APIs only.

Once a DOCA SE DPA handle (`doca_dpa_dev_sync_event_t`) has been retrieved it can be used within a DOCA DPA kernel as described in DOCA DPA Sync Event.

14.4.1.1.7.10 DOCA Sync Event Sample

This section provides DOCA SE sample implementation on top of the BlueField DPU.

The sample demonstrates how to share an SE between the host and the DPU while simultaneously interacting with the event from both the host and DPU sides using different handles.

Running DOCA Sync Event Sample

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample:

   ```
   cd /opt/mellanox/doca/samples/doca_common/sync_event_<local|remote>_pci
   meson /tmp/build
   ninja -C /tmp/build
   ```

   The binary `doca_sync_event_<local|remote>_pci` is created under `/tmp/build/`.
3. Sample usage:

   ```
   Usage: doca_sync_event_remote_pci [DOCA Flags] [Program Flags]

   DOCA Flags:
     -h, --help                        Print a help synopsis
     -v, --version                     Print program version information
     -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 
   30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
     --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICA
   L, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
     -j, --json <path>                 Parse all command flags from an input json file

   Program Flags:
     -d, --pci-addr                    Device PCI address
     -r, --rep-pci-addr                DPU representor PCI address
     --async                           Start DOCA Sync Event in asynchronous mode (synchronous mode by default)
     --async_num_tasks                 Async num tasks for asynchronous mode
     --atomic                          Update DOCA Sync Event using Add operation (Set operation by default)
   ```

   The flag `--rep-pci-addr` is relevant only for the DPU.
4. For additional information per sample, use the `-h` option:

   ```
   /tmp/build/doca_sync_event_<local|remote>_pci -h
   ```

Samples

Sync Event Remote PCIe

This sample should be run (on the DPU or on the host) before Sync Event Local PCIe.

This sample demonstrates creating an SE from an export which is associated with an SE on a local PCIe (host or the DPU) and interacting with the SE to achieve synchronization between the host and DPU.

The sample logic includes:

1. Reading configuration files and saving their content into local buffers.
2. Locating and opening DOCA devices and DOCA representors (if running on the DPU) matching the given PCIe addresses.
3. Initializing DOCA Comm Channel.
4. Receiving SE blob through Comm Channel.
5. Creating SE from export.
6. Starting the above SE in the requested operation mode (synchronous or asynchronous).
7. Interacting with the SE:
    a. Waiting for signal from the host - synchronously or asynchronously (with busy wait polling) according to user input.
    b. Signaling the SE for the host - synchronously or asynchronously, using set or atomic add, according to user input.
8. Cleaning all resources.

Reference:

- `/opt/mellanox/doca/samples/doca_common/sync_event_remote_pci/sync_event_remote_pci_sample.c`
- `/opt/mellanox/doca/samples/doca_common/sync_event_remote_pci/sync_event_remote_pci_main.c`
- `/opt/mellanox/doca/samples/doca_common/sync_event_remote_pci/meson.build`

Sync Event Local PCIe

> ⚠ This sample should run (on the DPU or on the Host) only after Sync Event Remote PCIe has been started.

This sample demonstrates how to initialize a SE to be shared with a remote PCIe (host or the DPU) how to export it to a remote PCIe, and how to interact with the SE to achieve synchronization between the host and DPU.

The sample logic includes:

1. Reading configuration files and saving their content into local buffers.
2. Locating and opening DOCA devices and DOCA representors (if running on the DPU) matching the given PCIe addresses.
3. Creating and configuring the SE to be shared with a remote PCIe.
4. Starting the above SE in the requested operation mode (synchronous or asynchronous).
5. Initializing DOCA Comm Channel.
6. Exporting the SE and sending it through the Comm Channel.
7. Interacting with the SE :
    a. Signaling the SE for the remote PCIe - synchronously or asynchronously, using set or atomic add, according to user input.

    b. Waiting for a signal - synchronously or asynchronously, with busy wait polling, according to user input.
8. Cleaning all resources.

Reference:

- `/opt/mellanox/doca/samples/doca_common/sync_event_local_pci/sync_event_local_pci_sample.c`
- `/opt/mellanox/doca/samples/doca_common/sync_event_local_pci/sync_event_local_pci_main.c`
- `/opt/mellanox/doca/samples/doca_common/sync_event_local_pci/meson.build`

14.4.1.1.8 Mmap Advise

14.4.1.1.8.1 Introduction

DOCA Mmap Advise is used to give advanced memory-related instructions to NVIDIA® BlueField® DPUs in order to improve system or application performance.

> ⚠ To use DOCA Mmap Advise with BlueField, the device must be configured to work in DPU mode as described in NVIDIA BlueField Modes of Operation.

The operations in the instructions are meant to influence the performance of the application, but not its semantics. The operations allow an application to inform the NIC how it expects it to use some mapped memory areas, so the BlueField's hardware can choose appropriate optimization techniques.

14.4.1.1.8.2 Prerequisites

DOCA Mmap Advise is a context and follows the architecture of a DOCA Core Context, it is recommended to read the following sections of the DOCA Core page before proceeding:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

14.4.1.1.8.3 Architecture

DOCA Mmap Advise is a DOCA Context as defined by DOCA Core. See DOCA Core Context for more information.

DOCA Mmap Advise currently supports the following list of advised operations:

- Cache Invalidate Operation

Cache Invalidate Operation

When data is processed by BlueField's cores it may be temporarily stored in the cores' system-level cache (i.e., L3 cache). When a cache line is occupied and new data must be written to it, the cache management sub-system evicts the existing data, usually based on LRU policy, by performing a write-back operation to store this data in the main (DDR) memory. When this data is not required to be stored in the BlueField's memory (e.g., it is host data and is no longer needed after it is copied to the host's memory), the cache's write-back operation wastes memory bandwidth that reduces overall system performance, which is undesirable. The simplest to avoid this write-back operation is to mark the appropriate cache lines as "invalid". This enables their immediate reuse, without additional operations.

The cache invalidate operation facilitates invalidating a set of cache lines.

14.4.1.1.8.4 Environment

Applications based on DOCA Mmap Advise can run on the BlueField target.

Objects

Device and Device Representor

The MMAP Advise context requires a DOCA Device to operate. The device is used to access memory and perform the copy operation. See DOCA Core Device Discovery.

> ⓘ For the same DPU, it does not matter which device is used (i.e., PF, VF, SF) as all these devices utilize the same hardware components.

> ⚠ The device must stay valid for as long as the MMAP Advise instance is not destroyed.

Memory Buffers

The cache invalidate task requires one DOCA Buffer containing the address space to invalidate depending on the allocation pattern of the buffers (refer to the table in section "Inventory Types"). To find what kind of memory is supported, refer to the table in section "Buffer Support".

Buffers must not be modified or read during the cache invalidate operation.

14.4.1.1.8.5 Configuration Phase

To start using the context, users must go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context, to allow execution of tasks and retrieval of events.

Configurations

The context can be configured to match the application's use case.

To find if a configuration is supported, or what the min/max value for it is, refer to section "Device Support".

Mandatory Configurations

These configurations are mandatory and must be set by the application before attempting to start the context:

- At least one task/event type must be configured. See configuration of tasks and/or events in sections "Tasks" and "Events" respectively for information.

- A device with appropriate support must be provided upon creation

Device Support

DOCA Mmap Advise requires a device to operate. To pick a device, refer to DOCA Core Device Discovery.

As device capabilities may change (see DOCA Core Device Support), it is recommended to select your device using the following method:

- `doca_mmap_advise_cap_task_cache_invalidate_is_supported`

Some devices expose different capabilities as follows:

- Maximum cache invalidate buffer size may differ.

Buffer Support

Tasks support buffers with the following features:

Buffer Type	Buffer
Local mmap buffer	Yes
MMAP from PCIe export buffer	No
MMAP from RDMA export buffer	No
Linked list buffer	No

14.4.1.1.8.6 Execution Phase

This section describes execution on the CPU using DOCA Core Progress Engine.

Tasks

DOCA Mmap Advise exposes asynchronous tasks that leverage DPU hardware according to the DOCA Core architecture. See DOCA Core Task for information.

Cache Invalidate Task

The cache invalidate task facilitates invalidating a set of cache lines, preventing them from being written back to the RAM (thus increasing performance).

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_mmap_advise_task_invalidate_cache_set_conf`	`doca_mmap_advise_cap_task_cache_invalidate_is_supported`
Number of tasks	`doca_mmap_advise_task_invalidate_cache_set_conf`	-
Maximal buffer size	-	`doca_mmap_advise_task_cache_invalidate_get_max_buf_size`
Maximal buffer list size	-	-

Task Input

Common input as described in DOCA Core Task.

Name	Description
buffer	Buffer that points to the memory to be invalidated

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task is completed successfully:

- The cache is invalidated

Task Completion Failure

If the task fails midway:

- The context may enter stopping state, if a fatal error occurs
- The cache is not invalidated

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the buffer should not be read/written to
- Other limitations are described in DOCA Core Task

Events

DOCA Mmap Advise exposes asynchronous events to notify on changes that happen unexpectedly, according to DOCA Core architecture.

The only events DOCA Mmap Advise exposes are common events as described in DOCA Core Event.

14.4.1.1.8.7 State Machine

DOCA Mmap Advise context follows the context state machine as described in DOCA Core Context State Machine.

The following section describes how to move states and what is allowed in each state.

Idle

In this state it is expected that the application:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to section "Configurations"
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop after making sure all tasks have been freed
Stopping	Call progress until all tasks are completed and freed

Starting

This state cannot be reached.

Running

In this state, it is expected that the application:

- Allocates and submits tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocating a previously configured task
- Submitting a task
- Calling stop

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

Stopping

In this state it is expected that the application:

- Calls progress to complete all in-flight tasks (tasks complete with failure)
- Frees any completed tasks

Allowed operations:

- Call progress

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and fatal error occurs
Running	Call stop without freeing all tasks

14.4.1.1.8.8 Alternative Datapath Options

DOCA Mmap Advise only supports datapath on the CPU. See section "Execution Phase".

14.4.1.1.8.9 Samples

Cache Invalidate Sample

The sample illustrates how to invalidate the cache for a memory range after copying it using DOCA DMA.

The sample logic includes:

1. Locating DOCA device.
2. Initializing needed DOCA core structures.
3. Populating DOCA memory map with two relevant buffers.
4. Allocating element in DOCA buffer inventory for each buffer.
5. Initializing DOCA DMA memory copy task object.
6. Initializing DOCA Mmap Advise cache invalidate task object
7. Submitting DMA task.
8. Polling for completion:
    a. Handling DMA task completion – submitting the cache invalidate task in the DMA task completion callback body.
    b. Handling cache invalidate task completion.
9. Polling for completion.
10. Destroying DMA, DOCA MMAP Advise, and DOCA Core objects.

Reference:

- `/opt/mellanox/doca/samples/doca_common/cache_invalidate/cache_invalidate_sample.c`
- `/opt/mellanox/doca/samples/doca_common/cache_invalidate/cache_invalidate_main.c`
- `/opt/mellanox/doca/samples/doca_common/cache_invalidate/meson.build`

14.4.1.2 DOCA Log

DOCA logging infrastructure allows printing DOCA SDK library error messages, and printing debug and error messages from applications.

To work with the DOCA logging mechanism, the header file `doca_log.h` must be included in every source code using it.

14.4.1.2.1 Log Verbosity Level Enumerations

The following verbosity levels are supported by the DOCA logging:

```
enum doca_log_level {
    DOCA_LOG_LEVEL_DISABLE = 10,    /**< Disable log messages */
    DOCA_LOG_LEVEL_CRIT = 20,       /**< Critical log level */
    DOCA_LOG_LEVEL_ERROR = 30,      /**< Error log level */
    DOCA_LOG_LEVEL_WARNING = 40,    /**< Warning log level */
    DOCA_LOG_LEVEL_INFO = 50,       /**< Info log level */
    DOCA_LOG_LEVEL_DEBUG = 60,      /**< Debug log level */
    DOCA_LOG_LEVEL_TRACE = 70,      /**< Trace log level */
};
```

⚠ The `DOCA_LOG_LEVEL_TRACE` verbosity level is available only if the macro `DOCA_LOGGING_ALLOW_TRACE` is set before the compilation.

See `doca_log.h` for more information.

14.4.1.2.2 Logging Backends

DOCA's logging backend is the target to which log messages are directed.

The following backend types are supported:

- FILE * - file stream which can be any open file or stdout/stderr
- file descriptor - any file descriptor that the system supports, including (but not limited to) raw files, sockets, and pipes
- buf - memory buffer (address and size) that can hold a single message and a callback to be called for every logged message
- syslog - system standard logging

Every logger is created with the following default lower and upper verbosity levels:

- Lower level - `DOCA_LOG_LEVEL_INFO`
- Upper level - `DOCA_LOG_LEVEL_CRIT`

SDK and application logging have different default configuration values and can be controlled separately using the appropriate API.

Every message is printed to every created backend if its verbosity level allows it.

14.4.1.2.3 Enabling DOCA SDK Libraries Logging

The DOCA SDK libraries print debug and error messages to all the backends created using the following functions:

- `doca_log_backend_create_with_file_sdk()`
- `doca_log_backend_create_with_fd_sdk()`
- `doca_log_backend_create_with_buf_sdk()`
- `doca_log_backend_create_with_syslog_sdk()`

A newly created SDK backend verbosity level is set to the SDK global verbosity level value. This value can be changed using `doca_log_level_set_global_sdk_limit()`.

`doca_log_level_set_global_sdk_limit()` sets the verbosity level for all existing SDK backends and sets the SDK global verbosity level.

`doca_log_backend_set_sdk_level()` sets the verbosity level of a specific SDK backend.

`doca_log_level_get_global_sdk_limit()` gets the SDK global verbosity level.

> ⚠ Messages may change between different versions of DOCA. Users cannot rely on message permanence or formatting.

14.4.1.2.4 Enabling DOCA Application Logging

Any source code that uses DOCA can use DOCA logging infrastructure.

Every debug and error messages is printed to all backends created using the following functions:

- `doca_log_backend_create_with_file()`
- `doca_log_backend_create_with_fd()`
- `doca_log_backend_create_with_buf()`
- `doca_log_backend_create_with_syslog()`

The lower and upper levels of a newly created backend are set to the default values. Those values can be changed using `doca_log_backend_set_level_lower_limit()` and `doca_log_backend_set_level_upper_limit()` .

`doca_log_backend_create_standard()` creates a default non-configurable set of two backends:

- `stdout` prints the range from a global minimum level up to `DOCA_LOG_LEVEL_INFO`
- `stderr` prints the range from `DOCA_LOG_LEVEL_WARNING` level up to `DOCA_LOG_LEVEL_CRIT`

`doca_log_backend_set_level_lower_limit_strict()` marks the lower log level limit of a backend as strict, preventing it from being lowered by future log level changes. It is both global and direct.

`doca_log_backend_set_level_upper_limit_strict()` marks the upper log level limit of a backend as strict, preventing it from being raised by future log level changes. It is both global and direct.

`doca_log_level_set_global_lower_limit()` sets the lower limit for all existing backends not marked as strict and sets the global application lower limit.

`doca_log_level_set_global_upper_limit()` sets the upper limit for all existing backends not marked as strict and sets the global application upper limit.

14.4.1.2.5 Logging DOCA Application Messages

To use the DOCA logging infrastructure with your source code to log its messages, users must call, at the beginning of the file, the macro `DOCA_LOG_REGISTER(source)` just before using the DOCA logging functionality. This macro handles the registration and the teardown from the DOCA logging.

Printing a message can be done by calling one of the following macros (with the same usage as `printf()`):

- `DOCA_LOG_CRIT(format, ...)`
- `DOCA_LOG_ERR(format, ...)`
- `DOCA_LOG_WARN(format, ...)`
- `DOCA_LOG_INFO(format, ...)`
- `DOCA_LOG_DBG(format, ...)`
- `DOCA_LOG_TRC(format, ...)`

The message is printed to all the application's backends with configured lower and upper logging limits.

14.4.2 DOCA Flow

This guide describes how to deploy the DOCA Flow library, the philosophy of the DOCA Flow API, and how to use it. The guide is intended for developers writing network function applications that focus on packet processing (such as gateways). It assumes familiarity with the network stack and DPDK.

14.4.2.1 Introduction

DOCA Flow is the most fundamental API for building generic packet processing pipes in hardware. The DOCA Flow library provides an API for building a set of pipes, where each pipe consists of match criteria, monitoring, and a set of actions. Pipes can be chained so that after a pipe-defined action is executed, the packet may proceed to another pipe.

Using DOCA Flow API, it is easy to develop hardware-accelerated applications that have a match on up to two layers of packets (tunneled).

- MAC/VLAN/ETHERTYPE
- IPv4/IPv6
- TCP/UDP/ICMP
- GRE/VXLAN/GTP-U/ESP/PSP
- Metadata

The execution pipe can include packet modification actions such as the following:

- Modify MAC address
- Modify IP address
- Modify L4 (ports)
- Strip tunnel
- Add tunnel
- Set metadata
- Encrypt/Decrypt

The execution pipe can also have monitoring actions such as the following:

- Count
- Policers

The pipe also has a forwarding target which can be any of the following:

- Software (RSS to subset of queues)
- Port
- Another pipe
- Drop packets

14.4.2.2 Prerequisites

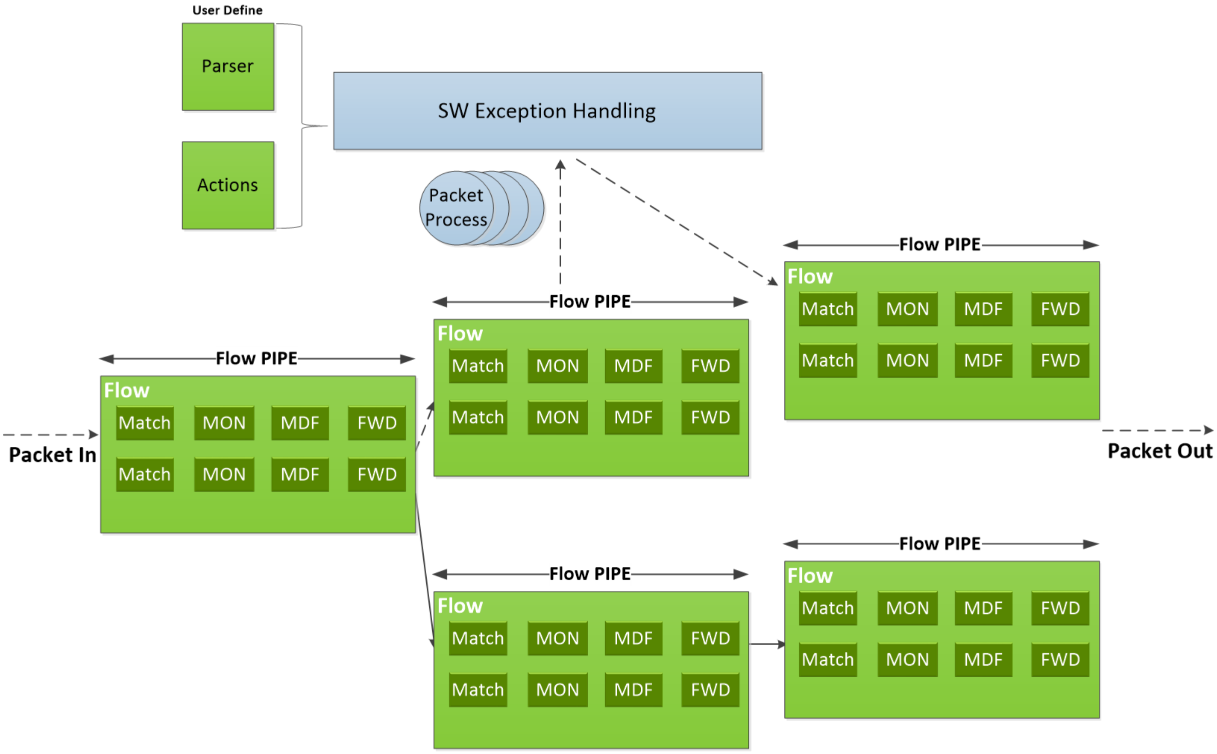
A DOCA Flow-based application can run either on the host machine or on an NVIDIA® BlueField® DPU target. Flow-based programs require an allocation of huge pages, hence the following commands are required:

```
echo '1024' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
sudo mkdir /mnt/huge
sudo mount -t hugetlbfs nodev /mnt/huge
```

> On some operating systems (RockyLinux, OpenEuler, CentOS 8.2) the default huge page size on the DPU (and Arm hosts) is larger than 2MB, often 512MB. Users can check the size of the huge pages on their OS using the following command:
>
> ```
> $ grep -i huge /proc/meminfo
>
> AnonHugePages:          0 kB
> ShmemHugePages:         0 kB
> FileHugePages:          0 kB
> HugePages_Total:        4
> HugePages_Free:         4
> HugePages_Rsvd:         0
> HugePages_Surp:         0
> Hugepagesize:      524288 kB
> Hugetlb:          6291456 kB
> ```
>
> In this case, instead of allocating 1024 pages, users should only allocate 4:
>
> ```
> echo '4' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-524288kB/nr_hugepages
> ```

14.4.2.3 Architecture

The following diagram shows how the DOCA Flow library defines a pipe template, receives a packet for processing, creates the pipe entry, and offloads the flow rule in NIC hardware.



Features of DOCA Flow:

- User-defined set of matches parser and actions
- DOCA Flow pipes can be created or destroyed dynamically

- Packet processing is fully accelerated by hardware with a specific entry in a flow pipe
- Packets that do not match any of the pipe entries in hardware can be sent to Arm cores for exception handling and then reinjected back to hardware

The DOCA Flow pipe consists of the following components:

- Monitor (MON in the diagram) - counts, meters, or mirrors
- Modify (MDF in the diagram) - modifies a field
- Forward (FWD in the diagram) - forwards to the next stage in packet processing

14.4.2.4 Steering Domains

DOCA Flow organizes pipes into high-level containers named domains to address the specific needs of the underlying architecture.

A key element in defining a domain is the packet direction and a set of allowed actions.

- A domain is a pipe attribute (also relates to shared objects)
- A domain restricts the set of allowed actions
- Transition between domains is well-defined (packets cannot cross domains arbitrarily)
- A domain may restrict the sharing of objects between packet directions
- Packet direction can restrict the move between domains

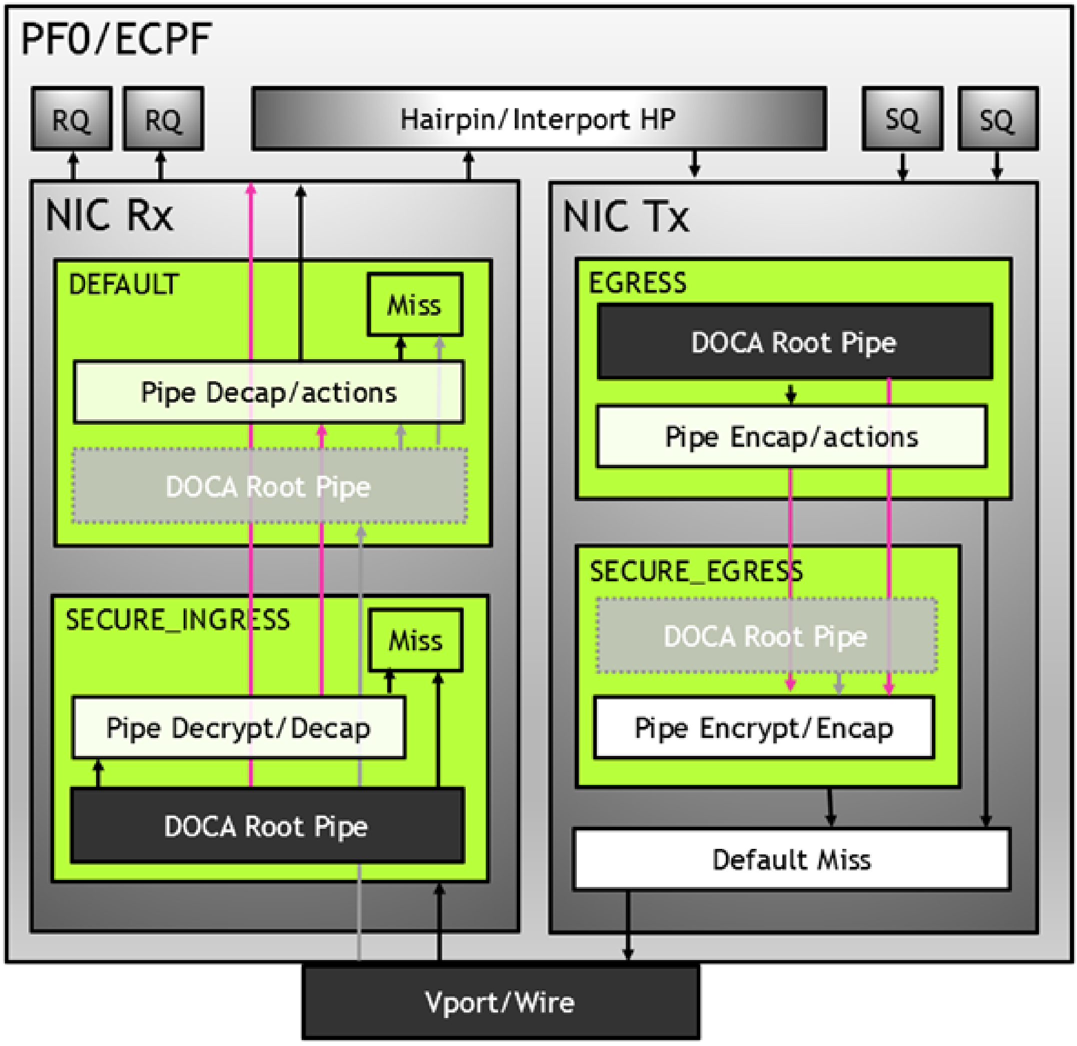
14.4.2.4.1 List of Steering Domains

DOCA Flow provides the following set of predefined steering domains:

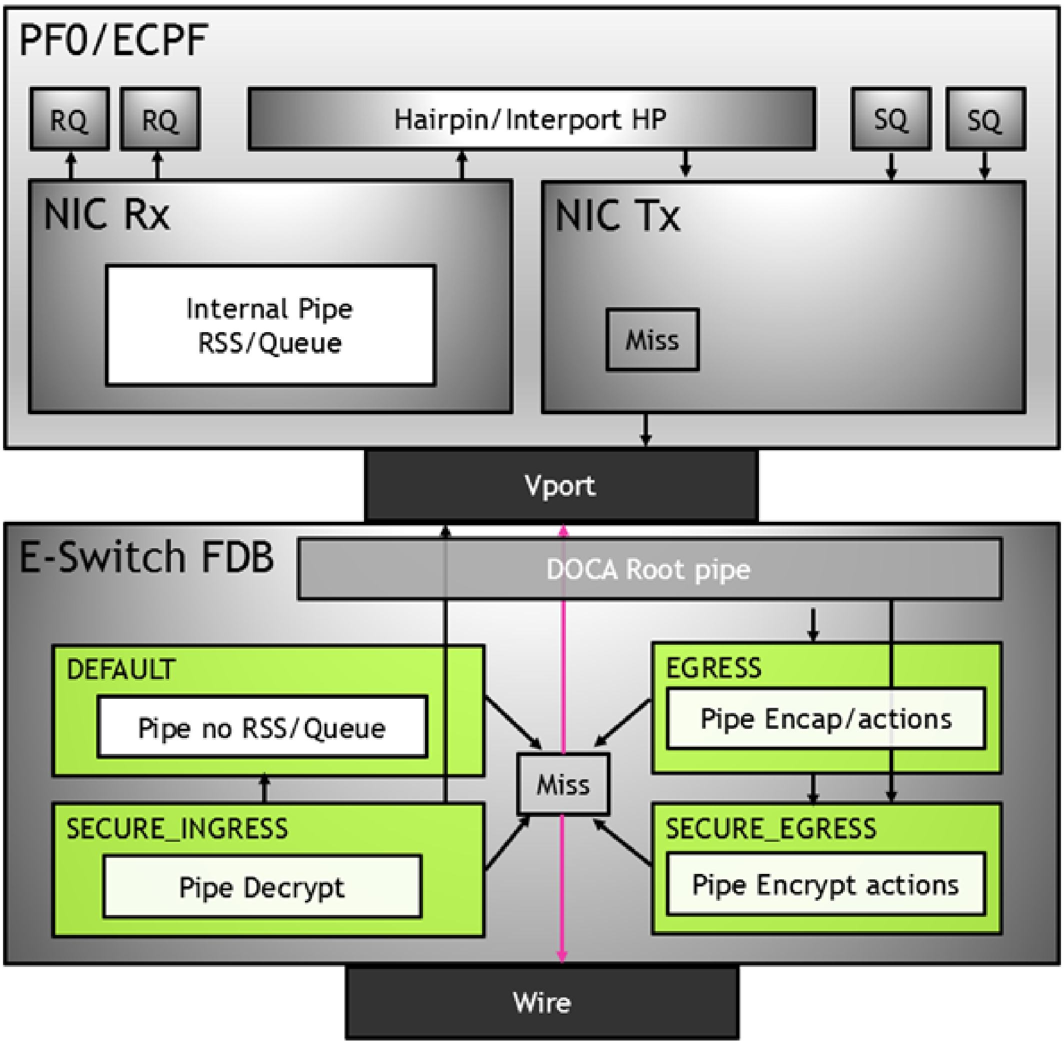
Domain	Description
`DOCA_FLOW_PIPE_DOMAIN_DEFAULT`	• Default domain for actions on ingress traffic • Encapsulated and secure actions are not allowed here • The next milestone is queue or pipe in the EGRESS domain • Miss action is: Drop
`DOCA_FLOW_PIPE_DOMAIN_SECURE_INGRESS`	• For secure actions on ingress traffic • Encapsulation and encrypting actions not allowed here • The only allowed domain for decrypting secure actions • The next milestone is queue or pipe in the DEFAULT or EGRESS domain • Only meta register is preserved • Miss action is: Drop • Memory may be optimized if set with `DOCA_FLOW_DIRECTION_NETWORK_TO_HOST` direction information
`DOCA_FLOW_PIPE_DOMAIN_EGRESS`	• Domain for actions on egress traffic • Decapsulation and secure actions are not allowed here • The next milestone is wire/representor or pipe in SECURE_EGRESS domain • Miss action is: Send to wire/representor

Domain	Description
`DOCA_FLOW_PIPE_DOMAIN_SECURE_EGRESS`	• Domain for secure actions on egress traffic • Decapsulation actions are not allowed here • The only allowed domain for encrypting secure action • The next milestone is wire/representor • Miss action is: Send to wire/representor • Memory may be optimized if set with `DOCA_FLOW_DIRECTION_HOST_TO_NETWORK` direction information

14.4.2.4.2 Domains in VNF Mode



14.4.2.4.3 Domains in Switch Mode



14.4.2.5 API

DOCA API is available through the NVIDIA DOCA Library APIs page.

> ⓘ The pkg-config (`*.pc` file) for the DOCA Flow library is `doca-flow` .

14.4.2.6 Flow Life Cycle

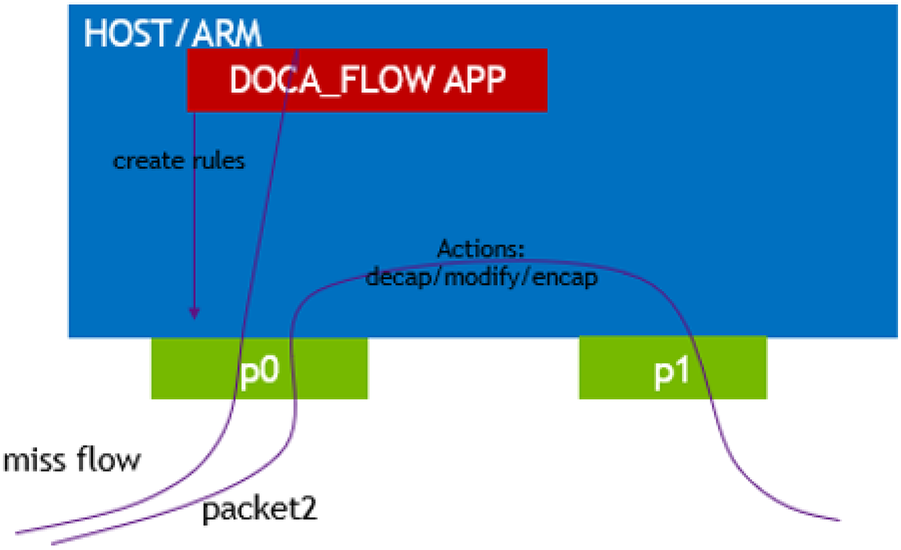
14.4.2.6.1 Initialization Flow

Before using any DOCA Flow function, it is mandatory to call DOCA Flow initialization, `doca_flow_init()` , which initializes all resources required by DOCA Flow.

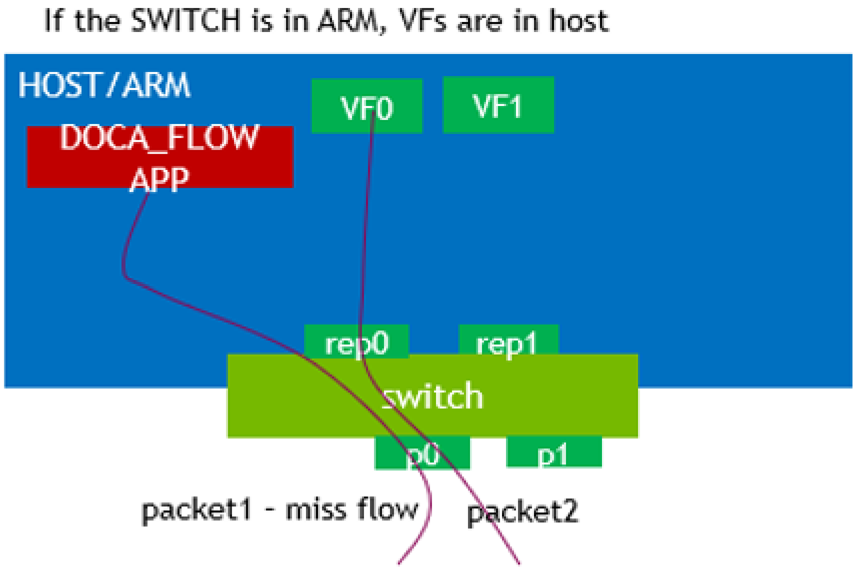
14.4.2.6.1.1 Pipe Mode

This mode (`mode_args`) defines the basic traffic in DOCA. It creates some miss rules when a DOCA port initializes. Currently, DOCA supports 3 modes:

- `vnf`
  A packet arriving from one of the device's ports is processed, and can be sent to another port. By default, missed packets go to RSS.
  The following diagram shows the basic traffic flow in `vnf` mode. Packet1 firstly misses and is forwarded to host RSS. The app captures this packet and decides how to process it and then creates a pipe entry. Packet2 will hit this pipe entry and do the action, for example, for VXLAN, will do decap, modify, and encap, then is sent out from P1.



- `switch`
  Used for internal switching, only representor ports are allowed, for example, uplink representors and SF/VF representors. Packet is forwarded from one port to another. If a packet arrives from an uplink and does not hit the rules defined by the user's pipe, then the packet is received on all RSS queues of the representor of the uplink.
  The following diagram shows the basic flow of traffic in `switch` mode. Packet1 firstly misses to host RSS queues. The app captures this packet and decides to which representor the packet goes, and then sets the rule. Packets hit this rule and go to representor0.



`doca_dev` field is mandatory in `doca_flow_port_cfg` (using `doca_flow_port_cfg_set_dev()`) and isolated mode should be specified.

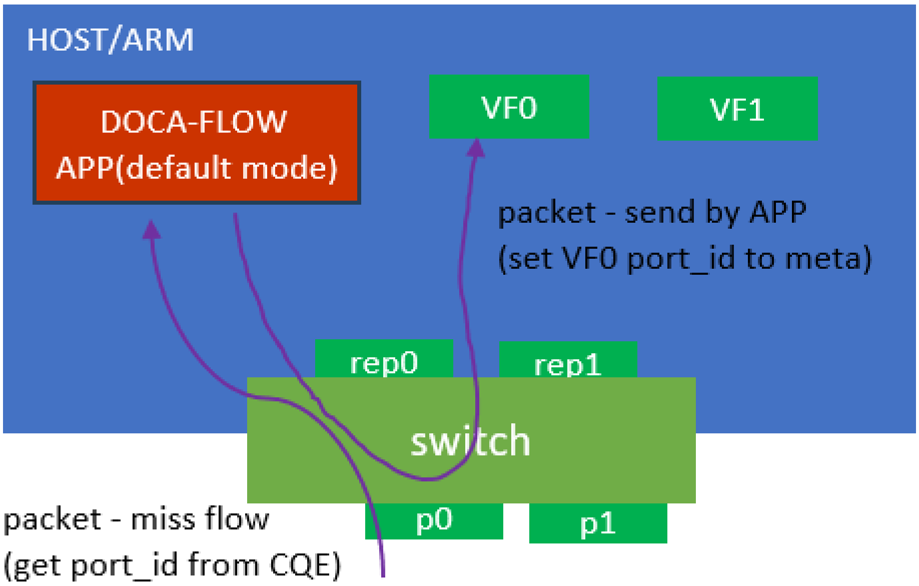
> The application must avoid initialization of the VF/SF representor ports in DPDK API (i.e., the following functions `rte_eth_dev_configure()` , `rte_eth_rx_queue_setup()` , `rte_eth_dev_start()` must not be called for VF/SF representor ports).

DOCA Flow switch mode unifies all the ports to the switch manager port for traffic management. This means that all the traffic is handled by switch manager port. Users only have to create an RSS pipe on the switch manager port to get the missed traffic, and they should only manage the pipes on the switch manager port. Switch mode can work with two different `mode_args` configurations: With or without `expert` . The way to retrieve the miss traffic source's `port_id` depends on this configuration:

> Only one RSS pipe is supported in switch mode, users can add multiple RSS pipe entries to that RSS pipe. Traffic missed from the user's pipe without a specified `fwd_miss` target is sent to the kernel if it is `isolated` mode, or sent to DOCA application (bypassing the kernel) if it is `non-isolated` (default) mode.

- If `expert` is not set, the traffic misses to software would be tagged with `port_id` information in the mbuf CQE field to allow users to deduce the source `port_id` . Meanwhile, users can set the destination `port_id` to mbuf meta and the packet is sent out directly to the destination port based on the meta information.

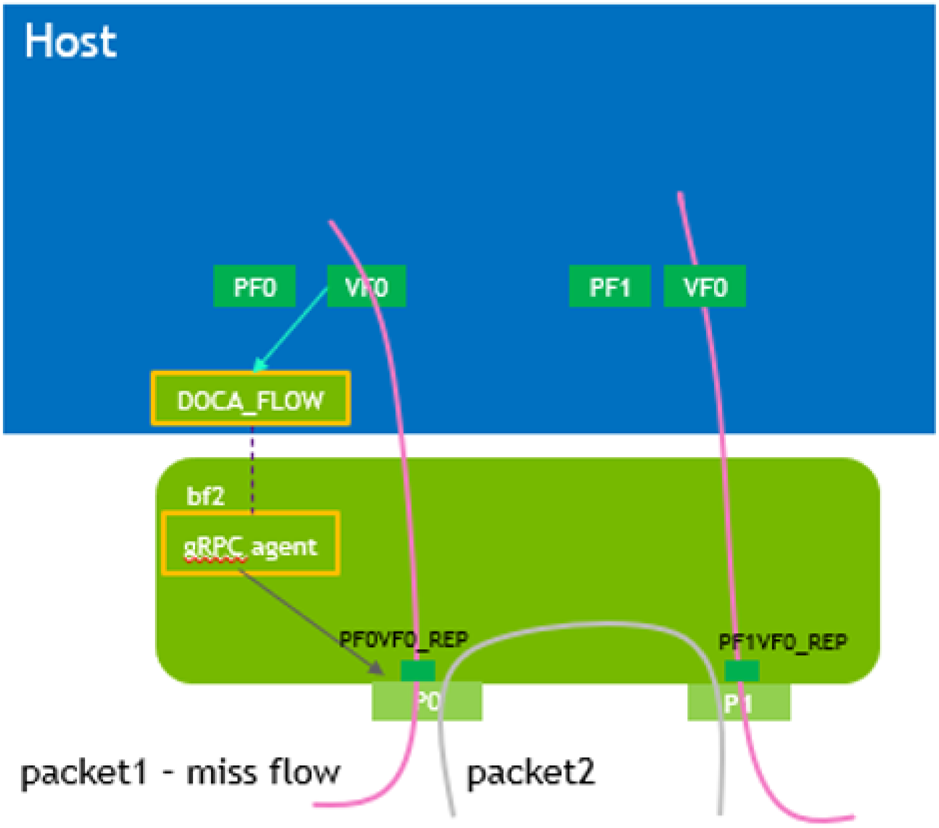
  > Please refer to the "Flow Switch to Wire" sample to get more information regarding the `port_id` management with missed traffic mbuf.



  - If `expert` is set, the `port_id` is not added to the packet. Users can configure the pipes freely to implement their own solution.

    > ⚠ Traffic cloned from the VF to the RSS pipe misses its `port_id` information due to firmware limitation.

- `remote-vnf`
  Remote mode is a BlueField mode only, with two physical ports (uplinks). Users must use `doca_flow_port_pair` to pair one physical port and one of its representors. A packet from this uplink, if it does not hit any rules from the users, is firstly received on this representor. Users must also use `doca_flow_port_pair` to pair two physical uplinks. If a packet is received from one uplink and hits the rule whose FWD action is to another uplink, then the packets are sent out from it.
  The following diagram shows the basic traffic flow in remote-vnf mode. Packet1, from BlueField uplink P0, firstly misses to host VF0. The app captures this packet and decides whether to drop it or forward it to another uplink (P1). Then, using gRPC to set rules on P0, packet2 hits the rule, then is either dropped or is sent out from P1.



14.4.2.6.2 Start Point

DOCA Flow API serves as an abstraction layer API for network acceleration. The packet processing in-network function is described from ingress to egress and, therefore, a pipe must be attached to the origin port. Once a packet arrives to the ingress port, it starts the hardware execution as defined by the DOCA API.

`doca_flow_port` is an opaque object since the DOCA Flow API is not bound to a specific packet delivery API, such as DPDK. The first step is to start the DOCA Flow port by calling `doca_flow_port_start()` . The purpose of this step is to attach user application ports to the DOCA Flow ports.

When DPDK is used, the following configuration must be provided:

```
enum doca_flow_port_type type = DOCA_FLOW_PORT_DPDK_BY_ID;
```

```
const char *devargs = "1";
```

The `devargs` parameter points to a string that has the numeric value of the DPDK `port_id` in decimal format. The port must be configured and started before calling this API. Mapping the DPDK port to the DOCA port is required to synchronize application ports with hardware ports.

14.4.2.6.3 Port Operation State

DOCA Flow ports can be initialized multiple times from different instances. Each instance prepares its pipeline, but only one actively receives port traffic at a time. The instance actively handling the port traffic depends on the operation state set by the `doca_flow_port_cfg_set_operation_state()` function:

- `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` - The instance actively handles incoming and outgoing traffic
- `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE_READY_TO_SWAP` - The instance handles traffic actively when no other active instance is available
- `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY` - The instance handles traffic only when no active or `active_ready_to_swap` instance is available
- `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED` - The instance does not handle traffic, regardless of the state of other instances

If the `doca_flow_port_cfg_set_operation_state()` function is not called, the default state `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` is applied.

⚠ When a port is configured with a state that expects to handle traffic, it takes effect only after root pipes are created for this port.

When the active port is closed, either gracefully or due to a crash, the standby instance automatically becomes active without any action required.

The port operation state can be modified after the port is started using the `doca_flow_port_operation_state_modify()` function.

14.4.2.6.3.1 Use Case Examples

Hot Upgrade

This operation state mechanism allows upgrading the DOCA Flow program without losing any traffic.

To upgrade an existing DOCA Flow program with ports started in `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` state (Instance A):

1. Open a new Instance B and start its ports in `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY` state.
2. Modify Instance A's ports from `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` to `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED` state. At this point, Instance B starts receiving traffic.
3. Close Instance A.
4. Open a new Instance C with `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED` state. Instance C is the upgraded version of Instance A.
5. Create the entire pipeline for Instance C.
6. Change Instance C's state from `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED` to `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE`. At this point, Instance B stops receiving traffic and Instance C starts.
7. Instance B can either be closed or kept as a backup should Instance C crash.

Swap Existing Instances

This mechanism also facilitates swapping two different DOCA Flow programs without losing any traffic.

To swap between two existing DOCA Flow programs with ports started in `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` and `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY` states (Instance A and Instance B, respectively):

1. Modify Instance A's ports from `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` to `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE_READY_TO_SWAP`.
2. Modify Instance B's ports from `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY` to `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE`. At this point, Instance B starts receiving traffic.
3. Modify Instance A's ports from `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE_READY_TO_SWAP` to `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY`.

14.4.2.6.3.2 Limitations

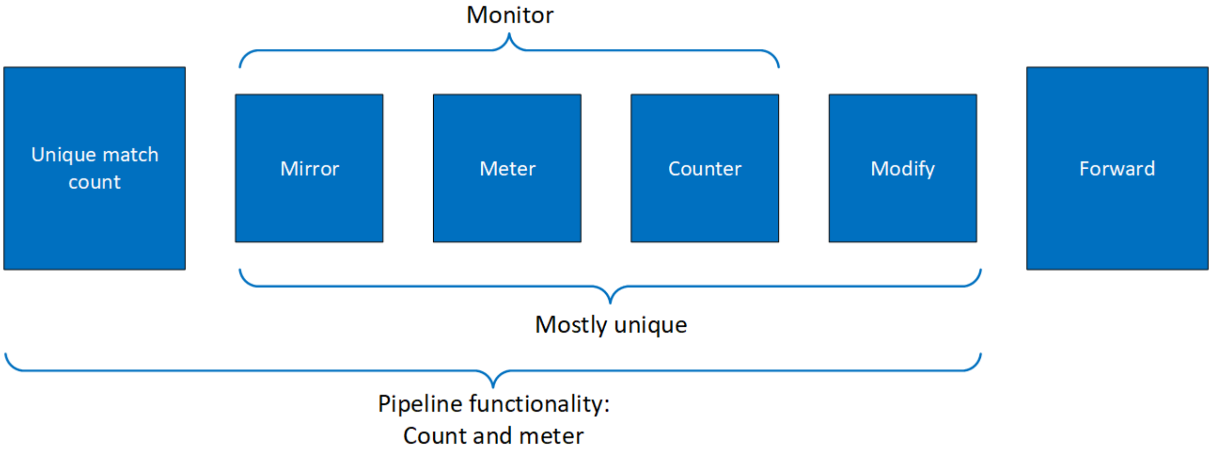
- Supported only in switch mode - the `mode_args` string must include `"switch"`.
- Only the switch port supports states; its representors are affected by its state. Starting a representor port or calling the modify function with a non-active operation state should fail.
- Two instances cannot be in the same operation state simultaneously, except for `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED`.

14.4.2.6.4 Create Pipe and Pipe Entry

Pipe is a template that defines packet processing without adding any specific hardware rule. A pipe consists of a template that includes the following elements:

- Match
- Monitor
- Actions
- Forward

The following diagram illustrates a pipe structure.



The creation phase allows the hardware to efficiently build the execution pipe. After the pipe is created, specific entries can be added. A subset of the pipe may be used (e.g., skipping the monitor completely, just using the counter, etc).

14.4.2.6.4.1 Pipe Matching or Action Applying

DOCA Flow allows defining criteria for matching on a packet or for taking actions on a matched packet by modifying it. The information defining these criteria is provided through the following pointers:

- Match or action pointer – given at pipe or entry creation
- Mask pointer – optionally given at pipe creation

Defining criteria for matching or actions on a packet can be done at the pipe level, where it applies to all packets of a pipe, or specified on a per entry basis, where each entry defines the operation on either the match, actions, or both.

In DOCA Flow terminology, when a field is identified as CHANGEABLE at pipe creation, this means that the actual criterion of the field is deferred to entry creation. Different entries can provide different criteria for a CHANGEABLE field.

A match or action field can be categorized, during pipe creation, as one of the following:

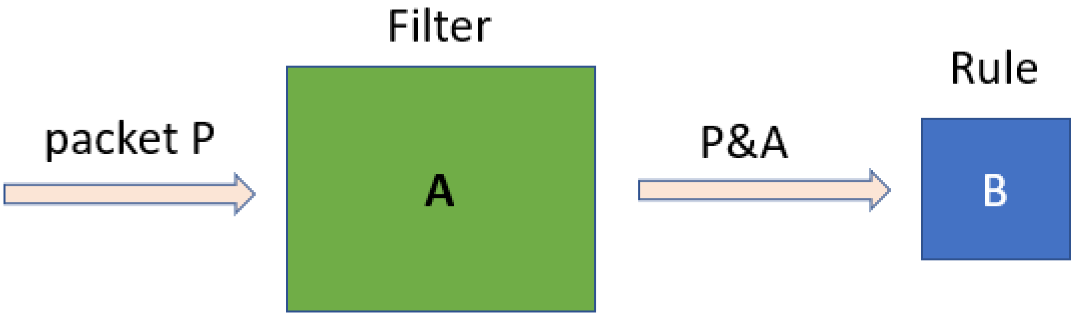
- IGNORED – Ignored in either the match or action taking process
- CHANGEABLE – When the actual behavior is deferred to the entry creation stage
- SPECIFIC – Value is used as is in either match or action process

A mask field can either be provided, in which case it is called it explicit matching, or action applying. If the mask pointer is NULL, we call it implicit matching or action applying. The following subsections provide the logic governing matching and action applying.

When a field value is specified as `0xffff` it means that all the field's bits are set (e.g., for TTL it means `0xff` and for IPv4 address it means `0xffffffff`).

Matching

Matching is the process of selecting packets based on their fields' values and steering them for further processing. Processing can either be further matching or actions applying.



The packet enters the green filter which modifies it by masking it with the value A. The output value, P&A, is then compared to the value B, and if they are equal, then that is a match.

The values of A and B are evaluated according to the values of the pipe configuration and entry configuration fields, according to the tables in sections "Implicit matching" and "Explicit matching".

Implicit Matching

Match Type	Pipe Match Value (V)	Pipe Match Mask (M)	Entry Match Value (E)	Filter (A)	Rule (B)
Ignore	0	NULL	N/A	0	0
Constant	0<V<0xffff	NULL	N/A	0xffff	V
Changeable (per entry)	0xffff	NULL	0≤E≤0xffff	0xffff	E

Explicit Matching

Match Type	Pipe Match Value (V)	Pipe Match Mask (M)	Entry Match Value (E)	Filter (A)	Rule (B)
Constant	V!=0xffff	0<M≤0xffff	0≤E≤0xffff	M	M&V
Changeable	V==0xffff	0<M≤0xffff	0≤E≤0xffff	M	M&E
Ignored	0≤V<0xffff	M==0	0≤E≤0xffff	0	0

Action Applying

Implicit Action Applying

Action Type	Pipe Action value (V)	Pipe Action Mask (M)	Entry Action value (E)	Action on the field
Ignore	0	NULL	N/A	none
Constant	0 < V < 0xffff	NULL	N/A	set to V
Changeable	0xffff	NULL	E	set to E

Implicit action applying example:

- Destination IPv4 address is 255.255.255.255
- No mask provided
- Entry value is 192.168.0.1
- Result - The action field is changeable. Therefore, the value is provided by the entry. If a match on the packet occurs, the packet destination IPv4 address is changed to 192.168.0.1.

Explicit Action Applying

Assume P is packet's field value.

Action Type	Pipe Action value (V)	Pipe Action Mask (M)	Entry Action value (E)	Action on the field
constant	V!=0xffff	0≤M≤0xffff	0≤E≤0xffff	set to (~M & P) | (M & V) In words: modify only bits that are set on the mask to the values in V
Changeable	V==0xffff	0<M≤0xffff	0≤E≤0xffff	set to (~M & P) | (M & E)
Ignored	0≤V<0xffff	M==0	0≤E≤0xffff	none

Explicit action applying example:

- Destination IPv4 address is 192.168.10.1
- Mask is provided and equals 255.255.0.0
- Entry value is ignored
- Result - If a match on the packet occurs, the packet destination IPv4 value changes to 192.168.0.0.

14.4.2.6.4.2 Setting Pipe Match or Action

Match is a mandatory parameter when creating a pipe. Using the `doca_flow_match` struct, users must define the packet fields to be matched by the pipe.

For each `doca_flow_match` field, users select whether the field type is:

- Ignore (match any) - the value of the field is ignored in a packet. In other words, match on any value of the field.
- Constant (specific) - all entries in the pipe have the same value for this field. Users should not put a value for each entry.
- Changeable - the value of the field is defined per entry. Users must provide it upon adding an entry.

  > ⚠ L4 type, L3 type, and tunnel type cannot be changeable.

The match field type can be defined either implicitly or explicitly using the `doca_flow_pipe_cfg_set_match(struct doca_flow_pipe_cfg *cfg, const doca_flow_match *match, const doca_flow_match *match_mask)` function. If `match_mask == NULL`, then it is done implicitly. Otherwise, it is explicit.

In the tables in the following subsections, an example is used of a 16-bit field (such as layer-4 destination port) where:

> ⚠ The same concept would apply to any other field (such as an IP address occupying 32 bits).

- P stands for the packet field value
- V stands for the pipe match field value
- M stands for the pipe mask field value
- E stands for the match entry field value

Implicit Match

Match Type	Pipe Match Value (V)	Pipe Match Mask (M)	Entry Match Value (E)	Filter (A)	Rule (B)
Ignore	0	NULL	N/A	0	0
Constant	0<V<0xffff	NULL	N/A	0xffff	V
Changeable (per entry)	0xffff	NULL	0≤E≤0xffff	0xffff	E

To match implicitly, the following considerations should be taken into account.

- Ignored fields:
    - Field is zeroed
    - Pipeline has no comparison on the field
- Constant fields – These are fields that have a constant value among all entries. For example, as shown in the following, the tunnel type is VXLAN:

  ```
  match.tun.type = DOCA_FLOW_TUN_VXLAN;
  ```

  These fields must only be configured once at pipe build stage, not once per new pipeline entry.
- Changeable fields – These are fields whose value may change per entry. For example, the following shows match on a destination IPv4 address of variable per-entry value (outer 5-tuple):

  ```
  match.outer.ip4.dst_ip = 0xffffffff;
  ```
- The following is an example of a match, where:
    - Outer 5-tuple
        - L3 type is IPv4 – constant among entries by design
        - L4 type is UDP – constant among entries by design
        - Tunnel type is `DOCA_FLOW_TUN_VXLAN` – constant among entries by design
        - IPv4 destination address varies per entry
        - UDP destination port is always `DOCA_VXLAN_DEFAULT_PORT`
        - VXLAN tunnel ID varies per entry
        - The rest of the packet fields are ignored
    - Inner 5-tuple
        - L3 type is IPv4 – constant among entries by design
        - L4 type is TCP – constant among entries by design
        - IPv4 source and destination addresses vary per entry
        - TCP source and destination ports vary per entry
        - The rest of the packet fields are ignored

```
// filter creation
static void build_underlay_overlay_match(struct doca_flow_match *match)
{
    //outer
    match->outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
    match->outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;
    match->tun.type = DOCA_FLOW_TUN_VXLAN;
    match->outer.ip4.dst_ip = 0xffffffff;
    match->outer.udp.l4_port.dst_port = DOCA_VXLAN_DEFAULT_PORT;
    match->tun.vxlan_tun_id = 0xffffffff;

    //inner
    match->inner.l3_type = DOCA_FLOW_L3_TYPE_IP4;
    match->inner.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_TCP;
    match->inner.ip4.dst_ip = 0xffffffff;
    match->inner.ip4.src_ip = 0xffffffff;
    match->inner.tcp.l4_port.src_port = 0xffff;
    match->inner.tcp.l4_port.dst_port = 0xffff;
}

// create entry specifying specific values to match upon
doca_error_t add_entry(struct doca_flow_pipe *pipe, struct doca_flow_port *port,
                       struct doca_flow_pipe_entry **entry)
{
    struct doca_flow_match match = {};
    struct entries_status status = {};
    doca_error_t result;

    match.outer.ip4.dst_ip = BE_IPV4_ADDR(7, 7, 7, 1);
    match.tun.vxlan_tun_id = RTE_BE32(9876);
    match.inner.ip4.src_ip = BE_IPV4_ADDR(8, 8, 8, 1);
    match.inner.ip4.dst_ip = BE_IPV4_ADDR(9, 9, 9, 1);
```

```
        match.inner.tcp.l4_port.src_port = rte_cpu_to_be_16(5678);
        match.inner.tcp.l4_port.dst_port = rte_cpu_to_be_16(1234);
        result = doca_flow_pipe_add_entry(0, pipe, &match, &actions, NULL, NULL, 0, &status, entry);
}
```

The fields of the `doca_flow_meta` struct inside the match are not subject to implicit match rules and must be paired with explicit mask values.

Explicit Match

Match Type	Pipe Match Value (V)	Pipe Match Mask (M)	Entry Match Value (E)	Filter (A)	Rule (B)
Constant	V!=0xffff	0<M≤0xffff	0≤E≤0xffff	M	M&V
Changeable	V==0xffff	0<M≤0xffff	0≤E≤0xffff	M	M&E
Ignored	0≤V<0xffff	M==0	0≤E≤0xffff	0	0

In this case, there are two `doca_flow_match` items, the following considerations should be considered:

- Ignored fields
  - M equals zero. This can be seen from the table where the rule equals 0. Since mask is also 0, the resulting packet after the filter is0. Thus, the comparison always succeeds.

```
match_mask.inner.ip4.dst_ip = 0;
```

- Constant fields
  These are fields that have a constant value. For example, as shown in the following, the inner 5-tuple match on IPv4 destination addresses belonging to the `0.0.0.0/24` subnet, and this match is constant among all entries:

```
// BE_IPV4_ADDR converts 4 numbers A,B,C,D to a big endian representation of IP address A.B.C.D
match.inner.ip4.dst_ip = 0;
match_mask.inner.ip4.dst_ip = BE_IPV4_ADDR(255, 255, 255, 0);
```

  For example, as shown in the following, the inner 5-tuple match on IPv4 destination addresses belonging to the `1.2.0.0/16` subnet, and this match is constant among all entries. The last two octets of the `match.inner.ip4.dst_ip` are ignored because the `match_mask` of `255.255.0.0` is applied:

```
// BE_IPV4_ADDR converts 4 numbers A,B,C,D to a big endian representation of IP address A.B.C.D
match.inner.ip4.dst_ip = BE_IPV4_ADDR(1, 2, 3, 4);
match_mask.inner.ip4.dst_ip = BE_IPV4_ADDR(255, 255, 0, 0);
```

  Once a field is defined as constant, the field's value cannot be changed per entry.

  Users should set constant fields to zero when adding entries for better code readability.

  A more complex example of constant matches may be achieved as follows:

```
match_mask.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(0xf0f0);
```

```
match.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(0x5020)
```

The following ports would be matched:

- 0x5020 - 0x502f
- 0x5120 - 0x512f
- ...
- 0x5f20 - 0x5f2f

Changeable fields

The following example matches on either FTP or TELNET well known port numbers and forwards packets to a server after modifying the destination IP address and destination port numbers. In the example, either FTP or TELNET are forwarded to the same server. FTP is forwarded to port 8000 and TELNET is forwarded to port 9000.

```
// at Pipe creation
pipe_cfg.attr.name = "PORT_MAPPER";
pipe_cfg.attr.type = DOCA_FLOW_PIPE_BASIC;
match.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(0xffff); // v
match_mask.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(0xffff); // M
pipe_cfg.match_mask = &match_mask;
pipe_cfg.match = &match;
actions_arr[0] = &actions;
pipe_cfg.actions = actions_arr;
pipe_cfg.attr.is_root = true;
pipe_cfg.attr.nb_actions = 1;

// Adding entries
// FTP
match.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(20); // E
actions.outer.ip4.src_ip = server_addr;
actions.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(8000);
result = doca_flow_pipe_add_entry(0, pipe, &match, &actions, NULL, NULL, 0, &status, entry);

// TELNET
match.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(23); // E
actions.outer.ip4.src_ip = server_addr;
actions.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(9000);
result = doca_flow_pipe_add_entry(0, pipe, &match, &actions, NULL, NULL, 0, &status, entry);
```

14.4.2.6.4.3 Relaxed Match

Relaxed matching is the default working mode in DOCA flow. However, it can be disabled per pipe using the `enable_strict_matching` pipe attribute. This mode grants the user more control on matching fields such that only explicitly set match fields by the user (either specific or changeable) are matched by the pipe.

Consider the following strict matching mode example. There are three pipes:

- Basic pipe `A` with `match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_TCP;` and `match.outer.tcp.flags = 1;`
- Basic pipe `B` with `match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;` and `match.outer.udp.l4_port.src_port = 8080;`
- Control pipe `X` with two entries to direct TCP traffic to pipe `A` and UDP to pipe `B`. The first entry has `match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_TCP;` while the second has `match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;`.

As a result, the hardware performs match on the L4 header type twice:

- First, when the packet enters the filter in control pipe `X` to decide the next pipe
- Second, when the packet enters the filter of pipe `A` or pipe `B` to do the match on L4 header fields

With particularly large pipelines, such double matches decrease performance and increase the memory footprint in hardware. Relaxed matching mode gives the user greater control of the match to solve the performance problems.

In relaxed mode, type selectors in the `outer`, `inner`, and `tun` parts of the `doca_flow_match` are used only for the type cast (or selectors) of the underlying unions. Header-type matches are available using the `parser_meta` API.

Thus, the aforementioned scenario may be overwritten in the following manner. There are three pipes:

- Basic pipe `A` with `match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_TCP;` and `match.outer.tcp.flags = 1;`
- Basic pipe `B` with `match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;` and `match.outer.udp.l4_port.src_port = 8080;`
- Control pipe `X` with two entries to direct TCP traffic to pipe `A` and UDP to pipe `B`. The first entry has `match.parser_meta.outer_l4_type = DOCA_FLOW_L4_META_TCP;` while the second has `match.parser_meta.outer_l4_type = DOCA_FLOW_L4_META_UDP;`.

As a result, the hardware performs the L4 header-type match only once, when the packet enters the filter of control pipe. Basic pipes' `match.outer.l4_type_ext` are used only for the selection of the `match.outer.tcp` or `match.outer.udp` structures.

Example

The following code snippet is used to demonstrate relaxed matching mode:

```
// filter creation
static void build_underlay_overlay_match(struct doca_flow_match *match)
{
    //outer
    match->outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
    match->outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;
    match->tun.type = DOCA_FLOW_TUN_VXLAN;
    match->outer.ip4.dst_ip = 0xffffffff;
    match->outer.udp.l4_port.src_port = 22;
    match->tun.vxlan_tun_id = 0xffffffff;
}
```

This match code above is an example of a match where:

- With relaxed matching disabled (i.e., enable_strict_matching attribute set to `true`), the following hardware matches are performed:
    - L3 type is IPv4 - constant among entries by design
    - L4 type is UDP - constant among entries by design
    - Tunnel type is `DOCA_FLOW_TUN_VXLAN` - constant among entries by design
    - IPv4 destination address varies per entry
    - UDP source port is constant among entries
    - VXLAN tunnel ID varies per entry
    - The rest of the packet fields are ignored
- With relaxed matching enabled (default mode), the following hardware matches are performed:
    - IPv4 destination address varies per entry
    - UDP source port is constant among entries
    - VXLAN tunnel ID varies per entry

In summary, with relaxed matching L3, L4, tunnel protocol types, and similar no longer indicate a match on the specific protocol. They are used solely as a selector for the relevant header fields. For example, to match on `outer.ip4.dst_ip`, users must set `outer.l3_type = DOCA_FLOW_L3_TYPE_IP4`. That is, the L3 header is checked for the IPv4 destination address. There is no check that it is of IPv4 type. It is user responsibility to make sure that packets arriving to such a filter indeed have an L3 header of type IPv4 (same goes for L4 UDP header/VXLAN tunnel).

Protocols/Tunnels Type Match

The following section explains how to match on a protocol's and a tunnel's type with relaxed matching.

To match on a specific protocol/tunnel type, consider the following:

- To match on an inner/outer L3/L4 protocol type, one can use relevant `doca_flow_parser_meta` fields (e.g., for outer protocols, `parser_meta.outer_l[3,4]_type` fields can be used).
- To match on a specific tunnel type (e.g., VXLAN/GRE and so on), users should match on a tunnel according to its specification (e.g., for VXLAN, a match on UDP destination port 4789 can be used). Another option is to use the L3 next protocol field (e.g., for IPv4 with next header GRE, one can match on the IPv4 header's next protocol field value to match GRE IP protocol number 47).

Example

Using the aforementioned example, to add the match on the same L3,L4 protocol type and on a VXLAN tunnel with relaxed matching enabled, the following function implementation should be considered:

```
// filter creation
static void build_underlay_overlay_match(struct doca_flow_match *match)
{
    //outer
    match->parser_meta.outer_l3_type = DOCA_FLOW_L3_META_IPV4;
    match->parser_meta.outer_l4_type = DOCA_FLOW_L4_META_UDP;
    match->outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
    match->outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;
    match->tun.type = DOCA_FLOW_TUN_VXLAN;
    match->outer.ip4.dst_ip = 0xffffffff;
    match->outer.udp.l4_port.src_port = 22;
    match->outer.udp.l4_port.dst_port = DOCA_VXLAN_DEFAULT_PORT;
    match->tun.vxlan_tun_id = 0xffffffff;
}
```

The match code above is an example of a match, where:

- With relaxed matching disabled (i.e., enable_strict_matching attribute set to `true`), the following hardware matches are performed:
    - L3 type is IPv4 – constant among entries by design
    - L4 type is UDP - constant among entries by design
    - Tunnel type is `DOCA_FLOW_TUN_VXLAN` - constant among entries by design
    - IPv4 destination address varies per entry
    - UDP source port is always 22
    - UDP destination port is always `DOCA_VXLAN_DEFAULT_PORT`
    - VXLAN tunnel ID varies per entry
    - The rest of the packet fields are ignored
- With relaxed matching enabled (default mode), the following hardware matches are performed:

- L3 type is IPv4 - constant among entries by design
- L4 type is UDP - constant among entries by design
- IPv4 destination address varies per entry
- UDP source port is always 22
- UDP destination port is always `DOCA_VXLAN_DEFAULT_PORT`
- VXLAN tunnel ID varies per entry

> ⚠ With relaxed matching, if any of the selectors is used without setting a relevant field, the pipe/entry creation would fail with the following error message:
>
> ```
> failed building active opcode - active opcode <opcode number> is protocol only
> ```

14.4.2.6.4.4 Setting Pipe Actions

Pipe Execution Order

When setting actions, they are executed in the following order:

1. Crypto (decryption)
2. Decapsulation
3. Pop
4. Meta
5. Outer
6. Tun
7. Push
8. Encapsulation
9. Crypto (encryption)

> ⚠ Modifying a field while simultaneously using it as a source for other modifications should be avoided, as the sequence of modification actions cannot be guaranteed.

Auto-modification

Similarly to setting pipe match, actions also have a template definition.

Similarly to `doca_flow_match` in the creation phase, only the subset of actions that should be executed per packet are defined. This is done in a similar way to match, namely by classifying a field of `doca_flow_match` to one of the following:

- Ignored field - field is zeroed, modify is not used.
- Constant fields - when a field must be modified per packet, but the value is the same for all packets, a one-time value on action definitions can be used
- Changeable fields - fields that may have more than one possible value, and the exact values are set by the user per entry

  ```
  actions.outer.ip4.dst_ip = 0xffffffff
  ```

> The `action_mask` should be set as `0xffffffff` and action as 0 if the user wants to configure 0 to this field.

Explicit Modification Type

It is possible to force constant modification or per-entry modification with action mask. For example:

```
static void
create_constant_modify_actions(struct doca_flow_actions *actions,
                               struct doca_flow_actions *actions_mask,
                               struct doca_flow_action_descs *descs)
{
        actions->outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;
        actions->outer.udp.src_port = 0x1234;
        actions_mask->outer.udp.src_port = 0xffff;
}
```

Copy Field

The action descriptor can be used to copy between the packet field and metadata. For example:

```
#define META_U32_BIT_OFFSET(idx) (offsetof(struct doca_flow_meta, u32[(idx)]) << 3)

static void
create_copy_packet_to_meta_actions(struct doca_flow_match *match,
                                   struct doca_flow_action_desc *desc)
{
        desc->type = DOCA_FLOW_ACTION_COPY;
        desc->field_op.src.field_string = "outer.ipv4.src_ip";
        desc->field_op.src.bit_offset = 0;
        desc->field_op.dst.field_string = "meta.data";
        desc->field_op.dst.bit_offset = META_U32_BIT_OFFSET(1); /* Bit offset of meta.u32[1] */;
}
```

Multiple Actions List

Creating a pipe is possible using a list of multiple actions. For example:

```
static void
create_multi_actions_for_pipe_cfg()
{
    struct doca_flow_actions *actions_arr[2];
    struct doca_flow_actions actions_0 = {0}, actions_1 = {0};
    struct doca_flow_pipe_cfg *pipe_cfg;

    /* input configurations for actions_0 and actions_1 */

    actions_arr[0] = &actions_0;
    actions_arr[1] = &actions_1;
    doca_flow_pipe_cfg_set_actions(pipe_cfg, actions_arr, NULL, NULL, 2);
}
```

Summary of Action Types

Pipe Creation				Entry Creation	Behavior
action_desc		Pipe Actions	Pipe Actions Mask	Entry Actions	
doca_flow_action_type	Configuration				
`DOCA_FLOW_ACTION_AUTO`/ `action_desc = NULL`	No specific config	`0`	`0`	N/A	Field ignored, no modification
		`0`	`mask` != 0	N/A	Apply `0` and `mask` to all entries
		`val` != 0 && `val` != 0xFF	`mask` != 0	N/A	Apply `val` and `mask` to all entries
		`val` = 0xFF	`mask` = 0	N/A	Apply `0xFF` to all entries
		`val` = 0xFF	`mask` != 0	Define `val` per entry	Apply entry's `val` and `mask`
`DOCA_FLOW_ACTION_ADD` Add field value or from `src`	Define only the `dst` field and width	`val` != 0	N/A	N/A	Apply this `val` to all entries
		`val` == 0	N/A	Define `val` per entry	Apply entry's `val`
	Define the `src` and `dst` fields and width	Define the source and destination fields. • Meta field → header field • Header field → meta field • Meta field → meta field	N/A	N/A	Add data from `src` fields to `dst` for all entries
`DOCA_FLOW_ACTION_COPY` Copy field to another field	N/A	Define the source and destination fields. • Meta field → header field • Header field → meta field • Meta field → meta field	N/A	N/A	Copy data between fields for all entries

14.4.2.6.4.5 Setting Pipe Monitoring

If a meter policer should be used, then it is possible to have the same configuration for all policers on the pipe or to have a specific configuration per entry. The meter policer is determined by the FWD action. If an entry has NULL FWD action, the policer FWD action is taken from the pipe.

If a mirror should be used, mirror can be shared on the pipe or configured to have a specific value per entry.

The monitor also includes the aging configuration, if the aging time is set, this entry ages out if timeout passes without any matching on the entry.

For example:

```
static void build_entry_monitor(struct doca_flow_monitor *monitor, void *user_ctx)
{
    monitor->aging_sec = 10;
}
```

Refer to Pipe Entry Aged Query for more information.

14.4.2.6.4.6 Setting Pipe Forwarding

The FWD (forwarding) action is the last action in a pipe, and it directs where the packet goes next. Users may configure one of the following destinations:

- Send to software (representor)
- Send to wire
- Jump to next pipe
- Drop packets

The FORWARDING action may be set for pipe create, but it can also be unique per entry.

A pipe can be defined with constant forwarding (e.g., always send packets on a specific port). In this case, all entries will have the exact same forwarding. If forwarding is not defined when a pipe is created, users must define forwarding per entry. In this instance, pipes may have different forwarding actions.

When a pipe includes meter monitor `<cir, cbs>`, it must have `fwd` defined as well as the policer.

If a pipe is created with a dedicate constant mirror with FWD, the pipe FWD can be from a mirror FWD or a pipe FWD and the two FWDs are exclusive. It is not allowed to specify a mirror with a FWD to a pipe with FWD also.

If a mirror FWD is not configured, the FWD is from the pipe configuration. The FWD of the pipe with a mirror cannot be direct RSS, only shared RSS from NULL FWD is allowed.

The following is an RSS forwarding example:

```
fwd->type = DOCA_FLOW_FWD_RSS;
fwd->rss_queues = queues;
fwd->rss_flags = DOCA_FLOW_RSS_IP | DOCA_FLOW_RSS_UDP;
fwd->num_of_queues = 4;
```

Queues point to the `uint16_t` array that contains the queue numbers. When a port is started, the number of queues is defined, starting from zero up to the number of queues minus 1. RSS queue numbers may contain any subset of those predefined queue numbers. For a specific match, a packet may be directed to a single queue by having RSS forwarding with a single queue.

Changeable RSS forwarding is supported. When creating the pipe, the `num_of_queues` must be set to `0xffffffff`, then different forwarding RSS information can be set when adding each entry.

```
fwd->num_of_queues = 0xffffffff;
```

The packet is directed to the port. In many instances the complete pipe is executed in the hardware, including the forwarding of the packet back to the wire. The packet never arrives to the software.

Example code for forwarding to port:

```
struct doca_flow_fwd *fwd = malloc(sizeof(struct doca_flow_fwd));
memset(fwd, 0, sizeof(struct doca_flow_fwd));
fwd->type = DOCA_FLOW_FWD_PORT;
fwd->port_id = port_id; // this should the same port_id that was set in doca_flow_port_cfg_set_devargs()
```

The type of forwarding is `DOCA_FLOW_FWD_PORT` and the only data required is the `port_id` as defined in `DOCA_FLOW_PORT`.

Changeable port forwarding is also supported. When creating the pipe, the `port_id` must be set to `0xffff`, then different forwarding `port_id` values can be set when adding each entry.

```
fwd->port_id = 0xffff;
```

14.4.2.6.4.7 Shared Resources

DOCA Flow supports several types of resources that can be shared. The supported types of resources can be:

- Meters
- Counters
- RSS queues
- Mirrors
- PSPs
- Encap
- Decap
- IPsec SA

Shared resources can be used by several pipes and can save device and memory resources while promoting better performance.

To create and configure shared resource, the user should go through the steps detailed in the following subsections.

Creating Shared Resource Configuration Object

Call `doca_flow_cfg_create(&flow_cfg)`, passing a pointer to `struct doca_flow_cfg` to be used to fill the required parameters for the shared resource.

⚠ The `struct doca_flow_cfg` object is used for configuring other resources besides the aforementioned shared resources, but this section only refers to the configuration of shared resources.

Setting Number of Shared Resources per Shared Resource Type

This can be done by calling `doca_flow_cfg_set_nr_shared_resource()`. Refer to the API documentation for details on the configuration process.

Conclude the configuration by calling `doca_flow_init()`.

Configuring Shared Resource

When shared resources are allocated, they are assigned identifiers ranging from 0 and increasing incrementally. For example, if the user configures two shared counters, they would bear the identifiers 0 and 1.

⚠ Note that each resource has its own identifier space. So, if users have two shared counters and three meters, they would bear identifiers 0..1 and 0..2 respectively.

Configuring the shared resources requires the user to call `doca_flow_shared_resource_set_cfg()`.

Binding Shared Resource

A shared resource must be bound by calling `doca_flow_shared_resources_bind()` which binds the resource to a pointer. The object to which the resource is bound is usually a `struct doca_flow_port` pointer.

Using Shared Resources

After a resource has been configured, it can be used by referring to its ID.

In the case of meters, counters, and mirrors, they are referenced through `struct doca_flow_monitor` during pipe creation or entry addition.

Querying Shared Resource

Querying shared resources can be done by calling `doca_flow_shared_resources_query()`. The function accepts the resource type and an array of resource numbers, and returns an array of `struct doca_flow_shared_resource_result` with the results.

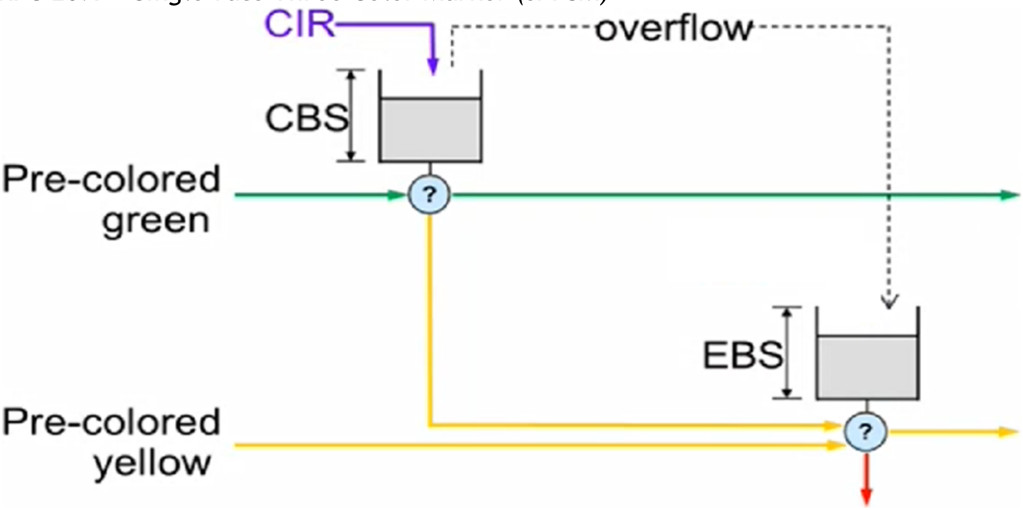
Shared Meter Resource

A shared meter can be used in multiple pipe entries (hardware steering mode support only).

The shared meter action marks a packet with one of three colors: Green, Yellow, and Red. The packet color can then be matched in the next pipe, and an appropriate action may be taken. For example, packets marked in red color are usually dropped. So, the next pipe to meter action may have an entry which matches on red and has fwd type `DOCA_FLOW_FWD_DROP`.

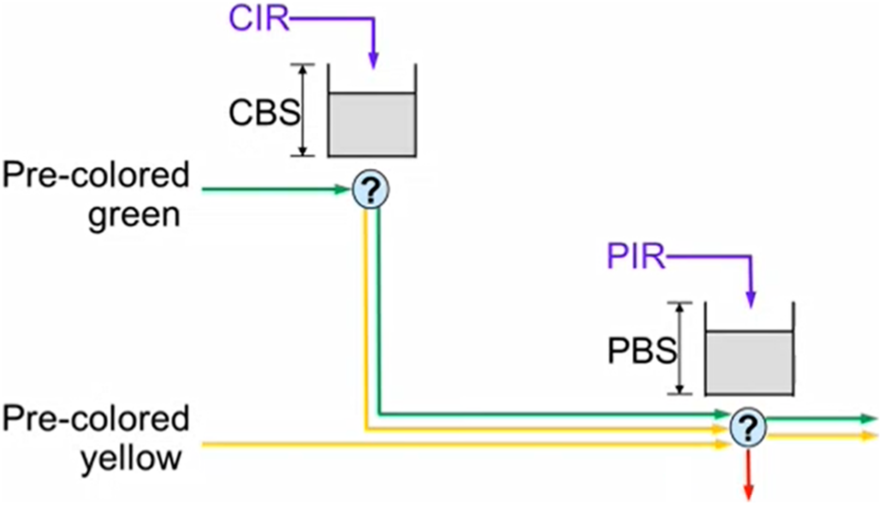
DOCA Flow supports three marking algorithms based on RFCs: 2697, 2698, and 4115.

RFC 2697 – Single-rate Three Color Marker (srTCM)



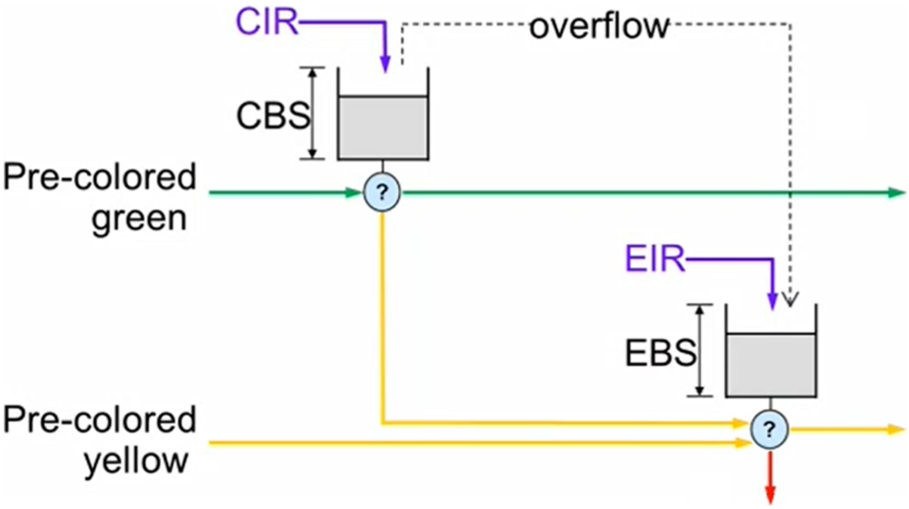
CBS (committed burst size) is the bucket size which is granted credentials at a CIR (committed information rate). If CBS overflow occurs, credentials are passed to the EBS (excess burst size) bucket. Packets passing through the meter consume credentials. A packet is marked green if it does not exceed the CBS, yellow if it exceeds the CBS but not the EBS, and red otherwise. A packet can have an initial color upon entering the meter. A pre-colored yellow packet will start consuming credentials from the EBS.

RFC 2698 – Two-rate Three Color Marker (trTCM)



CBS and CIR are defined as in RFC 2697. PBS (peak burst size) is a second bucket which is granted credentials at a PIR (peak information rate). There is no overflow of credentials from the CBS bucket to the PBS bucket. The PIR must be equal to or greater than the CIR. Packets consuming CBS credentials consume PBS credentials as well. A packet is marked red if it exceeds the PIR. Otherwise, it is marked either yellow or green depending on whether it exceeds the CIR or not. A packet can have an initial color upon entering the meter. A pre-colored yellow packet starts consuming credentials from the PBS.

RFC 4115 – trTCM without Peak-rate Dependency



EBS is a second bucket which is granted credentials at a EIR (excess information rate) and gets overflowed credentials from the CBS. For the packet marking algorithm, refer to RFC 4115.

The following sections present the steps for configuring and using shared meters to mark packets.

Shared IPsec SA Resource

The IPsec Security Association (SA) shared resource is used for IPsec ESP encryption protocol. The resource should be pointed from the `doca_flow_crypto_actions` struct that inside `doca_flow_actions`.

By default, the resource manages the state of the sequence number (SN), incrementing each packet on the encryption side, and performing anti-replay protection on the decryption side.

To control the SN in software, `sn_offload` should be disabled per port in the configuration for `doca_flow_port_start` (see DOCA API documentation for details). Once `sn_offload` is disabled, the following fields are ignored: `sn_offload_type`, `win_size`, `sn_initial`, and `lifetime_threshold`.

When shared resource query is called for an IPsec SA resource, the current SN is retrieved for the encryption resource and the lower bound of anti-replay window is retrieved for the decryption resource. Querying IPsec SA can only be called when `sn_offload` is enabled.

To maintain a valid state of the resource during its usage, `doca_flow_crypto_ipsec_resource_handle` should be called periodically.

Shared Mirror Resource

The mirror shared resource is used to clone packets to other pipes, vports (switch mode only), RSS queues (VNF mode only), or drop.

ⓘ The maximum supported mirror number is 4K.

ⓘ The maximum supported mirror clone destination is 254.

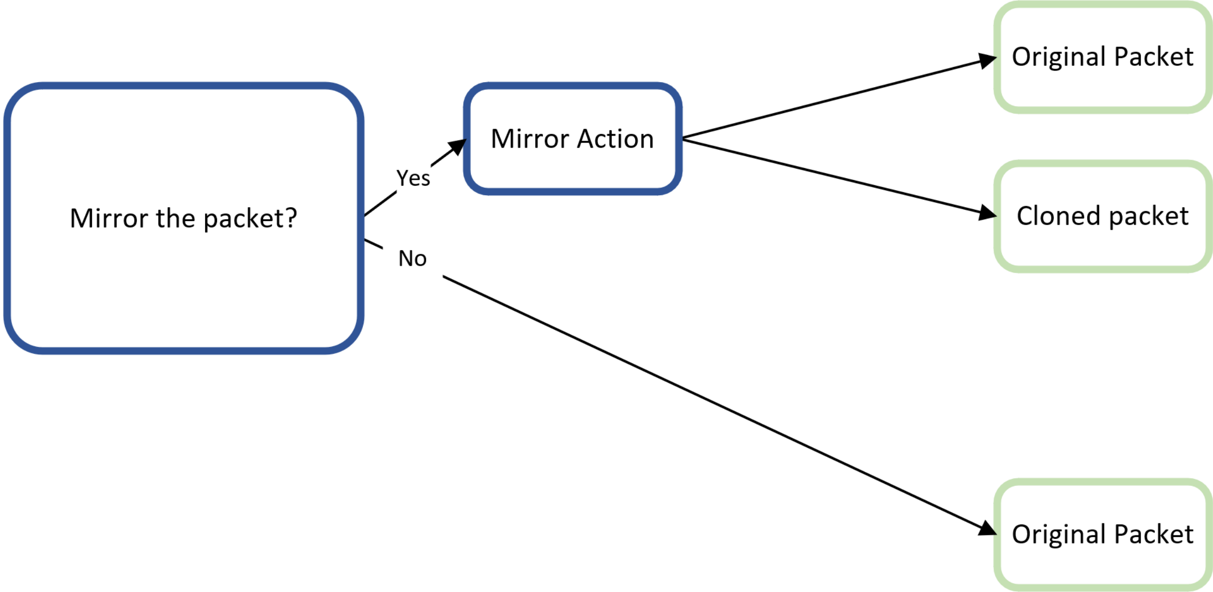
Mirror clone destination as `next_pipe` cannot be intermixed with `port` or `rss` types. Only clone destination and origin destination both as `next_pipe` is supported.

The register copy for packet after mirroring is not saved.

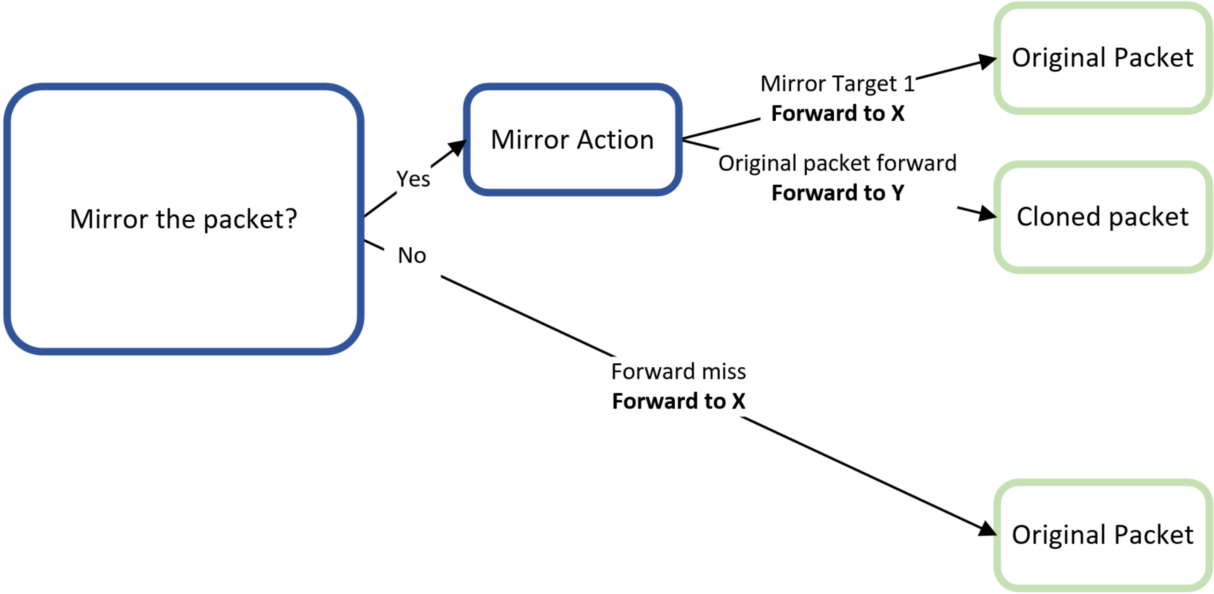
> ⚠ **Mirror limitations**
>
> For switch mode, there are several mirror limitations which should be noted:
>
> - Mirror should be cloned to `DOCA_FLOW_DIRECTION_BIDIRECTIONAL` pipe
> - The register copy for pkt after mirroring is not saved
> - Mirror should not be cloned to RSS pipe directly
> - Encap is supported while cloning a packet to a wire port only
> - Mirror must not be configured on a resizable pipe

If mirror creation fails, users should check the resulting syndrome for failure details.

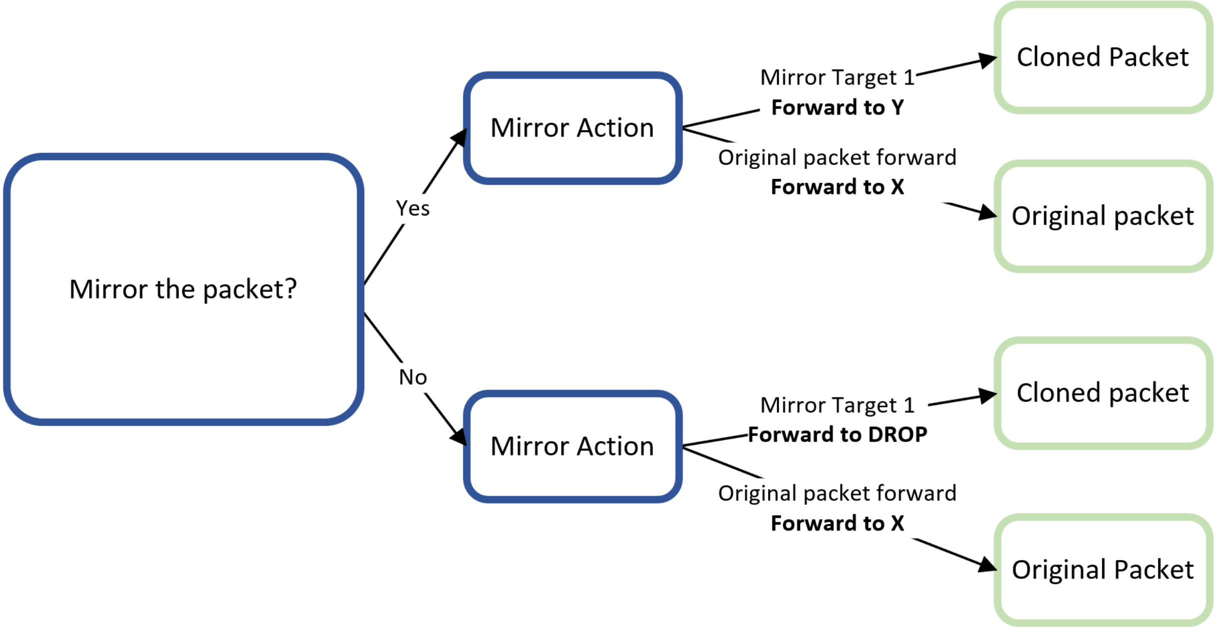
Mirroring and Packet Order



To maintain the order of the mirrored packets in relation to the non-mirrored ones, set a first mirror target forward destination equivalent to the non-mirrored packets as illustrated in the following diagram:



In NVIDIA® BlueField®-3, NVIDIA® ConnectX®-7, and lower, when using the mirror action in the egress domain, mirrored packets cannot preserve the order with the non-mirrored packets due to the high latency of the mirror operation. To maintain the order, use `DOCA_FLOW_FWD_DROP` as the target forward as illustrated in the following diagram:



Shared Encap Resource

The encap shared resource is used for encapsulation. A shared encap ID represents one kind of encap configuration and can be used in multiple pipes and entries (hardware steering mode support only).

The shared encap action encapsulates the packet with the configured tunnel information.

Shared Decap Resource

The decap shared resource is used for decapsulation. A shared decap ID represents one kind of decap configuration and can be used in multiple pipes and entries (hardware steering mode support only).

The shared decap action decapsulates the packet. Ethernet information should be provided when is_l2 is false.

Shared PSP Resource

The PSP shared resource is used for PSP encryption. The resource should be pointed to from the `doca_flow_crypto_actions` struct in `doca_flow_actions`.

The resource should be configured with a key to encrypt the packets. See NVIDIA DOCA Library API documentation for PSP key generation for a reference about key handling on decrypt side.

14.4.2.6.4.8 Basic Pipe Create

Once all parameters are defined, the user should call `doca_flow_pipe_create` to create a pipe.

The return value of the function is a handle to the pipe. This handle should be given when adding entries to pipe. If a failure occurs, the function returns `NULL`, and the error reason and message are put in the error argument if provided by the user.

Refer to the NVIDIA DOCA Library APIs to see which fields are optional and may be skipped. It is typically recommended to set optional fields to 0 when not in use. See Miss Pipe and Control Pipe for more information.

Once a pipe is created, a new entry can be added to it. These entries are bound to a pipe, so when a pipe is destroyed, all the entries in the pipe are removed. Please refer to section Pipe Entry for more information.

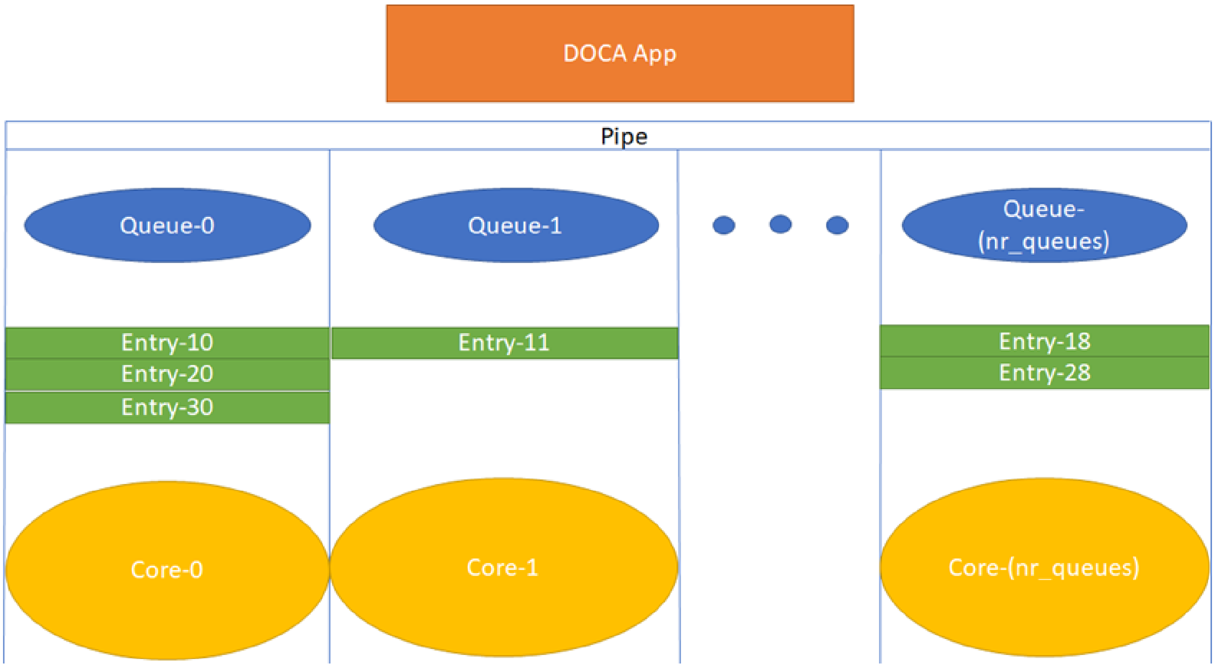
There is no priority between pipes or entries. The way that priority can be implemented is to match the highest priority first, and if a miss occurs, to jump to the next PIPE. There can be more than one PIPE on a root as long the pipes are not overlapping. If entries overlap, the priority is set according to the order of entries added. So, if two pipes have overlapping matching and PIPE1 has higher priority than PIPE2, users should add an entry to PIPE1 after all entries are added to PIPE2.

14.4.2.6.4.9 Pipe Entry (doca_flow_pipe_add_entry)

An entry is a specific instance inside of a pipe. When defining a pipe, users define match criteria (subset of fields to be matched), the type of actions to be done on matched packets, monitor, and, optionally, the FWD action.

When a user calls `doca_flow_pipe_add_entry()` to add an entry, they should define the values that are not constant among all entries in the pipe. And if FWD is not defined then that is also mandatory.

DOCA Flow is designed to support concurrency in an efficient way. Since the expected rate is going to be in millions of new entries per second, it is mandatory to use a similar architecture as the data path. Having a unique queue ID per core saves the DOCA engine from having to lock the data structure and enables the usage of multiple queues when interacting with hardware.



Each core is expected to use its own dedicated `pipe_queue` number when calling `doca_flow_pipe_entry`. Using the same `pipe_queue` from different cores causes a race condition and has unexpected results.

> ⚠ Applications are expected to avoid adding, removing, or updating pipe entries from within a `doca_flow_entry_process_cb`.

Failure Path

Entry insertion can fail in two places, `add_entry` and `add_entry_cb`.

- When `add_entry` fails, no cleanup is required.
- When `add_entry` succeeds, a handle is returned to the user. If the subsequent `add_entry_cb` fails, the user is responsible for releasing the handle through a `rm_entry` call. This `rm_entry` call is expected to return `DOCA_SUCCESS` and is expected to invoke `doca_rm_entry_cb` with a successful return code.

Pipe Entry Counting

By default, no counter is added. If defined in monitor, a unique counter is added per entry.

> ⚠ Having a counter per entry affects performance and should be avoided if it is not required by the application.

The retrieved statistics are stored in struct `doca_flow_query`.

Pipe Entry Aged Query

When a user calls `doca_flow_aging_handle()`, this query is used to get the aged-out entries by the time quota in microseconds. The user callback is invoked by this API with the aged entries.

Since the number of flows can be very large, the query of aged flows is limited by a quota in microseconds. This means that it may return without all flows and requires the user to call it again. When the query has gone over all flows, a full cycle is done.

14.4.2.6.4.10 Pipe Entry With Multiple Actions

Users can define multiple actions per pipe. This gives the user the option to define different actions per entry in the same pipe by providing the `action_idx` in `struct doca_flow_actions`.

For example, to create two flows with the same match but with different actions, users can provide two actions upon pipe creation, `Action_0` and `Action_1`, which have indices 0 and 1 respectively in the actions array in the pipe configuration. `Action_0` has `modify_mac`, and `Action_1` has `modify_ip`.

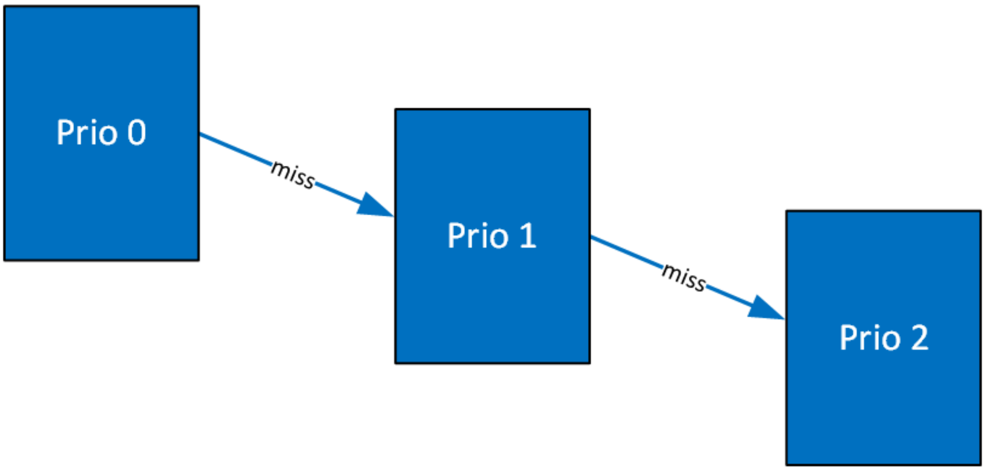
Users can also add two kinds of entries to the pipe, the first one with `Action_0` and the second with `Action_1`. This is done by assigning 0 in the `action_idx` field in `struct doca_flow_actions` when creating the first entry and 1 when creating the second one.

14.4.2.6.4.11 Miss Pipe and Control Pipe

> ⚠ Only one root pipe is allowed. If more than one is needed, create a control pipe as root and forward the packets to relevant non-root pipes.

To set priority between pipes, users must use miss-pipes. Miss pipes allow to look up entries associated with pipe X, and if there are no matches, to jump to pipe X+1 and perform a lookup on entries associated with pipe X+1.

The following figure illustrates the hardware table structure:



The first lookup is performed on the table with priority 0. If no hits are found, then it jumps to the next table and performs another lookup.

The way to implement a miss pipe in DOCA Flow is to use a miss pipe in FWD.
In struct `doca_flow_fwd`, the field `next_pipe` signifies that when creating a pipe, if a `fwd_miss` i s configured then if a packet does not match the specific pipe, steering should jump to `next_pipe` i n `fwd_miss`.

> `fwd_miss` is of type `struct doca_flow_fwd` but it only implements two forward types of this struct:
>
> - `DOCA_FLOW_FWD_PIPE` - forwards the packet to another pipe
> - `DOCA_FLOW_FWD_DROP` - drops the packet
>
> Other forwarding types (e.g., forwarding to port or sending to RSS queue) are not supported.

`next_pipe` is defined as `doca_flow_pipe` and created by `doca_flow_pipe_create`. To separate `miss_pipe` and a general one, `is_root` is introduced in struct `doca_flow_pipe_cfg`. If `is_root` is true, it means the pipe is a root pipe executed on packet arrival. Otherwise, the pipe is `next_pipe`.

When `fwd_miss` is not null, the packet that does not match the criteria is handled by `next_pipe` which is defined in `fwd_miss`.

In internal implementations of `doca_flow_pipe_create`, if `fwd_miss` is not null and the forwarding action type of `miss_pipe` is `DOCA_FLOW_FWD_PIPE`, a flow with the lowest priority is created that always jumps to the group for the `next_pipe` of the `fwd_miss`. Then the flow of `next_pipe` can handle the packets, or drop the packets if the forwarding action type of `miss_pipe` is `DOCA_FLOW_FWD_DROP`.

For example, VXLAN packets are forwarded as RSS and hairpin for other packets. The `miss_pipe` is for the other packets (non-VXLAN packets) and the match is for general Ethernet packets. The `fwd_miss` is defined by `miss_pipe` and the type is `DOCA_FLOW_FWD_PIPE`. For the VXLAN pipe, it is created by `doca_flow_create()` and `fwd_miss` is introduced.

Since, in the example, the jump flow is for general Ethernet packets, it is possible that some VXLAN packets match it and cause conflicts. For example, VXLAN flow entry for `ipA` is created. A VXLAN packet with `ipB` comes in, no flow entry is added for `ipB`, so it hits `miss_pipe` and is hairpinned.

A control pipe is introduced to handle the conflict. After creating a control pipe, the user can add control entries with different matches, forwarding, and priorities when there are conflicts.

The user can add a control entry by calling `doca_flow_control_pipe_add_entry()`.

`priority` must be defined as higher than the lowest priority (3) and lower than the highest one (0).

The other parameters represent the same meaning of the parameters in `doca_flow_pipe_create`. In the example above, a control entry for VXLAN is created. The VLXAN packets with `ipB` hit the control entry.

14.4.2.6.4.12 doca_flow_pipe_lpm

`doca_flow_pipe_lpm` uses longest prefix match (LPM) matching. LPM matching is limited to a single field of the `match` provided by the user at pipe creation (e.g., the outer destination IP). Each entry is consisted of a value and a mask (e.g., 10.0.0.0/8, 10.10.0.0/16, etc). The LPM match is defined as the entry that has the maximum matching bits. For example, using the two entries

10.7.0.0/16 and 10.0.0.0/8, the IP 10.1.9.2 matches on 10.0.0.0/8 and IP 10.7.9.2 matches on 10.7.0.0/16 because 16 bits are the longest prefix matched.

In addition to the longest prefix match logic, LPM supports exact match (EM) logic on the `meta.u32`, inner destination MAC and VNI. Only index `1` is supported for `meta.u32`. Any combination of these three fields can be chosen for EM. However, if inner destination MAC is chosen for LPM, then it should not be chosen for EM as well. If more than one field is chosen for EM, a logical AND is applied. Support for EM on meta allows working with any single field by copying its value to the `meta.u32[1]` on pipes before LPM. EM is performed at the same time as LPM matching (i.e., a logical AND is applied for both logics). For example, if there is a match on LPM logic, but the value in the fields chosen for EM is not exactly matched, this constitutes an LPM pipe miss.

To enable EM logic in an LPM pipe, two steps are required:

1. Provide `match_mask` to the LPM pipe creation with `meta.u32[1]` being fully masked and/or `inner.eth.dst_mac` and/or `tun.vxlan_tun_id`, while setting `match_mask.tun.type` to `DOCA_FLOW_TUN_VXLAN`. Thus, the `match` parameter is responsible for the choice of field for LPM logic, while the `match_mask` parameter is responsible for the enablement of EM logic. Separation into two parameters is done to distinguish which field is for LPM logic and which is for EM logic, when both fields can be used for LPM (e.g., destination IP address and source MAC address).
2. Per entry, provide values to do exact match using the `match` structure. `match_mask` is used only for LPM-related masks and is not involved into EM logic.

EM logic allows inserting many entries with different meta values for the same pair of LPM-related data. Regarding IPv4-based LPM logic with exact match enabled: LPM pipe can have 1.1.1.1/32 with `meta` 42, 555, and 1020. If a packet with 1.1.1.1/32 goes through such an LPM pipe, its `meta` value is compared against 42, 555, and 1020.

The actions and FWD of the DOCA Flow LPM pipe work the same as the basic DOCA Flow pipe.

> ⚠ The monitor only supports non-shared counters in the LPM pipe.

`doca_flow_pipe_lpm` insertion max latency can be measured in milliseconds in some cases and, therefore, it is better to insert it from the control path. To get the best insertion performance, entries should be added in large batches.

> ⚠ An LPM pipe cannot be a root pipe. You must create a pipe as root and forward the packets to the LPM pipe.

> ⚠ An LPM pipe can only do LPM matching on inner and outer IP and MAC addresses.

> ⚠ For monitoring, an LPM pipe only supports non-shared counters and does not support other capabilities of `doca_flow_monitor`.

14.4.2.6.4.13 doca_flow_pipe_acl

`doca_flow_pipe_acl` uses access-control list (ACL) matching. ACL matching is five tuple of the `doca_flow_match`. Each entry consists of a value and a mask (e.g., 10.0.0.0/8, 10.10.0.0/16, etc.) for IP address fields, port range, or specific port in the port fields, protocol, and priority of the entry.

ACL entry port configuration:

- Mask port is 0 ==> Any port
- Mask port is equal to match port ==> Exact port. Port with mask 0xffff.
- Mask port > match port ==> Match port is used as port from and mask port is used as port to

Monitor actions are not supported in ACL. FWD of the DOCA Flow ACL pipe works the same as the basic DOCA Flow pipe.

ACL supports the following types of FWD:

- `DOCA_FLOW_FWD_PORT`
- `DOCA_FLOW_FWD_PIPE`
- `DOCA_FLOW_FWD_DROP`

`doca_flow_pipe_lpm` insertion max latency can be measured in milliseconds in some cases and, therefore, it is better to insert it from the control path. To get the best insertion performance, entries should be added in large batches.

⚠ An ACL pipe can be a root pipe.

⚠ An ACL pipe can be in ingress and egress domain.

⚠ An ACL pipe must be accessed on a single queue. Different ACL pipes may be accessed on different queues.

⚠ Adding an entry to the ACL pipe after sending an entry with flag `DOCA_FLOW_NO_WAIT` is not supported.

⚠ Removing an entry from an ACL pipe is not supported.

14.4.2.6.4.14 doca_flow_pipe_ordered_list

`doca_flow_pipe_ordered_list` allows the user to define a specific order of actions and multiply the same type of actions (i.e., specific ordering between counter/meter and encap/decap).

An ordered list pipe is defined by an array of actions (i.e., sequences of actions). Each entry can be an instance one of these sequences. An ordered list pipe may consist of up to an array of 8 different actions. The maximum size of each action array is 4 elements. Resource allocation may be optimized when combining multiple action arrays in one ordered list pipe.

14.4.2.6.4.15 doca_flow_pipe_hash

`doca_flow_pipe_hash` allows the user to insert entries by index. The index represents the packet hash calculation.

An hash pipe gets `doca_flow_match` only on pipe creation and only mask. The mask provides all fields to be used for hash calculation.

The `monitor` , `actions` , `actions_descs` , and `FWD` of the DOCA Flow hash pipe works the same as the basic DOCA Flow pipe.

> The `nb_flows` in `doca_flow_pipe_attr` should be a power of 2.

14.4.2.6.4.16 Hardware Steering Mode

Users can enable hardware steering mode by setting devarg `dv_flow_en` to `2` .

The following is an example of running DOCA with hardware steering mode:

```
.... -a 03:00.0, dv_flow_en=2 -a 03:00.1, dv_flow_en=2....
```

The following is an example of running DOCA with software steering mode:

```
.... -a 03:00.0 -a 03:00.1 ....
```

The `dv_flow_en=2` means that hardware steering mode is enabled.

In the struct `doca_flow_cfg` , setting `mode_args` using (`doca_flow_cfg_set_mode_args()`) represents DOCA applications. If it is set with `hws` (e.g., `"vnf,hws"` , `"switch,hws"` , `"remmote_vnf,hws"`) then hardware steering mode is enabled.

In switch mode, `fdb_def_rule_en=0,vport_match=1,repr_matching_en=0,dv_xmeta_en=4` should be added to DPDK PMD devargs, which makes DOCA Flow switch module take over all the traffic.

To create an entry by calling `doca_flow_pipe_add_entry` , the parameter flags can be set as `DOCA_FLOW_WAIT_FOR_BATCH` or `DOCA_FLOW_NO_WAIT` :

- `DOCA_FLOW_WAIT_FOR_BATCH` means that this flow entry waits to be pushed to hardware. Batch flows then can be pushed only at once. This reduces the push times and enhances the insertion rate.
- `DOCA_FLOW_NO_WAIT` means that the flow entry is pushed to hardware immediately.

The parameter `usr_ctx` is handled in the callback set in struct `doca_flow_cfg` .

`doca_flow_entries_process` processes all the flows in this queue. After the flow is handled and the status is returned, the callback is executed with the status and `usr_ctx` .

If the user does not set the callback in `doca_flow_cfg`, the user can get the status using `doca_flow_entry_get_status` to check if the flow has completed offloading or not.

14.4.2.6.4.17 Isolated Mode

In non-isolated mode (default) any received packets (following an RSS forward, for example) can be processed by the DOCA application, bypassing the kernel. In the same way, the DOCA application can send packets to the NIC without kernel knowledge. This is why, by default, no replies are received when pinging a host with a running DOCA application. If only specific packet types (e.g., DNS packets) should be processed by the DOCA application, while other packets (e.g., ICMP ping) should be handled directly the kernel, then isolated mode becomes relevant.

In isolated mode, packets that match root pipe entries are steered to the DOCA application (as usual) while other packets are received/sent directly by the kernel.

If you plan to create a pipe with matches followed by action/monitor/forward operations, due to functional/performance considerations, it is advised that root pipes entries include the matches followed by a next pipe forward operation. In the next pipe, all the planned matches actions/monitor/forward operations could be specified. Unmatched packets are received and sent by the kernel.

> ⓘ In switch mode, DPDK must be in `isolated` mode. DOCA Flow may be in `isolated` or `non-isolated`.

To activate isolated mode, two configurations are required:

1. DOCA configuration: Update the string member `mode_args` (`struct doca_flow_cfg`) using `doca_flow_cfg_set_mode_args()` which represents the DOCA application mode and add "isolated" (separated by comma) to the other mode arguments. For example: `doca_flow_cfg_set_mode_args(cfg, "vnf,hws,isolated")` `doca_flow_cfg_set_mode_args(cfg, "switch,isolated")`
2. DPDK configuration: Set `isolated_mode` to 1 (`struct application_port_config`). For example, if DPDK is initialized by the API: `dpdk_queues_and_ports_init(struct application_dpdk_config *app_dpdk_config)`.

```
struct application_dpdk_config app_dpdk_config = {
    .port_config = {
        .isolated_mode = 1,
        .nb_ports = ...
        ...
    },
    ...
};
```

14.4.2.6.4.18 Pipe Resize

The move to HWS improves performance because rule insertion is implemented in hardware rather than software. However, this move imposes additional limitations, such as the need to commit in advance on the size of the pipes (the number of rule entries). For applications that require pipe sizes to grow over time, a static size can be challenging: Committing to a pipe size too small can cause the the application to fail once the number of rule entries exceeds the committed number, and pre-committing to an excessively high number of rules can result in memory over-allocation.

This is where pipe resizing comes in handy. This feature allows the pipe size to increase during runtime with support for all entries in a new resized pipe.

> ⓘ Pipe resizing is supported in a basic pipe and a control pipe.

Increasing Pipe Size

It is possible to set a congestion level by percentage (`CONGESTION_PERCENTAGE`). Once the number of entries in the pipe exceeds this value, a callback is invoked. For example, for a pipe with 1000 entries and a `CONGESTION_PERCENTAGE` of 80%, the `CONGESTION_REACHED` callback is invoked after the 800th entry is added.

Following the `CONGESTION_REACHED` callback, the application should call the pipe resize API (`resize()`). The following are optional callbacks during the resize callback:

- A callback on the new number of entries allocated to the pipe
- A callback on each entry that existed in the smaller pipe and is now allocated to the resized pipe

> ⓘ The pipe pointer remains the same for the application to use even after being resized.

Upon completion of the internal transfer of all entries from the small pipe to the resized pipe, a `RESIZED` callback is invoked.

A `CONGESTION_REACHED` callback is received exactly once before the `RESIZED` callback. Receiving another `CONGESTION_REACHED` only happens after calling `resize()` and receiving its completion with a `RESIZED` callback.

List of Callbacks

- `CONGESTION_REACHED` - on the updated number of entries in the pipe (if pipe is resizable)

  > ⓘ Receiving a `CONGESTION_REACHED` callback can occur after adding a small number of entries and for moving entries from a small to resized pipe. The application must always call pipe resize after receiving the `CONGESTION_REACHED` callback to handle such cases.

- `RESIZED` - upon completion of the resize operation

  > ⚠ Calling pipe resize returns immediately. It starts an internal process that ends later with the `RESIZED` callback.

- `NR_ENTRIES_CHANGED` (optional) - on the new max number of entries in the pipe
- `ENTRY_RELOCATE` (optional) - on each entry moved from the small pipe to the resized pipe

Order of Operations for Pipe Resizing

1. Set a process callback on flow configuration:

```
struct doca_flow_cfg *flow_cfg;
doca_flow_cfg_create(&flow_cfg);
doca_flow_cfg_set_cb_pipe_process(flow_cfg, <pipe-process-callback>);
```

> ⓘ This informs on `OP_CONGESTION_REACHED` and `OP_RESIZED` operations when applicable.

2. Set the following pipe attribute configurations:

```
struct doca_flow_pipe_cfg *pipe_cfg;
doca_flow_pipe_cfg_create(&pipe_cfg, port);
doca_flow_pipe_cfg_set_nr_entries(pipe_cfg, <initial-number-of-entries>);
doca_flow_pipe_cfg_set_is_resizable(pipe_cfg, true);
doca_flow_pipe_cfg_set_congestion_level_threshold(pipe_cfg, <CONGESTION_PERCENTAGE>);
doca_flow_pipe_cfg_set_user_ctx(pipe_cfg, <pipe-user-context>);
```

3. Start adding entries:

```
/* Basic pipe */
doca_flow_pipe_add_entry()
/* Contorl pipe */
doca_flow_pipe_control_add_entry()
```

4. Once the number of entries in the pipe crosses the congestion threshold, an `OP_CONGESTION_REACHED` operation callback is received.
5. Mark the pipe's congestion threshold event and, upon return, call `doca_flow_pipe_resize()`. For this call, add the following parameters:
   - The new threshold percentage for calculating the new size.
   - A callback on the new pipe size (optional):

     ```
     doca_flow_pipe_resize_nr_entries_changed_cb nr_entries_changed_cb
     ```

   - A callback on the entries to be transferred to the resized pipe:

     ```
     doca_flow_pipe_resize_entry_relocate_cb entry_relocation_cb
     ```

6. Call `doca_flow_entries_process()` to trigger the transfer of entries. It is relevant for both a basic pipe and a control pipe.
7. At this phase, adding new entries to the pipe is permitted. The entries are added directly to the resized pipe and therefore do not need to be transferred.
8. Once all entries are transferred, an `OP_RESIZED` operation callback is received. Also, at this point a new `OP_CONGESTION_REACHED` operation callback can be received again.
9. At this point calling `doca_flow_entries_process()` can be stopped for a control pipe. For a basic pipe an additional call is required to complete the call to `doca_flow_pipe_add_entry()`.

> ⓘ `doca_flow_entries_process()` has the following roles:
>
> - Triggering entry transfer from the smaller to the bigger pipe (until an `OP_RESIZED` callback is received)
> - Follow up API on previous `add_entries` API (basic pipe relevance only)

14.4.2.6.4.19 Hairpin Configuration

In switch mode, if `dev` is set in struct `doca_flow_port_cfg` (using `doca_flow_port_cfg_set_dev()`), then an internal hairpin is created for direct wire-to-wire fwd. Users may specify the hairpin configuration using `mode_args`. The supported options as follows:

- `hairpinq_num=[n]` - the hairpin queue number
- `use_huge_mem` - determines whether the Tx buffer uses hugepage memory
- `lock_rx_mem` - locks Rx queue memory

14.4.2.6.5 Teardown

14.4.2.6.5.1 Pipe Entry Teardown

When an entry is terminated by the user application or ages-out, the user should call the entry destroy function, `doca_flow_pipe_rm_entry()`. This frees the pipe entry and cancels hardware offload.

14.4.2.6.5.2 Pipe Teardown

When a pipe is terminated by the user application, the user should call the pipe destroy function, `doca_flow_pipe_destroy()`. This destroys the pipe and the pipe entries that match it.

When all pipes of a port are terminated by the user application, the user should call the pipe flush function, `doca_flow_port_pipes_flush()`. This destroys all pipes and all pipe entries belonging to this port.

During `doca_flow_pipe_destroy()` execution, the application must avoid adding/removing entries or checking for aged entries of any other pipes.

14.4.2.6.5.3 Port Teardown

When the port is not used anymore, the user should call the port stop function, `doca_flow_port_stop()`. This stops the DOCA port, disables the traffic, destroys the port and frees all resources of the port.

14.4.2.6.5.4 Flow Teardown

When the DOCA Flow is not used anymore, the user should call the flow destroy function, `doca_flow_destroy()`. This releases all the resources used by DOCA Flow.

14.4.2.7 Metadata

A scratch area exists throughout the pipeline whose maximum size is `DOCA_FLOW_META_MAX` bytes.

The user can set a value to metadata, copy from a packet field, then match in later pipes. Mask is supported in both match and modification actions.

The user can modify the metadata in different ways based on its actions' masks or descriptors:

- `ADD` - set metadata scratch value from a pipe action or an action of a specific entry. Width is specified by the descriptor.
- `COPY` - copy metadata scratch value from a packet field (including the metadata scratch itself). Width is specified by the descriptor.

> Refer to DOCA API documentation for details on `struct doca_flow_meta`.

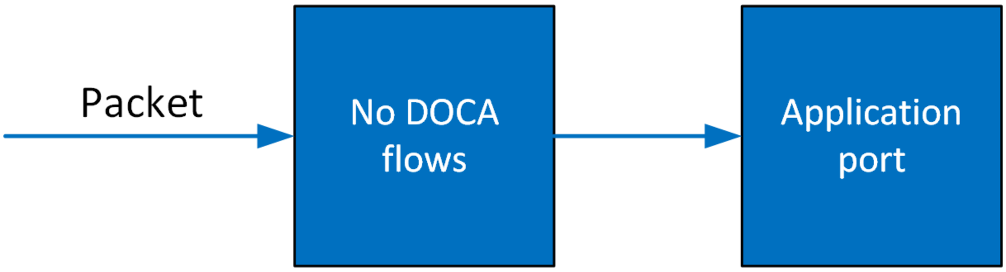
Some DOCA pipe types (or actions) use several bytes in the scratch area for internal usage. So, if the user has set these bytes in PIPE-1 and read them in PIPE-2, and between PIPE-1 and PIPE-2 there is PIPE-A which also uses these bytes for internal purpose, then these bytes are overwritten by the PIPE-A. This must be considered when designing the pipe tree.

The bytes used in the scratch area are presented by pipe type in the following table:

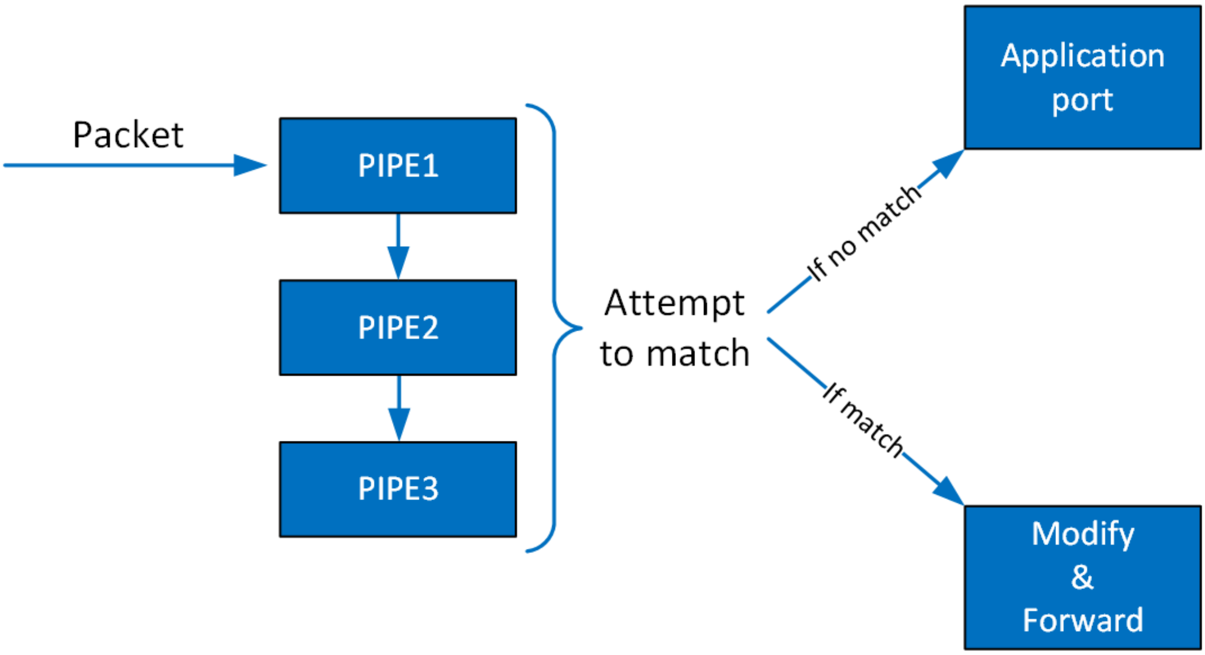
Pipe Type/Action	Bytes Used in Scratch
ordered_list	[0, 1, 2, 3]
LPM	[0, 1, 2, 3]
LPM EM	[0, 1, 2, 3, 4, 5, 6, 7]
Mirror	[0, 1, 2, 3]
ACL	[0, 1, 2, 3, 4, 5, 6, 7, 8 ,9, 10, 11, 12, 13, 14, 15]
Fwd from ingress to egress	[0, 1, 2, 3]

14.4.2.8 Packet Processing

In situations where there is a port without a pipe defined, or with a pipe defined but without any entry, the default behavior is that all packets arrive to a port in the software.



Once entries are added to the pipe, if a packet has no match then it continues to the port in the software. If it is matched, then the rules defined in the pipe are executed.



If the packet is forwarded in RSS, the packet is forwarded to software according to the RSS definition. If the packet is forwarded to a port, the packet is redirected back to the wire. If the packet is forwarded to the next pipe, then the software attempts to match it with the next pipe.

Note that the number of pipes impacts performance. The longer the number of matches and actions that the packet goes through, the longer it takes the hardware to process it. When there is a very large number of entries, the hardware must access the main memory to retrieve the entry context which increases latency.

14.4.2.9 Debug and Trace Features

DOCA Flow supports trace and debugging of DOCA Flow applications which enable collecting predefined internal key performance indicators (KPIs) and pipeline visualization.

14.4.2.9.1 Installation

The set of DOCA's SDK development packages include also a developer-oriented package that includes additional trace and debug features which are not included in the production libraries:

- `.deb` based systems - `libdoca-sdk-flow-trace`
- `.rpm` based systems - `doca-sdk-flow-trace`

These packages install the trace-version of the libraries under the following directories:

- `.deb` based systems - `/opt/mellanox/doca/lib/<arch>/trace`
- `.rpm` based systems - `/opt/mellanox/doca/lib64/trace`

14.4.2.9.2 Using Trace Libraries

The trace libraries are designed to allow a user to link their existing (production) program to the trace library without needing to recompile the program. To do so, one should simply update the matching environment variable so that the OS will prioritize loading libraries from the above trace directory:

```
LD_LIBRARY_PATH=/opt/mellanox/doca/lib/aarch64-linux-gnu/trace:${LD_LIBRARY_PATH} doca_ipsec_security_gw <program
parameters>
```

14.4.2.9.3 Trace Features

14.4.2.9.3.1 DOCA Log – Trace Level

DOCA's trace logging level (`DOCA_LOG_LEVEL_TRACE`) is compiled as part of this trace version of the library. That is, any program compiled against the library can activate this additional logging level through DOCA's API or even through DOCA's built-in argument parsing (ARGP) library:

```
LD_LIBRARY_PATH=/opt/mellanox/doca/lib/aarch64-linux-gnu/trace:${LD_LIBRARY_PATH} doca_ipsec_security_gw <program
parameters> --sdk-log-level 70
```

14.4.2.10 DOCA Flow Samples

This section provides DOCA Flow sample implementation on top of the BlueField.

> All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

14.4.2.10.1 Sample Prerequisites

A DOCA Flow-based program can either run on the host machine or on the BlueField.

Flow-based programs require an allocation of huge pages, hence the following commands are required:

```
echo '1024' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
sudo mkdir /mnt/huge
sudo mount -t hugetlbfs nodev /mnt/huge
```

> On some OSs (RockyLinux, OpenEuler, CentOS 8.2), the default huge page size on the BlueField (and Arm hosts) is larger than 2MB, often 512MB. Users can check the size of the huge pages on their OS using the following command:
>
> ```
> $ grep -i huge /proc/meminfo
>
> AnonHugePages:         0 kB
> ShmemHugePages:        0 kB
> FileHugePages:         0 kB
> HugePages_Total:       4
> HugePages_Free:        4
> HugePages_Rsvd:        0
> HugePages_Surp:        0
> Hugepagesize:     524288 kB
> Hugetlb:         6291456 kB
> ```
>
> In this case, instead of allocating 1024 pages, users should only allocate 4:

```
echo '4' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-524288kB/nr_hugepages
```

14.4.2.10.2 Running the Sample

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample:

   ```
   cd /opt/mellanox/doca/samples/doca_flow/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build
   ```

   > ⚠ The binary `doca_<sample_name>` will be created under `/tmp/build/`.

3. Sample (e.g., `flow_aging`) usage:

   ```
   Usage: doca_flow_aging [DPDK Flags] -- [DOCA Flags]

   DOCA Flags:
     -h, --help                              Print a help synopsis
     -v, --version                           Print program version information
     -l, --log-level                         Set the (numeric) log level for the program <10=DISABLE, 20=CRITI
   CAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
     --sdk-log-level                         Set the SDK (numeric) log level for the program <10=DISABLE, 20=C
   RITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
     -j, --json <path>                       Parse all command flags from an input json file
   ```

4. For additional information per sample, use the `-h` option after the `--` separator:

   ```
   /tmp/build/doca_<sample_name> -- -h
   ```

5. DOCA Flow samples are based on DPDK libraries. Therefore, the user is required to provide DPDK flags. The following is an example from an execution on the DPU:
   - CLI example for running the samples with "vnf" mode:

     ```
     /tmp/build/doca_<sample_name> -a auxiliary:mlx5_core.sf.2 -a auxiliary:mlx5_core.sf.3 -- -l 60
     ```

   - CLI example for running the VNF samples with `vnf,hws` mode:

     ```
     /tmp/build/doca_<sample_name> -a auxiliary:mlx5_core.sf.2,dv_flow_en=2 -a
     auxiliary:mlx5_core.sf.3,dv_flow_en=2 -- -l 60
     ```

   - CLI example for running the switch samples with `switch,hws` mode:

     ```
     /tmp/build/doca_<sample_name> -- -p 03:00.0 -r sf[2-3] -l 60
     ```

     > ⚠ When running on the BlueField with `switch,hws` mode, it is not necessary to configure the OVS.

DOCA switch sample hides the extra `fdb_def_rule_en=0,vport_match=1,repr_matching_en=0,dv_xmeta_en=4` DPDK devargs with a simple `-p` and `-r` to specify the PCIe ID and representor information.

⚠ When running on the DPU using the command above, sub-functions must be enabled according to the NVIDIA BlueField DPU Scalable Function User Guide.

⚠ When running on the host, virtual functions must be used according to the instructions in the NVIDIA DOCA Virtual Functions User Guide.

14.4.2.10.3 Samples

14.4.2.10.3.1 Flow ACL

This sample illustrates how to use the access-control list (ACL) pipe.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Building an ACL pipe that matches changeable:
        i. Source IPv4 address
        ii. Destination IPv4 address
        iii. Source port
        iv. Destination port
    b. Adding four example 5-tuple entries:
        i. The first entry with:
            - Full mask on source IPv4 address
            - Full mask on destination IPv4 address
            - Null mask on source port (any source port)
            - Null mask on destination port (any destination port)
            - TCP protocol
            - Priority 10
            - Action "deny" (drop action)
        ii. The second entry with:
            - Full mask on source IPv4 address
            - Full mask on destination IPv4 address
            - Null mask on source port (any source port)
            - Value set in mask on destination port is used as part of port range:
                - Destination port in match is used as port from
                - Destination port in mask is used as port to
            - UDP protocol
            - Priority 50

        - Action "allow" (forward port action)

    iii. The third entry with:
        - Full mask on source IPv4 address
        - Full mask on destination IPv4 address
        - Value set in mask on source port is equal to the source port in match. It is the exact port. ACL uses the port with full mask.
        - Null mask on destination port (any destination port)
        - TCP protocol
        - Priority 40
        - Action "allow" (forward port action)

    iv. The fourth entry with:
        - 24-bit mask on source IPv4 address
        - 24-bit mask on destination IPv4 address
        - Value set in mask on source port is used as part of port range : source port in match is used as port from, source port in mask is used as port to.
        - Value set in mask on destination port is equal to the destination port in match. It is the exact port. ACL uses the port with full mask.
        - TCP protocol
        - Priority 20
        - Action "allow" (forward port action)

  c. The sample shows how to run the ACL pipe on ingress and egress domains. To change the domain, use the global parameter `flow_acl_sample.c`.
    i. Ingress domain: ACL is created as root pipe
    ii. Egress domain:
        - Building a control pipe with one entry that forwards the IPv4 traffic hairpin port.
        - ACL is created as a root pipe on the hairpin port.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_acl/flow_acl_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_acl/flow_acl_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_acl/meson.build`

14.4.2.10.3.2 Flow Aging

This sample illustrates the use of DOCA Flow's aging functionality. It demonstrates how to build a pipe and add different entries with different aging times and user data.

The sample logic includes:

1. Initializing DOCA Flow with `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow port.
3. On each port:
    a. Building a pipe with changeable 5-tuple match and forward port action.
    b. Adding 10 entries with different 5-tuple match, a monitor with different aging time (5-60 seconds), and setting user data in the monitor. The user data will contain the port ID, entry number, and entry pointer.
4. Handling aging every 5 seconds and removing each entry after age-out.

5. Running these commands until all entries age out.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_aging/flow_aging_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_aging/flow_aging_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_aging/meson.build`

14.4.2.10.3.3 Flow Control Pipe

This sample shows how to use the DOCA Flow control pipe and decap action.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Building VXLAN pipe with match on VNI field, decap action, action descriptor for decap, and forwarding the matched packets to the second port.
    b. Building VXLAN-GPE pipe with match on VNI plus next protocol fields, and forwarding the matched packets to the second port.
    c. Building GRE pipe with match on GRE key field, decap and build `eth` header actions, action descriptor for decap, and forwarding the matched packets to the second port.
    d. Building NVGRE pipe with match on protocol is 0x6558, `vs_id`, `flow_id`, and inner UDP source port fields, and forwarding the matched packets to the second port. This pipe has a higher priority than the GRE pipe. The NVGRE packets are matched first.
    e. Building MPLS pipe with match on third MPLS label field, decap and build `eth` header actions, action descriptor for decap, and forwarding the matched packets to the second port.
    f. Building a control pipe with the following entries:
        - If L4 type is UDP and destination port is 4789, forward to VXLAN pipe
        - If L4 type is UDP and destination port is 4790, forward to VXLAN-GPE pipe
        - If L4 type is UDP and destination port is 6635, forward to MPLS pipe
        - If tunnel type and L4 type is GRE, forward to GRE pipe

⚠ When any tunnel is decapped, it is user responsibility to identify if it is an L2 or L3 tunnel within the action. If the tunnel is L3, the complete outer layer, tunnel, and inner L2 are removed and the inner L3 layer is exposed. To keep the packet valid, the user should provide the ETH header to encap the inner packet. For example:

```
actions.decap_type = DOCA_FLOW_RESOURCE_TYPE_NON_SHARED;
actions.decap_cfg.is_l2 = false;
/* append eth header after decap GRE tunnel */
SET_MAC_ADDR(actions.decap_cfg.eth.src_mac, src_mac[0], src_mac[1], src_mac[2], src_mac[3], src_mac[4],
src_mac[5]);
SET_MAC_ADDR(actions.decap_cfg.eth.dst_mac, dst_mac[0], dst_mac[1], dst_mac[2], dst_mac[3], dst_mac[4],
dst_mac[5]);
actions.decap_cfg.eth.type = DOCA_FLOW_L3_TYPE_IP4;
```

For a VXLAN tunnel, since VXLAN is a L2 tunnel, the user must indicate it within the action:

```
actions.decap_type = DOCA_FLOW_RESOURCE_TYPE_NON_SHARED;
actions.decap_cfg.is_l2 = true;
```

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_control_pipe/flow_control_pipe_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_control_pipe/flow_control_pipe_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_control_pipe/meson.build`

14.4.2.10.3.4 Flow Copy to Meta

This sample shows how to use the DOCA Flow copy-to-metadata action to copy the source MAC address and then match on it.

The sample logic includes:

1. Initializing DOCA Flow by indicating `ode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:

1. Building a pipe with changeable match on `meta_data` and forwarding the matched packets to the second port.
2. Adding an entry that matches an example source MAC that has been copied to metadata.
3. Building a pipe with changeable 5-tuple match, copying source MAC action, and fwd to the first pipe.
4. Adding example 5-tuple entry to the pipe.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_copy_to_meta/flow_copy_to_meta_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_copy_to_meta/flow_copy_to_meta_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_copy_to_meta/meson.build`

14.4.2.10.3.5 Flow Add to Metadata

This sample shows how to use the DOCA Flow add-to-metadata action to accumulate the source IPv4 address for double to meta and then match on the meta.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:

1. Building a pipe with changeable match on `meta_data` and forwarding the matched packets to the second port.
2. Adding an entry that matches an example double of source IPv4 address that has been added to metadata.

3. Building a pipe with changeable 5-tuple match, copying the source IPv4, and adding the value again to the meta action, and forwarding to the first pipe.
4. Adding an example 5-tuple entry to the pipe.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_add_to_meta/flow_add_to_meta_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_add_to_meta/flow_add_to_meta_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_add_to_meta/meson.build`

14.4.2.10.3.6 Flow Drop

This sample illustrates how to build a pipe with 5-tuple match, forward action drop, and forward miss action to the hairpin pipe. The sample also demonstrates how to dump pipe information to a file and query entry.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Building a hairpin pipe with an entry that matches all traffic and forwarding traffic to the second port.
    b. Building a pipe with a changeable 5-tuple match, forwarding action drop, and miss forward to the hairpin pipe. This pipe serves as a root pipe.
    c. Adding an example 5-tuple entry to the drop pipe with a counter as monitor to query the entry later.
4. Waiting 5 seconds and querying the drop entry (total bytes and total packets).
5. Dumping the pipe information to a file.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_drop/flow_drop_sample_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_drop/flow_drop_sample_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_drop/meson.build`

14.4.2.10.3.7 Flow ECMP

This sample illustrates ECMP feature using a hash pipe.

The sample enables users to determine how many port are included in ECMP distribution:

- The number of ports, `n`, is determined by DPDK device argument `representor=sf[0-m]` where `m=n-1`.
- CLI example for running this samples with `n=4` ports:

```
/tmp/build/doca_flow_ecmp -- -p 03:00.0 -r sf[0-3] -l 60 --sdk-log-level 60
```

- `n` should be power of 2. Max supported value is `n=8`.

The sample logic includes:

1. Calculate the number of SF representors (`n`) created by DPDK according to user input.
2. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` structure.
3. Starting DOCA Flow ports: Physical port and `n` SF representors.
4. On switch port:
    a. Constructing a hash pipe that signifies the `match_mask` structure to compute the hash based on the outer IPv6 flow label field.
    b. Adding `n` entries to the created pipe, each of which forwards packets to a different port representor.
5. Waiting 15 seconds and querying the entries.
6. Print the ECMP results per port (number packets in each port related to total packets).

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ecmp/flow_ecmp_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ecmp/flow_ecmp_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ecmp/meson.build`

14.4.2.10.3.8 Flow ESP

This sample illustrates how to match match ESP fields in two ways:

- Exact match for both `esp_spi` and `esp_en` fields using the `doca_flow_match` structure.
- Comparison match for `esp_en` field using the `doca_flow_match_condition` structure.

> ⚠ This sample is supported for ConnectX-7, BlueField-3, and above.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Building a control pipe with entry that match `esp_en > 3` (GT pipe).
    b. Building a control pipe with entry that match `esp_en < 3` (LT pipe).
    c. Building a root pipe with changeable `next_pipe` FWD and `esp_spi` match along with specific `esp_sn` match + IPv4 and ESP exitance (matching `parser_meta`).
    d. Adding example `esp_spi = 8` entry to the root pipe which forwards to GT pipe (and miss condition).
    e. Adding example `esp_spi = 5` entry to the root pipe which forwards to LT pipe (and hit condition).

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_esp/flow_esp_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_esp/flow_esp_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_esp/meson.build`

14.4.2.10.3.9 Flow Forward Miss

The sample illustrates how to use FWD miss query and update with or without miss counter.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Building a copy pipe with a changeable outer L3 type match and forwarding traffic to the second port.
    b. Add entries doing different copy action depending on the outer L3 type:
        i. `IPv4` - copy IHL field into Type Of Service field.
        ii. `IPv6` - copy Payload Length field into Traffic Class field.
    c. Building a pipe with a IPv4 addresses match, forwarding traffic to the second port, and miss forward to the copy pipe.
    d. Building an IP selector pipe with outer L3 type match, forwarding IPv4 traffic to IPv4 pipe, and miss forward to the copy pipe with miss counter.
    e. Building a root pipe with outer L3 type match, forwarding IPv4 and IPv6 traffic to IP selector pipe, and dropping all other traffic by miss forward with miss counter.
4. Waiting 5 seconds for first batch of traffic.
5. On each port:
    a. Querying the miss counters using `doca_flow_query_pipe_miss` API.
    b. Printing the miss results.
6. On each port:
    a. Building a push pipe that pushes VLAN header and forwarding traffic to the second port.
    b. Updating both IP selector and IPv4 pipes miss FWD pipe target to push pipe using `doca_flow_pipe_update_miss` API.
7. Waiting 5 seconds for second batch of traffic, same flow as before.
8. On each port:
    a. Querying again the miss counters using `doca_flow_query_pipe_miss` API.
    b. Printing the miss results again, the results should include miss packets coming either before or after miss action updating.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_fwd_miss/flow_fwd_miss_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_fwd_miss/flow_fwd_miss_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_fwd_miss/meson.build`

14.4.2.10.3.10 Flow Forward Target (DOCA_FLOW_TARGET_KERNEL)

The sample illustrates how to use `DOCA_FLOW_FWD_TARGET` type of forward, as well as the `doca_flow_get_target` API to obtain an instance of `struct doca_flow_target`.

The sample logic includes:

1. Initializing DOCA Flow with `"vnf,isolated,hws"`.

2. Initializing two ports.
3. Obtaining an instance of `doca_flow_target` by calling `doca_flow_get_target(DOCA_FLOW_TARGET_KERNEL, &kernel_target);`.
4. On each port, creating:
    a. Non-root basic pipe with 5 tuple match.
        i. If hit - forward the packet to another port.
        ii. If miss - forward the packet to the kernel for processing by using the instance of `doca_flow_target` obtained in previous steps.
        iii. Then add a single entry with a specific 5-tuple which is hit, and the rest is forwarded to the kernel.
    b. Root control pipe with a match on outer L3 type being IPv4.
        i. If hit - forward the packet to the non-root pipe.
        ii. If miss - drop the packet.
        iii. Add a single entry that implements the logic described.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_fwd_target/flow_fwd_target_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_fwd_target/flow_fwd_target_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_fwd_target/meson.build`

14.4.2.10.3.11 Flow GENEVE Encap

This sample illustrates how to use DOCA Flow actions to create a GENEVE tunnel.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Building ingress pipe with changeable 5-tuple match, copying to `pkt_meta` action, and forwarding port action.
    b. Building egress pipe with `pkt_meta` match and 4 different encapsulation actions:
        - L2 encap without options
        - L2 encap with options
        - L3 encap without options
        - L3 encap with options
    c. Adding example 5-tuple and encapsulation values entries to the pipes.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_geneve_encap/flow_geneve_encap_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_geneve_encap/flow_geneve_encap_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_geneve_encap/meson.build`

14.4.2.10.3.12 Flow GENEVE Options

This sample illustrates how to prepare a GENEVE options parser, match on configured options, and decap GENEVE tunnel.

> ⚠ This sample works only with PF. VFs and SFs are not supported.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Building GENEVE options parser, same input for all ports.
    b. Building match pipe with GENEVE VNI and options match and forwards decap pipe.
    c. Building decap pipe with more GENEVE options match, and 2 different decapsulation actions:
        - L2 decap
        - L3 decap with changeable mac addresses
    d. Adding example GENEVE options and MAC address values entries to the pipes.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_geneve_opt/flow_geneve_opt_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_geneve_opt/flow_geneve_opt_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_geneve_opt/meson.build`

14.4.2.10.3.13 Flow Hairpin VNF

This sample illustrates how to build a pipe with 5-tuple match and to forward packets to the other port.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Building a pipe with changeable 5-tuple match and forwarding port action.
    b. Adding example 5-tuple entry to the pipe.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_hairpin_vnf/flow_hairpin_vnf_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_hairpin_vnf/flow_hairpin_vnf_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_hairpin_vnf/meson.build`

14.4.2.10.3.14 Flow Switch to Wire

This sample illustrates how to build a pipe with 5-tuple match and forward packets from the wire back to the wire.

The sample shows how to build a basic pipe in a switch and hardware steering (HWS) mode. Each pipe contains two entries, each of which forwards matched packets to two different representors.

The sample also demonstrates how to obtain the switch port of a given port using `doca_flow_port_switch_get()` .

> The test requires one PF with three representors (either VFs or SFs).

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct .
2. Starting DOCA Flow ports with `doca_dev` in `struct doca_flow_port_cfg` .
3. On the switch's PF port:
    a. Building ingress, egress, vport, and RSS pipes with changeable 5-tuple match and forwarding port action.
    b. Adding example 5-tuple entry to the pipe.
    c. The matched traffic goes to its destination port, the missed traffic is handled by the `rx_tx` function and is sent to a dedicate port based on the protocol.

        - Ingress pipe:

        ```
        Entry 0: IP src 1.2.3.4 / TCP src 1234 dst 80 -> egress pipe
        Entry 1: IP src 1.2.3.5 / TCP src 1234 dst 80 -> vport pipe
        ```

        - Egress pipe (test ingress to egress cross domain):

        ```
        Entry 0: IP dst 8.8.8.8 / TCP src 1234 dst 80 -> port 0
        Entry 1: IP dst 8.8.8.9 / TCP src 1234 dst 80 -> port 1
        Entry 2: IP dst 8.8.8.10 / TCP src 1234 dst 80 -> port 2
        Entry 3: IP dst 8.8.8.11 / TCP src 1234 dst 80 -> port 3
        ```

        - Vport pipe (test ingress direct to vport):

        ```
        Entry 0: IP dst 8.8.8.8 / TCP src 1234 -> port 0
        Entry 1: IP dst 8.8.8.9 / TCP src 1234 -> port 1
        Entry 2: IP dst 8.8.8.10 / TCP src 1234-> port 2
        Entry 3: IP dst 8.8.8.11 / TCP src 1234-> port 3
        ```

        - RSS pipe (test miss traffic `port_id` get and destination `port_id` set):

        ```
        Entry 0: IPv4 / TCP -> port 0
        Entry 0: IPv4 / UDP -> port 1
        Entry 0: IPv4 / ICMP -> port 2
        ```

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_switch_to_wire/flow_switch_to_wire_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_to_wire/flow_switch_to_wire_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_to_wire/meson.build`

14.4.2.10.3.15 Flow Hash Pipe

This sample illustrates how to build a hash pipe in hardware steering (HWS) mode.

The hash pipe contains two entries, each of which forwards "matched" packets to two different SF representors. For each received packet, the hash pipe calculates the entry index to use based on the IPv4 destination address.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct .
2. Starting DOCA Flow ports: Physical port and two SF representors.
3. On switch port:
    a. Building a hash pipe while indicating which fields to use to calculate the hash in the `struct match_mask`.
    b. Adding two entries to the created pipe, each of which forwards packets to a different port representor.
4. Printing the hash result calculated by the software with the following message: `"hash value for" for dest ip = 192.168.1.1`.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_hash_pipe/flow_hash_pipe_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_hash_pipe/flow_hash_pipe_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_hash_pipe/meson.build`

14.4.2.10.3.16 Flow IPv6 Flow Label

This sample shows how to use DOCA Flow actions to update IPv6 flow label field after encapsulation.

As a side effect, it shows also example for IPv6 + MPLS encapsulation.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Building an ingress pipe with changeable L4 type and ports matching, which updates metadata and goes to the peer port.
    b. Adding example UDP/TCP type and ports and metadata values entries to the pipe. This pipe is L3 type agnostic.
    c. Building an egress pipe on the peer port with changeable metadata matching, which encapsulates packets with IPv6 + MPLS headers, and goes to the next pipe.
    d. Adding entries to the pipe, with different encapsulation values for different metadata values.
    e. Building another egress pipe on the peer port with changeable L3 inner type matching, which copies value into outer IPv6 flow label field.
    f. Adding two entries to the pipe:
        i. L3 inner type is IPv6 - copy IPv6 flow label from inner to outer.
        ii. L3 inner type is IPv6 - copy outer IPv6 flow label from metadata.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ipv6_flow_label/flow_ipv6_flow_label_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ipv6_flow_label/flow_ipv6_flow_label_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ipv6_flow_label/meson.build`

14.4.2.10.3.17 Flow Loopback

This sample illustrates how to implement packet re-injection, or loopback, in VNF mode.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Building a UDP pipe that matches a changeable source and destination IPv4 address, while the forwarding component is RSS to queues. Upon match, setting the packet meta on this UDP pipe which is referred to as an `RSS_UDP_IP` pipe.
    b. Adding one entry to the `RSS_UDP_IP` pipe that matches a packet with a specific source and destination IPv4 address and setting the meta to 10.
    c. Building a TCP pipe that matches changeable 4-tuple source and destination IPv4 and port addresses, while the forwarding component is RSS to queues (this pipe is called `RSS_TCP_IP` and it is the root pipe on ingress domain).
    d. Adding one entry to the `RSS_TCP_IP` pipe, that matches a packet with a specific source and destination port and IPv4 addresses.
    e. On the egress domain, creating the loopback pipe, which is root, and matching TCP over IPv4 with changeable 4-tuple source and destination port and IPv4 addresses, while encapsulating the matched packets with VXLAN tunneling and setting the destination and source MAC addresses to be changeable per entry.
    f. Adding one entry to the loopback pipe with specific values for the match and actions part while setting the destination MAC address to the port to which to inject the packet (in this case, it is the ingress port where the packet arrived).
    g. Starting to receive packets loop and printing the metadata
        - For packets that were re-injected, metadata equaling 10 is printed
        - Otherwise, 0 is be printed as metadata (indicating that it is the first time the packet has been encountered)

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_loopback/flow_loopback_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_loopback/flow_loopback_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_loopback/meson.build`

14.4.2.10.3.18 Flow LPM

This sample illustrates how to use LPM (Longest Prefix Match) pipe

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    1. Building an LPM pipe that matches changeable source IPv4 address.
    2. Adding two example 5-tuple entries:
        1. The first entry with full mask and forward port action
        2. The second entry with 16-bit mask and drop action
    3. Building a control pipe with one entry that forwards IPv4 traffic to the LPM pipe.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_lpm/flow_lpm_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_lpm/flow_lpm_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_lpm/meson.build`

14.4.2.10.3.19 Flow LPM with exact match (EM)

This sample illustrates how to use LPM (Longest Prefix Match) pipe with exact match logic (EM) enabled.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    1. Building LPM pipe that matches changeable source IPv4 address (using `match`) with exact-match logic on `meta.u32[1]` and the inner destination MAC and VNI (using `match_mask`).
    2. Adding five entries to the LPM:
        1. Default entry with IPv4 subnet 0 to drop the packets which are unmatched in LPM with EM
        2. Fully masked `1.2.3.4` IPv4 address with meta value `1`, inner destination mac `1:1:1:1:1:1`, VNI `0xabcde1` to forward to the next port
        3. Fully masked `1.2.3.4` IPv4 address with meta value `2`, inner destination mac `2:2:2:2:2:2`, VNI `0xabcde2` to forward to the next port
        4. Fully masked `1.2.3.4` IPv4 address with meta value `3`, inner destination mac `3:3:3:3:3:3`, VNI `0xabcde3` to drop
        5. First 16 bit masked `1.2.0.0` IPv4 address with meta value `3`, inner destination mac `3:3:3:3:3:3`, VNI `0xabcde3` to forward to the next port
    3. Building basic root pipe which matches everything, copies the `outer.eth_vlan0.tci` value to the `meta.u32[1]` and forwards the packet to the LPM pipe.
    4. Adding single entry to the main pipe.

The sample uses the counters to show the packets per entry. Here are the packets that can be used for the test and the expected response of the sample to them:

- `Ether()/Dot1Q(vlan=1)/IP(src="1.2.3.4")/UDP(dport=4789)/VXLAN(vni=0xabcde1)/Ether(dst="1:1:1:1:1:1")` - to be forwarded to next port by entry number 1

- `Ether()/Dot1Q(vlan=2)/IP(src="1.2.3.4")/UDP(dport=4789)/VXLAN(vni=0xabcde2)/Ether(dst="2:2:2:2:2:2")` - to be forwarded to next port by entry number 2
- `Ether()/Dot1Q(vlan=3)/IP(src="1.2.3.4")/UDP(dport=4789)/VXLAN(vni=0xabcde3)/Ether(dst="3:3:3:3:3:3")` - to be dropped by entry number 3
- `Ether()/Dot1Q(vlan=3)/IP(src="1.2.125.125")/UDP(dport=4789)/VXLAN(vni=0xabcde3)/Ether(dst="3:3:3:3:3:3")` - to be forwarded to next port by entry number 4
- `Ether()/Dot1Q(vlan=5)/IP(src="5.5.5.5")/UDP(dport=4789)/VXLAN(vni=0x424242)/Ether(dst="42:42:42:42:42:42")` - to be dropped by entry number 0 (default)
- `Ether()/Dot1Q(vlan=1)/IP(src="1.2.3.4")/UDP(dport=4789)/VXLAN(vni=0xabcde1)/Ether(dst="1:1:1:1:1:2")` - to be dropped by entry number 0 (default)
- `Ether()/Dot1Q(vlan=1)/IP(src="1.2.3.4")/UDP(dport=4789)/VXLAN(vni=0x424242)/Ether(dst="1:1:1:1:1:1")` - to be dropped by entry number 0 (default)

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_lpm_em/flow_lpm_em_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_lpm_em/flow_lpm_em_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_lpm_em/meson.build`

14.4.2.10.3.20 Flow Modify Header

This sample illustrates how to use DOCA Flow actions to modify the specific packet fields.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port, creating serial pipes and jumping to the next pipe if traffic is unmatched:
    a. Building a pipe with action `dec_ttl=true` and changeable `mod_dst_mac`. The pipe matches IPv4 traffic with a changeable destination IP and forwards the matched packets to the second port.
        - Adding an entry with an example destination IP (8.8.8.8) and `mod_dst_mac` valu e.
    b. Building a pipe with action-changeable `mod_vxlan_tun_rsvd1`. The pipe matches IPv4 traffic with a changeable UDP destination port and VXLAN-GPE tunnel ID then forwards the matched packets to the second port.
        - Adding an entry with an example VXLAN-GPE tunnel ID (100) and UDP destination port (4790), then `mod_vxlan_tun_rsvd1` value.
    c. Building a pipe with action-changeable `mod_vxlan_tun_rsvd1`. The pipe matches IPv4 traffic with a changeable UDP destination port and VXLAN tunnel ID then forwards the matched packets to the second port.
        - Adding an entry with an example VXLAN tunnel ID (100) and UDP destination port (4789), then `mod_vxlan_tun_rsvd1` value.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_modify_header/flow_modify_header_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_modify_header/flow_modify_header_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_modify_header/meson.build`

14.4.2.10.3.21 Flow Monitor Meter

This sample illustrates how to use DOCA Flow monitor meter.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Building a pipe with monitor meter flag and changeable 5-tuple match. The pipe forwards the matched packets to the second port.
    b. Adding an entry with an example CIR and CBS values.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_monitor_meter/flow_monitor_meter_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_monitor_meter/flow_monitor_meter_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_monitor_meter/meson.build`

14.4.2.10.3.22 Flow Multi-actions

This sample shows how to use a DOCA Flow array of actions in a pipe.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Building a pipe with changeable source IP match which forwards the matched packets to the second port and sets different actions in the actions array:
        - Changeable modify source MAC address
        - Changeable modify source IP address
    b. Adding two entries to the pipe with different source IP match:
        i. The first entry with an example modify source MAC address.
        ii. The second with a modify source IP address.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_multi_actions/flow_multi_actions_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_multi_actions/flow_multi_actions_main.c`

- `/opt/mellanox/doca/samples/doca_flow/flow_multi_actions/meson.build`

14.4.2.10.3.23 Flow Multi-fwd

This sample shows how to use a different forward in pipe entries.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
   a. Building a pipe with changeable source IP match and sending NULL in the forward.
   b. Adding two entries to the pipe with different source IP match, and different forward:
      - The first entry with forward to the second port
      - The second with drop

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_multi_fwd/flow_multi_fwd_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_multi_fwd/flow_multi_fwd_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_multi_fwd/meson.build`

14.4.2.10.3.24 Flow Ordered List

This sample shows how to use a DOCA Flow ordered list pipe.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
   a. Building a root pipe with changeable 5-tuple match and forwarding to an ordered list pipe with a changeable index.
   b. Adding two entries to the pipe with an example value sent to a different index in the ordered list pipe.
   c. Building ordered list pipe with two lists, one for each entry:
      - First list uses meter and then shared counter
      - Second list uses shared counter and then meter
4. Waiting 5 seconds and querying the entries (total bytes and total packets).

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ordered_list/flow_ordered_list_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ordered_list/flow_ordered_list_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ordered_list/meson.build`

14.4.2.10.3.25 Flow Parser Meta

This sample shows how to use some of `match.parser_meta` fields from 3 families:

- IP fragmentation – matching on whether a packet is IP fragmented
- Integrity bits – matching on whether a specific protocol is OK (length, checksum etc.)
- Packet types – matching on a specific layer packet type

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Building a root pipe with outer IP fragmentation match:
        - If a packet is IP fragmented – forward it to the second port regardless of next pipes in the pipeline
        - If a packet is not IP fragmented – proceed with the the pipeline by forwarding it to integrity pipe

    b. Building an "integrity" pipe with a single entry which continues to the next pipe when:
        - The outer IPv4 checksum is OK
        - The inner L3 is OK (incorrect length should be dropped)

    c. Building a "packet type" pipe which forwards packets to the second port when:
        - The outer L3 type is IPv4
        - The inner L4 type is either TCP or UDP
4. Waiting 5 seconds for traffic to arrive.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_parser_meta/flow_parser_meta_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_parser_meta/flow_parser_meta_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_parser_meta/meson.build`

14.4.2.10.3.26 Flow Random

This sample shows how to use `match.parser_meta.random` field for 2 different use-cases:

- Sampling – sampling certain percentage of traffic regardless of flow content
- Distribution – distributing traffic in 8 different queues

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Building a root pipe with changeable 5-tuple match and forwarding to specific use-case pipe according to changeable source IP address.

    b. Adding two entries to the pipe with different source IP match, and different forward:
        - The first entry with forward to the sampling pipe.
        - The second entry with forward to the distribution pipe.

    c. Building a "sampling" pipe with a single entry and preparing the entry to sample 12.5% of traffic.

    d. Building a "distribution" hash pipe with 8 entries and preparing the entries to get 12.5% of traffic for each queue.

4. Waiting 15 seconds and querying the entries (total packets after sampling/distribution related to total packets before).

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_random/flow_random_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_random/flow_random_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_random/meson.build`

14.4.2.10.3.27 Flow RSS ESP

This sample shows how to use DOCA Flow forward RSS according to ESP SPI field, and distribute the traffic between queues.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Building a pipe with both L3 and L4 types match, copy the `SPI` field into packet meta data, and forwarding to RSS with 7 queues.
    b. Adding an entry with both IPv4 and ESP existence matching.
4. Waiting 15 seconds for traffic to arrived.
5. On each port:
    a. Calculates the traffic percentage distributed into each port and prints the result.
    b. Printing for each packet its `SPI` value. (only in debug mode, `-l ≥ 60`)

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_rss_esp/flow_rss_esp_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_rss_esp/flow_rss_esp_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_rss_esp/meson.build`

14.4.2.10.3.28 Flow RSS Meta

This sample shows how to use DOCA Flow forward RSS, set meta action, and then retrieve the matched packets in the sample.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Building a pipe with a changeable 5-tuple match, forwarding to RSS queue with index 0, and setting changeable packet meta data.
    b. Adding an entry with an example 5-tuple and metadata value to the pipe.
4. Retrieving the packets on both ports from a receive queue, and printing the packet metadata value.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_rss_meta/flow_rss_meta_sample.c`

- `/opt/mellanox/doca/samples/doca_flow/flow_rss_meta/flow_rss_meta_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_rss_meta/meson.build`

14.4.2.10.3.29 Flow Sampling

This sample shows how to sample certain percentage of traffic regardless of flow content using `doca_flow_match_condition` structure with `parser_meta.random.value` field string.

> ⚠ This sample is supported for ConnectX-7/BlueField-3 and above.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct .
2. Starting DOCA Flow ports: Physical port and two SF representors.
3. On switch port:
   a. Building a root pipe with changeable 5-tuple match and forwarding to sampling pipe.
   b. Adding entry with an example 5-tuple to the pipe.
   c. Building a "sampling" control pipe with a single entry.
   d. calculating the requested random value for getting 35% of traffic.
   e. Adding entry with an example condition random value to the pipe.
4. Waiting 15 seconds and querying the entries (total packets after sampling related to total packets before).

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_sampling/flow_sampling_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_sampling/flow_sampling_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_sampling/meson.build`

14.4.2.10.3.30 Flow Set Meta

This sample shows how to use the DOCA Flow set metadata action and then match on it.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:

1. Building a pipe with a changeable match on metadata and forwarding the matched packets to the second port.
2. Adding an entry that matches an example metadata value.
3. Building a pipe with changeable 5-tuple match, changeable metadata action, and fwd to the first pipe.
4. Adding entry with an example 5-tuple and metadata value to the pipe.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_set_meta/flow_set_meta_sample.c`

- `/opt/mellanox/doca/samples/doca_flow/flow_set_meta/flow_set_meta_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_set_meta/meson.build`

14.4.2.10.3.31 Flow Shared Counter

This sample shows how to use the DOCA Flow shared counter and query it to get the counter statistics.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Binding the shared counter to the port.
    b. Building a pipe with changeable 5-tuple match with UDP protocol, changeable shared counter ID and forwarding the matched packets to the second port.
    c. Adding an entry with an example 5-tuple match and shared counter with ID= `port_id`.
    d. Building a pipe with changeable 5-tuple match with TCP protocol, changeable shared counter ID and forwarding the matched packets to the second port.
    e. Adding an entry with an example 5-tuple match and shared counter with ID= `port_id`.
    f. Building a control pipe with the following entries:
        - If L4 type is UDP, forwards the packets to the UDP pipe
        - If L4 type is TCP, forwards the packets to the TCP pipe
4. Waiting 5 seconds and querying the shared counters (total bytes and total packets).

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_shared_counter/flow_shared_counter_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_shared_counter/flow_shared_counter_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_shared_counter/meson.build`

14.4.2.10.3.32 Flow Shared Meter

This sample shows how to use the DOCA Flow shared meter.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Config a shared meter with specific cir and cbs values.
    b. Binding the shared meter to the port.
    c. Building a pipe with a changeable 5-tuple match with UDP protocol, changeable shared meter ID and forwarding the matched packets to the second port.
    d. Adding an entry with an example 5-tuple match and shared meter with ID= `port_id`.
    e. Building a pipe with a changeable 5-tuple match with TCP protocol, changeable shared meter ID and forwarding the matched packets to the second port.
    f. Adding an entry with an example 5-tuple match and shared meter with ID= `port_id`.

   g. Building a control pipe with the following entries:
      - If L4 type is UDP, forwards the packets to the UDP pipe
      - If L4 type is TCP, forwards the packets to the TCP pipe

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_shared_meter/flow_shared_meter_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_shared_meter/flow_shared_meter_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_shared_meter/meson.build`

14.4.2.10.3.33 Flow Switch Control Pipe

This sample shows how to use the DOCA Flow control pipe in switch mode.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct .
2. Starting two DOCA Flow ports.
3. On each port:
   a. Building control pipe with match on VNI field.
   b. Adding two entries to the control pipe, both matching TRANSPORT (UDP or TCP proto) over IPv4 with source port 80 and forwarding to the other port, where the first entry matches destination port 1234 and the second 12345.
   c. Both entries have counters, so that after the successful insertions of both entries, the sample queries those counters to check the number of matched packets per entry.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_switch_control_pipe/flow_switch_control_pipe_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_control_pipe/flow_switch_control_pipe_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_control_pipe/meson.build`

14.4.2.10.3.34 Flow Switch - Multiple Switches

This sample illustrates how to use two switches working concurrently on two different physical functions.

It shows how to build a basic pipe in a switch and hardware steering (HWS) mode. Each pipe contains two entries, each of which forwards matched packets to two different representors.

The sample also demonstrates how to obtain the switch port of a given port using `doca_flow_port_switch_get()` .

> ⚠ The test requires two PFs with two (either VF or SF) representors on each.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct .
2. Starting DOCA Flow ports: Two physical ports and two representors each (totaling six ports).
3. On the switch port:
    a. Building a basic pipe while indicating which fields to match on using `struct doca_flow_match match`.
    b. Adding two entries to the created pipe, each of which forwards packets to a different port representor.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_switch/flow_switch_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch/flow_switch_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch/meson.build`

14.4.2.10.3.35 Flow Switch - Single Switch

This sample is identical to the previous sample, before the flow switch sample was extended to take advantage of the capabilities of DOCA to support multiple switches concurrently, each based on a different physical device.

The reason we add this original version is that it removes the constraints imposed by the modified flow switch version, allowing to use arbitrary number of representors in the switch configuration.

The logic of this sample is identical to that of the previous sample with 2 new pipes.

- A user RSS pipe which receives the packets which missed TC rules (in the kernel domain in this case)
- A simple pipe forwarding packets to kernel domain by using `DOCA_FLOW_FWD_TARGET`

In the `to_kernel_pipe`, all the IPv4 packets are forwarded to the kernel (i.e., entry 0 in `to_kernel_pipe`). In the kernel domain, all the IPv4 packets are missed to the NIC domain if there is no TC rule. In the NIC domain, the IPv4 packets missed from the NIC domain are forwarded to slow path (i.e., the representor of the PF/VF).

- Root pipe:

```
Entry 0: IP src 1.2.3.4 / dst 8.8.8.8 / TCP src 1234 dst 80 -> port 0
Entry 1: IP src 1.2.3.5 / dst 8.8.8.9 / TCP src 1234 dst 80 -> port 1
Miss: -> To kernel pipe
```

- To kernel pipe:

```
Entry 0: IPv4 -> send to kernel
IPv6 traffic would be dropped
```

- RSS pipe:

```
Entry 0: IPv4 -> port 0 rss queue 0
```

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_switch_single/flow_switch_single_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_single/flow_switch_single_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_single/meson.build`

14.4.2.10.3.36 Flow Switch (Direction Info)

This sample illustrates how to give a hint to the driver for potential optimizations based on the direction information.

> ⓘ This sample requires a single PF with two representors (either VF or SF).

The sample also demonstrates usage of the `match.parser_meta.port_meta` to detect by the switch pipe the source from where the packet has arrived.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct .
2. Starting 3 DOCA Flow ports, 1 physical port and 2 representors.
3. On the switch port:
    a. Network-to-host pipe:
        i. Building basic pipe with a changeable `ipv4.next_proto` field and configuring the pipe with the hint of direction by setting `attr.dir_info = DOCA_FLOW_DIRECTION_NETWORK_TO_HOST`.
        ii. Adding two entries:
            - If `ipv4.next_proto` is TCP, the packet is forwarded to the first representor, to the host.
            - If `ipv4.next_proto` is UDP, the packet is forwarder to the second representor, to the host.
    b. Host-to-network pipe:
        i. Building a basic pipe with a match on `aa:aa:aa:aa:aa:aa` as a source MAC address and configuring a pipe with the hint of direction by setting `attr.dir_info = DOCA_FLOW_DIRECTION_HOST_TO_NETWORK`.
        ii. Adding an entry. If the source MAC is matched, forward the packet to the physical port (i.e., to the network).
    c. Switch pipe:
        i. Building a basic pipe with a changeable `parser_meta.port_meta` to detect where the packet has arrived from.
        ii. Adding 3 entries:
            - If the packet arrived from port 0 (i.e., the network), forward it to the network-to-host pipe to decide for further logic
            - If the packet arrived from port 1 (i.e., the host's first representor), forward it to the host-to-network pipe to decide for further logic
            - If the packet arrived from port 2, (i.e., the host's second representor), forward it to the host-to-network pipe to decide for further logic

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_switch_direction_info/flow_switch_direction_info_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_direction_info/flow_switch_direction_info_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_direction_info/meson.build`

14.4.2.10.3.37 Flow Switch Hot Upgrade

This sample demonstrates how to use the port operation state mechanism for a hot upgrade use case. It shows how to configure the state of a port during initialization and how to modify the state after the port has already been started.

Prerequisites

The test requires two physical functions (PFs) with two (either VFs or SFs) representors on each.

Command-line Arguments

The sample allows users to specify the operation state of the instance using the `--state <value>` argument. The relevant values are:

- `0` for `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE`
- `1` for `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE_READY_TO_SWAP`
- `2` for `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY`

Sample Logic

1. Initialize DOCA Flow:
    - Indicate `mode_args="switch"` in the `doca_flow_cfg` structure.
2. Start DOCA Flow ports:
    - Two physical ports and two representors each (totaling six ports) are started.
    - Both switch ports are configured with `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED` state.
3. Configure each switch port:
    a. Build a basic pipe with a miss counter matching on outer L3 type (specific IPv4) and outer L4 type (changeable).
    b. Add two entries to the created pipe with counters, each forwarding packets to a different port representor.
    c. Modify the port operation state from `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED` to the required state.
4. Traffic handling:
    - Wait for traffic until a SIGQUIT signal (Ctrl+) is received.
    - While traffic is being received, traffic statistics are printed to stdout.

Hot Upgrade Use Case

To illustrate the hot upgrade use case, follow these steps:

1. Create two different instances in separate windows with different states.

DPDK prevents users from creating two primary instances. To avoid this limitation, use the `--file-prefix` EAL argument.

- Example for the "active" instance:

```
/tmp/build/samples/doca_flow_switch_hot_upgrade -- -p 08:00.0 -p 08:00.1 -r vf[0-1] -r
vf[0-1] -l 70
```

- Example for the "stand-by" instance:

```
/tmp/build/samples/doca_flow_switch_hot_upgrade --file-prefix standby -- -p 08:00.0 -p
08:00.1 -r vf[0-1] -r vf[0-1] -l 70 --state 2
```

2. Close the active process by typing Ctrl+\ while traffic is being received. The traffic statistics will start printing in the standby instance.
3. Restart the first instance. The traffic statistics will stop printing in the standby instance and start printing in the active instance again.

Swap Use Case

When both instances are running, the swap use case can be demonstrated by typing Ctrl+C:

- Typing Ctrl+C in the active instance changes its state to `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE_READY_TO_SWAP`
- Typing Ctrl+C in the standby instance changes its state to `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE`
- Typing Ctrl+C in the active instance again changes its state to `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY`

References

- `/opt/mellanox/doca/samples/doca_flow/flow_switch_hot_upgrade/flow_switch_hot_upgrade_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_hot_upgrade/flow_switch_hot_upgrade_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_hot_upgrade/meson.build`

14.4.2.10.3.38 Flow VXLAN Encap

This sample shows how to use DOCA Flow actions to create a VXLAN/VXLANGPE/VXLANGBP tunnel as well as illustrating the usage of matching TCP and UDP packets in the same pipe.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Building a pipe with changeable 5-tuple match, encap action, and forward port action.
    b. Adding example 5-tuple and encapsulation values entry to the pipe. Every TCP or UDP over IPv4 packet with the same 5-tuple is matched and encapsulated.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_vxlan_encap/flow_vxlan_encap_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_vxlan_encap/flow_vxlan_encap_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_vxlan_encap/meson.build`

14.4.2.10.3.39 Flow Shared Mirror

This sample shows how to use the DOCA Flow shared mirror.

> ⚠ A current limitation does not allow using shared mirror IDs bearing the value zero.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    a. Configuring a shared mirror with a clone destination hairpin to the second port.
    b. Binding the shared mirror to the port.
    c. Building a pipe with a changeable 5-tuple match with UDP protocol, changeable shared mirror ID, and forwarding the matched packets to the second port.
    d. Adding an entry with an example 5-tuple match and shared mirror with ID= `port_id+1`.
    e. Building a pipe with a changeable 5-tuple match with TCP protocol, changeable shared mirror ID, and forwarding the matched packets to the second port.
    f. Adding an entry with an example 5-tuple match and shared mirror with ID= `port_id+1`.
    g. Building a control pipe with the following entries:
        - If L4 type is UDP, forwards the packets to the UDP pipe
        - If L4 type is TCP, forwards the packets to the TCP pipe
    h. Waiting 15 seconds to clone any incoming traffic. Should see the same two packets received on the second port (one from the clone and another from the original).

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_shared_mirror/flow_shared_mirror_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_shared_mirror/flow_shared_mirror_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_shared_mirror/meson.build`

14.4.2.10.3.40 Flow Match Comparison

This sample shows how to use the DOCA Flow match with a comparison result.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:

    a. Building a pipe with a changeable match on `meta_data[0]` and forwarding the matched packets to the second port.
    b. Adding an entry that matches on `meta_data[0]` equal with TCP header length.
    c. Building a control pipe for comparison purpose.
    d. Adding an entry to the control pipe match with comparison result the `meta_data[0]` value greater than `meta_data[1]` and forwarding the matched packets to match with the meta pipe.
    e. Building a pipe with a changeable 5-tuple match, copying `ipv4.total_len` to `meta_data[1]`, and accumulating `ipv4.version_ihl << 2` `tcp.data_offset << 2` to `meta_data[1]`, then forwarding to the second pipe.
    f. Adding an example 5-tuple entry to the pipe.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_match_comparison/flow_match_comparison_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_match_comparison/flow_match_comparison_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_match_comparison/meson.build`

14.4.2.10.3.41 Flow Pipe Resize

This sample shows how the DOCA Flow pipe resize feature behaves as pipe size increases. The pipe type under resize (basic or control) can be specified in the command line.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws,cpds"` in the `doca_flow_cfg` struct.

    > The CPDS (control pipe dynamic size) argument is relevant for a control pipe only. By default, a control pipe's internal tables have a default size of 64 entries. Using the CPDS mode, each table's initial size matches the control pipe size.

2. Starting a PF with two representors of SFs or VFs and selecting the pipe type under resize. For example:

    ```
    ./doca_flow_pipe_resize -- --pipe-type <basic|control> -p 08:00.0 -r sf\[0-1\] -l 60 --sdk-log-level 50
    ```

3. Starting with a pipe of a max size of 10 entries then adding 80 entries. Instead of failing on adding the 11th entry, the pipe continues increasing in the following manner:
    a. Receiving a `CONGESTION_REACHED` callback whenever the number of current entries exceeds a threshold level of 80%.
    b. Calling `doca_flow_pipe_resize()` with threshold percentage of 50%. Roughly, the new size is calculated as: (current entries) / (50%) rounded up to the nearest power of 2. A callback can indicate the exact number of entries.
    c. Receiving a callback on the exact new calculated size of the pipe:

```
typedef doca_error_t (*doca_flow_pipe_resize_nr_entries_changed_cb)(void *pipe_user_ctx, uint32_t
nr_entries);
```

d. Start calling `doca_flow_entries_process()` in a loop on each thread ID to trigger the entry relocations.

> ⓘ The loop should continue as long as the resize process was not ended.

e. Receiving a callback on each entry relocated to the new resized pipe:

```
typedef doca_error_t (*doca_flow_pipe_resize_entry_relocate_cb)(void *pipe_user_ctx, uint16_t
pipe_queue, void *entry_user_ctx, void **new_entry_user_ctx)
```

f. Receiving a `PIPE_RESIZED` callback upon completion of the resize process. At this point, in case of a control pipe, calling `doca_flow_entries_process()` should stop. In case of a basic pipe, continue calling `doca_flow_entries_process()` to process the last entries being added to the pipe.

> ⓘ The resize cycles described above repeats five times increasing the pipe sizes in these steps: 10 -> 16 -> 32 -> 64 -> 128.

> ⓘ The pipe control entries define a match on increasing destination IP address. The fwd action send packet to the other port.

g. Waiting 5 seconds to send any traffic that matches the flows and seeing them on the other port.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_pipe_resize/flow_pipe_resize_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_pipe_resize/flow_pipe_resize_main.c`
- /opt/mellanox/doca/samples/doca_flow/flow_pipe_resize/meson.build

14.4.2.10.3.42 Flow Entropy

This sample shows how to use the DOCA Flow entropy calculation.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct .
2. Starting one DOCA Flow port.
3. Configuring the `doca_flow_entropy_format` structure with 5-tuple values.
4. Calling to `doca_flow_port_calc_entropy` to get the calculated entropy.
5. Logging the calculated entropy.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_entropy/flow_entropy_sample.c`

- `/opt/mellanox/doca/samples/doca_flow/flow_entropy/flow_entropy_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_entropy/meson.build`

14.4.2.10.3.43 Flow VXLAN Shared Encap

This sample shows how to use DOCA Flow actions to create a VXLAN tunnel as well as illustrating the usage of matching TCP and UDP packets in the same pipe.

The VXLAN tunnel is created by `shared_resource_encap`.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
   a. Configure and bind shared encap resources. The encap resources are for VXLAN encap.
   b. Building a pipe with changeable 5-tuple match, `shared_encap_id`, and forward port action.
   c. Adding example 5-tuple and encapsulation values entry to the pipe. Every TCP or UDP over IPv4 packet with the same 5-tuple is matched and encapsulated.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_vxlan_shared_encap/flow_vxlan_shared_encap_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_vxlan_shared_encap/flow_vxlan_shared_encap_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_vxlan_shared_encap/meson.build`

14.4.2.11 Field String Support Appendix

14.4.2.11.1 Supported Field String

The following is a list of all the API fields available for matching criteria and action execution.

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
`meta.data` (bit_offset < 32)	`meta.pkt_meta`	`meta.pkt_meta`	✓	✓	✓	✓	✓	✓	✓
`meta.data` (bit_offset ≥ 32)	`meta.u32[i]`	`meta.u32[i]`	✓	✓	✓	✓	✓	✓	✓
`meta.mark`	`meta.mark`	`meta.mark`	✘	✘	✘	✘	✘	✘	✘
`parser_meta.hash.result`	None. See section "Copy Hash Result" for details.		N/A	N/A	✓	N/A	✓	✘	✘
`parser_meta.port.id`	`parser_meta.port_meta`		N/A	N/A	✘	N/A	✘	✘	✘

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
`parser_meta.ipsec.syndrome`	`parser_meta.ipsec_syndrome`		N/A	N/A	✘	N/A	✘	✘	✘
`parser_meta.psp.syndrome`	`parser_meta.psp_syndrome`		N/A	N/A	✘	N/A	✘	✘	✘
`parser_meta.random.value`	`parser_meta.random`		N/A	N/A	✘	N/A	✘	✓	✘
`parser_meta.meter.color`	`parser_meta.meter_color`		N/A	N/A	✘	N/A	✘	✘	✘
`parser_meta.packet_type.l2_outer`	`parser_meta.outer_l2_type`		N/A	N/A	N/A	N/A	N/A	N/A	N/A
`parser_meta.packet_type.l3_outer`	`parser_meta.outer_l3_type`		N/A	N/A	N/A	N/A	N/A	N/A	N/A
`parser_meta.packet_type.l4_outer`	`parser_meta.outer_l4_type`		N/A	N/A	N/A	N/A	N/A	N/A	N/A
`parser_meta.packet_type.l2_inner`	`parser_meta.inner_l2_type`		N/A	N/A	N/A	N/A	N/A	N/A	N/A
`parser_meta.packet_type.l3_inner`	`parser_meta.inner_l3_type`		N/A	N/A	N/A	N/A	N/A	N/A	N/A
`parser_meta.packet_type.l4_inner`	`parser_meta.inner_l4_type`		N/A	N/A	N/A	N/A	N/A	N/A	N/A
`parser_meta.outer_ip_fragmented.flag`	`parser_meta.outer_ip_fragmented`		N/A	N/A	N/A	N/A	N/A	N/A	N/A
`parser_meta.inner_ip_fragmented.flag`	`parser_meta.inner_ip_fragmented`		N/A	N/A	N/A	N/A	N/A	N/A	N/A
`parser_meta.outer_integrity.l3_ok`	`parser_meta.outer_l3_ok`		N/A	N/A	N/A	N/A	N/A	N/A	N/A
`parser_meta.outer_integrity.ipv4_checksum_ok`	`parser_meta.outer_ip4_checksum_ok`		N/A	N/A	N/A	N/A	N/A	N/A	N/A
`parser_meta.outer_integrity.l4_ok`	`parser_meta.outer_l4_ok`		N/A	N/A	N/A	N/A	N/A	N/A	N/A
`parser_meta.outer_integrity.l4_checksum_ok`	`parser_meta.outer_l4_checksum_ok`		N/A	N/A	N/A	N/A	N/A	N/A	N/A
`parser_meta.inner_integrity.l3_ok`	`parser_meta.inner_l3_ok`		N/A	N/A	N/A	N/A	N/A	N/A	N/A

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
`parser_meta.inner_integrity.ipv4_checksum_ok`	`parser_meta.inner_ip4_checksum_ok`		N/A	N/A	N/A	N/A	N/A	N/A	N/A
`parser_meta.inner_integrity.l4_ok`	`parser_meta.inner_l4_ok`		N/A	N/A	N/A	N/A	N/A	N/A	N/A
`parser_meta.inner_integrity.l4_checksum_ok`	`parser_meta.inner_l4_checksum_ok`		N/A	N/A	N/A	N/A	N/A	N/A	N/A
`outer.eth.dst_mac`	`outer.eth.dst_mac`	`outer.eth.dst_mac`	✓	✘	✓	✓	✓	✘	✘
`outer.eth.src_mac`	`outer.eth.src_mac`	`outer.eth.src_mac`	✓	✘	✓	✓	✓	✘	✘
`outer.eth.type`	`outer.eth.type`	`outer.eth.type`	✓	✘	✓	✓	✓	✘	✘
`outer.eth_vlan0.tci`	`outer.eth_vlan[0].tci`	`outer.eth_vlan[0].tci`	✓	✘	✓	✓	✓	✘	✘
`outer.eth_vlan1.tci`	`outer.eth_vlan[1].tci`	`outer.eth_vlan[1].tci`	✘	✘	✘	✘	✘	✘	✘
`outer.ipv4.src_ip`	`outer.ip4.src_ip`	`outer.ip4.src_ip`	✓	✘	✓	✓	✓	✘	✘
`outer.ipv4.dst_ip`	`outer.ip4.dst_ip`	`outer.ip4.dst_ip`	✓	✘	✓	✓	✓	✘	✘
`outer.ipv4.dscp_ecn`	`outer.ip4.dscp_ecn`	`outer.ip4.dscp_ecn`	✓	✘	✓	✓	✓	✘	✘
`outer.ipv4.next_proto`	`outer.ip4.next_proto`	`outer.ip4.next_proto`	✓	✘	✓	✓	✓	✘	✘
`outer.ipv4.ttl`	`outer.ip4.ttl`	`outer.ip4.ttl`	✓	✓	✓	✓	✓	✘	✘
`outer.ipv4.version_ihl`	`outer.ip4.version_ihl`	`outer.ip4.version_ihl`	✓	✓	✓	✓	✓	✘	✘
`outer.ipv4.total_len`	`outer.ip4.total_len`	`outer.ip4.total_len`	✓	✓	✓	✓	✓	✘	✘
`outer.ipv6.src_ip`	`outer.ip6.src_ip`	`outer.ip6.src_ip`	✓	✘	✓	✓	✓	✘	✘
`outer.ipv6.dst_ip`	`outer.ip6.dst_ip`	`outer.ip6.dst_ip`	✓	✘	✓	✓	✓	✘	✘

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
`outer.ipv6.traffic_class`	`outer.ip6.traffic_class`	`outer.ip6.traffic_class`	✓	✘	✓	✓	✓	✘	✘
`outer.ipv6.flow_label`	`outer.ip6.flow_label`	`outer.ip6.flow_label`	✓	✘	✓	✓	✓	✘	✘
`outer.ipv6.next_proto`	`outer.ip6.next_proto`	`outer.ip6.next_proto`	✓	✘	✓	✓	✓	✘	✘
`outer.ipv6.hop_limit`	`outer.ip6.hop_limit`	`outer.ip6.hop_limit`	✓	✓	✓	✓	✓	✘	✘
`outer.ipv6.payload_len`	`outer.ip6.payload_len`	`outer.ip6.payload_len`	✓	✓	✓	✓	✓	✘	✘
`outer.udp.src_port`	`outer.udp.l4_port.src_port`	`outer.udp.l4_port.src_port`	✓	✘	✓	✓	✓	✘	✘
`outer.udp.dst_port`	`outer.udp.l4_port.dst_port`	`outer.udp.l4_port.dst_port`	✓	✘	✓	✓	✓	✘	✘
`outer.transport.src_port`	`outer.transport.src_port`	`outer.transport.src_port`	✓	✘	✓	✓	✓	✘	✘
`outer.transport.dst_port`	`outer.transport.dst_port`	`outer.transport.dst_port`	✓	✘	✓	✓	✓	✘	✘
`outer.tcp.src_port`	`outer.tcp.l4_port.src_port`	`outer.tcp.l4_port.src_port`	✓	✘	✓	✓	✓	✘	✘
`outer.tcp.dst_port`	`outer.tcp.l4_port.dst_port`	`outer.tcp.l4_port.dst_port`	✓	✘	✓	✓	✓	✘	✘
`outer.tcp.flags`	`outer.tcp.flags`	`outer.tcp.flags`	✘	✘	✘	✘	✘	✘	✘
`outer.tcp.data_offset`	`outer.tcp.data_offset`	`outer.tcp.data_offset`	✘	✓	✓	✓	✓	✘	✘
`outer.icmp4.type`	`outer.icmp.type`	`outer.icmp.type`	✘	✘	✘	✘	✘	✘	✘
`outer.icmp4.code`	`outer.icmp.code`	`outer.icmp.code`	✘	✘	✘	✘	✘	✘	✘
`outer.icmp4.ident`	`outer.icmp.ident`	`outer.icmp.ident`	✘	✘	✘	✘	✘	✘	✘
`outer.icmp6.type`	`outer.icmp.type`	`outer.icmp.type`	✘	✘	✘	✘	✘	✘	✘
`outer.icmp6.code`	`outer.icmp.code`	`outer.icmp.code`	✘	✘	✘	✘	✘	✘	✘
`tunnel.gre.protocol`	`tun.protocol`	`tun.protocol`	✘	✘	✘	✘	✘	✘	✘

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
`tunnel.gre_key.value`	`tun.gre_key`	`tun.gre_key`	✘	✘	✘	✘	✘	✘	✘
`tunnel.nvgre.protocol`	`tun.protocol`	`tun.protocol`	✘	✘	✘	✘	✘	✘	✘
`tunnel.nvgre.nvgre_vs_id`	`tun.nvgre_vs_id`	`tun.nvgre_vs_id`	✘	✘	✘	✘	✘	✘	✘
`tunnel.nvgre.nvgre_flow_id`	`tun.nvgre_flow_id`	`tun.nvgre_flow_id`	✘	✘	✘	✘	✘	✘	✘
`tunnel.vxlan.vni`	`tun.vxlan_tun_id`	`tun.vxlan_tun_id`	✓	✘	✓	✓	✓	✘	✘
`tunnel.vxlan_gpe.vni`	`tun.vxlan_tun_id`	`tun.vxlan_tun_id`	✓	✘	✓	✓	✓	✘	✘
`tunnel.vxlan_gbp.vni`	`tun.vxlan_tun_id`	`tun.vxlan_tun_id`	✓	✘	✓	✓	✓	✘	✘
`tunnel.vxlan_gpe.next_proto`	`tun.vxlan_next_protocol`		✘	✘	✘	✘	✘	✘	✘
`tunnel.vxlan_gbp.policy_id`	`tun.vxlan_group_policy_id`		✘	✘	✘	✘	✘	✘	✘
`tunnel.vxlan.rsvd1`		`tun.vxlan_tun_rsvd1` [1]	✓	✘	✘	✓	✓	✘	✘
`tunnel.vxlan_gpe.rsvd1`		`tun.vxlan_tun_rsvd1` [1]	✓	✘	✘	✓	✓	✘	✘
`tunnel.vxlan_gbp.rsvd1`		tun.vxlan_tun_rsvd1 [1]	✓	✘	✘	✓	✓	✘	✘
`tunnel.gtp.teid`	`tun.gtp_teid`	`tun.gtp_teid`	✓	✘	✓	✓	✓	✘	✘
`tunnel.esp.spi`	`tun.esp_spi`	`tun.esp_spi`	✓	✘	✓	✓	✓	✘	✘
`tunnel.esp.sn`	`tun.esp_sn`	`tun.esp_sn`	✓	✘	✓	✓	✓	✓	✓
`tunnel.psp.nexthdr`	`tun.psp.nexthdr`	`tun.psp.nexthdr`	✓	✘	✓	✓	✓	✘	✘
`tunnel.psp.hdrextlen`	`tun.psp.hdrextlen`	`tun.psp.hdrextlen`	✓	✘	✓	✓	✓	✘	✘
`tunnel.psp.res_cryptofst`	`tun.psp.res_cryptofst`	`tun.psp.res_cryptofst`	✓	✘	✓	✓	✓	✘	✘
`tunnel.psp.s_d_ver_v`	`tun.psp.s_d_ver_v`	`tun.psp.s_d_ver_v`	✓	✘	✓	✓	✓	✘	✘
`tunnel.psp.spi`	`tun.psp.spi`	`tun.psp.spi`	✓	✘	✓	✓	✓	✘	✘

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
`tunnel.psp.iv`	`tun.psp.iv`	`tun.psp.iv`	✓	✘	✓	✓	✓	✘	✘
`tunnel.psp.vc`	`tun.psp.vc`	`tun.psp.vc`	✓	✘	✓	✓	✓	✘	✘
`tunnel.mpls[0].label`	`tun.mpls[0].label`	`tun.mpls[0].label`	✘	✘	✓	✘	✓	✘	✘
`tunnel.mpls[1].label`	`tun.mpls[1].label`	`tun.mpls[1].label`	✘	✘	✓	✘	✓	✘	✘
`tunnel.mpls[2].label`	`tun.mpls[2].label`	`tun.mpls[2].label`	✘	✘	✓	✘	✓	✘	✘
`tunnel.mpls[3].label`	`tun.mpls[3].label`	`tun.mpls[3].label`	✘	✘	✓	✘	✓	✘	✘
`tunnel.mpls[4].label`	`tun.mpls[4].label`	`tun.mpls[4].label`	✘	✘	✓	✘	✓	✘	✘
`tunnel.geneve.ver_opt_len`	`tun.geneve.ver_opt_len`	`tun.geneve.ver_opt_len`	✘	✘	✘	✘	✘	✘	✘
`tunnel.geneve.o_c`	`tun.geneve.o_c`	`tun.geneve.o_c`	✘	✘	✘	✘	✘	✘	✘
`tunnel.geneve.next_proto`	`tun.geneve.next_proto`	`tun.geneve.next_proto`	✘	✘	✘	✘	✘	✘	✘
`tunnel.geneve.vni`	`tun.geneve.vni`	`tun.geneve.vni`	✓	✘	✓	✓	✓	✘	✘
`tunnel.geneve_opt[i].type`	None. See section "Copy Geneve Options" for details.		✓	✘	✓	✓	✓	✘	✘
`tunnel.geneve_opt[i].class`			✓	✘	✓	✓	✓	✘	✘
`tunnel.geneve_opt[i].data`			✓	✘	✓	✓	✓	✘	✘
`inner.eth.dst_mac`	`inner.eth.dst_mac`		✘	✘	✓	✘	✓	✘	✘
`inner.eth.src_mac`	`inner.eth.src_mac`		✘	✘	✓	✘	✓	✘	✘
`inner.eth.type`	`inner.eth.type`		✘	✘	✓	✘	✓	✘	✘
`inner.eth_vlan0.tci`	`inner.eth_vlan[0].tci`		✘	✘	✘	✘	✘	✘	✘
`inner.eth_vlan1.tci`	`inner.eth_vlan[1].tci`		✘	✘	✘	✘	✘	✘	✘
`inner.ipv4.src_ip`	`inner.ip4.src_ip`		✘	✘	✓	✘	✓	✘	✘

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
`inner.ipv4.dst_ip`	`inner.ip4.dst_ip`		✘	✘	✓	✘	✓	✘	✘
`inner.ipv4.dscp_ecn`	`inner.ip4.dscp_ecn`		✘	✘	✓	✘	✓	✘	✘
`inner.ipv4.next_proto`	`inner.ip4.next_proto`		✘	✘	✘	✘	✘	✘	✘
`inner.ipv4.ttl`	`inner.ip4.ttl`		✘	✘	✓	✘	✓	✘	✘
`inner.ipv4.version_ihl`	`inner.ip4.version_ihl`		✘	✘	✓	✘	✓	✘	✘
`inner.ipv4.total_len`	`inner.ip4.total_len`		✘	✘	✓	✘	✓	✘	✘
`inner.ipv6.src_ip`	`inner.ip6.src_ip`		✘	✘	✓	✘	✓	✘	✘
`inner.ipv6.dst_ip`	`inner.ip6.dst_ip`		✘	✘	✓	✘	✓	✘	✘
`inner.ipv6.traffic_class`	`inner.ip6.traffic_class`		✘	✘	✓	✘	✓	✘	✘
`inner.ipv6.flow_label`	`inner.ip6.flow_label`		✘	✘	✓	✘	✓	✘	✘
`inner.ipv6.next_proto`	`inner.ip6.next_proto`		✘	✘	✘	✘	✘	✘	✘
`inner.ipv6.hop_limit`	`inner.ip6.hop_limit`		✘	✘	✓	✘	✓	✘	✘
`inner.ipv6.payload_len`	`inner.ip6.payload_len`		✘	✘	✓	✘	✓	✘	✘
`inner.udp.src_port`	`inner.udp.l4_port.src_port`		✘	✘	✓	✘	✓	✘	✘
`inner.udp.dst_port`	`inner.udp.l4_port.dst_port`		✘	✘	✓	✘	✓	✘	✘
`inner.transport.src_port`	`inner.transport.src_port`		✘	✘	✓	✘	✓	✘	✘
`inner.transport.dst_port`	`inner.transport.dst_port`		✘	✘	✓	✘	✓	✘	✘
`inner.tcp.src_port`	`inner.tcp.l4_port.src_port`		✘	✘	✓	✘	✓	✘	✘
`inner.tcp.dst_port`	`inner.tcp.l4_port.dst_port`		✘	✘	✓	✘	✓	✘	✘
`inner.tcp.flags`	`inner.tcp.flags`		✘	✘	✘	✘	✘	✘	✘

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
`inner.tcp.data_offset`	`inner.tcp.data_offset`		✘	✘	✓	✘	✓	✘	✘
`inner.icmp4.type`	`inner.icmp.type`		✘	✘	✘	✘	✘	✘	✘
`inner.icmp4.code`	`inner.icmp.code`		✘	✘	✘	✘	✘	✘	✘
`inner.icmp4.ident`	`inner.icmp.ident`		✘	✘	✘	✘	✘	✘	✘
`inner.icmp6.type`	`inner.icmp.type`		✘	✘	✘	✘	✘	✘	✘
`inner.icmp6.code`	`inner.icmp.code`		✘	✘	✘	✘	✘	✘	✘

1. `tun.vxlan_tun_rsvd1` modifications only work for traffic with the default UDP destination port (i.e., 4789 for VXLAN and VXLAN-GBP and 4790 for VXLAN-GPE) ↩ ↩ ↩

14.4.2.11.2 Supported Non-field String

Users can modify fields which are not included in `doca_flow_match` structure.

14.4.2.11.2.1 Copy Hash Result

Users can copy the the matcher hash calculation into other fields using the `"parser_meta.hash"` string.

14.4.2.11.2.2 Copy GENEVE Options

User can copy GENEVE option type/class/data using the following strings:

- `"tunnel.geneve_opt[i].type"` - Copy from/to option type (only for option configured with `DOCA_FLOW_PARSER_GENEVE_OPT_MODE_MATCHABLE`).
- `"tunnel.geneve_opt[i].class"` - Copy from/to option class (only for option configured with `DOCA_FLOW_PARSER_GENEVE_OPT_MODE_MATCHABLE`).
- `"tunnel.geneve_opt[i].data"` - Copy from/to option data, the bit offset is from the start of the data.

`i` is the index of the option in `tlv_list` array provided in `doca_flow_parser_geneve_opt_create`.

14.4.2.12 DOCA Flow Connection Tracking

This guide provides an overview and configuration instructions for DOCA Flow CT API.

14.4.2.12.1 Introduction

DOCA Flow Connection Tracking (CT) is a 5-tuple table which supports the following:

- Track 5-tuple sessions (or 6-tuple when a zone is available)
- Zone based – virtual tables
- Aging (i.e., removes idle connections)
- Sets metadata for a connection
- Bidirectional packet handling
- High rate of connections per second (CPS)

The CT module makes it simple and efficient to track connections by leveraging hardware resources. The module supports both autonomous and managed mode.

14.4.2.12.2 Architecture

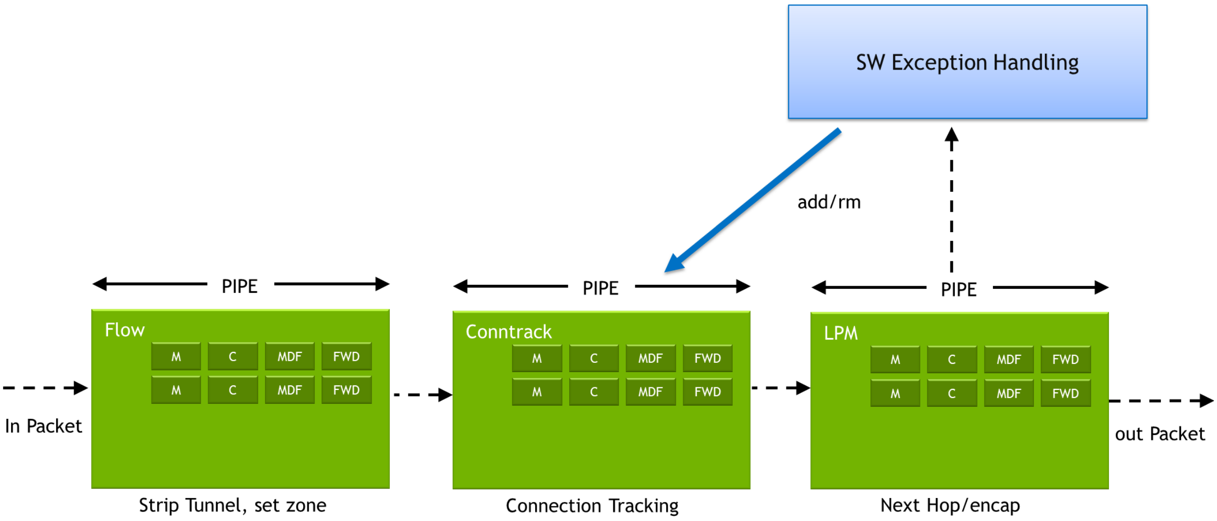
DOCA Flow CT pipe handles non-encapsulated TCP and UDP packets. The CT pipe only supports forward to next pipe or miss to next pipe actions:

- All packets matching known connection 6-tuples are forwarded to the CT's forward pipe
- Non-matching packets are forwarded to the miss pipe

The user application must handle packets accordingly.

The DOCA Flow CT API is built around four major parts:

- CT module manipulation – configuring CT module resources
- CT connection entry manipulation – adding, removing, or updating connection entries
- Callbacks – handling asynchronous entry processing result
- Pipe and entry statistics



14.4.2.12.2.1 Aging

Aging time is a time in seconds that sets the maximum allowed time for a session to be maintained without a packet seen. If that time elapses with no packet being detected, the session is terminated.

To support aging, a dedicated aging thread is started to poll and check counters for all connections.

14.4.2.12.2.2 Autonomous Mode

In this mode, DOCA runs multiple CT workers internally, to handle connections in parallel.

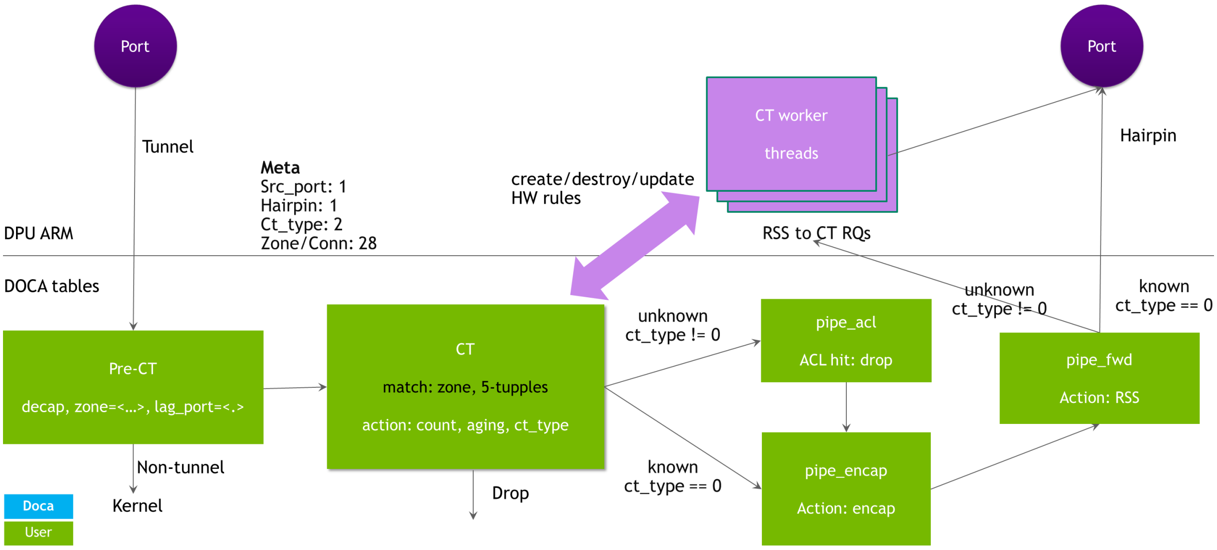
A connection's lifecycle is controlled by the connection state encapsulated in the packet and time-based aging.

CT workers establish and close connections automatically based on the connection's state stored in packet meta.

Packet meta is defined as follows:

```
uint32_t src : 1;       /**< Source port in multi-port E-Switch mode */
uint32_t hairpin : 1;   /**< Subject to forward using hairpin. */
uint32_t type : 2;      /**< CT packet type: New, End or Update */
uint32_t data : 28;     /**< Zone set by user or reserved after CT pipe. */
```

- `data` - CT table matches on packet meta (zone) and 5-tuples
- `type` - can have the following values:
  - `NONE` - (known) if packet hit any connection rule
  - `NEW` - if new TCP or UDP connection
  - `END` - if TCP connection closed
- `src` and `hairpin` - used for forwarding pipe and worker to deliver packet



14.4.2.12.2.3 Managed Mode

The application is responsible for managing the worker threads in this mode, parsing and handling the connection's lifecycle.

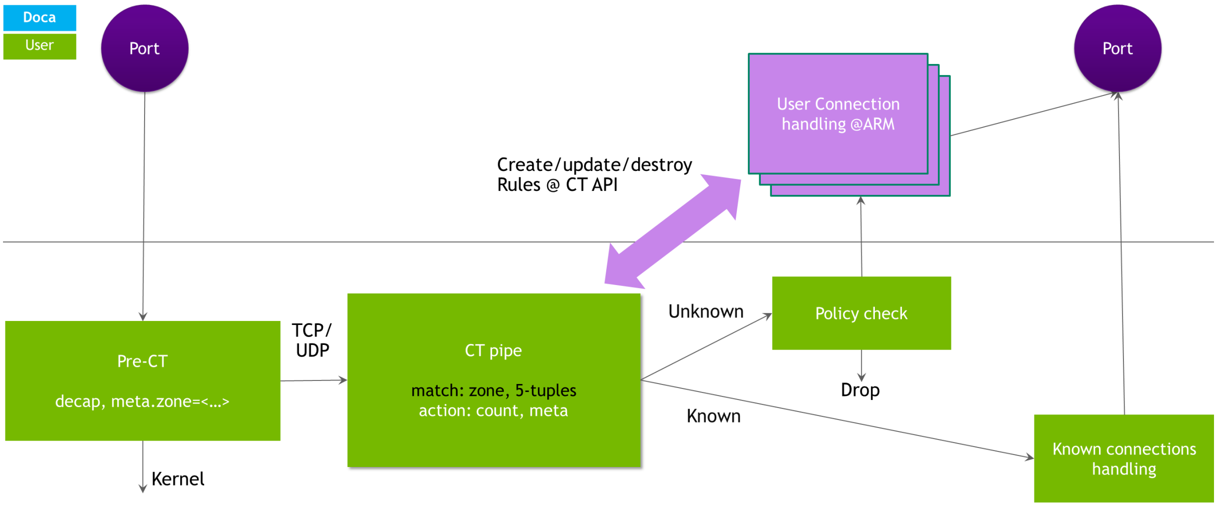
Managed mode uses DOCA Flow CT management APIs to create or destroy the connections.

The CT aging module notifies on aged out connections by calling callbacks.

Users can create connection rules with a different pattern, meta, or counter, for each packet direction.

Users are responsible for defining meta and mask to `match` and `modify`.

Users can create one rule of a connection first, then create another rule using API `doca_flow_ct_entry_add_dir()`.



DOCA Flow API can be used to process CT entries with a CT-dedicated queue.

- `doca_flow_entries_process` - process pipe entries in queue
- `doca_flow_aging_handle` - handle pipe entries aging

Other DOCA Flow APIs like CT entry status query and pipe miss query are not supported.

14.4.2.12.3 Prerequisites

14.4.2.12.3.1 DPU

NVIDIA® BlueField®-3 and above is required to support IPv6.

To enable DOCA Flow CT on the DPU, perform the following on the Arm:

1. Enable `iommu.passthrough` in Linux boot commands (or disable SMMU from the DPU BIOS):
   a. Run:

   ```
   sudo vim /etc/default/grub
   ```

   b. Set `GRUB_CMDLINE_LINUX="iommu.passthrough=1"`.
   c. Run:

   ```
   sudo update-grub
   sudo reboot
   ```

2. Configure DPU firmware with `LAG_RESOURCE_ALLOCATION=1`:

```
sudo mlxconfig -d <device-id> s LAG_RESOURCE_ALLOCATION=1
```

> Retrieve `device-id` from the output of the `mst status -v` command. If, under the MST tab, the value is N/A, run the `mst start` command.

3. Update `/etc/mellanox/mlnx-bf.conf` as follows:

```
ALLOW_SHARED_RQ="no"
```

4. Perform power cycle on the host and Arm sides.
5. If working with a single port, set the DPU into e-switch mode:

```
sudo devlink dev eswitch set pci/<pcie-address> mode switchdev
sudo devlink dev param set pci/<pcie-address> name esw_multiport value false cmode runtime
```

> Retrieve `pcie-address` from the output of the `mst status -v` command.

6. If working with two PF ports, set the DPU into multi-port e-switch mode (for the 2 PCIe devices):

```
sudo devlink dev param set pci/<pcie-address> name esw_multiport value true cmode runtime
```

> Retrieve `pcie-address` from the output of the `mst status -v` command.

7. Define huge pages (see DOCA Flow prerequisites).

14.4.2.12.3.2 ConnectX

To enable DOCA Flow CT on the NVIDIA® ConnectX®, perform the following:

1. Configure firmware with `LAG_RESOURCE_ALLOCATION=1`:

```
sudo mlxconfig -d <device-id> s LAG_RESOURCE_ALLOCATION=1
```

> Retrieve `device-id` from the output of the `mst status -v` command. If, under the MST tab, the value is N/A, run the `mst start` command.

2. Perform power cycle.
3. If working with a single port:

```
sudo devlink dev eswitch set pci/<pcie-address> mode switchdev
sudo devlink dev param set pci/<pcie-address> name esw_multiport value false cmode runtime
```

> Retrieve `pcie-address` from the output of the `mst status -v` command.

4. If working with two PF ports:

   ```
   sudo devlink dev eswitch set pci/<pcie-address0> mode switchdev
   sudo devlink dev eswitch set pci/<pcie-address1> mode switchdev
   sudo devlink dev param set pci/<pcie-address0> name esw_multiport value true cmode runtime
   sudo devlink dev param set pci/<pcie-address1> name esw_multiport value true cmode runtime
   ```

   ⓘ Retrieve `pcie-address` from the output of the `mst status -v` command.

5. Define huge pages (see DOCA Flow prerequisites).

14.4.2.12.4 Actions

DOCA Flow CT supports actions based on meta and NAT operations. Each action can be defined as either shared or non-shared.

14.4.2.12.4.1 Shared Actions

Actions that can be shared between entries. Shared actions are predefined and reused in multiple entries.

The user gets a handle per shared action created and uses this handle as a reference to the action where required.

ⓘ It is user responsibility to track shared actions and to remove them when they become irrelevant.

Shared actions are defined using a control queue (see struct doca_flow_ct_cfg).

14.4.2.12.4.2 Non-shared Actions

Actions provided with their data during entry create/update.

These actions are completely managed by DOCA Flow CT and cannot be reused in multiple flows (i.e., NAT operations).

14.4.2.12.4.3 Action Sets in Pipe Creation

Users must define action sets during DOCA Flow CT pipe creation (as with any other pipe).

ⓘ Only actions for meta and NAT are accepted (according to struct doca_flow_ct_actions).

During entry create/update, different actions can be provided per direction (different action content and/or different type).

14.4.2.12.4.4 Feature Enable

To enable user actions, configure the following parameters:

- User action templates during DOCA Flow CT pipe creation
- Maximum number of user actions (`nb_user_actions` on DOCA Flow CT init)

14.4.2.12.4.5 Using Actions in Autonomous Mode

Init

Configure the following parameters on `doca_flow_ct_init()` :

- `nb_ctrl_queues` - number of control queues for defining shared actions
- `nb_user_actions` - maximum number of actions (shared and non-shared)
- `worker_cb` - callbacks required to communicate with the user

Create DOCA Flow CT Pipe

Configure actions sets on `doca_flow_pipe_create()` .

Create Shared Actions

Use `doca_flow_ct_actions_add_shared()` with one of the control queues.

Shared actions can be added at any time before use.

Implement Worker Callbacks

Callbacks are called from each worker thread to acquire synchronization with the user code and on the first packet of a flow.

On `doca_flow_ct_rule_pkt_cb` :

- Determine how the packet should be treated
- If rules are required, return the actions handles to use

14.4.2.12.4.6 Using Actions in Managed Mode

Init

Configure the following parameters on `doca_flow_ct_init()` :

- `nb_ctrl_queues` - number of control queues for defining shared actions
- `nb_user_actions` - maximum number of user actions. Both shared control queues and non-shared control queues cache actions IDs to speed up ID allocation, each queue cache max 1024 IDs. The user must configure expected number of actions + total queues * 1024. The number can't exceed the number of actions hardware supported.

Create DOCA Flow CT Pipe

Configure actions sets on `doca_flow_pipe_create()` .

Create Shared Actions

Use `doca_flow_ct_actions_add_shared()` with one of the control queues.

Shared actions can be added at any time before use.

Add Entry

Entry can be created in one of the following ways:

- Using an action handle of a predefined shared action
- Using action data, which is specific to the flow, not sharable (e.g., for NAT operations)

The entry can have different actions and/or different action types per direction.

Remove Entry

Non-shared actions associated with an entry are implicitly destroyed by DOCA Flow CT.

Shared actions are not destroyed. They can be used by the user until they decide to remove them.

Update Entry

Entry actions can be updated per direction. All combinations of shared/non-shared actions are applicable (e.g., update from shared to non-shared).

14.4.2.12.5 Changeable Forward

DOCA Flow CT allows using a different forward pipe per flow direction.

DOCA Flow CT supports the forward pipe in two levels:

- Pipe level - a single forward pipe defined during DOCA Flow CT pipe creation and used for all entries
- Entry level - forward pipe defined during entry create
- DOCA Flow CT operates in one of the two levels

DOCA CT forward in entry level has the following characteristics:

- Supports only `DOCA_FLOW_FWD_PIPE` (up to 4 different forward pipes)
- Supports forward pipe per flow direction (both directions can have same/different forward pipe)
- Must set forward pipes on each entry create (no default forward pipe)

Turn on the feature:

1. Create DOCA Flow CT pipe with forward type = `DOCA_FLOW_FWD_PIPE` and `next_pipe` = `NULL` .
2. Call to `doca_flow_ct_fwd_register` to register forward pipes and get `fwd_handles` in return.

14.4.2.12.5.1 Using Changeable Forward in Managed Mode

1. Initialize DOCA Flow CT (`doca_flow_ct_init`).
2. Register forward pipes (`doca_flow_ct_fwd_register`).
    - Define pipes that can be used for forward
3. Create DOCA Flow CT pipe (`doca_flow_pipe_create`) with definition of possible forward pipes.
4. Add entry (`doca_flow_ct_add_entry`).
    - Set origin and/or reply `fwd_handles` returned from `doca_flow_ct_fwd_register` .
5. Update forward for entry direction (`doca_flow_ct_update_entry`).

⚠ Updating forward handle requires setting all other parameters with their previous values.

14.4.2.12.5.2 Using Changeable Forward in Autonomous Mode

1. Initialize DOCA Flow CT (`doca_flow_ct_init`).
2. Register forward pipes (`doca_flow_ct_fwd_register`).
   - Define pipes that can be used for forward.
3. Create DOCA Flow CT pipe (`doca_flow_pipe_create`) with definition of possible forward pipes.
4. CT workers start to handle traffic.
5. On the first flow packet, `doca_flow_ct_rule_pkt` callback is called.
   - In this callback, determine if the entry should be created, and which actions and/or forward handles should be used for this entry.

ⓘ Update forward for entry direction is not supported.

14.4.2.12.6 API

For the library API reference, refer to DOCA Flow and CT API documentation in the NVIDIA DOCA Library APIs.

⚠ The pkg-config (`*.pc` file) for the Flow CT library is included in DOCA's regular definitions : `doca` .

The following sections provide additional details about the library API.

14.4.2.12.6.1 enum doca_flow_ct_flags

DOCA Flow CT configuration optional flags.

Flag	Description
`DOCA_FLOW_CT_FLAG_STATS = 1u << 0`	Enable internal pipe counters for packet tracking purposes. Call `doca_flow_pipe_dump(<ct_pipe>)` to dump counter values. Each call dumps values changed.
`DOCA_FLOW_CT_FLAG_WORKER_STATS = 1u << 1,`	Enable worker thread internal debug counter periodical dump. Autonomous mode only.
`DOCA_FLOW_CT_FLAG_NO_AGING = 1u << 2,`	Disable aging
`DOCA_FLOW_CT_FLAG_SW_PKT_PARSING = 1u << 3,`	Enable CT worker software packet parsing to support VLAN, IPv6 options, or special tunnel types

Flag	Description
`DOCA_FLOW_CT_FLAG_MANAGED = 1u << 4,`	Enable managed mode in which user application is responsible for managing packet handling, and calling the CT API to manipulate CT connection entries
`DOCA_FLOW_CT_FLAG_ASYMMETRIC = 1u << 5,`	Allows different 6-tuple table definitions for the origin and reply directions. Default to symmetric mode, uses same meta and reverse 5-tuples for reply direction. Managed mode only.
`DOCA_FLOW_CT_FLAG_ASYMMETRIC_COUNTER = 1u << 6,`	Enable different counters for the origin and reply directions. Managed mode only.
`DOCA_FLOW_CT_FLAG_NO_COUNTER = 1u << 7,`	Disable counter and aging to save aging thread CPU cycles
`DOCA_FLOW_CT_FLAG_DEFAULT_MISS = 1u << 8,`	Check TCP SYN flags and UDP in CT miss flow to identify ADD type packets.
`DOCA_FLOW_CT_FLAG_WIRE_TO_WIRE = 1u << 9,`	Hint traffic comes from uplink wire and forwards to uplink wire. ⚠ If this flag is set, the direction info must be `DOCA_FLOW_DIRECTION_NETWORK_TO_HOST`.
`DOCA_FLOW_CT_FLAG_CALC_TUN_IP_CHKSUM = 1u << 10,`	Enable hardware to calculate and set the checksum on L3 header (IPv4)
`DOCA_FLOW_CT_FLAG_DUP_FILTER_UDP_ONLY = 1u << 11,`	Apply the connection duplication filter for UDP connections only

14.4.2.12.6.2 enum doca_flow_ct doca_flow_ct_entry_flags

DOCA Flow CT Entry optional flags.

Flag	Description
`DOCA_FLOW_CT_ENTRY_FLAGS_NO_WAIT = (1 << 0)`	Entry is not buffered; send to hardware immediately
`DOCA_FLOW_CT_ENTRY_FLAGS_DIR_ORIGIN = (1 << 1)`	Apply flags to origin direction
`DOCA_FLOW_CT_ENTRY_FLAGS_DIR_REPLY = (1 << 2)`	Apply flags to reply direction
`DOCA_FLOW_CT_ENTRY_FLAGS_IPV6_ORIGIN = (1 << 3)`	Origin direction is IPv6; origin match union in struct `doca_flow_ct_match` is IPv6
`DOCA_FLOW_CT_ENTRY_FLAGS_IPV6_REPLY = (1 << 4)`	Reply direction is IPv6; reply match union in struct `doca_flow_ct_match` is IPv6
`DOCA_FLOW_CT_ENTRY_FLAGS_COUNTER_ORIGIN = (1 << 5)`	Apply counter to origin direction
`DOCA_FLOW_CT_ENTRY_FLAGS_COUNTER_REPLY = (1 << 6)`	Apply counter to reply direction

Flag	Description
`DOCA_FLOW_CT_ENTRY_FLAGS_COUNTER_SHARED = (1 << 7)`	Counter is shared for both direction (origin and reply)
`DOCA_FLOW_CT_ENTRY_FLAGS_FLOW_LOG = (1 << 8)`	Enable flow log on entry removed
`DOCA_FLOW_CT_ENTRY_FLAGS_ALLOC_ON_MISS = (1 << 9)`	Allocate on entry not found when calling `doca_flow_ct_entry_prepare()` API
`DOCA_FLOW_CT_ENTRY_FLAGS_DUP_FILTER_ORIGIN = (1 << 10)`	Enable duplication filter on origin direction
`DOCA_FLOW_CT_ENTRY_FLAGS_DUP_FILTER_REPLY = (1 << 11)`	Enable duplication filter on reply direction

14.4.2.12.6.3 enum doca_flow_ct_rule_opr

Options for handling flows in autonomous mode with shared actions. The decision is taken on the first flow packet.

Operation	Description
`DOCA_FLOW_CT_RULE_OK`	Flow should be defined in the CT pipe using the required shared actions handles
`DOCA_FLOW_CT_RULE_DROP`	Flow should not be defined in the CT pipe. The packet should be dropped.
`DOCA_FLOW_CT_RULE_TX_ONLY`	Flow should not be defined in the CT pipe. The packet should be transmitted.

14.4.2.12.6.4 struct direction_cfg

Managed mode configuration for origin or reply direction.

Field	Description
`bool match_inner`	5-tuple match pattern applies to packet inner layer
`struct doca_flow_meta *zone_match_mask`	Mask to indicate meta field and bits to match
`struct doca_flow_meta *meta_modify_mask`	Mask to indicate meta field and bits to modify on connection packet match

14.4.2.12.6.5 struct doca_flow_ct_worker_callbacks

Set of callbacks for using shared actions in autonomous mode.

Field	Description
`doca_flow_ct_sync_acquire_cb worker_init`	Called at the start of a worker thread to sync with the user context

Field	Description
`doca_flow_ct_sync_release_cb worker_release`	Called at the end of a worker thread
`doca_flow_ct_rule_pkt_cb rule_pkt`	Called on the first packet of a flow

14.4.2.12.6.6 struct doca_flow_ct_cfg

DOCA Flow CT configuration.

```
uint32_t nb_arm_queues;
uint32_t nb_ctrl_queues;
uint32_t nb_user_actions;
uint32_t nb_arm_sessions[DOCA_FLOW_CT_SESSION_MAX];
uint32_t flags;
uint16_t aging_core;
uint16_t aging_query_delay_s;
doca_flow_ct_flow_log_cb flow_log_cb;
struct doca_flow_ct_aging_ops *aging_ops;
uint32_t base_core_id;
uint32_t dup_filter_sz;
union {
                /* Managed mode configuration for origin and reply direction. */
                struct direction_cfg direction[2];

                /* Below fields are dedicate for autonomous mode */
                struct {
                        uint16_t tcp_timeout_s;
                        uint16_t tcp_session_del_s;
                        uint16_t udp_timeout_s;
                        enum doca_flow_tun_type tunnel_type;
                        uint16_t vxlan_dst_port;
                        enum doca_flow_ct_hash_type hash_type;
                        uint32_t meta_user_bits;
                        uint32_t meta_action_bits;
                        struct doca_flow_meta *meta_zone_mask;
                        struct doca_flow_meta *connection_id_mask;
                        struct doca_flow_ct_worker_callbacks worker_cb;
               };
       };
```

Where:

Field	Description
`uint32_t nb_arm_queues`	Number of CT queues. In autonomous mode, also the number of worker threads.
`uint32_t nb_ctrl_queues`	Number of CT control queues used for defining shared actions
`uint32_t nb_user_actions`	Maximum number of user actions supported (shared and non-shared) Minimum value is 1K * `nb_ctrl_queues`
`uint32_t nb_arm_sessions[DOCA_FLOW_CT_SESSION_MAX]`	Maximum number of IPv4 and IPv6 CT connections
`uint32_t flags`	CT configuration flags
`uint16_t aging_core`	CPU core ID for CT aging thread to bind.
`uint16_t aging_core_delay`	CT aging code delay.
`doca_flow_ct_flow_log_cb flow_log_cb`	Flow log callback function, when set
`struct doca_flow_ct_aging_ops *aging_ops`	User-defined aging logic callback functions. Fallback to default aging logic

Field	Description
`uint32_t base_core_id`	Base core ID for the workers
`uint32_t dup_filter_sz`	Number of connections to cache in the duplication filter
`struct direction_cfg direction`	Managed mode configuration for origin or reply direction
`uint16_t tcp_timeout_s`	TCP timeout in seconds
`uint16_t tcp_session_del_s`	Time to delay or kill TCP session after RST/FIN
`enum doca_flow_tun_type tunnel_type`	Encapsulation tunnel type
`uint16_t vxlan_dst_port`	VXLAN outer UDP destination port in big endian
`enum doca_flow_ct_hash_type hash_type`	Type of connection hash table type: `NONE` or `SYMMETRIC_HASH`
`uint32_t meta_user_bits`	User packet meta bits to be owned by the user
`uint32_t meta_action_bits`	User packet meta bits to be carried by identified connection packet
`struct doca_flow_meta *meta_zone_mask`	Mask to indicate meta field and bits saving zone information
`struct doca_flow_meta *connection_id_mask`	Mask to indicate meta field and bits for CT internal connection ID
`struct doca_flowct_worker_callbacks worker_cb`	Worker callbacks to use shared actions

14.4.2.12.6.7 struct doca_flow_ct_actions

This structure is used in the following cases:

- For defining shared actions. In this case, action data is provided by the user. The action handle is returned by DOCA Flow CT.
- For defining an entry with actions. The structure can be filled with two options:
  - With action handle of a previously created shared action
  - With non-shared action data

DOCA Flow CT action structure.

```
enum doca_flow_resource_type  resource_type;
union {
               /* Used when creating an entry with a shared action. */
               uint32_t action_handle;

               /* Used when creating an entry with non-shared action or when creating a shared action. */
               struct {
                          uint32_t action_idx;
                          struct doca_flow_meta meta;
                          struct doca_flow_header_l4_port l4_port;
                          union {
                                    struct doca_flow_ct_ip4 ip4;
                                    struct doca_flow_ct_ip6 ip6;
                           };
               } data;
       };
```

Where:

Field	Description
`enum doca_flow_resource_type resource_type`	Shared/non-shared action
`uint32_t action_handle`	Shared action handle
`uint32_t action_idx`	Actions template index
`struct doca_flow_meta meta`	Modify meta values
`struct doca_flow_header_l4_port l4_port`	UDP or TCP source and destination port
`struct doca_flow_ct_ip4 ip4`	Source and destination IPv4 addresses
`struct doca_flow_ct_ip6 ip6`	Source and destination IPv6 addresses

14.4.2.12.7 DOCA Flow Connection Tracking Samples

This section describes DOCA Flow CT samples based on the DOCA Flow CT pipe.

The samples illustrate how to use the library API to manage TCP/UDP connections.

> All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

14.4.2.12.7.1 Running the Samples

1. Refer to the following documents:
    - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
    - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample:

```
cd /opt/mellanox/doca/samples/doca_flow/flow_ct_udp
meson /tmp/build
ninja -C /tmp/build
```

> The binary `doca_flow_ct_udp` is created under `/tmp/build/samples/`.

3. Sample (e.g., `doca_flow_ct_udp`) usage:

```
Usage: doca_<sample_name> [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                              Print a help synopsis
  -v, --version                           Print program version information
  -l, --log-level                         Set the (numeric) log level for the program <10=DISABLE, 20=CRITI
CAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                         Set the SDK (numeric) log level for the program <10=DISABLE, 20=C
RITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                       Parse all command flags from an input json file

Program Flags:
  -p, --pci_addr <PCI-ADDRESS>            PCI device address
```

4. For additional information per sample, use the `-h` option:

```
/tmp/build/samples/<sample_name> -h
```

5. The following is a CLI example for running the samples when port `03:00.0` is configured (multi-port e-switch) as manager port:

```
/tmp/build/samples/doca_<sample_name> -- -p 03:00.0 -l 60
```

> To avoid the test being impacted by unexpected packets, it only accepts packets like the following examples:
>
> - IPv4 destination address is "1.1.1.1"
> - IPv6 destination address is "0101:0101:0101:0101:0101:0101:0101:0101"

14.4.2.12.7.2 Samples

> **Duplication filter**
>
> All CT UDP samples demonstrate the usage of the connection's duplication filter. Duplication filter is used if the user is interested in preventing same connection rule insertion in a high-rate workload environment.

Flow CT 2 Ports

This sample illustrates how to create a simple pipeline on two standalone e-switches. Multi-port e-switch must be disabled.

```
sudo devlink dev eswitch set pci/<pcie-address0> mode switchdev
sudo devlink dev eswitch set pci/<pcie-address1> mode switchdev
sudo devlink dev param set pci/<pcie-address0> name esw_multiport value false cmode runtime
```

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct .
2. Initializing DOCA Flow CT.
3. Starting two DOCA Flow uplink ports where port 0 and 1 each has a special role of being a switch manager port.

   > Ports are configured according to the parameters provided to `doca_dpdk_port_probe()` in the main function.

4. Creating a pipeline on each port:
   a. Building an UDP pipe to filter non-UDP packets.
   b. Building a CT pipe to hold UDP session entries.
   c. Building a counter pipe with an example 5-tuple entry to which non-unidentified UDP sessions should be sent.
   d. Building a hairpin pipe to send back packets.
   e. Building an RSS pipe from which all packets are directed to the sample main thread for parsing and processing.

5. Packet processing on each port:
    a. The first UDP packet triggers the miss flow as the CT pipe is empty.
    b. Performing 5-tuple packet parsing.
    c. Calling `doca_flow_ct_add_entry()` to create a hardware rule according to the parsed 5-tuple info.
    d. The second UDP packet based on the the same 5-tuple should be sent again. Packet hits the hardware rule inserted before and sent back to egress.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp/flow_ct_2_ports_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp/flow_ct_2_ports_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp/meson.build`

Flow CT UDP

This sample illustrates how to create a simple UDP pipeline with a CT pipe in it.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct .
2. Initializing DOCA Flow CT.
3. Starting two DOCA Flow uplink representor ports where port 0 has a special role of being a switch manager port.

    > ⓘ Ports are configured according to the parameters provided to `doca_dpdk_port_probe()` in the main function.

4. Creating a pipeline on the main port:
    a. Building an UDP pipe to filter non-UDP packets.
    b. Building a CT pipe to hold UDP session entries.
    c. Building a counter pipe with an example 5-tuple entry to which non-unidentified UDP sessions should be sent.
    d. Building a VXLAN encapsulation pipe to encapsulate all identified UDP sessions.
    e. Building an RSS pipe from which all packets are directed to the sample main thread for parsing and processing.
5. Packet processing:
    a. The first UDP packet triggers the miss flow as the CT pipe is empty.
    b. 5-tuple packet parsing is performed.
    c. `doca_flow_ct_add_entry()` is called to create a hardware rule according to the parsed 5-tuple info.
    d. The second UDP packet based on the the same 5-tuple should be sent again. Packet hits the HW rule inserted before and directed to port 0 after VXLAN encapsulation.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp/flow_ct_udp_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp/flow_ct_udp_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp/meson.build`

Flow CT UDP Query

This sample illustrates how to query a Flow CT UDP session entry. The query can be done according to session direction (origin or reply). The pipeline is identical to that of the Flow CT UDP sample.

This sample adds the following logic:

1. Dumping port 0 information into a file at `./port_0_info.txt`.
2. Querying UDP session hardware entry created after receiving the first UDP packet:
    - Origin total bytes received
    - Origin total packets received
    - Reply total bytes received
    - Reply total packets received

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp_query/flow_ct_udp_query_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp_query/flow_ct_udp_query_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp_query/meson.build`

Flow CT UDP Update

This sample illustrates how a CT entry can be updated after creation.

The pipeline is identical to that of the Flow CT UDP sample. In case of non-active UDP sessions, a relevant entry shall be updated with an aging timeout.

This sample adds the following logic:

1. Querying all UDP sessions for the total number of packets received in both the origin and reply directions.
2. Updating entry aging timeout to 2 seconds once a session is not active (i.e., no packets received on either side).
3. Waiting until all non-active session are aged and deleted.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp_update/flow_ct_udp_update_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp_update/flow_ct_udp_update_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp_update/meson.build`

Flow CT UDP Single Match

This sample is based on the Flow CT UDP sample. The sample illustrates that a hardware entry can be created with a single match (matching performed in one direction only) in the API call `doca_flow_ct_add_entry()`.

Flow CT Aging

This sample illustrates the use of the DOCA Flow CT aging functionality. It demonstrates how to build a pipe and add different entries with different aging times and user data.

No packets need to be sent for this sample.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct .
2. Initializing DOCA Flow CT.
3. Starting two DOCA Flow uplink representor ports where port 0 has a special role of being a switch manager port.

   > ⓘ Ports are configured according to the parameters provided to `doca_dpdk_port_probe()` in the main function.

4. Building a UDP pipe to serve as the root pipe.
5. Building a counter pipe with an example 5-tuple entry to which CT forwards packets.
6. Adding 32 entries with a different 5-tuple match, different aging time (3-12 seconds), and setting user data. User data will contain the port ID, entry number, and status.
7. Handling aging in small intervals and removing each entry after age-out.
8. Running these commands until all 32 entries age out.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ct_aging/flow_ct_aging_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_aging/flow_ct_aging_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_aging/meson.build`

Flow CT TCP

This sample illustrates how to manage TCP flags with CT to achieve better control over TCP sessions.

> ⓘ The sample expects to receive at least `SYN` and `FIN` packets.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct .
2. Initializing DOCA Flow CT.
3. Starting two DOCA Flow uplink representor ports where port 0 has a special role of being a switch manager port.

   > ⓘ Ports are configured according to the parameters provided to `doca_dpdk_port_probe()` in the main function.

4. Creating a pipeline on the main port:
   a. Building an TCP pipe to filter non-TCP packets.
   b. Building a CT pipe to hold TCP session entries.
   c. Building a CT miss pipe which forwards all packets to RSS pipe.

    d. Building an RSS pipe from which all packets are directed to the sample main thread for parsing and processing.
    e. Building a TCP flags filter pipe which identifies the TCP flag inside the packets. `SYN` , `FIN` , and `RST` packets are forwarded the to RSS pipe while all others are forwarded to the EGRESS pipe.
    f. Building an EGRESS pipe to forward packets to uplink representor port 1.

5. Packet processing:
    a. The first TCP packet triggers the miss flow as the CT pipe is empty.
    b. 5-tuple packet parsing is performed.
    c. TCP flag is examined.
        - In case of a `SYN` flag, a hardware entry is created.
        - For `FIN` or `RST` flags, the HW entry is removed and all packets are transferred to uplink representor port 1 using `rte_eth_tx_burst()` on port 0 (proxy port) by rte_flow_dynf_metadata_set() to 1.
    d. From this point on, all TCP packets belonging to the above session are offloaded directly to uplink port representor 1.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp/flow_ct_tcp_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp/flow_ct_tcp_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp/meson.build`

Flow CT TCP Actions

This sample illustrates how a to add shared and non-shared actions to CT TCP sessions. The pipeline is identical to that of the Flow CT TCP sample.

> The sample expects to receive at least `SYN` and `FIN` packets.

This sample adds a shared action on one side of the session that placed the value 1 in the packet's metadata, while on the other side of the session a non-shared action is placed. The non-shared action simply flips the order of the source-destination IP addresses and port numbers.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp_actions/flow_ct_tcp_actions_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp_actions/flow_ct_tcp_actions_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp_actions/meson.build`

Flow CT TCP Flow Log

This sample illustrate how to use the flow log callback to alert when a session is aged/removed.

> The sample expects to receive at least `SYN` and `FIN` packets.

This sample is based on the Flow CT TCP sample. Once a session is removed (after receiving `FIN` packet), the callback is triggered and session counters are queried.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp_flow_log/flow_ct_tcp_flow_log_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp_flow_log/flow_ct_tcp_flow_log_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp_flow_log/meson.build`

Flow CT TCP IPv4/IPv6

This sample illustrates how to manage a flow with a different IP type per direction.

In case of a `SYN` flag:

1. A single HW entry of IPv4 is created as origin direction
2. An additional HW entry of IPv6 is created as reply direction
3. From this point on, all IPv4 TCP packets (belonging to the origin direction) and all IPv6 TCP packets (belonging to the reply direction) are offloaded.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp/flow_ct_tcp_sample_ipv4_ipv6.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp/flow_ct_tcp_ipv4_ipv6_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp/meson.build`

14.4.2.13 DOCA Flow Tune Server

This guide provides an overview and configuration instructions for DOCA Flow Tune Server API.

14.4.2.13.1 Introduction

DOCA Flow Tune Server (TS), DOCA Flow subcomponent, exposes an API to collect predefined internal key performance indicators (KPIs) and pipeline visualization of a running DOCA Flow application.

Supported port KPIs:

- Total add operations across all queues
- Total update operations across all queues
- Total remove operations across all queues
- Pending operations number across all queues
- Number of `NO_WAIT` flag operations across all queues
- Number of shared resources and counters
- Number of pipes

Supported application KPIs:

- Number of ports
- Number of queues

- Queues depth

Pipeline information is saved to a JSON file to simplify its structure. Visualization is supported for the following DOCA Flow pipes:

- Basic
- Control

Each pipe contains the following fields:

- Type
- Name
- Domain
- Is root
- Match
- Match mask
- FWD
- FWD miss

Supported entry information:

- Basic
    - FWD
- Control
    - FWD
    - Match
    - Match mask
    - Priority

14.4.2.13.2 Prerequisites

DOCA Flow Tune Server API is available only by using the DOCA Flow and DOCA Flow Tune Server trace libraries.

ⓘ For more detailed information, refer to section "DOCA Flow Debug and Trace" under *DOCA Flow*.

14.4.2.13.3 API

ⓘ For more detailed information on DOCA Flow API, refer to NVIDIA DOCA Library APIs.

The following subsections provide additional details about the library API.

14.4.2.13.3.1 enum doca_flow_tune_server_kpi_type

DOCA Flow TS KPI flags.

Flag	Description
`TUNE_SERVER_KPI_TYPE_NR_PORTS,`	Retrieve port number

Flag	Description
`TUNE_SERVER_KPI_TYPE_NR_QUEUES,`	Retrieve queue number
`TUNE_SERVER_KPI_TYPE_QUEUE_DEPTH,`	Retrieve queue depth
`TUNE_SERVER_KPI_TYPE_NR_SHARED_RESOURCES,`	Retrieve shared resource and counter numbers
`TUNE_SERVER_KPI_TYPE_NR_PIPES,`	Retrieve number of pipes per port
`TUNE_SERVER_KPI_TYPE_ENTRIES_OPS_ADD,`	Retrieve entry add operations per port
`TUNE_SERVER_KPI_TYPE_ENTRIES_OPS_UPDATE,`	Retrieve entry update operations per port
`TUNE_SERVER_KPI_TYPE_ENTRIES_OPS_REMOVE,`	Retrieve entry remove operations per port
`TUNE_SERVER_KPI_TYPE_PENDING_OPS,`	Retrieve entry pending operations per port
`TUNE_SERVER_KPI_TYPE_NO_WAIT_OPS,`	Retrieve entry `NO_WAIT` flag operations per port

14.4.2.13.3.2 struct doca_flow_tune_server_shared_resources_kpi_res

Holds the number of each shared resources and counters per port.

Field	Description
`uint64_t nr_meter`	Number of meters
`uint64_t nr_counter`	Number of counters
`uint64_t nr_rss`	Number of RSS
`uint64_t nr_mirror`	Number of mirrors
`uint64_t nr_psp`	Number of PSP
`uint64_t nr_encap`	Number of encap
`uint64_t nr_decap`	Number of decap

14.4.2.13.3.3 struct doca_flow_tune_server_kpi_res

Holds the KPI result.

> ⚠ This structure is required when calling `doca_flow_tune_server_get_kpi` or `doca_flow_tune_server_get_port_kpi`.

Field	Description
`enum doca_flow_tune_server_kpi_type type`	KPI result type
`struct doca_flow_tune_server_shared_resources_kpi_res shared_resources_kpi`	Shared resource result values

Field	Description
`uint64_t val`	Result value

14.4.2.13.3.4 doca_flow_tune_server_cfg_create

Creates DOCA Flow Tune Server configuration structure.

```
doca_error_t doca_flow_tune_server_cfg_create(struct doca_flow_tune_server **cfg);
```

14.4.2.13.3.5 doca_flow_tune_server_cfg_set_bind_path

Adds local path to the configuration struct on which the DOCA Flow Tune Server AF_UNIX socket binds.

```
doca_error_t doca_flow_tune_server_cfg_set_bind_path(struct doca_flow_tune_server *cfg, const char *path, size_t
path_len);
```

14.4.2.13.3.6 doca_flow_tune_server_cfg_destroy

Destroys DOCA Flow Tune Server configuration structure.

```
doca_error_t doca_flow_tune_server_cfg_destroy(struct doca_flow_tune_server *cfg);
```

14.4.2.13.3.7 doca_flow_tune_server_init

Initializes DOCA Flow Tune Server internal structures.

```
doca_error_t doca_flow_tune_server_init(void);
```

14.4.2.13.3.8 doca_flow_tune_server_destroy

Destroys DOCA Flow Tune Server internal structures.

```
void doca_flow_tune_server_destroy(void);
```

14.4.2.13.3.9 doca_flow_tune_server_query_pipe_line

Queries and dumps pipeline info for all ports to a JSON file pointed by fp.

```
doca_error_t doca_flow_tune_server_query_pipe_line(FILE *fp);
```

14.4.2.13.3.10 doca_flow_tune_server_get_port_ids

Retrieves ports identification numbers.

```
doca_error_t doca_flow_tune_server_get_port_ids(uint16_t *port_id_arr, uint16_t port_id_arr_len, uint16_t
*nr_ports);
```

14.4.2.13.3.11 doca_flow_tune_server_get_kpi

Retrieves application scope KPI.

```
doca_error_t doca_flow_tune_server_get_kpi(enum doca_flow_tune_server_kpi_type kpi_type,
                       struct doca_flow_tune_server_kpi_res *res)
```

14.4.2.13.3.12 doca_flow_tune_server_get_port_kpi

Retrieves port scope KPI.

```
doca_error_t doca_flow_tune_server_get_port_kpi(uint16_t port_id,
                        enum doca_flow_tune_server_kpi_type kpi_type,
                        struct doca_flow_tune_server_kpi_res *res);
```

14.4.2.13.4 DOCA Flow Tune Server Samples

This section describes DOCA Flow Tune Server samples.

The samples illustrate how to use the library API to retrieve KPIs or save pipeline information into a JSON file.

> All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

14.4.2.13.4.1 Running the Samples

1. Refer to the following documents:
    - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
    - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample:

    ```
    cd /opt/mellanox/doca/samples/doca_flow/flow_tune_server_dump_pipeline
    meson /tmp/build
    ninja -C /tmp/build
    ```

    > The binary `doca_flow_tune_server_dump_pipeline` is created under `/tmp/build/samples/`.

3. Sample (e.g., `doca_flow_tune_server_dump_pipeline`) usage:

    ```
    Usage: doca_<sample_name> [DOCA Flags] [Program Flags]

    DOCA Flags:
      -h, --help                              Print a help synopsis
      -v, --version                           Print program version information
    ```

```
  -l, --log-level                         Set the (numeric) log level for the program <10=DISABLE, 20=CRITI
CAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                         Set the SDK (numeric) log level for the program <10=DISABLE, 20=C
RITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                       Parse all command flags from an input json file
```

4. For additional information per sample, use the `-h` option:

```
/tmp/build/samples/<sample_name> -h
```

5. The following is a CLI example for running the samples:

```
/tmp/build/doca_<sample_name> -a auxiliary:mlx5_core.sf.2,dv_flow_en=2 -a
auxiliary:mlx5_core.sf.3,dv_flow_en=2 -- -l 60
```

14.4.2.13.4.2 Samples

Flow Tune Server KPI

This sample illustrates how to use DOCA Flow Tune Server API to retrieve KPIs.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting a single DOCA Flow port.
3. Creating a server configuration struct using the `doca_flow_tune_server_cfg_create` function.
4. Initializing DOCA Flow server using the `doca_flow_tune_server_init` function. This must be done after calling the `doca_flow_port_start` function (or the `init_doca_flow_ports` helper function).
5. Querying existing port IDs using the `doca_flow_tune_server_get_port_ids` function.
6. Querying application level KPIs using `doca_flow_tune_server_get_kpi` function. The following KPI are read:
    - Number of queues
    - Queue depth
7. KPIs per port on which the basic pipe is created:
    a. Add operation entries.
8. Adding 20 entries followed by a second call to query entries add operations.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_tune_server_kpi/flow_tune_server_kpi_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_tune_server_kpi/flow_tune_server_kpi_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_tune_server_kpi/meson.build`

Flow Tune Server Dump Pipeline

This sample illustrates how to use DOCA Flow Tune Server API to dump pipeline information into a JSON file.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.
3. Creating server configuration struct using the `doca_flow_tune_server_cfg_create` function.
4. Initializing DOCA Flow server using `doca_flow_tune_server_init` function.

   > This must be done after calling `init_foca_flow_ports` function.

5. Opening a file called `sample_pipeline.json` for writing.
6. For each port:
   a. Creating a pipe to drop all traffic.
   b. Creating a pipe to hairpin traffic from port 0 to port 1
   c. Creating FWD pipe to forward traffic based on 5-tuple.
   d. Adding two entries to FWD pipe, each entry with different 5-tuple.
   e. Creating a control pipe and adding the FWD pipe as an entry.
7. Dumping the pipeline information into a file.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_tune_server_dump_pipeline/flow_tune_server_dump_pipeline_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_tune_server_dump_pipeline/flow_tune_server_dump_pipeline_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_tune_server_dump_pipeline/meson.build`

14.4.2.13.5 Flow Visualization

Once a DOCA Flow application pipeline has been exported to a JSON file, it is easy to visualize it using tools such as Mermaid.

1. Save the following Python script locally to a file named `doca-flow-viz.py` (or similar). This script converts a given JSON file produced by DOCA Flow TS to a Mermaid diagram embedded in a markdown document.

```
#!/usr/bin/python3

#
# Copyright (c) 2024 NVIDIA CORPORATION & AFFILIATES, ALL RIGHTS RESERVED.
#
# This software product is a proprietary product of NVIDIA CORPORATION &
# AFFILIATES (the "Company") and all right, title, and interest in and to the
# software product, including all associated intellectual property rights, are
# and shall remain exclusively with the Company.
#
# This software product is governed by the End User License Agreement
# provided with the software product.
#

import glob
import json
import sys
import os.path

class MermaidConfig:
    def __init__(self):
        self.prefix_pipe_name_with_port_id = False
        self.show_match_criteria = False
        self.show_actions = False

class MermaidFormatter:
    def __init__(self, cfg):
        self.cfg = cfg
```

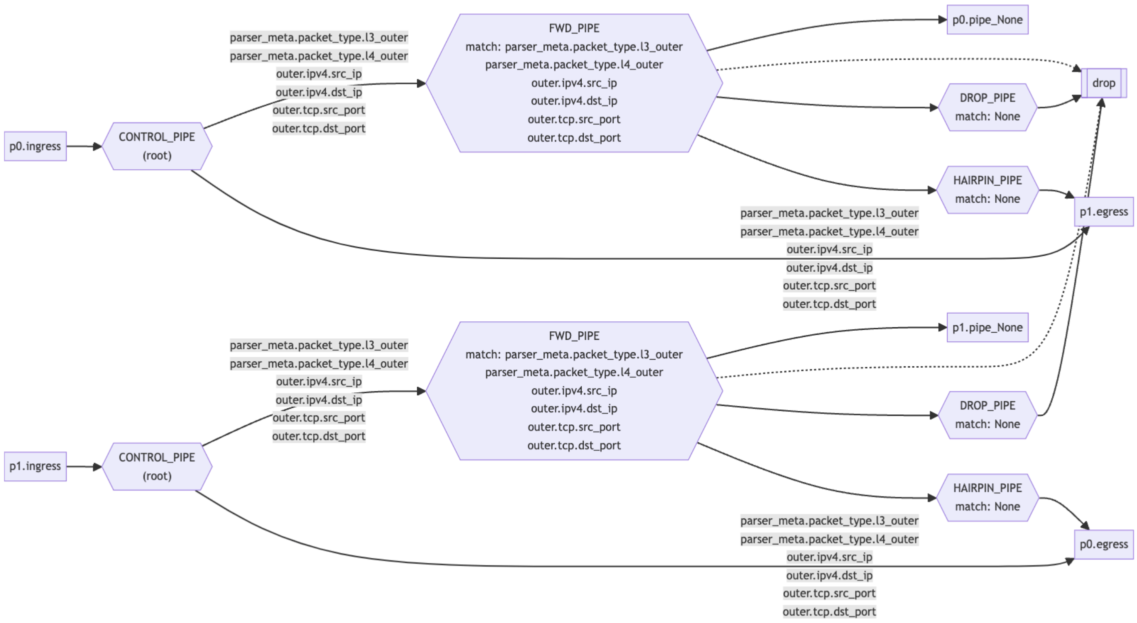
````
        self.syntax = ''
        self.prefix_pipe_name_with_port_id = cfg.prefix_pipe_name_with_port_id

    def format(self, data):
        self.prefix_pipe_name_with_port_id = self.cfg.prefix_pipe_name_with_port_id and len(data.get('ports
', []])) > 0

        if not 'ports' in data:
            port_id = data.get('port_id', 0)
            data = {
                'ports': [
                    {
                        'port_id': port_id,
                        'pipes': data['pipes']
                    }
                ]
            }

        self.syntax = ''
        self.append('```mermaid')
        self.append('graph LR')

        self.declare_terminal_states(data)

        for port in data['ports']:
            self.process_port(port)

        self.append('```')
        return self.syntax

    def append(self, text, endline = "\n"):
        self.syntax += text + endline

    def declare_terminal_states(self, data):
        all_fwd_types = self.get_all_fwd_types(data)
        if 'drop' in all_fwd_types:
            self.append('    drop[[drop]]')
        if 'rss' in all_fwd_types:
            self.append('    RSS[[RSS]]')

    def get_all_fwd_types(self, data):
        # Gather all 'fwd' and 'fwd_miss' types from pipes and 'fwd' types from entries
        all_fwd_types = {
            fwd_type
            for port in data.get('ports', [])
            for pipe in port.get('pipes', [])
            for tag in ['fwd', 'fwd_miss'] # Process both 'fwd' and 'fwd_miss' for each pipe
            for fwd_type in [pipe.get(tag, {}).get('type', None)]  # Extract the 'type'
            if fwd_type
        } | {
            fwd_type
            for port in data.get('ports', [])
            for pipe in port.get('pipes', [])
            for tag in ['fwd']
            for entry in pipe.get('entries', []) # Process all entries in each pipe
            for fwd_type in [entry.get(tag, {}).get('type', None)]
            if fwd_type
        }
        return all_fwd_types

    def process_port(self, port):
        port_id = port['port_id']
        pipe_names = self.resolve_pipe_names(port)
        self.declare_pipes(port, pipe_names)
        for pipe in port.get('pipes', []):
            self.process_pipe(pipe, port_id)

    def resolve_pipe_names(self, port):
        pipe_names = {}

        port_id = port['port_id']
        for pipe in port.get('pipes', []):
            id = pipe['pipe_id']
            name = pipe['attributes'].get('name', f"pipe_{id}")
            if self.prefix_pipe_name_with_port_id:
                name = f"p{port_id}.{name}"
            pipe_names[id] = name
        return pipe_names

    def declare_pipes(self, port, pipe_names):
        port_id = port['port_id']
        for pipe in port.get('pipes', []):
            id = pipe['pipe_id']
            name = pipe_names[id]
            self.declare_pipe(port_id, pipe, name)

    def declare_pipe(self, port_id, pipe, pipe_name):
        id = pipe['pipe_id']
        attr = "\n(root)" if self.pipe_is_root(pipe) else ""
        if self.cfg.show_match_criteria and not self.pipe_is_ctrl(pipe):
            fields_matched = self.pipe_match_criteria(pipe, 'match')
            attr += f"\nmatch: {fields_matched}"
        self.append(f'    p{port_id}.pipe_{id}{{{{"{pipe_name}{attr}"}}}}')

    def pipe_match_criteria(self, pipe, key: ['match', 'match_mask']):
        return "\n".join(self.extract_match_criteria_paths(None, pipe.get(key, {}))) or 'None'

    def extract_match_criteria_paths(self, prefix, match):
        for k,v in match.items():
            if isinstance(v, dict):
                new_prefix = f"{prefix}.{k}" if prefix else k
                for x in self.extract_match_criteria_paths(new_prefix, v):
                    yield x
````

```
            else:
                # ignore v, the match value
                yield f"{prefix}.{k}" if prefix else k

    def pipe_is_ctrl(self, pipe):
        return pipe['attributes']['type'] == 'control'

    def pipe_is_root(self, pipe):
        return pipe['attributes'].get('is_root', False)

    def process_pipe(self, pipe, port_id):
        pipe_id = f"pipe_{pipe['pipe_id']}"
        is_ctrl = self.pipe_is_ctrl(pipe)
        self.declare_fwd(port_id, pipe_id, '-->', self.get_fwd_target(pipe.get('fwd', {}), port_id))
        self.declare_fwd(port_id, pipe_id, '-.->', self.get_fwd_target(pipe.get('fwd_miss', {}), port_id))

        for entry in pipe.get('entries', []):
            fields_matched = self.pipe_match_criteria(entry, 'match') if is_ctrl else None
            fields_matched = f'|"{fields_matched}"|' if fields_matched else ''
            self.declare_fwd(port_id, pipe_id, f'-->{fields_matched}', self.get_fwd_target(entry.get('fwd',
{}), port_id))

        if self.pipe_is_root(pipe):
            self.declare_fwd(port_id, None, '-->', f"p{port_id}.{pipe_id}")

    def get_fwd_target(self, fwd, port_id):
        fwd_type = fwd.get('type', None)
        if not fwd_type:
            return None
        if fwd_type == 'changeable':
            return None
        elif fwd_type == 'pipe':
            pipe_id = fwd.get('pipe_id', fwd.get('value', None))
            target = f"p{port_id}.pipe_{pipe_id}"
        elif fwd_type == 'port':
            port_id = fwd.get('port_id', fwd.get('value', None))
            target = f"p{port_id}.egress"
        else:
            target = f"{fwd_type}"
        return target

    def declare_fwd(self, port_id, pipe_id, arrow, target):
        if target:
            src = f"p{port_id}.{pipe_id}" if pipe_id else f"p{port_id}.ingress"
            self.append(f"    {src} {arrow} {target}")

def json_to_md(infile, outfile, cfg):
    formatter = MermaidFormatter(cfg)
    data = json.load(infile)
    mermaid_syntax = formatter.format(data)
    outfile.write(mermaid_syntax)

def json_dir_to_md_inplace(dir, cfg):
    for infile in glob.glob(dir + '/**/*.json', recursive=True):
        outfile = os.path.splitext(infile)[0] + '.md'
        print(f"{infile} --> {outfile}")
        json_to_md(open(infile, 'r'), open(outfile, 'w'), cfg)

def main() -> int:
    cfg = MermaidConfig()
    cfg.show_match_criteria = True

    if len(sys.argv) == 2 and os.path.isdir(sys.argv[1]):
        json_dir_to_md_inplace(sys.argv[1], cfg)

    else:
        infile = open(sys.argv[1], 'r') if len(sys.argv) > 1 else sys.stdin
        outfile = open(sys.argv[2], 'w') if len(sys.argv) > 2 else sys.stdout
        json_to_md(infile, outfile, cfg)

if __name__ == '__main__':
    sys.exit(main())
```

2. The resulting Markdown can be viewed in several ways, including:
    - Microsoft Visual Studio Code (using an available Mermaid plugin, such as this one)
    - In the GitHub and GitLab built-in Markdown renderer (after committing the output to a Git repo)
    - By pasting only the Flowchart content into the Online FlowChart and Diagram Editor
3. The Python script can be invoked as follows:

```
python3 doca-flow-viz.py sample_pipeline.json sample_pipeline.md
```

In the case of the `flow_tune_server_dump_pipeline` sample, the script produces the following diagram:

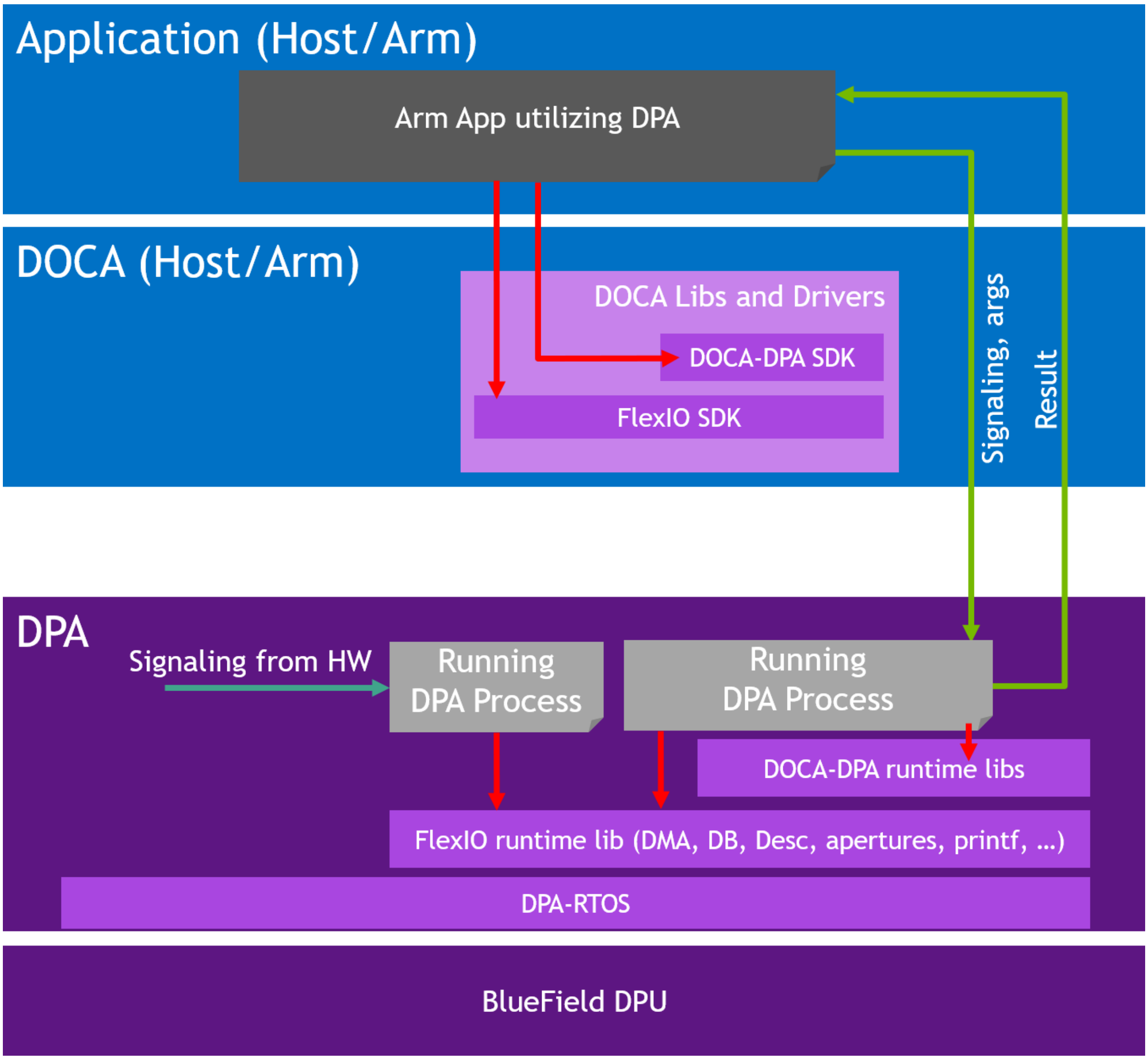


14.4.3 DPA Subsystem

The NVIDIA® BlueField®-3 data-path accelerator (DPA) is an embedded subsystem designed to accelerate workloads that require high-performance access to the NIC engines in certain packet and I/O processing workloads. Applications leveraging DPA capabilities run faster on the DPA than on host. Unlike other programmable embedded technologies, such as FPGAs, the DPA enables a high degree of programmability using the C programming model, multi-process support, tools chains like compilers and debuggers, SDKs, dynamic application loading, and management.

The DPA architecture is optimized for executing packet and I/O processing workloads. As such, the DPA subsystem is characterized by having many execution units that can work in parallel to overcome latency issues (such as access to host memory) and provide an overall higher throughput.

The following diagram illustrates the DPA subsystem. The application accesses the DPA through the DOCA library (DOCA DPA) or the DOCA driver layer (FlexIO SDK). On the host or DPU side, the application loads its code into the DPA (shown as "Running DPA Process") as well as allocates memory, NIC queues, and more resources for the DPA process to access. The DPA process can use device side libraries to access the resources. The provided APIs support signaling of the DPA process from the host or DPU to explicitly pass control or to obtain results from the DPA.



The threads on the DPA can react independently to incoming messages via interrupts from the hardware, thereby providing full bypass of DPU or Arm CPU for datapath operations.

The following sections provide an overview of the DPA platform design.

14.4.3.1 Multiple Processes on Multiple Execution Units

The DPA platform supports multiple processes with each process having multiple threads. Each thread can be mapped to a different execution unit to achieve parallel execution. The processes operate within their own address spaces and their execution contexts are isolated. Processes are loaded and unloaded dynamically per the user's request. This is achieved by the platform's hardware design (i.e., privilege layers, memory translation units, DMA engines) and a light-weight real-time operating system (RTOS). The RTOS enforces the privileges and isolation among the different processes.

14.4.3.2 DPA RTOS

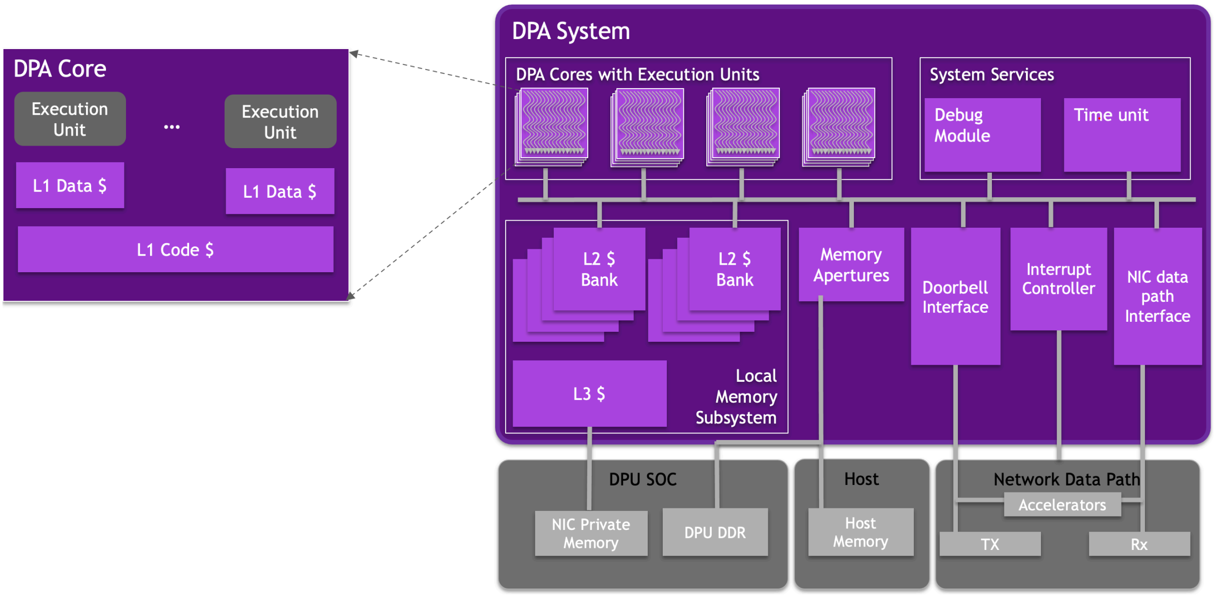
The RTOS is designed to rely on hardware-based scheduling to enable low activation latency for the execution handlers. The RTOS works in a cooperative run-to-completion scheduling model.

Under cooperative scheduling, an execution handler can use the execution unit without interrupts until it relinquishes it. Once relinquished, the execution unit is handed back to the RTOS to schedule the next handler. The RTOS sets a watchdog for the handlers to prevent any handler from unduly monopolizing the execution units.

14.4.3.3 DPA Memory and Caches

The following diagram illustrates the DPA memory hierarchy. Memory accessed by the DPA can be cached at three levels (L1, L2, and L3). Each execution unit has a private L1 data cache. The L1 code cache is shared among all the execution units in a DPA core. The L2 cache is shared among all the DPA cores. The DPA execution units can access external memory via load/store operations through the Memory Apertures.

The external memory that is fetched can be cached directly in L1. The DPA caches are backed by NIC private memory, which is located in the DPU's DDR memory banks. Therefore, the address spaces are scalable and bound only by the size of the NIC's private memory, which in turn is limited only by the DPU's DDR capacity.



See "Memory Model" for more details.

14.4.3.4 DPA Access to NIC Accelerators

The DPA can send and receive any kind of packet toward the NIC and utilize all the accelerators that reside on the BlueField DPU (e.g., encryption/decryption, hash computation, compression/decompression).

The DPA platform has efficient DMA accelerators that enable the different execution units to access any memory location accessible by the NIC in parallel and without contention. This includes both synchronous and asynchronous DMA operations triggered by the execution units. In addition, the NIC can DMA data to the DPA caches to enable low-latency access and fast processing. For example, a packet received from the wire may be "DMA-gathered" directly to the DPA's last level caches.

14.4.3.5 DPA Development

14.4.3.5.1 Overview

14.4.3.5.1.1 DOCA Libs and Drivers

The NVIDIA DOCA framework is the key for unlocking the potential of NVIDIA® BlueField®-3 platforms.

DOCA's software environment allows developers to program the DPA to accelerate workloads. Specifically, DOCA includes:

- DOCA DPA SDK – a high-level SDK for application-level protocol acceleration
- DOCA FlexIO SDK – a low-level SDK to load DPA programs into the DPA, manage the DPA memory, create the execution handlers and the needed hardware rings and contexts
- DPACC – DPA toolchain for compiling and ELF file manipulation of the DPA code

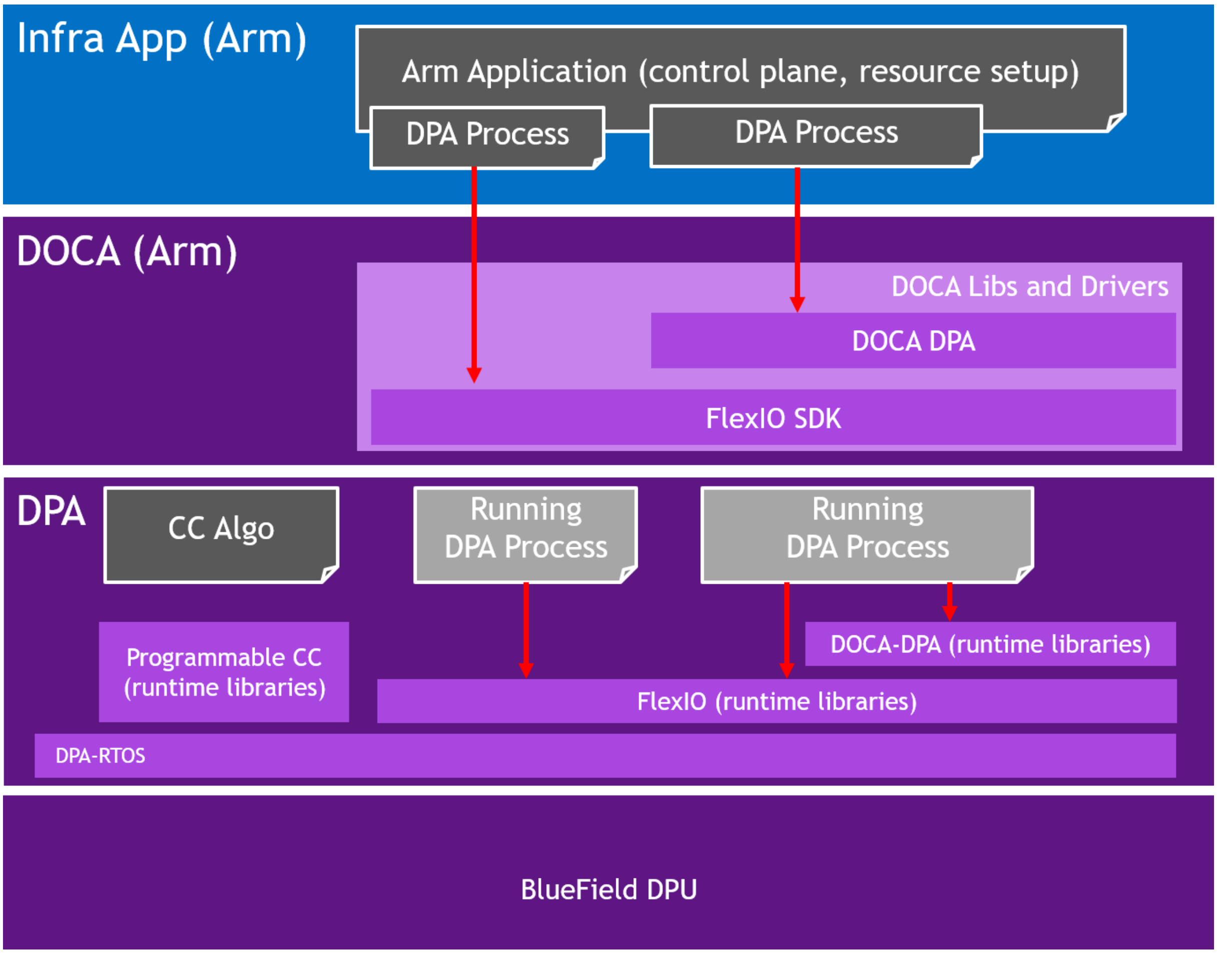
14.4.3.5.1.2 Programming Model

The DPA is intended to accelerate datapath operations for the DPU and host CPU. The accelerated portion of the application using DPA is presented as a library for the host application. The code within the library is invoked in an event-driven manner in the context of a process that is running on the DPA. One or many DPA execution units may work to handle the work associated with network events. The programmer specifies different conditions when each function should be called using the appropriate SDK APIs on the host or DPU.

The DPA cannot be used as a standalone CPU.

Management of the DPA, such as loading processes and allocating memory, is performed from a host or DPU process. The host process discovers the DPA capabilities on the device and drives the control plane to set up the different DPA objects. The DPA objects exist as long as the host process exists. When the host process is destroyed, the DPA objects are freed. The host process decides which functions it wants to accelerate using the DPA: Either its entire data plane or only a part of it.

The following diagram illustrates the different processes that exist in the system:



Compiler

DPACC is a compiler for the DPA processor. It compiles code targeted for the DPA processor into an executable and generates a DPA program. A DPA program is a host library with interfaces encapsulating the DPA executable.

This DPA program is linked with the host application to generate a host executable. The host executable can invoke the DPA code through the DPA SDK's runtime.

Compiler Keywords

DPACC implements the following keywords:

Keyword	Application Usage	Comment
`__dpa_global__`	Annotate all event handlers that execute on the DPA and all common user-defined datatypes (including user-defined structures) which are passed from the host to the DPA as arguments.	Used by the compiler to generate entry points in the DPA executable and automatically replicate user-defined datatypes between the host and DPA.
`__dpa_rpc__`	Annotate all RPC calls which are invoked by the host and execute on the DPA. RPC calls return a value of `uint64_t`.	Used by the compiler to generate RPC specific entry points.

Please refer to NVIDIA DOCA DPACC Compiler for more details.

FlexIO

Supported at beta level.

FlexIO is a low-level event-driven library to program and accelerate functions on the DPA.

FlexIO Execution Model

To load an application onto the DPA, the user must create a process on the DPA, called a FlexIO process. FlexIO processes are isolated from each other like standard host OS processes.

FlexIO supports the following options for executing a user-defined function on the DPA:

1. FlexIO event hander – the event handler executes its function each time an event occurs. An event on this context is a completion event (CQE) received on the NIC completion queue (CQ) when the CQ was in the armed state. The event triggers an internal DPA interrupt that activates the event handler. When the event handler is activated, it is provided with a user-defined argument. The argument in most cases is a pointer to the software execution context of the event handler.
   The following pseudo-code example describes how to create an event handler and attach it to a CQ:

```
// Device code
__dpa_global__ void myFunc(flexio_uintptr_t myArg){
      struct my_db *db = (struct my_db *)myArg;
      get_completion(db->myCq)
      work();
      arm_cq(myCq);
      // reschedule the thread
      flexio_dev_thread_reschedule();
}

// Host code
main() {

      /* Load the application code into the DPA */
      flexio_process_create(device, application, &myProcess);

      /* Create event handler to run my_func with my_arg */
      flexio_event_handler_create(myProcess, myFunc, myArg, &myEventHandler);

      /* Associate the event hanlder with a specific CQ */
      create_cq(&myCQ,… , myEventHandler)

      /* Start the event handler */
      flexio_event_handler_run(myEventHandler)
      …
}
```

2. RPC – remote, synchronous, one-time call of a specific function. RPC is mainly used for the control path to update DPA memory contexts of a process. The RPC's return value is reported back to the host application.
   The following pseudo-code example describes how to use the RPC:

```
// Device code
__dpa_rpc__ uint64_t myFunc(myArg) {
      struct my_db *db = (struct my_db *)myArg;
      if (db->flag) return 1;
      db->flag = 1;
      return 0;
}

// Host code
main() {
      …

      /* Load the application code into the DPA */
      flexio_process_create(device, application, &myProcess);

      /* run the function */
      flexio_process_call(myProcess, myFunc, myArg, &returnValue);
      …
}
```

FlexIO Memory Management

The DPA process can access several memory locations:

- Global variables defined in the DPA process.
- Stack memory – local to the DPA execution unit. Stack memory is not guaranteed to be preserved between different execution of the same handler.
- Heap memory – this is the process' main memory. The heap memory contents are preserved as long as the DPA process is active.
- External registered memory – remote to the DPA but local to the server. The DPA can access any memory location that can be registered to the local NIC using the provided API. This includes BlueField DRAM, external host DRAM, GPU memory, and more.

The heap and external registered memory locations are managed from the host process. The DPA execution units can load/store from stack/heap and external memory locations. Note that for external memory locations, the window should be configured appropriately using FlexIO Window APIs.

FlexIO allows the user to allocate and populate heap memory on the DPA. The memory can later be used by in the DPA application as an argument to the execution context (RPC and event handler):

```
/* Load the application code into the DPA */
flexio_process_create(device, application, &myProcess);

/* allocate some memory */
flexio_buf_dev_alloc(process, size, ptr)

/* populate it with user defined data */
flexio_host2dev_memcpy(process, src, size, ptr)

/* run the function */
flexio_process_call(myProcess, function, ptr, &return value);
```

FlexIO allows accessing external registered memory from the DPA execution units using FlexIO Window. FlexIO Window maps a memory region from the DPA process address space to an external registered memory. A memory key for the external memory region is required to be associated with the window. The memory key is used for address translation and protection. FlexIO window is created by the host process and is configured and used by the DPA handler during execution. Once configured, LD/ST from the DPA execution units access the external memory directly.

The access for external memory is not coherent. As such, an explicit memory fencing is required to flush the cached data to maintain consistency. See section "Memory Fences" for more.

The following example code demonstrates the window management:

```
// Device code
__dpa_rpc__ uint64_t myFunc(arg1, arg2, arg3)
{
    struct flexio_dev_thread_ctx *dtctx;
    flexio_dev_get_thread_ctx(&dtctx);
    uint32_t windowId = arg1;
    uint32_t mkey = arg2;
    uint64_t *dev_ptr;
    flexio_dev_window_config(dtctx, windowId, mkey );
    /* get ptr to the external memory (arg3) from the DPA process address space */
    flexio_dev_status status = flexio_dev_window_ptr_acquire (dtctx, arg3, dev_ptr);
    /* will set the external memory */
    *dev_ptr = 0xff;
    /* flush the data out */
    __dpa_thread_window_writeback();
    return 0;
}

// Host code
main() {
      /* Load the application code into the DPA */
      flexio_process_create(device, application, &myProcess);
      /* define an array on host */
      uint64_t var= {0};
      /* register host buffer  */
      mkey =ibv_reg_mr(&var, …)
      /* create the window */
      flexio_window_create(process, doca_device->pd, mkey, &window_ctx);
      /* run the function */
      flexio_process_call(myProcess, myFunc, flexio_window_get_id(window_ctx), mkey, &var, &returnValue);
}
```

Send and Receive Operation

A DPA process can initiate send and receive operations using the FlexIO outbox object. The FlexIO outbox contains memory-mapped IO registers that enable the DPA application to issue device doorbells to manage the send and receive planes. The DPA outbox can be configured during run time to perform send and receive from a specific NIC function exposed by the DPU. This capability is not available for Host CPUs that can only access their assigned NIC function.

Each DPA execution engine has its own outbox. As such, each handler can efficiently use the outbox without needing to lock to protect against accesses from other handlers. To enforce the required security and isolation, the DPA outbox enables the DPA application to send and receive only for queues created by the DPA host process and only for NIC functions the process is allowed to access.

Like the FlexIO window, the FlexIO outbox is created by the host process and configured and used at run time by the DPA process.

```
// Device code
__dpa_rpc__ uint64_t myFunc(arg1,arg2,arg3) {

    struct flexio_dev_thread_ctx *dtctx;

    flexio_dev_get_thread_ctx(&dtctx);

    uint32_t outbox = arg1;
    flexio_dev_outbox_config (dtctx, outbox);

    /* Create some wqe and post it on sq */

    /* Send DB on sq*/

    flexio_dev_qp_sq_ring_db(dtctx, sq_pi,arg3);

    /* Poll CQ (cq number is in arg2) */
    return 0;
}

// Host code
main() {

      /* Load the application code into the DPA */
      flexio_process_create(device, application, &myProcess);

      /* Allocate uar */
      uar = ibv_alloc_uar(ibv_ctx);

      /* Create queues*/
      flexio_cq_create(myProcess, ibv_ctx, uar, cq_attr, &myCQ);
      my_hwcq = flexio_cq_get_hw_cq (myCQ);

      flexio_sq_create(myProcess, ibv_ctx, myCQ, uar, sq_attr, &mySQ);
      my_hwsq = flexio_sq_get_hw_sq(mySQ);

      /* Outbox will allow access only for queues created with the same UAR*/
      flexio_outbox_create(process, ibv_ctx, uar, &myOutbox);

      /* Run the function */
      flexio_process_call(myProcess, myFunc, myOutbox, my_hwcq->cq_num, my_hwsq->sq_num,  &return_value);
}
```

Synchronization Primitives

The DPA execution units support atomic instructions to protect from concurrent access to the DPA process heap memory. Using those instructions, multiple synchronization primitives can be designed.

FlexIO currently supports basic spin lock primitives. More advanced thread pipelining can be achieved using DOCA DPA events.

DOCA DPA

Supported at beta level.

The DOCA DPA SDK eases DPA code management by providing high-level primitives for DPA work offloading, synchronization, and communication. This leads to simpler code but lacks the low-level control that FlexIO SDK provides.

User-level applications and libraries wishing to utilize the DPA to offload their code may choose DOCA DPA. Use-cases closer to the driver level and requiring access to low-level NIC features would be better served using FlexIO.

The implementation of DOCA DPA is based on the FlexIO API. The higher level of abstraction enables the user to focus on their program logic and not the low-level mechanics.

ⓘ Refer to DOCA DPA documentation for more details.

Memory Model

The DPA offers a coherent but weakly ordered memory model. The application is required to use fences to impose the desired memory ordering. Additionally, where applicable, the application is required to write back data for the data to be visible to NIC engines (see the coherency table).

The memory model offers "same address ordering" within a thread. This means that, if a thread writes to a memory location and subsequently reads that memory location, the read returns the contents that have previously been written.

The memory model offers 8-byte atomicity for aligned accesses to atomic datatypes. This means that all eight bytes of read and write are performed in one indivisible transaction.

The DPA does not support unaligned accesses, such as accessing `N` bytes of data from an address not evenly divisible by `N`.

The DPA processes memory can be divided into the following memory spaces:

Memory Space	Definition
Heap	Memory locations within the DPA process heap. Referenced as `__DPA_HEAP` in the code.
Memory	Memory locations belonging to the DPA process (including stack, heap, BSS and data segment) except the memory-mapped IO. Referenced as `__DPA_MEMORY` in the code.
MMIO (memory-mapped I/O)	External memory outside the DPA process accessed via memory-mapped IO. Window and Outbox accesses are considered MMIO. Referenced as `__DPA_MMIO` in the code.
System	All memory locations accessible to the thread within Memory and MMIO spaces as described above. Referenced as `__DPA_SYSTEM` in the code.

The coherency between the DPA threads and NIC engines is described in the following table:

Producer	Observer	Coherency	Comments
DPA thread	NIC engine	Not coherent	Data to be read by the NIC must be written back using the appropriate intrinsic (see section "Memory Fence and Cache Control Usage Examples").

Producer	Observer	Coherency	Comments
NIC engine	DPA Thread	Coherent	Data written by the NIC is eventually visible to the DPA threads. The order in which the writes are visible to the DPA threads is influenced by the ordering configuration of the memory region (see `IBV_ACCESS_RELAXED_ORDERING`). In a typical example of the NIC writing data and generating a completion entry (CQE), it is guaranteed that when the write to the CQE is visible, the DPA thread can read the data without additional fences.
DPA thread	DPA thread	Coherent	Data written by a DPA thread is eventually visible to the other DPA threads without additional fences. The order in which writes made by a thread are visible to other threads is undefined when fences are not used. Programmers can enforce ordering of updates using fences (see section "Memory Fences").

Memory Fences

Fence APIs are intended to impose memory access ordering. The fence operations are defined on the different memory spaces. See information on memory spaces under section "Memory Model".

The fence APIs apply ordering between the operations issued by the calling thread. As a performance note, the fence APIs also have a side effect of writing back data to the memory space used in the fence operation. However, programmers should not rely on this side effect. See section "Cache Control" for explicit cache control operations. The fence APIs have an effect of a compiler-barrier which means that memory accesses are not reordered around the fence API invocation by the compiler.

A fence applies between the "predecessor" and the "successor" operations. The predecessor and successor ops can be refenced using `__DPA_R`, `__DPA_W`, and `__DPA_RW` in the code.

The generic memory fence operation can operate on any memory space and any set of predecessor and successor operations. The other fence operations are provided as convenient shortcuts that are specific to the use case. It is preferable for programmers to use the shortcuts when possible.

Fence operations can be included using the `dpaintrin.h` header file.

Generic Fence

```
void __dpa_thread_fence(memory_space, pred_op, succ_op);
```

This fence can apply to any DPA thread memory space. Memory spaces are defined under section "Memory Model". The fence ensures that all operations (`pred_op`) performed by the calling thread, before the call to `__dpa_thread_fence()`, are performed and made visible to all threads in the DPA, host, NIC engines, and peer devices as occurring before all operations (`succ_op`) to the memory space after the call to `__dpa_thread_fence()`.

System Fence

```
void __dpa_thread_system_fence();
```

This is equivalent to calling `__dpa_thread_fence(__DPA_SYSTEM, __DPA_RW, __DPA_RW)`.

Outbox Fence

```
void __dpa_thread_outbox_fence(pred_op, succ_op);
```

This is equivalent to calling `__dpa_thread_fence(__DPA_MMIO, pred_op, succ_op)`.

Window Fence

```
void __dpa_thread_window_fence(pred_op, succ_op);
```

This is equivalent to calling `__dpa_thread_fence(__DPA_MMIO, pred_op, succ_op)`.

Memory Fence

```
void __dpa_thread_memory_fence(pred_op, succ_op);
```

This is equivalent to calling `__dpa_thread_fence(__DPA_MEMORY, pred_op, succ_op)`.

Cache Control

Cache control operations allow the programmer to exercise fine-grained control over data resident in the DPA's caches. They have an effect of a compiler-barrier. The operations can be included using the `dpaintrin.h` header file.

Window Read Contents Invalidation

```
void __dpa_thread_window_read_inv();
```

The DPA can cache data that was fetched from external memory using a window. Subsequent memory accesses to the window memory location may return the data that is already cached. In some cases, it is required by the programmer to force a read of external memory (see example under "Polling Externally Set Flag"). In such a situation, the window read contents cached must be dropped.

This function ensures that contents in the window memory space of the thread before the call to `__dpa_thread_window_read_inv()` are invalidated before read operations made by the calling thread after the call to `__dpa_thread_window_read_inv()`.

Window Writeback

```
void __dpa_thread_window_writeback();
```

Writes to external memory must be explicitly written back to be visible to external entities.

This function ensures that contents in the window space of the thread before the call to `__dpa_thread_window_writeback()` are performed and made visible to all threads in the DPA, host, NIC engines, and peer devices as occurring before any write operation after the call to `__dpa_thread_window_writeback()`.

Memory Writeback

```
void __dpa_thread_memory_writeback();
```

Writes to DPA memory space may need to be written back. For example, the data must be written back before the NIC engines can read it. Refer to the coherency table for more.

This function ensures that the contents in the memory space of the thread before the call to `__dpa_thread_writeback_memory()` are performed and made visible to all threads in the DPA, host, NIC engines, and peer devices as occurring before any write operation after the call to `__dpa_thread_writeback_memory()`.

Memory Fence and Cache Control Usage Examples

These examples illustrate situations in which programmers must use fences and cache control operations.

In most situations, such direct usage of fences is not required by the application using FlexIO or DOCA DPA SDKs as fences are used within the APIs.

Issuing Send Operation

In this example, a thread on the DPA prepares a work queue element (WQE) that is read by the NIC to perform the desired operation.

The ordering requirement is to ensure the WQE data contents are visible to the NIC engines read it. The NIC only reads the WQE after the doorbell (MMIO operation) is performed. Refer to coherency table.

#	User Code - WQE Present in DPA Memory	Comment
1	Write WQE	Write to memory locations in the DPA (memory space = `__DPA_MEMORY`)
2	`__dpa_thread_memory_writeback();`	Cache control operation
3	Write doorbell	MMIO operation via Outbox

In some cases, the WQE may be present in external memory. See the description of `flexio_qmem` below. The table of operations in such a case is below.

#	User Code - WQE Present in External Memory	Comment
1	Write WQE	Write to memory locations in the DPA (memory space = `__DPA_MMIO`)
2	`__dpa_thread_window_writeback();`	Cache control operation
3	Write doorbell	MMIO operation via Outbox

Posting Receive Operation

In this example, a thread on the DPA is writing a WQE for a receive queue and advancing the queue's producer index. The DPA thread will have to order its writes and writeback the doorbell record contents so that the NIC engine can read the contents.

#	User Code - WQE Present in DPA Memory	Comment
1	Write WQE	Write to memory locations in the DPA (memory space = `__DPA_MEMORY`)
2	`__dpa_thread_memory_fence(__DPA_W, __DPA_W);`	Order the write to the doorbell record with respect to WQE
3	Write doorbell record	Write to memory locations in the DPA (memory space = `__DPA_MEMORY`)
4	`__dpa_thread_memory_writeback();`	Ensure that contents of doorbell record are visible to the NIC engine

Polling Externally Set Flag

In this example, a thread on the DPA is polling on a flag that will be updated by the host or other peer device. The memory is accessed by the DPA thread via a window. The DPA thread must invalidate the contents so that the underlying hardware performs a read.

User Code - Flag Present in External Memory	Comment

`while (!flag) {`
`  __dpa_thread_window_read_inv();`
`}`
	`flag` is a memory location read using a window

Thread-to-thread Communication

In this example, a thread on the DPA is writing a data value and communicating that the data is written to another thread via a flag write. The data and flag are both in DPA memory.

User Code - Thread 1	User Code - Thread 2	Comment
		Initial condition, `flag = 0`
`var1 = x;`	`while(*((volatile int *)&flag) !=1);`	- Thread 1 - write to var1 - Thread 2 - `flag` is accessed as a volatile variable, so the compiler preserves the intended program order of reads
`__dpa_thread_memory_fence(__DPA_W, __DPA_W);`		Thread 1 - write to flag cannot bypass write to `var1`
	`var_t2 = var1;`	
`flag = 1;`	`assert(var_t2 == x);`	`var_t2` must be equal to `x`

Setting Flag to be Read Externally

In this example, a thread on the DPA sets a flag that is observed by a peer device. The flag is written using a window.

User Code - Flag Present in External Memory	Comment
`flag = data;`	`flag` is updated in local DPA memory
`__dpa_thread_window_writeback();`	Contents from DPA memory for the window are written to external memory

Polling Completion Queue

In this example, a thread on the DPA reads a NIC completion queue and updates its consumer index.

First, the DPA thread polls the memory location for the next expected CQE. When the CQE is visible, the DPA thread processes it. After processing is complete, the DPA thread updates the CQ's consumer index. The consumer index is read by the NIC to determine whether a completion queue entry has been read by the DPA thread. The consumer index is used by the NIC to monitor a potential completion queue overflow situation.

User Code - CQE in DPA Memory	Comment
`while(*((volatile uint8_t *)&cq->op_own) & 0x1 == hw_owner);`	Poll CQ owner bit in DPA memory until the value indicates the CQE is in software ownership. Coherency model ensures update to the CQ is visible to the DPA execution unit without additional fences or cache control operations. Coherency model ensures that data in the CQE or referenced by it are visible when the CQE changes ownership to software.
`process_cqe();`	User processes the CQE according to the application's logic.
`cq->cq_index++; // next CQ index. Handle wraparound if necessary`	Calculate the next CQ index taking into account any wraparound of the CQ depth.
`update_cq_dbr(cq, cq_index); // writes cq_index to DPA memory`	Memory operation to write the new consumer index.
`__dpa_thread_memory_writeback();`	Ensures that write to CQ's consumer index is visible to the NIC. Depending on the application's logic, the `__dpa_thread_memory_writeback()` may be coalesced or eliminated if the CQ is configured in overrun ignore mode.
`arm_cq();`	Arm the CQ to generate an event if this handler is going to call `flexio_dev_thread_reschedule()`. Arming the CQ is not required if the handler calls `flexio_dev_thread_finish()`.

DPA-specific Operations

The DPA supports some platform-specific operations. These can be accessed using the functions described in the following subsections. The operations can be included using the `dpaintrin.h` header file.

Clock Cycles

```
uint64_t __dpa_thread_cycles();
```

Returns a counter containing the number of cycles from an arbitrary start point in the past on the execution unit the thread is currently scheduled on.

Note that the value returned by this function in the thread is meaningful only for the duration of when the thread remains associated with this execution unit.

This function also acts as a compiler barrier, preventing the compiler from moving instructions around the location where it is used.

Timer Ticks

```
uint64_t __dpa_thread_time();
```

Returns the number of timer ticks from an arbitrary start point in the past on the execution unit the thread is currently scheduled on.

Note that the value returned by this function in the thread is meaningful only for the duration of when the thread remains associated with this execution unit.

This intrinsic also acts as a compiler barrier, preventing the compiler from moving instructions around the location where the intrinsic is used.

Instructions Retired

```
uint64_t __dpa_thread_inst_ret();
```

Returns a counter containing the number of instructions retired from an arbitrary start point in the past by the execution unit the thread is currently scheduled on.

Note that the value returned by this function in the software thread is meaningful only for the duration of when the thread remains associated with this execution unit.

This intrinsic also acts as a compiler barrier, preventing the compiler from moving instructions around the location where the intrinsic is used.

Fixed Point Log2

```
int __dpa_fxp_log2(unsigned int);
```

This function evaluates the fixed point Q16.16 base 2 logarithm. The input is an unsigned integer.

Fixed Point Reciprocal

```
int __dpa_fxp_rcp(int);
```

This function evaluates the fixed point Q16.16 reciprocal (1/x) of the value provided.

Fixed Point Pow2

```
int __dpa_fxp_pow2(int);
```

This function evaluates the fixed point Q16.16 power of 2 of the provided value.

14.4.3.5.2 FlexIO

This chapter provides an overview and configuration instructions for DOCA FlexIO SDK API.

The DPA processor is an auxiliary processor designed to accelerate packet processing and other data-path operations. The FlexIO SDK exposes an API for managing the DPA device and executing native code over it.

The DPA processor is supported on NVIDIA® BlueField®-3 DPUs and later generations.

After DOCA installation, FlexIO SDK headers may be found under `/opt/mellanox/flexio/include` and libraries may be found under `/opt/mellanox/flexio/lib/` .

14.4.3.5.2.1 Prerequisites

DOCA FlexIO applications can run either on the host machine or on the target DPU.

Developing programs over FlexIO SDK requires knowledge of DPU networking queue usage and management.

14.4.3.5.2.2 Architecture

FlexIO SDK library exposes a few layers of functionality:

- `libflexio` - library for Host-side operations. It is used for resource management.
- `libflexio_dev` - library for DPA-side operations. It is used for data path implementation.
- `libflexio_libc` - a lightweight C library for DPA device code. `libflexio_libc` may expose very partial functionality compared to a standard `libc` .

A typical application is composed of two parts: One running on the host machine or the DPU target and another running directly over the DPA.

14.4.3.5.2.3 API

Please refer to the NVIDIA DOCA Driver APIs.

14.4.3.5.2.4 Resource Management

DPA programs cannot create resources. The responsibility of creating resources, such as FlexIO process, thread, outbox and window, as well as queues for packet processing (completion, receive and send), lies on the DPU program. The relevant information should be communicated (copied) to the DPA side and the address of the copied information should be passed as an argument to the running thread.

Example

Host side:

1. Declare a variable to hold the DPA buffer address.

```
flexio_uintptr_t app_data_dpa_daddr;
```

2. Allocate a buffer on the DPA side.

```
flexio_buf_dev_alloc(flexio_process, sizeof(struct my_app_data), &app_data_dpa_daddr);
```

3. Copy application data to the DPA buffer.

```
flexio_host2dev_memcpy(flexio_process, (uintptr_t)app_data, sizeof(struct my_app_data),
app_data_dpa_daddr);
```

`struct my_app_data` should be common between the DPU and DPA applications so the DPA application can access the struct fields.
The event handler should get the address to the DPA buffer with the copied data:

```
flexio_event_handler_create(flexio_process, net_entry_point, app_data_dpa_daddr, NULL, flexio_outbox,
&app_ctx.net_event_handler)
```

DPA side:

```
__dpa_rpc__ uint64_t event_handler_init(uint64_t thread_arg)
{
      struct my_app_data *app_data;
      app_data = (my_app_data *)thread_arg;
      ...
}
```

14.4.3.5.2.5 DPA Memory Management

As mentioned previously, the DPU program is responsible for allocating buffers on the DPA side (same as resources). The DPU program should allocate device memory in advance for the DPA program needs (e.g., queues data buffer and rings, buffers for the program functionality, etc).

The DPU program is also responsible for releasing the allocated memory. For this purpose, the FlexIO SDK API exposes the following memory management functions:

```
flexio_status flexio_buf_dev_alloc(struct flexio_process *process, size_t buff_bsize, flexio_uintptr_t
*dest_daddr_p);
flexio_status flexio_buf_dev_free(flexio_uintptr_t daddr_p);
flexio_status flexio_host2dev_memcpy(struct flexio_process *process, void *src_haddr, size_t buff_bsize,
flexio_uintptr_t dest_daddr);
flexio_status flexio_buf_dev_memset(struct flexio_process *process, int value, size_t buff_bsize, flexio_uintptr_t
dest_daddr);
```

Allocating NIC Queues for Use by DPA

The FlexIO SDK exposes an API for allocating work queues and completion queues for the DPA. This means that the DPA may have direct access and control over these queues, allowing it to create doorbells and access their memory.

When creating a FlexIO SDK queue, the user must pre-allocate and provide memory buffers for the queue's work queue elements (WQEs). This buffer may be allocated on the DPU or the DPA memory.

To this end, the FlexIO SDK exposes the `flexio_qmem` struct, which allows the user to provide the buffer address and type (DPA or DPU).

Memory Allocation Best Practices

To optimize process device memory allocation, it is recommended to use the following allocation sizes (or closest to it):

- Up to 1 page (4KB)
- 2^6 pages (256KB)

- 2^{11} pages (8MB)
- 2^{16} pages (256MB)

Using these sizes minimizes memory fragmentation over the process device memory heap. If other buffer sizes are required, it is recommended to round the allocation up to one of the listed sizes and use it for multiple buffers.

14.4.3.5.2.6 DPA Window

DPA windows are used to access external memory, such as on the DPU's DDR or host's memory. DPA windows are the software mechanism to use the Memory Apertures mentioned in section "DPA Memory and Caches". To use the window functionality, DPU or host memory must be registered for the device using the `ibv_reg_mr()` call.

Both the address and size provided to this call must be 64 bytes aligned for the window to operate. This alignment may be obtained using the `posix_memalign()` allocation call.

14.4.3.5.2.7 DPA Event Handler

Default Window/Outbox

The DPA event handler expects a DPA window and DPA outbox structs upon creation. These are used as the default for the event handler thread. Users may choose to set one or both to NULL, in which case there would be no valid default value for one/both of them.

Upon thread invocation on the DPA side, the thread context is set for the provided default IDs. If, at any point, the outbox/window IDs are changed, then the thread context on the next invocation is restored to the default IDs. This means that the DPA Window MKey must be configured each time the thread is invoked, as it has no default value.

Execution Unit Management

DPA execution units (EUs) are the equivalent to logical cores. For a DPA program to execute, it must be assigned an EU.

It is possible to set EU affinity for an event handler upon creation. This causes the event handler to execute its DPA program over specific EUs (or a group of EUs).

DPA supports three types of affinity: `none` , `strict` , `group` .

The affinity type and ID, if applicable, are passed to the event handler upon creation using the `affinity` field of the `flexio_event_handler_attr` struct.

For more information, please refer to NVIDIA DOCA DPA Execution Unit Management Tool.

Execution Unit Partitions

To work over DPA, an EU partition must be created for the used device. A partition is a selection of EUs marked as available for a device. For the DPU ECPF, a default partition is created upon boot with all EUs available in it. For any other device (i.e., function), the user must create a partition. This means that running an application on a non-ECPF function without creating a partition would result in failure.

FlexIO SDK uses `strict` and `none` affinity for internal threads, which require a partition with at least one EU for the participating devices. Failing to comply with this assumption may cause failures.

Virtual Execution Units

Users should be aware that beside the default EU partition, which is exposed to the real EU numbers, all other partitions created use virtual EUs.

For example, if a user creates a partition with the range of EUs 20-40, querying the partition info from one of its virtual HCAs (vHCAs) it would display EUs from 0-20. So, the real EU number, 39 in this example, would correspond to the virtual EU number 19.

14.4.3.5.2.8 Version API and Backward Compatibility

FlexIO SDK supports partial backward compatibility. The may follow one of the following options:

1. Work only with the latest version. The user must align their entire code according to the changes in the FlexIO SDK API listed in the document accompanying each version.
2. Ensure partial backward compatibility for the working code. The user must inform the SDK which version they intend to work with. The SDK provides a set of tools that ensure backward compatibility. The set consists of compile-time and runtime tools.

Version API Toolkit

To support backward compatibility, the FlexIO SDK uses the macros `FLEXIO_VER` for the host and `FLEXIO_DEV_VER` for the DPA device. The macros have 3 parameters, where the first is the major version (year), the second is the minor version (month), and the third is the sub-minor version (not used, always 0).

Compile-time

This toolkit is available for both the host and DPA device. The header files `flexio_ver.h` and `flexio_dev_ver.h` contain the macros `FLEXIO_VER` and `FLEXIO_VER_LATEST` for the host and `FLEXIO_DEV_VER` and `FLEXIO_DEV_VER_LATEST` for the DPA device. For example, to set backward compatibility for version 24.04, the user must declare the following construct for the host:

```
#include <libflexio/flexio_ver.h>
#define FLEXIO_VER_USED FLEXIO_VER(24, 4, 0)
#include <libflexio/flexio.h>
```

And the user must declare the following construct for the DPA device:

```
#include <libflexio-dev/flexio_dev_ver.h>
#define FLEXIO_DEV_VER_USED FLEXIO_DEV_VER(24, 4, 0)
#include <libflexio-dev/flexio_dev.h>
```

Where `24` is the major version, and `4` is the minor version.

> The files `flexio.h` and `flexio_dev.h` have the macros `FLEXIO_CURRENT_VERSION` and `FLEXIO_LAST_SUPPORTED_VERSION` for the host `FLEXIO_DEV_CURRENT_VERSION` and `FLEXIO_DEV_LAST_SUPPORTED_VERSION` for the DPA device. These versions are provided for internal use and user information. The user should not use these macros.

Runtime

This toolkit is only present for the host. For backward compatibility in runtime, the user can call the function `flexio_status flexio_version_set(uint64_t version);` in `flexio.h` once before calling any other function from the API, with the version parameter they wish to work with. The function returns an error in the following cases:

- If the specified version is less than `FLEXIO_LAST_SUPPORTED_VERSION`
- If it exceeds `FLEXIO_CURRENT_VERSION`
- If the function is called again with a version value different from the previous one

```
status = flexio_version_set(FLEXIO_VER(24, 4, 0));
if (status == FLEXIO_STATUS_FAILED)
{
    return ERROR;
}
```

It is recommended to use the `FLEXIO_VER_USED` macro as a parameter:

```
flexio_version_set(FLEXIO_VER_USED);
```

End of Backward Compatibility

The backward compatibility tools are designed to have an endpoint. With each new version, it is possible to gradually raise the value of `FLEXIO_LAST_SUPPORTED_VERSION` for the host and `FLEXIO_DEV_LAST_SUPPORTED_VERSION` for the DPA device. If `FLEXIO_VER_USED` equals `FLEXIO_LAST_SUPPORTED_VERSION`, then the compiler will issue a warning. This is a sign for the user to start transitioning to a newer version. This way the user has time at least until the next version to modify their code to comply with the older version. If `FLEXIO_VER_USED` is lower than `FLEXIO_LAST_SUPPORTED_VERSION`, then the compiler will issue errors. This is a sign for the user to immediately transition to a newer version. The same behavior for the DPA device.

14.4.3.5.2.9 Application Debugging

Because application execution is divided between the host side and the DPA processor services, debugging may be somewhat challenging, especially since the DPA side does not have a terminal allowing the use of the C stdio library printf services.

Using Device Messaging Stream API

Another logging (messaging) option is to use FlexIO SDK infrastructure to send strings or formatted text in general, from the DPA side to the host side console or file. The host side's `flexio.h` file provides the `flexio_msg_stream_create` API function for initializing the required infrastructures to support this. Once initialized, the DPA side must have the thread context, which can be obtained by calling `flexio_dev_get_thread_ctx`. `flexio_dev_msg` can then be called to write a string generated on the DPA side to the stream created (using its ID) on the host side, where it is directed to the console or a file, according to user configuration in the creation stage.

It is important to call `flexio_msg_stream_destroy` when exiting the DPU application to ensure proper clean-up of the print mechanism resources.

Device messages use an internal QP for communication between the DPA and the DPU. When running over an InfiniBand fabric, the user must ensure that the subnet is well-configured, and that the relevant device's port is in `active` state.

Message Stream Functionality

The user can create as many streams as they see fit, up to a maximum of `FLEXIO_MSG_DEV_MAX_STREAMS_AMOUNT` as defined in `flexio.h`.

Every stream has its own messaging level which serves as a filter where messages with a level below that of the stream are filtered out.

The first stream created is the `default_stream` gets stream ID 0, and it is created with messaging level `FLEXIO_MSG_DEV_INFO` by default.

The stream ID defined by `FLEXIO_MSG_DEV_BROADCAST_STREAM` serves as a broadcast stream which means it messagaes all open streams (with the proper messaging level).

A stream can be configured with a synchronization mode attribute according to the following options:

- `sync` - displays the messages as soon as they are sent from the device to the host side using the verb SEND.
- `async` - uses the verb RDMA write. When the programmer calls the stream's flush functionality, all the messages in the buffer are displayed (unless there was a wraparound due to the size of messages being bigger than the size allocated for them). In this synchronization mode, the flush should be called at the end of the run.
- `batch` - uses RDMA write and RDMA write with immediate. It works similarly to the async mode, except the fact each batch size of messages is being flushed and therefore displayed automatically in every batch. The purpose is to allow the host to use fewer resources for device messaging.

Device Messaging Assumptions

Device messaging uses RPC calls to create, modify, and destroy streams. By default, these RPC calls run with affinity `none`, which requires at least one available EU on the default group. If the user wants to set the management affinity of a stream to a different option (any affinity option is supported, including forcing `none`, which is the default behavior) they should specify this in the stream attributes using the `mgmt_affinity` field.

Printf Support

Only limited functionality is implemented for printf. Not all libc printf is supported.

Please consult the following list for supported modifiers:

- Formats - `%c`, `%s`, `%d`, `%ld`, `%u`, `%lu`, `%i`, `%li`, `%x`, `%hx`, `%hxx`, `%lx`, `%X`, `%lX`, `%lo`, `%p`, `%%`

- Flags - `.` , `*` , `-` , `+` , `#`
- General supported modifiers:
  - "0" padding
  - Min/max characters in string
- General unsupported modifiers:
  - Floating point modifiers - `%e` , `%E` , `%f` , `%lf` , `%LF`
  - Octal modifier `%o` is partially supported
  - Precision modifiers

Core Dump

If the DPA process encounters a fatal error, the user can create a core dump file to review the application's status at that point using a GDB app.

Creating a core dump file can be done after the process has crashed (as indicated by the `flexio_err_status` API) and before the process is destroyed by calling the `flexio_coredump_create` API.

Recommendations for opening DPA core dump file using GDB:

- Use the `gdb-multiarch` application
- The `Program` parameter for GDB should be the device-side ELF file
  - Use the `dpacc-extract` tool (provided with the DPACC package) to extract the device-side ELF file from the application's ELF file

14.4.3.5.2.10 FlexIO Samples

This section describes samples based on the FlexIO SDK. These samples illustrate how to use the FlexIO API to configure and execute code on the DPA.

Running FlexIO Sample

The FlexIO SDK samples serve as a reference for building and running FlexIO-based DPA applications. They provide a collection of out-of-the-box working DPA applications that encompass the basic functionality of the FlexIO SDK.

Documentation

- Refer to NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software
- Refer to NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples

Minimal Requirements

The user must have the following installed:

- DOCA DPACC package
- DOCA RDMA package
- pkg-config package
- Python3 package
- Gcc with version 7.0 or higher
- Meson package with version 0.53.0 or higher

- Ninja package
- DOCA FlexIO SDK

Sample Structure

Each sample is situated in its own directory and is accompanied by a corresponding description in README files. Every sample comprises two applications:

- The first, located in the `device` directory, is designed for DPA
- The second, found in the `host` directory, is intended for execution on the DPU or host in a Linux OS environment

Additionally, there is a `common` directory housing libraries for the examples. These libraries are further categorized into `device` and `host` directories to facilitate linking with similar applications. Beyond containing functions and macros, these libraries also serve as illustrative examples for how to use them.

The list of the samples:

- `flexio_rpc` - sample demonstrating how to run RPC functions from DPA
- `packet_processor` - sample demonstrating how to process a package

Building the Samples

```
cd /opt/mellanox/fleio/samples/
./build.sh --check-compatibility --rebuild
```

Samples

flexio_rpc

This sample application executes FlexIO with a remote process call.

The device program calculates the sum of 2 input parameters, prints the result, and copies the result back to the host application.

This sample demonstrates how applications are built (DPA and host), how to create processes and message streams, how to open the IBV device, and how to use RPC from the host to DPA function.

Compilation

```
cd /opt/mellanox/flexio/samples/
./build.sh --check-compatibility --rebuild
```

The output path:

```
/opt/mellanox/flexio/samples/build/flexio_rpc/host/flexio_rpc
```

Usage

```
<sample_root>/build/flexio_rpc/host/flexio_rpc <mlx5_device> <arg1> <arg2>
```

Where:

- `mlx5_device` - IBV device with DPA
- `arg1` - first numeric argument
- `arg2` - second numeric argument

Example:

```
$/opt/mellanox/flexio/samples/build/flexio_rpc/host/flexio_rpc mlx5_0 44 55
Welcome to 'Flex IO RPC' sample
Registered on device mlx5_0
/  2/Calculate: 44 + 55 = 99
Result: 99
Flex IO RPC sample is done
```

flexio_packet_process

This example demonstrates packet processing handling.

The device application implements a handler for `flexio_pp_dev` that receives packets from the network, swaps MAC addresses, inserts some text into the packet, and sends it back.

This allows the user to send UDP packets (with a packet length of 65 bytes) and check the content of returned packets. Additionally, the console displays the execution of packet processing, printing each new packet index. Device messaging operates in synchronous mode (i.e., each message from the device received by the host is output immediately).

This sample illustrates how applications work with libraries (DPA and host), how to create SQ, RQ, CQ, memory keys, and doorbell rings, how to create and use DPA memory buffers, how to use UAR, and how to create and run event handlers.

Compilation

```
cd /opt/mellanox/flexio/samples/
./build.sh --check-compatibility --rebuild
```

The output path:

```
/opt/mellanox/flexio/samples/build/packet_processor/host/flexio_packet_processor
```

Usage

```
<sample_root>/build/packet_processor/host/flexio_packet_processor <mlx5_device>
```

Where:

- `mlx5_device` - name of IB device with DPA
- `--nic-mode` - optional parameter indicating that the application is run from the host. If the application is run from DPU, then the parameter should not be used.

For example

```
$sudo /build/packet_processor/host/flexio_packet_processor mlx5_0
```

The application must run with root privileges.

Running with Traffic

Run host-side sample:

```
$ cd <sample_root>
$ sudo ./build/packet_processor/host/flexio_packet_processor mlx5_0
```

Use another machine connected to the setup running the application. Bring the interface used as packet generator up:

```
$ sudo ifconfig my_interface up
```

Use `scapy` to run traffic to the device the application is running on:

```
$ python

>>> from scapy.all import *
>>> from scapy.layers.inet import IP, UDP, Ether

>>> sendp(Ether(src="02:42:7e:7f:eb:02", dst='52:54:00:79:db:d3')/IP()/UDP()/Raw(load="===============12345678"),
iface="my_interface")
```

Source MAC must be same as above as the application defines a steering rule for it. Destination MAC can be anything.

The load should be kept the same as above, as the application looks for this pattern and changes it during processing.

Interface name should be changed to the interfaced used for traffic generation.

The packets can be viewed using `tcpdump`:

```
$ sudo tcpdump -i my_interface -en host 127.0.0.1 -X
```

Example output

```
Example output:
11:53:51.422075 02:42:7e:7f:eb:02 > 52:54:00:12:34:56, ethertype IPv4 (0x0800), length 65: 127.0.0.1.domain > 127.0.0
.1.domain: 15677 op7+% [b2&3=0x3d3d] [15677a] [15677q] [15677n] [15677au][|domain]
        0x0000:  4500 0033 0001 0000 4011 7cb7 7f00 0001  E..3....@.|.....
        0x0010:  7f00 0001 0035 0035 001f 42c6 3d3d 3d3d  .....5.5..B.==== <-- Original data
        0x0020:  3d3d 3d3d 3d3d 3d3d 3d3d 3d31 3233 3435  ===========12345
        0x0030:  3637 38                                  678
11:53:51.700038 52:54:00:12:34:56 > 02:42:7e:7f:eb:02, ethertype IPv4 (0x0800), length 65: 127.0.0.1.domain > 127.0.0
.1.domain: 26144 op8+% [b2&3=0x4576] [29728a] [25966q] [25701n] [28015au][|domain]
        0x0000:  4500 0033 0001 0000 4011 7cb7 7f00 0001  E..3....@.|.....
        0x0010:  7f00 0001 0035 0035 001f 42c6 6620 4576  .....5.5..B.f.Ev <-- Modified data
        0x0020:  656e 7420 6465 6d6f 2a2a 2a2a 2a2a 2a2a  ent.demo********
        0x0030:  2a2a 2a                                  ***
```

14.4.3.5.3 DPA Application Authentication

DPA Application Authentication is supported at beta level for BlueField-3.

DPA Application Authentication is currently only supported with statically linked libraries. Dynamically linked libraries are currently not supported.

This section provides instructions for developing, signing, and using authenticated BlueField-3 data-path accelerator (DPA) applications. It includes information on:

- Principles of root of trust and structures supporting it
- Device ownership transfer/claiming flow (i.e., how the user should configure the device so that it will authenticate the DPA applications coming from the user)
- Crypto signing flow and ELF file structure and tools supporting it

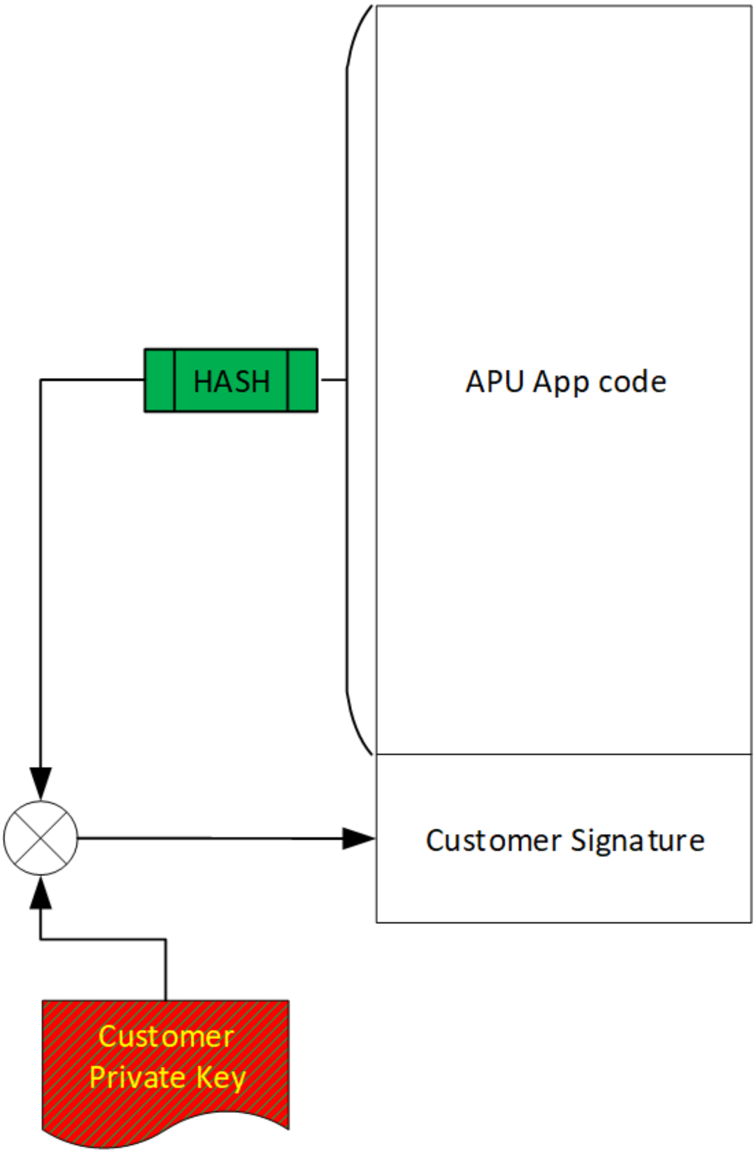
14.4.3.5.3.1 Root of Trust Principles

Signing of 3rd Party DPA App Code

NVIDIA® BlueField®-3 introduces the ability for customers/device owners to sign applications running on the DPA with their private key and have it authenticated by a device-embedded certificate chain. This provides the benefit of ensuring that only code permitted by the customer can run on the DPA. The customer can be any party writing code intended to run on the DPA (e.g., a cloud service provider, OEM, etc).

The following figure illustrates the signature of customer code. This signature will allow NVIDIA firmware to authenticate the source of the application's code.

Example of Customer DPA Code Signed by Customer for Authentication



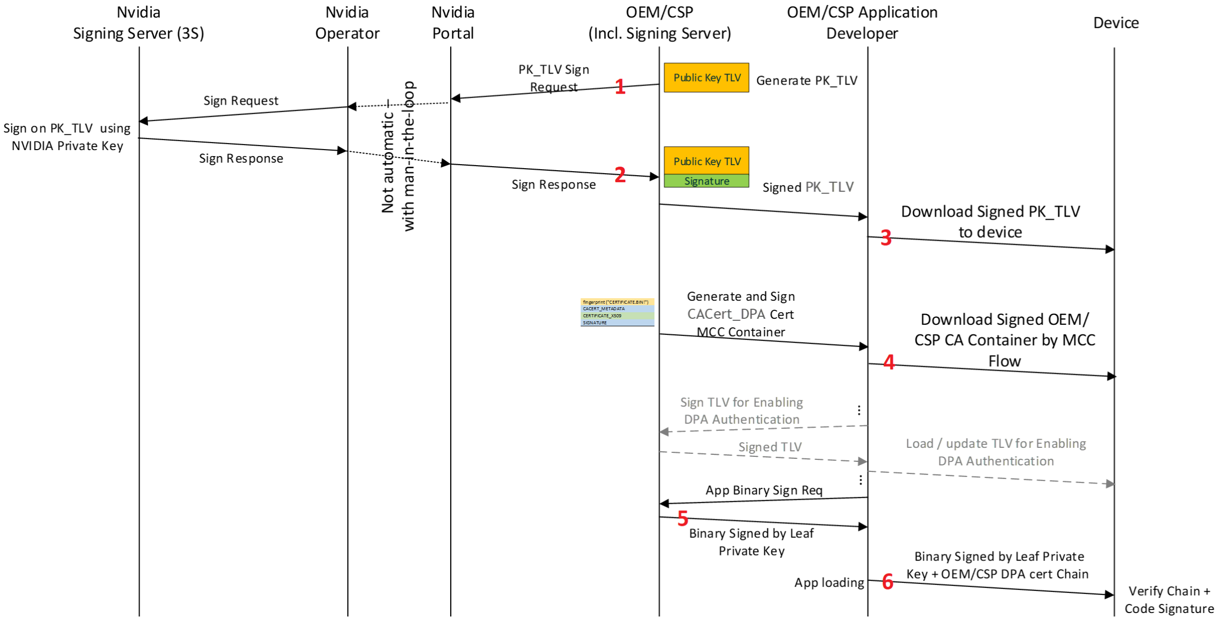
The high-level scheme is as follows (see figure "Loading of Customer Keys and CA Certificates and Provision of DPA Firmware to BlueField-3 Device"):

The numbers of these steps correspond to the numbers indicated in the figure below.

1. Customer provides NVIDIA Enterprise Support the public key for device ownership.
2. NVIDIA signs the customer's public key and sends it back to the customer.
3. Customer uploads the NVIDIA-signed public key to the device, enabling "Transfer of Ownership" to the customer (from NVIDIA).
4. Using the private key corresponding to the public key uploaded to the device, the customer can now enable DPA authentication and load the root certificate used for authentication of DPA App code.
5. DPA app code crypto-signed by the customer serves to authenticate the source of the app code.
   The public key used to authenticate the DPA app is provided as part of the certificate chain (leaf certificate), together with the DPA firmware image.

6. App code and the owner signature serves to authorize the app execution by the NVIDIA firmware (similar to NVIDIA own signature).

Loading of Customer Keys and CA Certificates and Provision of DPA Firmware to BlueField-3 Device



The following sections provide more details about this high-level process.

Verification of Authenticity of DPA App Code

Authentication of application firmware code before authorization to execute shall consist of validation of the customer certificate chain and customer signature using the customer's public key.

Public Keys (Infrastructure, Delivery, and Verification)

For the purposes of the authentication verification of the application firmware, the public key must be securely provided to the hardware. To do so, a secure Management Component Control (MCC) Flow shall be used. Using this, the content of the downloaded certificate is enveloped in an MCC Download Container and signed by NVIDIA Private Key.

The following is an example of how to use the MCC flow describes in detail the procedures, tools and structures supporting this (Section "Loading of CSP CA Certificates and Keys and Provisioning of DPA Firmware to Device" describes the high-level flow for this).

The following command burns the certificate container:

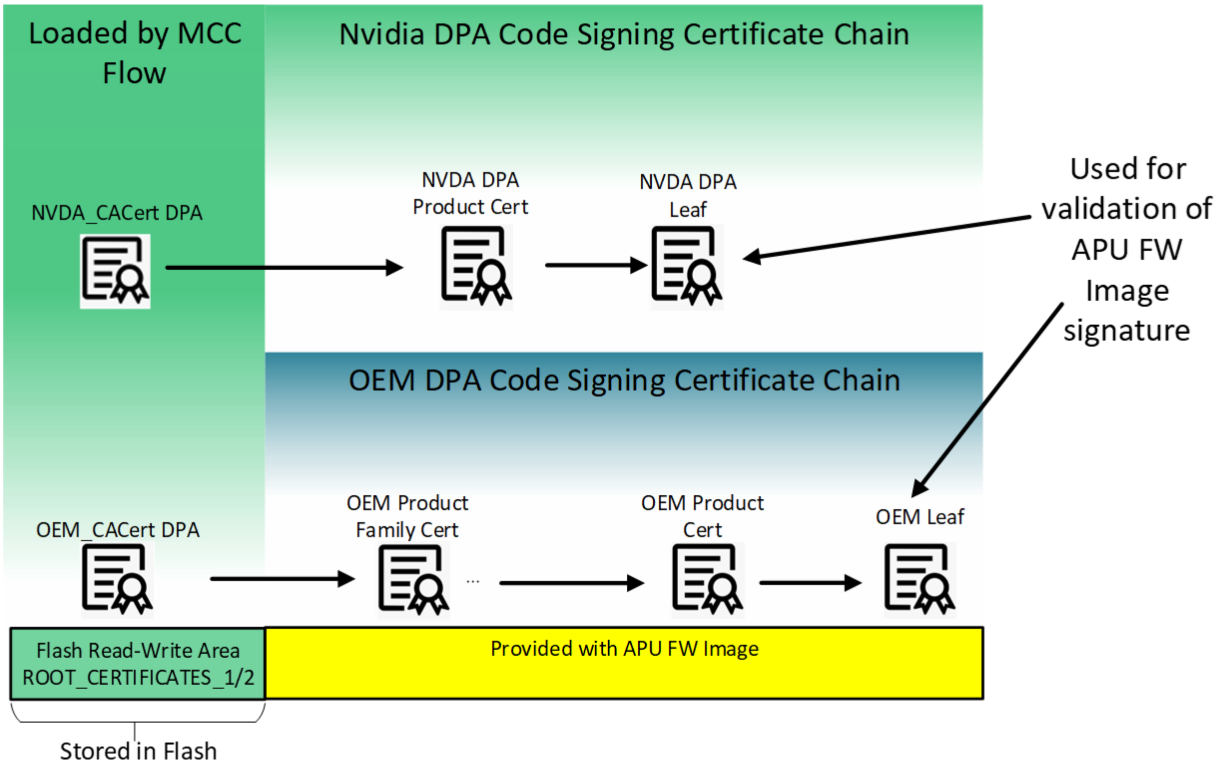
```
flint -d <mst device> -i <signed-certificate-container> burn
```

Two use cases are possible:

- The DPA application is developed internally in NVIDIA, and the authentication is based on internal NVIDIA keys and signing infrastructure
- The DPA application is developed by a customer, and the authentication is based on the customer certificate chain

In either case, the customer must download the relevant CA certificate to the device.

ROT Certificate Chain


Loaded by MCC Flow
Nvidia DPA Code Signing Certificate Chain
NVDA_CACert DPA
NVDA DPA Product Cert
NVDA DPA Leaf
Used for validation of APU FW Image signature
OEM DPA Code Signing Certificate Chain
OEM_CACert DPA
OEM Product Family Cert
...
OEM Product Cert
OEM Leaf
Flash Read-Write Area ROOT_CERTIFICATES_1/2
Provided with APU FW Image
Stored in Flash

This figure illustrates the build of the certificate chain used for validation of DPA app images. The leaf certificate of these chains is used to validate the DPA application supplied by the customer (with ROT from customer CA). The NVIDIA certificate chain for validation of DPA applications (built internally in NVIDIA) is structured in a very similar way. OEMDpaCert CA is the root CA which can be used by the customer to span their certificate chain up to the customer leaf certificate which is used for validating the signature of the application's image. Similarly, NVDADpaCert CA is the root CA used internally in NVIDIA to build the DPA certificate chain for validation of NVIDIA DPA apps.

Customer private keys must be kept secure and are the sole responsibility of the customer to maintain. It is recommended to have a set of keys ready and usable by customer for redundancy purposes.The whole customer certificate chain, including root CA and leaf, must not exceed 4 certificates.

The `NVDA_CACert_DPA` and `OEM_CACert_DPA` certificates are self-signed and trusted because they are loaded by the secure MCC flow and authenticated by the firmware.

The customer certificate chain beyond `OEM_CACert_DPA` is delivered with the DPA image, including the leaf certificate that is used for validating the cryptographic signature of the DPA firmware (see table "ELF Crypto Data Section Fields Description").

For more details on the certificates and their location in the flash, contact NVIDIA Enterprise Support to obtain the *Flash Application Note*. The rest of the certificate chain used for the DPA firmware authentication includes:

- For NVIDIA-signed images (e.g., figure "ROT Certificate Chain"): NVDA DPA root certificate (`NVDA_CACert_DPA` can be downloaded here)
- For customer-signed images (e.g., figure "ROT Certificate Chain"): Customer CA certificate, customer product, and customer leaf certificates

In both cases (NVIDIA internal and customer-signed) these parts of the certificate chain are attached to the DPA firmware image.

Loading of CSP CA Certificates and Keys and Provisioning of DPA Firmware to Device

The figure "Loading of Customer Keys and CA Certificates and Provision of DPA Firmware to BlueField-3 Device" shows, at high-level, the procedures for loading user public keys to the device, signing and loading of customer certificates MCC container, and downloading the DPA firmware images.

For clarity, the hierarchy of ROT validation is as follows:

1. Customer public key to be used for customer TLVs and `CACert_DPA` certificate validation, `PK_TLV` (i.e., `NV_LC_NV_PUBLIC_KEY`):
   a. For a device whose DPA authentication ability the customer wishes to enable for the first time, the customer must get it signed and authenticated by NVIDIA keys by reaching out to NVIDIA Enterprise Support. The complete flow is described in "Device Ownership Claiming Flow".
   b. After `PK_TLV` is loaded, it can be updated by authenticating the update using either the same `PK_TLV`. The complete flow is described in "Device Ownership Claiming Flow".
   c. Authentication of TLV for enabling/disabling DPA authentication is also validated by the `PK_TLV`. The complete flow is described in section "DPA Authentication Enablement".
2. Loading of CA certificate (`CACert_DPA`) to be used for DPA code validation. It is authenticated using the same `PK_TLV`.
   The complete flow is described in "Uploading DPA Root CA Certificate".
3. The public key in the leaf of the certificate chain anchored by `CACert_DPA` is used for authentication of the DPA firmware Image.
   The structure of the ELF file containing the DPA app and the certificate chain is described in "ELF File Structure".

A scalable and reliable infrastructure is required to support many users. The customer must also have an infrastructure to support their own code signing process according to their organization's security policy. Both matters are out of the scope of this document.

> ⚠ Trying to utilize the DPA signing flow in a firmware version prior to DOCA 2.2.0 is not supported.

Device Ownership Claiming Flow

NVIDIA networking devices allow the user of the device to customize the configurations, and in some cases change the behavior of the device. This set of available customizations is controlled by higher level NVIDIA configurations that come either as part of the device firmware or as a separate update file. To allow customers/device owners to change the set of available configurations and allowed behaviors, each device can have a device owner who is allowed to change the default behaviors and configurations of the device, and to change what configurations are exposed to the user.

The items controlled by the customer/device owner are:

- Device configurations: The customer/device owner can change the default value of any configuration available to users. They can also prevent users from changing the value.
- Trusted root certificates: The customer/device owner can control what root certificates the device trusts. These certificates control various behaviors (e.g., what 3rd party code the BlueField DPA accepts).

After the device has the public key of the owner, whenever an NVconfig file is signed with this key, one of two things must be true:

- The `nv_file_id` field in the NVconfig file must have the parameter `keep_same_priority` as `True`; or
- The NVconfig file must contain the public key itself (so the public key is rewritten to the device)

Otherwise, the public key is removed from the device, and as such will not accept files signed by the matching private key.

Detailed Ownership Claiming Flow

1. Customer generates a private-public key pair, and a UUID for the key pair.
    a. Generating UUID for the key pair:

    ```
    uuidgen -t
    ```

    Example output:

    ```
    77dd4ef0-c633-11ed-9e20-001dd8b744ff
    ```

    b. Generating an RSA key pair:

    ```
    openssl genrsa -out OEM.77dd4ef0-c633-11ed-9e20-001dd8b744ff.pem 4096
    ```

    Example output:

    ```
    Generating RSA private key, 2048 bit long modulus
    ...........+++
    ..............+++
    e is 65537 (0x10001)
    ```

    c. Extracting the public key file from the RSA key pair:

    ```
    openssl rsa -in OEM.77dd4ef0-c633-11ed-9e20-001dd8b744ff.pem -out OEM.77dd4ef0-
    c633-11ed-9e20-001dd8b744ff.public -pubout -outform PEM
    ```

    Output:

    ```
    writing RSA key
    ```

    The public key should look similar to the following:

    ```
    -----BEGIN PUBLIC KEY-----
    MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAxfijde+27A3pQ7MoZnlm
    mtpyuHO1JY9AUeKaHUXkWRiopL9Puswx1KcGfWJSNzlEPZRevTHraYlLQCru4ofr
    W9NBE/qIwS2n7kiFwCCvZK6FKUUqZAuMJTpfuNtv9o4C4v0ZiX4TQqWDND8hy+1L
    hPf3QLRiJ/ux4G6uHIFwENSwagershuKD0RI6BaZ1g9S9IxdXcD0vTdEuDPqQ0m4
    CwEs/3xnksNRLUM+TiPEZoc5MoEoKyJv4GFbGttabhDCt5sr9RqAqTNUSDI9B0jr
    XoQBQQpqRgYd3lQ31Fhh3G9GjtoAcUQ6l0Gct3DXKFTAADV3Lyo1vjFNrOKUhdhT
    pjDKzNmZAsxyIZI0buc24TCgj1yPyFboJtpnHmltyxfm9e+EJsdSIpRiX8YTWwkN
    ```

```
aIzNj08VswULwbKow5Gu5FFpE/uXDE3cXjLOUNnKihszFv4qkqsQjKaK4GszXge+
jfiEwsDKwS+cuWd9ihnyLrIWF23+OX0S5xjFXDJE8UthOD+3j3gGmP3kze1Iz2YP
Qvh3ITPRsqQltaiYh+CivqaCHC0voIMOP1ilAEZ/rW85pi6LA8EsudNMG2ELrUyl
SznBzZI/OxMk4qKx9nGgjaP2YjmcPw2Ffc9zZcwl57ThEOhlyS6w3E9xwBvZINLe
gMuOIWsu1FK3lIGxMSCUZQsCAwEAAQ==
-----END PUBLIC KEY-----
```

2. Customer provides NVIDIA Enterprise Support the public key for device ownership with its UUID.
3. NVIDIA generates a signed NVconfig file with this public key and sends it to the customer. This key may only be applied to devices that do not have a device ownership key installed yet.
4. Customer uses `mlxconfig` to install the OEM key on the needed devices.

```
mlxconfig -d /dev/mst/<dev> apply oem_public_key_nvconfig.bin
```

To check if the upload process has been successful, the customer can use `mlxconfig` to query the device and check if the new public key has been applied. The relevant parameters to query are `LC_NV_PUB_KEY_EXP`, `LC_NV_PUB_KEY_UUID`, and `LC_NV_PUB_KEY_0_255`.

Example of query command and expected response:

```
mlxconfig -d <dev>-e q LC_NV_PUB_KEY_0_255
```

Uploading DPA Root CA Certificate

After uploading a device ownership public key to the device, the owner can upload DPA root CA certificates to the device. There can be multiple DPA root CA certificates on the device at the same time.

If the owner wants to upload authenticated DPA apps developed by NVIDIA, they must upload the NVIDIA DPA root CA certificate found here.

If the owner wants to sign their own DPA apps, they must create another public-private key pair (in addition to the device ownership key pair), create a certificate containing the DPA root CA public key, and create a container with this certificate using `mlxdpa`.

To upload a signed container with a DPA root CA certificate to the device, `mlxdpa` must be used. This can be done both for either NVIDIA or customer-created certificates.

Generating DPA Root CA Certificate

1. Create a DER encoded certificate containing the public key used to validate DPA apps.
    a. Generating a certificate and a new key pair:

    ```
    openssl req -x509 -newkey rsa:4096 -keyout OEM-DPA-root-CA-key.pem -outform der -out OEM-DPA-root-CA-cert.crt -sha256 -nodes -subj "/C=XX/ST=OEMStateName/L=OEMCityName/O=OEMCompanyName/OU=OEMCompanySectionName/CN=OEMCommonName" -days 3650
    ```

    > Both SHA256 and SHA512 are supported in cert. Only a RSA 4096 key is supported. The size of each certificate in DER format must be less than 1792 bytes.

    Output:

```
Generating a 4096 bit RSA private key
......++
......................++
writing new private key to 'OEM-DPA-root-CA-key.pem'
-----
```

2. Create a container for the certificate and sign it with the device ownership private key.
    a. To create and add a container:

    ```
    mlxdpa --cert_container_type add -c <cert.der> -o <path to output> --life_cycle_priority <Nvidia/
    OEM/User> create_cert_container
    ```

    Output example:

    ```
    Certificate container created successfully!
    ```

    b. To sign a container:

    ```
    mlxdpa --cert_container <path to container> -p <key file> --keypair_uuid <uuid> --cert_uuid <uuid>
    --life_cycle_priority <Nvidia/OEM/User> -o <path-to-output> sign_cert_container

    Certificate container signed successfully!
    ```

Manually Signing Container

If the server holding the private key cannot run `mlxdpa`, it is possible to manually sign the certificate container and add the signature to the container. In that case, the following process should be followed:

1. Generate unsigned cert container:

    ```
    mlxdpa --cert_container_type add -c <.DER-formatted-certificate> -o <unsigned-container-path> --
    keypair_uuid <uuid> --cert_uuid <uuid> --life_cycle_priority OEM create_cert_container
    ```

2. Generate signature field header:

    ```
    echo "90 01 02 0C 10 00 00 00 00 00 00 00" | xxd -r -p - <signature-header-path>
    ```

3. Generate signature of container (in whatever way, this is an example only):

    ```
    openssl dgst -sha512 -sign <private-key-pem-file> -out <container-signature-path> <unsigned-container-path>
    ```

4. Concatenate unsigned container, signature header, and signature into one file:

    ```
    cat <unsigned-container-path> <signature-header-path> <container-signature-path> > <signed-container-path>
    ```

Uploading Certificates

Upload each signed container containing the desired certificates for the device.

```
flint -d <dev> -i  <signed-container> -y b
```

Output example:

```
-I- Downloading FW ...
FSMST_INITIALIZE -   OK
Writing DIGITAL_CACERT_REMOVAL component -   OK
-I- Component FW burn finished successfully.
```

Removing Certificates

To remove root CA certificates from the device, the user must apply a certificate removal container signed by the device ownership private key.

There are two ways to remove certificates, either removing all certificates, or removing all installed certificates:

- Removing all root CA certificates from the device:
    a. Generate a signed container to remove all certificates.
        i. Created certificate container:

        ```
        mlxdpa --cert_container_type remove --remove_all_certs -o <path-to-container> --
        life_cycle_priority <Nvidia/OEM/User> create_cert_container
        ```

        Output example:

        ```
        Certificate container created successfully!
        ```

        ii. Sign certificate container:

        ```
        mlxdpa --cert_container <path-to-container> -p <key-file> --keypair_uuid <uuid> --
        life_cycle_priority <Nvidia/OEM/User> -o <path-to-signed-container> sign_cert_container
        ```

        Output example:

        ```
        Certificate container signed successfully!
        ```

    b. Apply the container to the device.

    ```
    flint -d <dev> -i <signed-container> -y b
    ```

    Output example:

    ```
    -I- Downloading FW ...
    FSMST_INITIALIZE -   OK
    Writing DIGITAL_CACERT_REMOVAL component -   OK
    -I- Component FW burn finished successfully.
    ```

- Removing specific root CA certificates according to their UUID:
    a. Generate a signed container to remove certificate based on UUID.
        i. Create the container.

        ```
        mlxdpa --cert_container_type remove--cert_uuid <uuid> -o <path-to-container> --
        life_cycle_priority <Nvidia/OEM/User> create_cert_container
        ```

        Output example:

        ```
        Certificate container created successfully!
        ```

        ii. Sign the container:

        ```
        mlxdpa --cert_container <path-to-container> -p <key-file> --keypair_uuid <uuid> --cert_uuid
        <uuid> --life_cycle_priority <Nvidia/OEM/User> -o <path to output> sign_cert_container
        ```

        Output example:

        ```
        Certificate container signed successfully!
        ```

b. Apply the container to the device:

```
flint -d <dev> -i <signed container> -y b
```

Output:

```
-I- Downloading FW ...
FSMST_INITIALIZE -   OK
Writing DIGITAL_CACERT_REMOVAL component -   OK
-I- Component FW burn finished successfully.
```

DPA Authentication Enablement

After the device has a device ownership key and DPA root CA certificates installed, the owner of the device can enable DPA authentication. To do this, they must create an NVconfig file, sign it with the device ownership private key, and upload the NVconfig to the device.

Generating NVconfig Enabling DPA Authentication

1. Create XML with TLVs to enable DPA authentication.
   a. Get list of available TLVs for this device:

```
mlxconfig -d /dev/mst/<dev> gen_tlvs_file enable_dpa_auth.txt
```

Output:

```
Saving output...
Done!
```

Example part of the generated text file:

```
file_applicable_to                                0
file_comment                                      0
file_signature                                    0
file_dbg_fw_token_id                              0
file_cs_token_id                                  0
file_btc_token_id                                 0
file_mac_addr_list                                0
file_public_key                                   0
file_signature_4096_a                             0
file_signature_4096_b                             0
…
```

   b. Edit the text file to contain the following TLVs:

```
file_applicable_to                                1
nv_file_id_vendor                                 1
nv_dpa_auth                                       1
```

   c. Convert the `.txt` file to XML format with another mlxconfig command:

```
mlxconfig -a gen_xml_template enable_dpa_auth.txt enable_dpa_auth.xml
```

Output:

```
Saving output...
Done!
```

The generated `.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<config xmlns="http://www.mellanox.com/config">
<file_applicable_to ovr_en='1' rd_en='1' writer_id='0'>
     <psid></psid>
     <psid_branch></psid_branch>
 </file_applicable_to>

<nv_file_id_vendor ovr_en='1' rd_en='1' writer_id='0'>

    <!-- Legal Values: False/True -->
    <disable_override></disable_override>

    <!-- Legal Values: False/True -->
    <keep_same_priority></keep_same_priority>

    <!-- Legal Values: False/True -->
    <per_tlv_priority></per_tlv_priority>

    <!-- Legal Values: False/True -->
    <erase_lower_priority></erase_lower_priority>
    <file_version></file_version>
    <day></day>
    <month></month>
    <year></year>
    <seconds></seconds>
    <minutes></minutes>
    <hour></hour>

</nv_file_id_vendor>

<nv_dpa_auth ovr_en='1' rd_en='1' writer_id='0'>
    <!-- Legal Values: False/True -->
    <dpa_auth_en></dpa_auth_en>

</nv_dpa_auth>
</config>
```

d. Edit the XML file and add the information for each of the TLVs, as seen in the following example XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://www.mellanox.com/config">

<file_applicable_to ovr_en='0' rd_en='1' writer_id='0'>
    <psid>TODO</psid>
    <psid_branch>TODO</psid_branch>
</file_applicable_to>

<nv_file_id_vendor ovr_en='0' rd_en='1' writer_id='0'>
    <disable_override>False</disable_override>
    <keep_same_priority>True</keep_same_priority>
    <per_tlv_priority>False</per_tlv_priority>
    <erase_lower_priority>False</erase_lower_priority>
    <file_version>TODO</file_version>
    <day>TODO</day>
    <month>TODO</month>
    <year>TODO</year>
    <seconds>TODO</seconds>
    <minutes>TODO</minutes>
    <hour>TODO</hour>
</nv_file_id_vendor>

<nv_dpa_auth ovr_en='0' rd_en='1' writer_id='0'>
    <dpa_auth_en>True</dpa_auth_en>
</nv_dpa_auth>
</config>
```

In `nv_file_id_vendor` , `keep_same_priority` must be `True` to avoid removing the ownership public key from the device. More information they can be found in section "Device Ownership Claiming Flow".

The `ovr_en` should be set to 0. This can ignore user priority changing `nv_dpa_auth` .

2. Convert XML file to binary NVconfig file and sign it using `mlxconfig` :

```
mlxconfig -p OEM.77dd4ef0-c633-11ed-9e20-001dd8b744ff.pem -u 77dd4ef0-c633-11ed-9e20-001dd8b744ff
create_conf enable_dpa_auth.xml enable_dpa_auth.bin
```

Output of `create_conf` command:

```
Saving output...
Done!
```

3. Upload NVconfig file to device by writing the file to the device:

```
mlxconfig -d /dev/mst/<dev> apply enable_dpa_auth.bin
```

Output:

```
Saving output...
Done!
```

4. Verify that the device has DPA authentication enabled by reading the status of DPA authentication from the device:

```
mlxconfig -d /dev/mst/<dev> -e q DPA_AUTHENTICATION
```

Output:

```
Device #1:
----------

Device type:    BlueField3
…
…
Configurations:                                      Default          Current           Next Boot
 RO     DPA_AUTHENTICATION                           True(1)          True(1)           True(1)
```

The DPU's factory default setting is configured with `dpa_auth_en=0` (i.e., DPA applications can run without authentication). To prevent configuration change by any user, it is strongly recommended for the customer to generate and install NVconfig with `dpa_auth_en=0/1`, according to their preferences, with `ovr_en=0`.

Manually Signing NVconfig File

If the server holding the private key cannot run mlxconfig, it is possible to manually sign the binary NVconfig file and add the signature to the file. In this case, the following process should be followed instead of step 2:

1. Generate unsigned NVconfig bin file from the XML file:

```
mlxconfig create_conf <xml-nvconfig-path> <unsigned-nvconfig-path>
```

2. Generate random UUID for signature:

```
uuidgen -r | xxd -r -p - <signature-uuid-path>
```

3. Generate signature of NVconfig bin file (in whatever way, this is an example only):

```
openssl dgst -sha512 -sign <private-key-pem-file> -out <nvconfig-signature-path> <unsigned-nvconfig-path>
```

4. Split the signature into two parts:

```
head -c 256 <nvconfig-signature-path> > <signature-part-1-path> && tail -c 256 <nvconfig-signature-path> >
<signature-part-2-path>
```

5. Add signing key UUID:

```
echo "<signing-key-UUID>" | xxd -r -p - <signing-key-uuid-path>
```

Use the signing key UUID, which must have a length of exactly 16 bytes, in a format like `aa9c8c2f-8b29-4e92-9b76-2429447620e0`.

6. Generate headers for signature struct:

```
echo "03 00 01 20 06 00 00 0B 00 00 00 00" | xxd -r -p - <signature-1-header-path>
echo "03 00 01 20 06 00 00 0C 00 00 00 00" | xxd -r -p - <signature-2-header-path>
```

7. Concatenate everything:

```
cat <unsigned-nvconfig-path> <signature-1-header-path> <signature-uuid-path> <signing-key-uuid-path>
<signature-part-1-path> <signature-2-header-path> <signature-uuid-path> <signing-key-uuid-path> <signature-
part-2-path> > <signed-nvconfig-path>
```

Device Ownership Transfer

The device owner may change the device ownership key to change the owner of the device or to remove the owner altogether.

First Installation

To install the first `OEM_PUBLIC_KEY` on the device, the user must upload an NVCONFIG file signed by NVIDIA. This file would contain the 3 `FILE` `_OEM_PUBLIC_KEY` TLVs of the current user.

Removing Device Ownership Key

Before removing the device ownership key completely, it is recommended that the device owner reverts any changes made to the device since it is not possible to undo them after the key is removed. Mainly, the root CA certificates installed by the owner should be removed.

1. To remove device ownership key completely, follow the steps in section "Generating NVconfig Enabling DPA Authentication" to create an XML file with TLVs.
2. Edit the XML file to contain the following TLVs:

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://www.mellanox.com/config">

<file_applicable_to ovr_en='0' rd_en='1' writer_id='0'>
    <psid> MT_0000000911</psid>
    <psid_branch> </psid_branch>
</file_applicable_to>

<nv_file_id_vendor ovr_en='0' rd_en='1' writer_id='0'>
    <disable_override>False</disable_override>
    <keep_same_priority>False</keep_same_priority>
    <per_tlv_priority>False</per_tlv_priority>
    <erase_lower_priority>False</erase_lower_priority>
    <file_version>0</file_version>
    <day>17</day>
    <month>7</month>
    <year>7e7</year>
    <seconds>1</seconds>
    <minutes>e</minutes>
    <hour>15</hour>
</nv_file_id_vendor>
</config>
```

The TLVs in this file are the only TLVs that will have OEM priority after this file is applied, and as the device ownership key will no longer be on the device, the OEM will no longer be able to change the TLVs. To have OEM priority TLVs on the device after removing the device ownership key, add to this XML any TLV that must stay as default on the device.

3. Convert the XML file to a binary NVconfig TLV file signed by the device ownership key as described in section "Generating NVconfig Enabling DPA Authentication".

4. Apply the NVconfig file to the device as described in section "Generating NVconfig Enabling DPA Authentication".

Changing Device Ownership Key

To transfer ownership of the device to another entity, the previous owner can change the device ownership public key to the public key of the new owner.

To do this, they can use an NVconfig file, and include in it the following TLVs:

```
<nv_ls_nv_public_key_0 ovr_en='0' rd_en='1' writer_id='0'>
    <public_key_exp>65537</public_key_exp>
    <keypair_uuid>77dd4ef0-c633-11ed-9e20-001dd8b744ff</keypair_uuid>
</nv_ls_nv_public_key_0>

<nv_ls_nv_public_key_1 ovr_en='0' rd_en='1' writer_id='0'>
    <key>
    c5:f8:a3:75:ef:b6:ec:0d:e9:43:b3:28:66:79:
    66:9a:da:72:b8:73:b5:25:8f:40:51:e2:9a:1d:45:
    e4:59:18:a8:a4:bf:4f:ba:cc:31:d4:a7:06:7d:62:
    52:37:39:44:3d:94:5e:bd:31:eb:69:89:4b:40:2a:
    ee:e2:87:eb:5b:d3:41:13:fa:88:c1:2d:a7:ee:48:
    85:c0:20:af:64:ae:85:29:45:2a:64:0b:8c:25:3a:
    5f:b8:db:6f:f6:8e:02:e2:fd:19:89:7e:13:42:a5:
    83:34:3f:21:cb:ed:4b:84:f7:f7:40:b4:62:27:fb:
    b1:e0:6e:ae:1c:81:70:10:d4:b0:6a:07:ab:b2:1b:
    8a:0f:44:48:e8:16:99:d6:0f:52:f4:8c:5d:5d:c0:
    f4:bd:37:44:b8:33:ea:43:49:b8:0b:01:2c:ff:7c:
    67:92:c3:51:2d:43:3e:4e:23:c4:66:87:39:32:81:
    28:2b:22:6f:e0:61:5b:1a:db:5a:6e:10:c2:b7:9b:
    2b:f5:1a:80:a9:33:54:48:32:3d:07:48:eb:5e:84:
    01:41:0a:6a:46:06:1d:de:54:37:d4:58:61:dc:6f:
    46:8e:da:00:71:44:3a:97:41:9c:b7:70:d7:28:54:
    c0:00:35:77:2f:2a:35:be:31:4d:ac:e2:94:85:d8:
    53:a6:
    </key>
</nv_ls_nv_public_key_1>

<nv_ls_nv_public_key_2 ovr_en='0' rd_en='1' writer_id='0'>
    <key>
    30:ca:cc:d9:99:02:cc:72:21:92:34:6e:e7:
    36:e1:30:a0:8f:5c:8f:c8:56:e8:26:da:67:1e:69:
    6d:cb:17:e6:f5:ef:84:26:c7:52:22:94:62:5f:c6:
    13:5b:09:0d:68:8c:cd:8f:4f:15:b3:05:0b:c1:b2:
    a8:c3:91:ae:e4:51:69:13:fb:97:0c:4d:dc:5e:32:
    ce:50:d9:ca:8a:1b:33:16:fe:2a:92:ab:10:8c:a6:
    8a:e0:6b:33:5e:07:be:8d:f8:84:c2:c0:ca:c1:2f:
    9c:b9:67:7d:8a:19:f2:2e:b2:16:17:6d:fe:39:7d:
    12:e7:18:c5:5c:32:44:f1:4b:61:38:3f:b7:8f:78:
    06:98:fd:e4:cd:ed:48:cf:66:0f:42:f8:77:21:33:
    d1:b2:a4:25:b5:a8:98:87:e0:a2:be:a6:82:1c:2d:
    2f:a0:83:0e:3f:58:a5:00:46:7f:ad:6f:39:a6:2e:
    8b:03:c1:2c:b9:d3:4c:1b:61:0b:ad:4c:a5:4b:39:
    c1:cd:92:3f:3b:13:24:e2:a2:b1:f6:71:a0:8d:a3:
    f6:62:39:9c:3f:0d:85:7d:cf:73:65:cc:25:e7:b4:
    e1:10:e8:65:c9:2e:b0:dc:4f:71:c0:1b:d9:20:d2:
    de:80:cb:8e:21:6b:2e:d4:52:b7:94:81:b1:31:20:
    94:65:0b
    </key>
</nv_ls_nv_public_key_2>
```

If the transfer is internal, the owner should set `keep_same_priority=True in` `nv_file_id_vendor` TLV and only include the 3 `nv_ls_nv_public_key_*` TLVs, `file_applicable_to` and `nv_file_id_vendor` TLVs in the NVconfig file.

If the transfer is to another OEM/CSP, the owner should clean the device (similarly to removing the device ownership key) and set `keep_same_priority=False` in `nv_file_id_vendor` TLV.

14.4.3.5.3.2 ELF File Structure

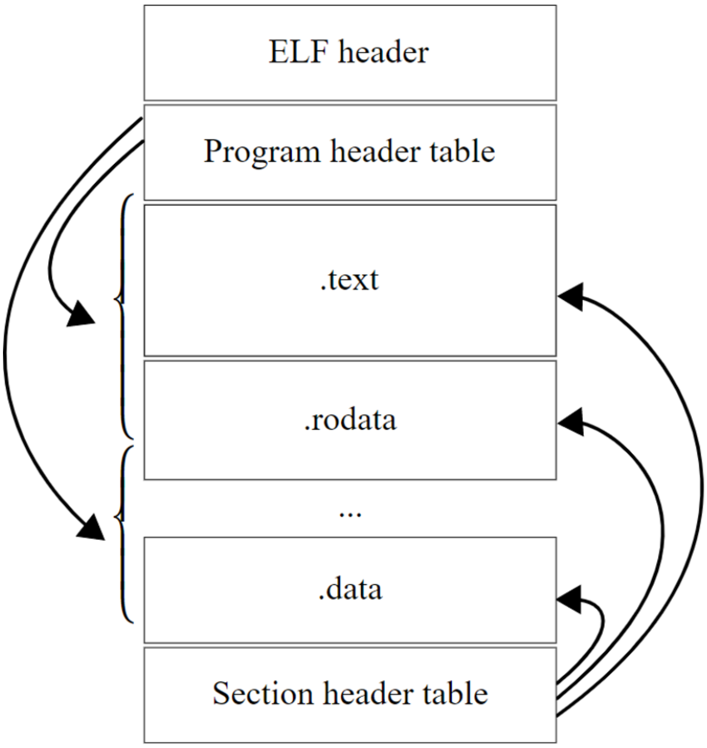
For maximal firmware code reuse, the format of the DPA image loaded from driver should be the same as for the file loaded from flash. As for files loaded from the host, ELF is the default file format. This is chosen as the format for the DPA image, both for flash and for files loaded from the host.

The following figure shows, schematically, a generic ELF file structure.

To support DPA Code authentication additional information needs to be presented to firmware. This info must include:

- Cryptographic signature of the DPA code
- Customer certificate chain including a Leaf Certificate with the public key to be used for signature validation (as described in section "Public Keys (Infrastructure, Delivery, and Verification)")

ELF File Structure Schematic



Crypto Signing Flow

The host ELF includes parts which run on the host, and those that run on DPA. DPA code files are incorporated in the "big" host ELF as binaries. Each host file may include several DPA applications.

When it is required to sign the DPA applications, the following steps need to be performed by the MFT Signing Tool (also see figure "Crypto Signing Flow"):

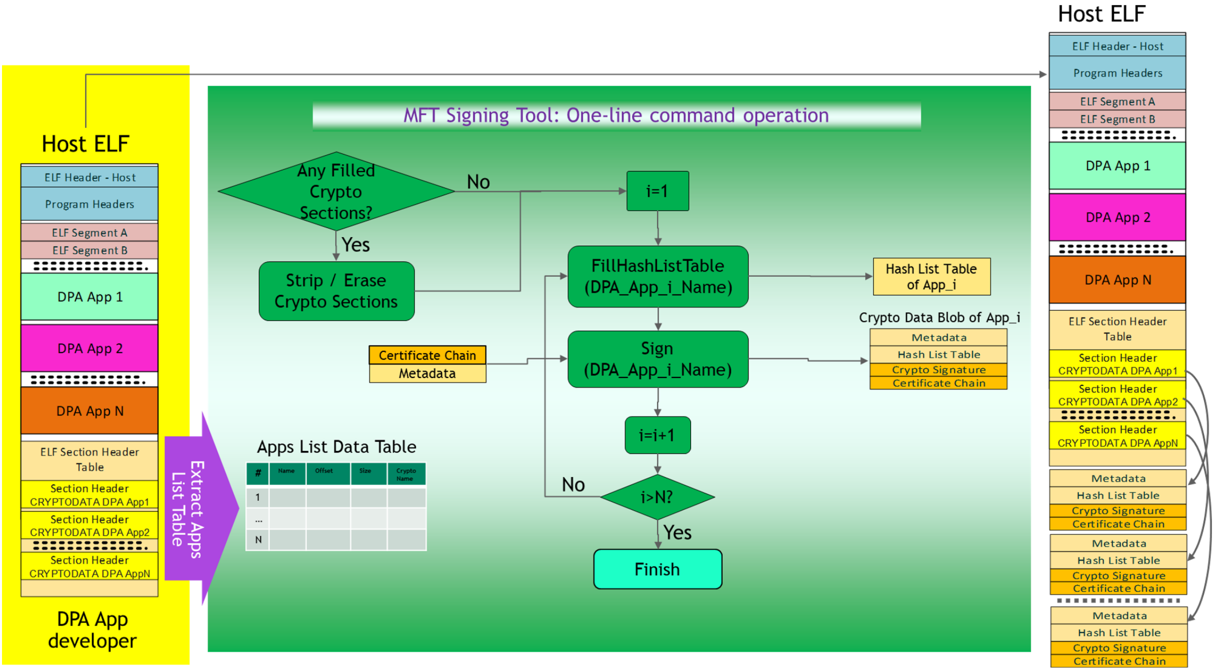
1. Using ELF manipulation library APIs of DPACC, extract Apps List Table
    a. Input - host ELF
    b. Output - apps list data table to include:
        i. DPA app index
        ii. DPA app name
        iii. Offset in host ELF
        iv. Size of app
        v. Name of corresponding crypto data section
        For each DPA application (from i=1 to i=N, N- number of DPA apps in the host ELF) run steps 2 and 3.
2. Fill hash list table:
    - Input: `Dpa_App_i`

    - Output: Hash list table
3. Sign the crypto data:
    - Input: {Metadata, Hash List Table}, key handle (e.g., UUID from leaf of the Certificate Chain)
    - Output: `Crypto_Data` "Blob", including: Metadata, Hash List Table, Crypto Signature, Certificate Chain
4. Add crypto data section to host ELF:
    - Inputs: Host ELF, crypto data section name to use
    - Output: File name of host ELF with signature added

The structures used in the flow (hash list table, metadata, etc.) are described in sections "ELF Crypto Data Section Content" and "Hash List Table Layout".

Signing the crypto data may be done using a signing server or a locally stored key.

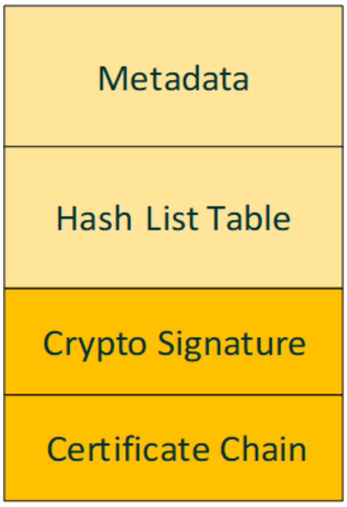
Crypto Signing Flow



ELF Cryptographic Data Section

This figure shows, schematically, the layout of the cryptographic data section, and the following subsections provide details about the ELF section header and the rest of the structures.

ELF Cryptographic Data Section Layout



Crypto Data ELF Section Header

Defined according to the ELF section header format.

ELF Section Header

Name	Offset	Range	Description
`sh_name`	0x0	4B	&("Cryptographic Data Section DPA App X") An offset to a string (in the `.shstrtab` section of ELF) which represents the name of this section
`sh_type`	0x4	4B	0x70000666 `SHT_CRYPTODATA` - the section is proprietary and holds crypto information defined in this document
`sh_flags`	0X8	8B	0 - no flags
`sh_addr`	0x10	8B	Virtual address of the section in memory, for sections that are loaded
`sh_offset`	0x18	8B	Offset of the section in the file image
`sh_size`	0x20	8B	Size in bytes of the section in the file image. Depends on the content (e.g., presence and type of public key certificate chain and signature).
`sh_link`	0x28	4B	0 - `=SHN_UNDEF`, no link information
`sh_info`	0x2C	4B	0 - no extra information about the section
`sh_addralign`	0x30	8B	Contains the required alignment of the section. This field must be a power of two.
`sh_entsize`	0x38	8B	0
	0x40		End of section header (size)

ELF Crypto Data Section Content

ELF Crypto Data Section Fields Description

Name	Offset	Range	Description
metadata_version	0x0	15:0	Version metadata structure format. Initial version is 0.
Reserved (`DPA_fw_type`)	0x4	15:8	Reserved
Reserved	0x8	31:0	Reserved
Reserved	0xC	31:0	Reserved. Shall be set to all zeros.
Reserved	0x10	16B	Reserved. Shall be set to all zeros.
Reserved	0x20	4 bytes	Reserved. Shall be set to all zeros.
Reserved	0x24	24B	Reserved. Shall be set to all zeros.
signature_type	0x3c	15:0	Signature Type. Only relevant for signed firmware: • 0, 1 - Reserved • 2 - RSA_ SHA_512 • >3 - Reserved
Hash List Table	0x40	HashTableLength	
Crypto Signature	0x40 + HashTableLength	Signature_Length	Signature_Length depends on the signature_type.
Certificate_Chain	0x40 + HashTableLength + Signature_Length	CrtChain_Length	Structure given the table under section "Certificate Chain Layout".
Padding			FF-padding to align the full size of the data to multiples of DWords (DWs)

The full length of the ELF crypto data section shall be a multiple of DWs (due to firmware legacy implementation). Thus, the MFT (as part of the flow described in figure "Crypto Signing Flow") shall add FF-padding for this structure to align to multiple of DW.

Hash List Table Layout

This table specifies the hash table layout (proposal).

The table contains two parts:

- The 1st part corresponds to the segments of the ELF file, as referenced by the Program Header Table of the EFL file
- The 2nd part corresponds to the sections of the ELF file, as referenced by the Section Header Table

The hash algorithm to be used is SHA-256.

Hash List Table Layout (Proposal)

Name	Offset	Range	Description
Hash Table Magic Pattern	0x0	8 bytes	ASCII "HASHLIST' string: 0x0: 31:24 - "H", 23:16 - "A", 15:8 - "S", 7:0 - "H" 0x4: 31:24 - "L", 23:16 - "I", 15:8 - "S", 7:0 - "T"
Number of Entries - Segments	0x8	7:0	Number of entries in Hashes Segments part, N_Segments.
Reserved	0x8	31:8	Reserved
Number of Entries - Sections	0xc	7:0	Number of entries in Hashes Sections part, N_Sections. Minimum - 0
Reserved	0xc	31:8	Reserved
Reserved	0x10	16 bytes	Reserved
DPA Application ELF Hash	0x20	32 bytes	Hash of the full ELF App file
ELF Header Hash	0x40	32 bytes	Hash of the ELF Header
Program Header Hash	0x60	32 bytes	Hash of the program header
Hash of 1st Segment referenced in the Program Header Table	0x80	32 bytes	Hash of 1st segment referenced in the Program Header Table
Hash of 2nd Segment referenced in the Program Header Table	0xA0	32 bytes	Hash of 2nd Segment referenced in the Program Header Table
……	……	…..	……
Hash of N_Segments (last) Segment referenced in the Program Header Table	0x60 + N_Segments*0x20	32 bytes	Hash of 2nd segment referenced in the Program Header Table
Section Header Table Hash	0x80 + N_Segments*0x20	32 bytes	Hash of the Section Header Table
Hash of 1st Section referenced in the Section Header Table	+ 0x20	32 bytes	Hash of 1st section referenced in the Section Header Table
Hash of 2nd Section referenced in the Section Header Table	+ 0x20	32 bytes	Hash of 2nd section referenced in the Section Header Table
……	……	…..	……
Hash of N_Sections (last) Section referenced in the Section Header Table	+ 0x20	32 bytes	Hash of N_Sections (last) section referenced in the Section Header Table

The 32-bytes hash fields of different sections/segments in the previous table shall follow Big-Endian convention, as illustrated here:

Hash Fields (Big Endian) Bytes Alignment

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	offset
hash[0]								hash[1]								hash[2]								hash[3]								0h
hash[4]								hash[5]								hash[6]								hash[7]								4h
...								...								...								...								...
hash[28]								hash[29]								hash[30]								hash[31]								1Ch

Certificate Chain Layout

The following table specifies the certificate chain layout. The leaf (the last certificate) of the chain is used as the public key for authentication of the DPA code. This structure is aligned with the certificate chain layout as defined in the *Flash Application Note*.

Certificate Chain Layout

Name	Offset	Range	Description
Type	0x0	3:0	Chain type. Shall be set to 1. 3rd party code authentication certificate chain.
Count	0x0	7:4	Number of certificates in this chain
Length	0x0	23:8	Total length of the certificate chain, in bytes, including all fields in this table
Reserved	0x4	31:0	31:0 - Reserved
CRC	0x8	15:0	The CRC of the header, for header integrity check, covering DWs in 0x0, 0x4
Certificates	0xC-0x1000		One or more ASN.1 DER-encoded X509v3 certificates. The ASN.1 DER encoding of each individual certificate can be analyzed to determine its length. The certificates shall be listed in hierarchical order, with the leaf certificate being the last on the list.

14.4.3.5.4 Known Limitations

14.4.3.5.4.1 Supported Devices

- BlueField-3 based DPUs

14.4.3.5.4.2 Supported Host OS

- Windows is not supported

14.4.3.5.4.3 Supported SDKs

- DOCA FlexIO at beta level
- DOCA DPA at beta level

14.4.3.5.4.4 Toolchain

- DPA image-signing and signature-verification are not currently supported
- Debugger (GDB) is currently not supported

14.4.3.5.4.5 FlexIO

- When `flexio_dev_outbox_config_uar_extension` API is called with a `device_id` parameter different than PF/ECPF ID (i.e., move to SF/VF outbox) and the APIs `flexio_dev_yield()` , `flexio_dev_print()` , or `flexio_dev_msg()` are called, then when either of those 3 APIs return, the user cannot work with the SF/VF queues.

14.4.3.6 DOCA DPA

14.4.3.6.1 Introduction

⚠ Supported at beta level.

This chapter provides an overview and configuration instructions for DOCA DPA API.

The DOCA DPA library offers a programming model for offloading communication-centric user code to run on the DPA processor on NVIDIA® BlueField®-3 networking platform. DOCA DPA provides a high-level programming interface to the DPA processor.

DOCA DPA offers:

- Full control over DPA threads -
  - The user can control the thread function (kernel) that runs on DPA and their placement on DPA EUs
  - The user can associate a DPA thread with a DPA Completion Context. When the completion context receives a notification, the DPA thread is scheduled.
- Abstraction to allow a DPA thread to issue asynchronous operations
- Abstraction to execute a blocking one-time call from host application to execute the kernel on the DPA from the host application (RPC)
- Abstraction for memory services
- Abstraction for remote communication primitives (integrated with remote event signaling)
- Full control on execution-ordering and notifications/synchronization of the DPA and host/Target BlueField
- A set of debugging APIs that allow diagnosing and troubleshooting any issue on the device, as well as accessing real-time information from the running application
- C API for application developers

DPACC is used to compile and link kernels with DOCA DPA device libraries to get DPA applications that can be loaded from the host program to execute on the DPA (similar to CUDA usage with NVCC). For more information on DPACC, refer to the NVIDIA DOCA DPACC Compiler.

14.4.3.6.2 Prerequisites

DOCA DPA applications can run either on the host or on the Target BlueField. Running on the host machine requires EU pre-configuration using the `dpaeumgmt` tool. For more information, please refer to NVIDIA DOCA DPA EU Management Tool.

14.4.3.6.3 Library Changes From Previous Releases

14.4.3.6.3.1 Changes in 2.8.0

The following subsection(s) detail the `doca_dpa` library updates in version 2.8.0.

Added Features

- `doca_error_t doca_dpa_device_extend(struct doca_dpa *dpa, struct doca_dev *other_dev, struct doca_dpa **extended_dpa)`
  - Extended DPA with another device/GVMI
- `doca_error_t doca_dpa_get_dpa_handle(struct doca_dpa *dpa, doca_dpa_dev_t *handle)`
- `void doca_dpa_dev_device_set(doca_dpa_dev_t dpa_handle)`

Development Flow

DOCA enables developers to program the DPA processor using both DOCA DPA library and a suite of other tools (mainly DPACC).

The following are the main steps to start DPA offload programming:

1. Write DPA device code, or kernels, (`.c` files) with:
   - The `__dpa_global__` keyword before DPA thread function (see "Examples" section)
   - The `__dpa_rpc__` keyword before RPC function (see "Examples" section)
2. Use DPACC to build a DPA program (i.e., a host library which contains an embedded device executable). Inputs for DPACC are:
   - Kernels from the previous step
   - DOCA DPA device libraries
3. Build host executable using a host compiler. Inputs for the host compiler are:
   - DPA program from the previous step
   - User host application source files
   - DOCA DPA host library

DPACC is provided by the DOCA SDK installation. For more information, please refer to the NVIDIA DOCA DPACC Compiler.

14.4.3.6.4 Software Architecture

14.4.3.6.4.1 Deployment View

DOCA DPA is composed of the following libraries that come with the DOCA SDK installation:

- Host/Target BlueField library and header file (used by user host application)

- `doca_dpa.h`
- `libdoca_dpa.a` / `libdoca_dpa.so`

- Two device libraries and header files
  - `doca_dpa_dev.h`
  - `doca_dpa_dev_rdma.h`
  - `doca_dpa_dev_sync_event.h`
  - `doca_dpa_dev_buf.h`
  - `libdoca_dpa_dev.a` - DOCA DPA device library for common utilities (e.g., log, trace, completion, sync event, etc.)
  - `libdoca_dpa_dev_comm.a` - DOCA DPA device library for communication utilities (e.g., RDMA)

14.4.3.6.4.2 DPA Queries

- Before invoking the DPA API, make sure that DPA is indeed supported on the relevant device. The API which checks whether a device supports DPA is:

```
doca_error_t doca_devinfo_get_is_dpa_supported(const struct doca_devinfo *devinfo)
```

> ⓘ Only if this call returns `DOCA_SUCCESS` can the user invoke DOCA DPA API on the device.

- To use a valid EU ID for the DPA EU Affinity of a DPA thread, use the following APIs to query EU ID and core valid values:

```
doca_error_t doca_dpa_get_core_num(struct doca_dpa *dpa, unsigned int *num_cores)
doca_error_t doca_dpa_get_num_eus_per_core(struct doca_dpa *dpa, unsigned int *eus_per_core)
doca_error_t doca_dpa_get_total_num_eus_available(struct doca_dpa *dpa, unsigned int *total_num_eus)
```

- There is a limitation on the maximum number of DPA threads that can run a single kernel. This can be retrieved by calling the host API:

```
doca_error_t doca_dpa_get_max_threads_per_kernel(struct doca_dpa *dpa, unsigned int *value)
```

- Each kernel launched into the DPA has a maximum runtime limit. This can be retrieved by calling the host API:

```
doca_error_t doca_dpa_get_kernel_max_run_time(struct doca_dpa *dpa, unsigned long long *value)
```

> ⚠ If the kernel execution time on the DPA exceeds this maximum runtime limit, it may be terminated and cause a fatal error. To recover, the application must destroy the DPA context and create a new one.

14.4.3.6.4.3 Overview of DOCA DPA Software Objects

Term	Definition
DPA context	Software construct for the host process that encapsulates the state associated with a DPA process (on a specific device).
DPA Application	Interface with the DPACC compiler to produce a DPA program (app) which is obtained by the DPA context to begin working on DPA.
Kernel	User function (and its arguments) to be executed on DPA. A kernel may be executed by one or more DPA threads.
DPA EU Affinity	An object used to control which EU to use for DPA thread.
DPA Thread	DOCA DPA provides APIs to create/manage DPA thread which runs a given kernel.
DPA Completion Context	An object used to receive/handle a completion notification. The user can associate a DPA thread with a completion context. When the completion context receives a notification, DPA thread is scheduled.
DPA Thread Notification	A mechanism for one DPA thread to notify another DPA thread.
DPA Async Ops	An object used to allow a DPA thread to issue asynchronous operations, like memcpy or post_wait operations.
DPA RPC	A blocking one-time call from host application to execute a kernel on DPA. RPC is mainly used for control path. The RPC's return value is reported back to the host application.
DPA Memory	DOCA DPA provides an API to allocate/manage DPA memory, as well as handling host/Target BlueField memory that has been exported to DPA.
Sync Event	Data structure in either CPU, Target BlueField, GPU, or DPA-heap. An event contains a counter that can be updated and waited on.
RDMA	Abstraction around a network transport object. Allows executing various RDMA operations.
DPA Hash Table	DOCA DPA provides an API to create a Hash Table on DPA. This data structure is managed on DPA using relevant device APIs.
DPA Logger/Tracer	DOCA DPA provides a set of debugging APIs to allow the user to diagnose and troubleshoot any issue on the device, as well as accessing real-time information from the running application.

⚠ The DOCA DPA SDK does not use any means of multi-thread synchronization primitives. All DOCA DPA objects are non-thread-safe. Developers should make sure the user program and kernels are written to avoid race conditions.

14.4.3.6.4.4 Initialization

The DPA context encapsulates the DPA device and a DPA process (program). Within this context, the application creates various DPA SDK objects and controls them. After verifying DPA is supported for the chosen device, the DPA context is created.

Use the following host-side APIs to create/set the DPA context and it is expected to be the first programming step:

- To create/destroy DPA context:

```
doca_error_t doca_dpa_create(struct doca_dev *dev, struct doca_dpa **dpa)
doca_error_t doca_dpa_destroy(struct doca_dpa *dpa)
```

- To start/stop DPA context:

```
doca_error_t doca_dpa_start(struct doca_dpa *dpa)
doca_error_t doca_dpa_stop(struct doca_dpa *dpa)
```

14.4.3.6.4.5 Interface to DPACC

DPA Application

To associate a DPA program (app) with a DPA context, use the following host-side APIs:

```
doca_error_t doca_dpa_set_app(struct doca_dpa *dpa, struct doca_dpa_app *app)
doca_error_t doca_dpa_get_app(struct doca_dpa *dpa, struct doca_dpa_app **app)
doca_error_t doca_dpa_app_get_name(struct doca_dpa_app *app, char *app_name, size_t *app_name_len)
```

The `app` variable name used in `doca_dpa_set_app()` API must be the token passed to DPACC `--app-name` parameter.

Example (Pseudo Code)

For example, when using the following `dpacc` command line:

```
dpacc \
    kernels.c \
    -o dpa_program.a \
    -hostcc=gcc \
    -hostcc-options="..." \
    --devicecc-options="..." \
    -device-libs="-L/opt/mellanox/doca/include -ldoca_dpa_dev -ldoca_dpa_dev_comm" \
    --app-name="dpa_example_app"
```

The user must use the following commands to set the `app` of a DPA context:

```
extern struct doca_dpa_app *dpa_example_app;
doca_dpa_create(&dpa);
doca_dpa_set_app(dpa, dpa_example_app);
doca_dpa_start(dpa);
```

14.4.3.6.4.6 Affinity

The user can control which EU to use for a DPA thread using DPA EU affinity object.

A DPA EU affinity object can be configured for one EU ID at a time.

Use the following host-side APIs to manage it:

- To create/destroy DPA EU affinity object:

```
doca_error_t doca_dpa_eu_affinity_create(struct doca_dpa *dpa, struct doca_dpa_eu_affinity **affinity)
doca_error_t doca_dpa_eu_affinity_destroy(struct doca_dpa_eu_affinity *affinity)
```

- To set/clear EU ID in DPA EU affinity object:

```
doca_error_t doca_dpa_eu_affinity_set(struct doca_dpa_eu_affinity *affinity, unsigned int eu_id)
doca_error_t doca_dpa_eu_affinity_clear(struct doca_dpa_eu_affinity *affinity)
```

- To get EU ID of a DPA EU affinity object:

```
doca_error_t doca_dpa_eu_affinity_get(struct doca_dpa_eu_affinity *affinity, unsigned int *eu_id)
```

14.4.3.6.4.7 Threading

DOCA DPA thread used to run a user function “DPA kernel” on DPA.

User can control on which EU to run DPA kernel by attaching a DPA EU affinity object to the thread.

The thread can be triggered on DPA using two methods:

1. DPA Thread Notification - Notifying one DPA thread from another DPA thread.
2. DPA Completion Context - A completion is arrived at a DPA completion context which is attached to the thread.

DPA Thread

Host-side API

- To create/destroy DPA thread:

```
doca_error_t doca_dpa_thread_create(struct doca_dpa *dpa, struct doca_dpa_thread **dpa_thread)

doca_error_t doca_dpa_thread_destroy(struct doca_dpa_thread *dpa_thread)
```

- To set/get thread user function and it's argument:

```
doca_error_t doca_dpa_thread_set_func_arg(struct doca_dpa_thread *thread, doca_dpa_func_t *func, uint64_t
arg)

doca_error_t doca_dpa_thread_get_func_arg(struct doca_dpa_thread *dpa_thread, doca_dpa_func_t **func,
uint64_t *arg)
```

- To set/get DPA EU Affinity:

```
doca_error_t doca_dpa_thread_set_affinity(struct doca_dpa_thread *thread, struct
doca_dpa_eu_thread_affinity *eu_affinity)

doca_error_t doca_dpa_thread_get_affinity(struct doca_dpa_thread *dpa_thread, const struct
doca_dpa_eu_affinity **affinity)
```

- Thread Local Storage (TLS)
  User can ask to store an opaque for a DPA thread in host side application using the following API:

```
doca_error_t doca_dpa_thread_set_local_storage(struct doca_dpa_thread *dpa_thread, doca_dpa_dev_uintptr_t
dev_ptr)

doca_error_t doca_dpa_thread_get_local_storage(struct doca_dpa_thread *dpa_thread, doca_dpa_dev_uintptr_t
*dev_ptr)
```

  `dev_ptr` is a pre-allocated DPA memory.
  In kernel, user can retrieve the stored opaque using the relevant device API (see below API).
  This opaque is stored/retrieved using the Thread Local Storage (TLS) mechanism.

- To start/stop DPA thread:

```
doca_error_t doca_dpa_thread_start(struct doca_dpa_thread *thread)

doca_error_t doca_dpa_thread_stop(struct doca_dpa_thread *dpa_thread)
```

- To run DPA thread:

```
doca_error_t doca_dpa_thread_run(struct doca_dpa_thread *dpa_thread)
```

This API sets the thread to run state.
This function must be called after DPA thread is:

a. Created, set and started.
b. In case of DPA thread is attached to DPA Completion Context, the completion context must be started before, see below pseudo code example:

Device-side API

Device APIs are used by user-written kernels.

- Thread Restart APIs
  DPA thread can ends its run using one of the following device APIs:
  - Reschedule API:

    ```
    void doca_dpa_dev_thread_reschedule(void)
    ```

    - DPA thread still active.
    - DPA thread resources are back to RTOS.
    - DPA thread can be triggered again.
  - Finish API:

    ```
    void doca_dpa_dev_thread_finish(void)
    ```

    - DPA thread is marked as finished.
    - DPA thread resources are back to RTOS.
    - DPA thread can’t be triggered again.
- To get TLS:

  ```
  doca_dpa_dev_uintptr_t doca_dpa_dev_thread_get_local_storage(void)
  ```

  This function returns DPA thread local storage which was set previously using host API `doca_dpa_thread_set_local_storage()` .

Example (Host-side Pseudo Code)

```
extern doca_dpa_func_t hello_kernel;

// create DPA thread
doca_dpa_thread_create(&dpa_thread);

// set thread kernel
doca_dpa_thread_set_func_arg(dpa_thread, &hello_kernel, func_arg);

// set thread affinity
doca_dpa_eu_affinity_create(&eu_affinity);
doca_dpa_eu_affinity_set(eu_affinity, 10 /* EU ID */);
doca_dpa_thread_set_affinity(dpa_thread, eu_affinity);

// set thread local storage
doca_dpa_mem_alloc(&tls_dev_ptr);
doca_dpa_thread_set_local_storage(dpa_thread, tls_dev_ptr);

// start thread
doca_dpa_thread_start(dpa_thread);

// create and initialize DPA Completion Context
doca_dpa_completion_create(&dpa_comp);
doca_dpa_completion_set_thread(dpa_comp, dpa_thread);
doca_dpa_completion_start(dpa_comp);

// run thread only after both thread is started and the attached completion context is started
doca_dpa_thread_run(dpa_thread);
```

Completion Context

To tie the user application closely with the DPA native model of event-driven scheduling/computation, we introduced DPA Completion Context.

User associates a DPA Thread with a completion context. When the completion context receives a notification, DPA Thread is triggered.

User can choose not to associate it with DPA Thread and to poll it manually.

User has the option to continue receiving new notifications or ignore them.

DOCA DPA provides a generic completion context that can be shared for Message Queues, RDMA, Ethernet and as well as DPA Async Ops.

Host-side API

- To create/destroy DPA Completion Context:

  ```
  doca_error_t doca_dpa_completion_create(struct doca_dpa *dpa, unsigned int queue_size, struct
  doca_dpa_completion **dpa_comp)

  doca_error_t doca_dpa_completion_destroy(struct doca_dpa_completion *dpa_comp)
  ```

- To get queue size:

  ```
  doca_error_t doca_dpa_completion_get_queue_size(struct doca_dpa_completion *dpa_comp, unsigned int *size)
  ```

- To attach to a DPA Thread:

  ```
  doca_error_t doca_dpa_completion_set_thread(struct doca_dpa_completion *dpa_comp, struct doca_dpa_thread
  *thread)

  doca_error_t doca_dpa_completion_get_thread(struct doca_dpa_completion *dpa_comp, struct doca_dpa_thread
  **thread)
  ```

  Attaching to a thread is only required if the user wants triggering of the thread when a completion is arrived at the completion context.

- To start/stop DPA Completion Context:

  ```
  doca_error_t doca_dpa_completion_start(struct doca_dpa_completion *dpa_comp)
  ```

```
doca_error_t doca_dpa_completion_stop(struct doca_dpa_completion *dpa_comp)
```

- To get DPA handle:

  ```
  doca_error_t doca_dpa_completion_get_dpa_handle(struct doca_dpa_completion *dpa_comp,
  doca_dpa_dev_completion_t *handle)
  ```

  Use output parameter `handle` for below device APIs which can be used in thread kernel.

Device-side API

Device APIs are used by user-written kernels.

Kernels get `doca_dpa_dev_completion_t` handle and invoke the following API:

- To get a completion element:

  ```
  int doca_dpa_dev_get_completion(doca_dpa_dev_completion_t dpa_comp_handle,
  doca_dpa_dev_completion_element_t *comp_element)
  ```

  Use the returned `comp_element` to retrieve completion info using below APIs.

- To get completion element type:

  ```
  typedef enum {
      DOCA_DPA_DEV_COMP_SEND = 0x0,                /**< Send completion */
      DOCA_DPA_DEV_COMP_RECV_RDMA_WRITE_IMM = 0x1, /**< Receive RDMA Write with Immediate completion */
      DOCA_DPA_DEV_COMP_RECV_SEND = 0x2,           /**< Receive Send completion */
      DOCA_DPA_DEV_COMP_RECV_SEND_IMM = 0x3,       /**< Receive Send with Immediate completion */
      DOCA_DPA_DEV_COMP_SEND_ERR = 0xD,            /**< Send Error completion */
      DOCA_DPA_DEV_COMP_RECV_ERR = 0xE             /**< Receive Error completion */
  } doca_dpa_dev_completion_type_t;

  doca_dpa_dev_completion_type_t doca_dpa_dev_get_completion_type(doca_dpa_dev_completion_element_t
  comp_element)
  ```

- To get completion element user data:

  ```
  uint32_t doca_dpa_dev_get_completion_user_data(doca_dpa_dev_completion_element_t comp_element)
  ```

  This API returns user data which was set previously in either host APIs:

  a. doca_dpa_async_ops_create(..., `user_data`, ...)
     When DPA Completion Context is attached to DPA Async Ops.
  b. doca_ctx_set_user_data(..., `user_data`)
     When DPA Completion Context is attached to DOCA context, such as DOCA RDMA context.

- To get completion element immediate data:

  ```
  uint32_t doca_dpa_dev_get_completion_immediate(doca_dpa_dev_completion_element_t comp_element)
  ```

  This API returns immediate data for a completion element of type:

  a. DOCA_DPA_DEV_COMP_RECV_RDMA_WRITE_IMM
  b. DOCA_DPA_DEV_COMP_RECV_SEND_IMM

- Acknowledge that the completions have been read on DPA Completion Context:

  ```
  void doca_dpa_dev_completion_ack(doca_dpa_dev_completion_t dpa_comp_handle, uint64_t num_comp)
  ```

This API releases resources of the acked completion elements in completion context.
This acknowledgment enables receiving new `num_comp` completions.

- To request notification on DPA Completion Context:

```
void doca_dpa_dev_completion_request_notification(doca_dpa_dev_completion_t dpa_comp_handle)
```

This API enables requesting new notifications on DPA Completion Context.
Without calling this function, DPA Completion Context is not being notified on new arrived completion elements, hence new completions are not populated in DPA Completion Context.

Example (Device-side Pseudo Code)

```
__dpa_global__ void hello_kernel(uint64_t arg)
{
    // User is expected to pass in some way the attached completion context handle "dpa_comp_handle" to kernel such
as func_arg or a shared memory.
    DOCA_DPA_DEV_LOG_INFO("Hello from kernel\n");

    doca_dpa_dev_completion_element_t comp_element;
    found = doca_dpa_dev_get_completion(dpa_comp_handle, &comp_element);
    if (found) {
        comp_type = doca_dpa_dev_get_completion_type(comp_element);
        // process the completion according to completion type...

        // ack on 1 completion
        doca_dpa_dev_completion_ack(dpa_comp_handle, 1);

        // enable getting more completions and triggering the thread
        doca_dpa_dev_completion_request_notification(dpa_comp_handle);
  }

    // reschedule thread
    doca_dpa_dev_thread_reschedule();
}
```

Thread Notification

Thread Activation is a mechanism for one DPA thread to trigger another DPA Thread.

Thread activation is done without receiving a completion on the attached thread. Therefore it is expected that user of this method of thread activation passes the message in another fashion – such as shared memory.

Thread Activation can be achieved using DPA Notification Completion object.

Host-side API

- To create/destroy DPA Notification Completion:

```
doca_error_t doca_dpa_notification_completion_create(struct doca_dpa *dpa, struct doca_dpa_thread
*dpa_thread, struct doca_dpa_notification_completion **notify_comp)

doca_error_t doca_dpa_notification_completion_destroy(struct doca_dpa_notification_completion *notify_comp)
```

Attaching DPA Notification Completion to a DPA Thread is done using the given parameter `dpa_thread`.

- To get attached DPA Thread:

```
doca_error_t doca_dpa_notification_completion_get_thread(struct doca_dpa_notification_completion
*notify_comp, struct doca_dpa_thread **dpa_thread)
```

- To start/stop DPA Notification Completion:

```
doca_error_t doca_dpa_notification_completion_start(struct doca_dpa_notification_completion *notify_comp)
doca_error_t doca_dpa_notification_completion_stop(struct doca_dpa_notification_completion *notify_comp)
```

- To get DPA handle:

  ```
  doca_error_t doca_dpa_notify_completion_get_dpa_handle(struct doca_dpa_notification_completion
  *notify_comp, doca_dpa_dev_notification_completion_t *comp_handle)
  ```

  Use output parameter `comp_handle` for below device API which can be used in thread kernel.

Device-side API

Device API is used by user-written kernels.

Kernels get `doca_dpa_dev_notification_completion_t` handle and invoke the following API:

```
void doca_dpa_dev_thread_notify(doca_dpa_dev_notification_completion_t comp_handle)
```

Calling this API triggers the attached DPA Thread (the one that is specified in `dpa_thread` parameter in host-side API `doca_dpa_notification_completion_create()`).

Example (Pseudo Code)

- Host-side

  ```
  extern doca_dpa_func_t hello_kernel;

  // create DPA thread
  doca_dpa_thread_create(&dpa_thread);

  // set thread kernel
  doca_dpa_thread_set_func_arg(dpa_thread, &hello_kernel, func_arg);

  // start thread
  doca_dpa_thread_start(dpa_thread);

  // create and start DPA notification completion
  doca_dpa_notification_completion_create(dpa, dpa_thread, &notify_comp);

  doca_dpa_notification_completion_start(notify_comp);

  // get its DPA handle
  doca_dpa_notification_completion_get_dpa_handle(notify_comp, &notify_comp_handle);

  // run thread only after both thread is started and attached notification completion is started
  doca_dpa_thread_run(dpa_thread);
  ```

- Device-side
  Whenever some DPA Thread calls:

  ```
  doca_dpa_dev_thread_notify(notify_comp_handle);
  ```

  This call triggers `dpa_thread` .

Asynchronous Ops

DPA Async Ops allows DPA Thread to issue asynchronous operations, like memcpy or post_wait.

This feature requires the user to create an “asynchronous ops” context and attach to a completion context.

User is expected to adhere to `queue_size` limit on the device when posting operations.

The completion context can raise activation if it is attached to a DPA Thread.

User can also choose to progress the completion context via polling it manually.

User can provide DPA Async Ops `user_data`, and retrieve this metadata in device using relevant device API.

Host-side API

- To create/destroy DPA Async Ops:

  ```
  doca_error_t doca_dpa_async_ops_create(struct doca_dpa *dpa, unsigned int queue_size, uint64_t user_data,
  struct doca_dpa_async_ops **async_ops)

  doca_error_t doca_dpa_async_ops_destroy(struct doca_dpa_async_ops *async_ops)
  ```

  Please use the following define for valid user_data values:

  ```
  #define DOCA_DPA_COMPLETION_LOG_MAX_USER_DATA   (24)
  ```

- To get queue size/user_data:

  ```
  doca_error_t doca_dpa_async_ops_get_queue_size(struct doca_dpa_async_ops *async_ops, unsigned int
   *queue_size)

  doca_error_t doca_dpa_async_ops_get_user_data(struct doca_dpa_async_ops *async_ops, uint64_t *user_data)
  ```

- To attach to a DPA Completion Context:

  ```
  doca_error_t doca_dpa_async_ops_attach(struct doca_dpa_async_ops *async_ops, struct doca_dpa_completion
  *dpa_comp)
  ```

- To start/stop DPA Async Ops:

  ```
  doca_error_t doca_dpa_async_ops_start(struct doca_dpa_async_ops *async_ops)

  doca_error_t doca_dpa_async_ops_stop(struct doca_dpa_async_ops *async_ops)
  ```

- To get DPA handle:

  ```
  doca_error_t doca_dpa_async_ops_get_dpa_handle(struct doca_dpa_async_ops *async_ops,
  doca_dpa_dev_async_ops_t *handle)
  ```

  Use output parameter `handle` for below device API which can be used in thread kernel.

Device-side API

Device APIs are used by user-written kernels.

Kernels get `doca_dpa_dev_async_ops_t` handle and invoke the following API:

- To post memcpy operation using `doca_buf`:

  ```
  void doca_dpa_dev_post_buf_memcpy(doca_dpa_dev_async_ops_t async_ops_handle, doca_dpa_dev_buf_t
  dst_buf_handle, doca_dpa_dev_buf_t src_buf_handle, bool completion_requested)
  ```

  This API copies data between two DOCA buffers.
  The destination buffer, specified by `dst_buf_handle` will contain the copied data after memory copy is complete.
  This is a non-blocking routine.
  Use `completion_requested` to raise a completion when copy data operation is done (any value greater than 0).

If `completion_requested` was set and the attached DPA Completion Context is attached to a DPA Thread, then the thread is triggered once the memcpy operation is done.

- To post memcpy operation using `doca_mmap` and an explicit addresses:

  ```
  void doca_dpa_dev_post_memcpy(doca_dpa_dev_async_ops_t async_ops_handle,
                                doca_dpa_dev_mmap_t dst_mmap_handle,
                                uint64_t dst_addr,
                                doca_dpa_dev_mmap_t src_mmap_handle,
                                uint64_t src_addr,
                                size_t length,
                                uint32_t completion_requested)
  ```

  This API copies data between two DOCA Mmaps.
  The destination DOCA Mmap, specified by `dst_mmap_handle`, `dst_addr` will contain the copied data in source DOCA Mmap specified by `src_mmap_handle`, `src_addr` and `length` after memory copy is complete.
  This is a non-blocking routine.
  Use `completion_requested` to raise a completion when copy data operation is done (any value greater than 0).
  If `completion_requested` was set and the attached DPA Completion Context is attached to a DPA thread, then the thread is triggered once the memcpy operation is done.

  > Use this API for memcpy instead of using doca_buf memcpy API to gain better performance.

- To post wait greater operation on a DOCA Sync Event:

  ```
  void doca_dpa_dev_sync_event_post_wait_gt(doca_dpa_dev_async_ops_t async_ops_handle,
  doca_dpa_dev_sync_event_t wait_se_handle, uint64_t value)
  ```

  This function posts a wait operation on the DOCA Sync Event using DPA Async Ops to obtain a DPA Thread activation.
  Attached thread is activated when value of DOCA Sync Event is greater than a given value.
  This is a non-blocking routine.

  > Valid values must be in the range [0, 254] and can be called for event with value in the range [0, 254]. Invalid values leads to undefined behavior.

- To post wait not equal operation on a DOCA Sync Event:

  ```
  void doca_dpa_dev_sync_event_post_wait_ne(doca_dpa_dev_async_ops_t async_ops_handle,
  doca_dpa_dev_sync_event_t wait_se_handle, uint64_t value)
  ```

  This function posts a wait operation on the DOCA Sync Event using the DPA Async Ops to obtain a DPA Thread activation.
  Attached thread is activated when value of DOCA Sync Event is not equal to a given value.
  This is a non-blocking routine.

Example (Host-side Pseudo Code)

```
doca_dpa_thread_create(&dpa_thread);
doca_dpa_thread_set_func_arg(dpa_thread);
doca_dpa_thread_start(dpa_thread);
```

```
doca_dpa_completion_create(&dpa_comp);
doca_dpa_completion_set_thread(dpa_comp, dpa_thread);
doca_dpa_completion_start(dpa_comp);

doca_dpa_thread_run(dpa_thread);

doca_dpa_async_ops_create(&async_ops);
doca_dpa_async_ops_attach(async_ops, dpa_comp);
doca_dpa_async_ops_start(async_ops);

doca_dpa_async_ops_get_dpa_handle(async_ops, &handle); // use this handle in relevant Async Ops device APIs
```

Thread Group

Thread group is used to aggregate individual DPA threads to a single group.

Please see below host-side APIs for creating/managing thread group.

- To create/destroy DPA Thread Group:

  ```
  doca_error_t doca_dpa_thread_group_create(struct doca_dpa *dpa, unsigned int num_threads, struct
  doca_dpa_tg **tg)

  doca_error_t doca_dpa_thread_group_destroy(struct doca_dpa_tg *tg)
  ```

- To get number of threads:

  ```
  doca_error_t doca_dpa_thread_group_get_num_threads(struct doca_dpa_tg *tg, unsigned int *num_threads);
  ```

- To set DPA Thread at 'rank' in DPA Thread Group:

  ```
  doca_error_t doca_dpa_thread_group_set_thread(struct doca_dpa_tg *tg, struct doca_dpa_thread *thread,
  unsigned int rank)
  ```

  Thread rank is an index of the thread (between 0 and (num_threads - 1)) within the group.

- To start/stop DPA Thread Group:

  ```
  doca_error_t doca_dpa_thread_group_start(struct doca_dpa_tg *tg)

  doca_error_t doca_dpa_thread_group_stop(struct doca_dpa_tg *tg)
  ```

14.4.3.6.4.8 Memory Subsystem

The user can allocate (from the host API) and access (from both the host and device API) several memory locations using the relevant DOCA DPA API.

DOCA DPA supports access from the host/Target BlueField to DPA heap memory and also enables device access to host memory (e.g., kernel writes to host heap).

The normal memory usage flow would be to:

1. Allocate memory (Host/Target BlueField/DPA).
2. Register the memory.
3. Get a DPA handle for the registered memory so it can be accessed by DPA kernels.
4. Access/use the memory from the kernel (see relevant device-side APIs).

Host-side API

- To allocate DPA heap memory:

```
doca_dpa_mem_alloc(doca_dpa_t dpa, size_t size, doca_dpa_dev_uintptr_t *dev_ptr)
```

- To free previously allocated DPA memory:

```
doca_dpa_mem_free(doca_dpa_dev_uintptr_t dev_ptr)
```

- To copy previously allocated memory from a host pointer to a DPA heap device pointer:

```
doca_dpa_h2d_memcpy(doca_dpa_t dpa, doca_dpa_dev_uintptr_t src_ptr, void *dst_ptr, size_t size)
```

- To copy previously allocated memory from a DOCA Buffer to a DPA heap device pointer:

```
doca_error_t doca_dpa_h2d_buf_memcpy(struct doca_dpa *dpa, doca_dpa_dev_uintptr_t dst_ptr, struct doca_buf
*buf, size_t size)
```

- To copy previously allocated memory from a DPA heap device pointer to a host pointer:

```
doca_dpa_d2h_memcpy(doca_dpa_t dpa, void *dst_ptr, doca_dpa_dev_uintptr_t src_ptr, size_t size)
```

- To copy previously allocated memory from a DPA heap device pointer to a DOCA Buffer:

```
doca_error_t doca_dpa_d2h_buf_memcpy(struct doca_dpa *dpa, struct doca_buf *buf, doca_dpa_dev_uintptr_t
src_ptr, size_t size)
```

- To set memory:

```
doca_dpa_memset(doca_dpa_t dpa, doca_dpa_dev_uintptr_t dev_ptr, int value, size_t size)
```

- To get a DPA handle to use in kernels, the user must use a DOCA Core Memory Inventory Object in the following manner (refer to "DOCA Memory Subsystem"):
    - When the user wants to use device APIs with DOCA Buffer, use the following pseudo code:

      ```
      doca_buf_arr_create(&buf_arr);
      doca_buf_arr_set_target_dpa(buf_arr, doca_dpa);
      doca_buf_arr_start(buf_arr);
      doca_buf_arr_get_dpa_handle(buf_arr, &handle);
      ```

      Use output parameter `handle` in relevant device APIs in thread kernel.
    - When the user wants to use device APIs with DOCA Mmap, use the following pseudo code:

      ```
      doca_mmap_create(&mmap);
      doca_mmap_set_dpa_memrange(mmap, doca_dpa, dev_ptr, dev_ptr_len); // dev-ptr is a pre-allocated DPA
      memory
      doca_mmap_start(mmap);
      doca_mmap_dev_get_dpa_handle(mmap, doca_dev, &handle);
      ```

      Use output parameter `handle` in relevant device APIs in thread kernel.

Device-side API

Device APIs are used by user-written kernels.

Memory APIs supplied by the DOCA DPA SDK are all asynchronous (i.e., non-blocking).

The user can acquire either:

1. Pre-configured DOCA Buffers (previously configured with `doca_buf_arr_set_params`).

2. Non-configured DOCA Buffers and use below device setters to configure them.

Device-side API operations:

- To obtain a single buffer handle from the buf array handle:

  ```
  doca_dpa_dev_buf_t doca_dpa_dev_buf_array_get_buf(doca_dpa_dev_buf_arr_t buf_arr, const uint64_t buf_idx)
  ```

- To set/get the address pointed to by the buffer handle:

  ```
  void doca_dpa_dev_buf_set_addr(doca_dpa_dev_buf_t buf, uintptr_t addr)
  uintptr_t doca_dpa_dev_buf_get_addr(doca_dpa_dev_buf_t buf)
  ```

- To set/get the length of the buffer:

  ```
  void doca_dpa_dev_buf_set_len(doca_dpa_dev_buf_t buf, size_t len)
  uint64_t doca_dpa_dev_buf_get_len(doca_dpa_dev_buf_t buf)
  ```

- To set/get the DOCA Mmap associated with the buffer:

  ```
  void doca_dpa_dev_buf_set_mmap(doca_dpa_dev_buf_t buf, doca_dpa_dev_mmap_t mmap)
  doca_dpa_dev_mmap_t doca_dpa_dev_buf_get_mmap(doca_dpa_dev_buf_t buf)
  ```

- To get a pointer to external memory registered on the host using DOCA Buffer:

  ```
  doca_dpa_dev_uintptr_t doca_dpa_dev_buf_get_external_ptr(doca_dpa_dev_buf_t buf)
  ```

- To get a pointer to external memory registered on the host using an explicit address and DOCA Mmap:

  ```
  doca_dpa_dev_uintptr_t doca_dpa_dev_mmap_get_external_ptr(doca_dpa_dev_mmap_t mmap_handle, uint64_t addr)
  ```

14.4.3.6.4.9 Sync Events

Sync events fulfill the following roles:

- DOCA DPA execution model is asynchronous and sync events are used to control various threads running in the system (allowing order and dependency)
- DOCA DPA supports remote sync events, so the programmer is capable of invoking remote nodes by means of DOCA sync events

Host-side API

Please refer to "DOCA Sync Event".

Device-side API

- To get the current event value:

  ```
  doca_dpa_dev_sync_event_get(doca_dpa_dev_sync_event_t event, uint64_t *value)
  ```

- To add/set to the current event value:

  ```
  doca_dpa_dev_sync_event_update_<add|set>(doca_dpa_dev_sync_event_t event, uint64_t value)
  ```

- To wait until event is greater than threshold:

  ```
  doca_dpa_dev_sync_event_wait_gt(doca_dpa_dev_sync_event_t event, uint64_t value, uint64_t mask)
  ```

  Use mask to apply (bitwise AND) on the DOCA sync event value for comparison with the wait threshold.

14.4.3.6.4.10 Communication Model

DOCA DPA communication primitives allow sending data from one node to another.

The object used for the communication between nodes is called an RDMA DPA handle. RDMA DPA handles can be used by kernels only.

RDMAs represent a unidirectional communication pipe between two nodes.

RDMA DPA handles are created when setting a DOCA RDMA context to DPA data path. For more information, please refer to DOCA RDMA.

To track the completion of all communications, the user can attach DOCA RDMA context to a DPA Completion Context.

DPA Completion Context can be associated with a DPA Thread. When the completion context receives a completion on a communication operation, DPA Thread is triggered.

> The user can choose not to associate it with a DPA Thread and to poll it manually.

Host-side API

- To create DOCA RDMA context on DPA, the user must use the following API for the DOCA RDMA context:

  ```
  doca_error_t doca_ctx_set_datapath_on_dpa(struct doca_ctx *ctx, struct doca_dpa *dpa)
  ```

- To attach a DOCA RDMA context to a DPA Completion Context:

  ```
  doca_error_t doca_rdma_dpa_completion_attach(struct doca_rdma *rdma, struct doca_dpa_completion *dpa_comp)
  ```

- To obtain a DPA RDMA handle:

  ```
  doca_error_t doca_rdma_get_dpa_handle(struct doca_rdma *rdma, doca_dpa_dev_rdma_t *dpa_rdma)
  ```

  Use output parameter `handle` in relevant device APIs in the thread kernel.

  > DPA RDMAs are not thread safe and, therefore, must not be used from different kernels/threads concurrently.

Device-side API

DOCA DPA offers two work models for each device RDMA operation:

- An API for RDMA operation using DOCA Buffer
- An API for RDMA operation using DOCA Mmap and an explicit memory address

The user may choose to also raise a completion when the operation is done.

- To read to a local buffer from the remote side buffer:

```
void doca_dpa_dev_rdma_post_read(doca_dpa_dev_rdma_t rdma,
                                 doca_dpa_dev_mmap_t dst_mmap_handle,
                                 uint64_t dst_addr,
                                 doca_dpa_dev_mmap_t src_mmap_handle,
                                 uint64_t src_addr,
                                 size_t length,
                                 uint32_t completion_requested)
void doca_dpa_dev_rdma_post_buf_read(doca_dpa_dev_rdma_t rdma,
                                     doca_dpa_dev_buf_t dst_buf_handle,
                                     doca_dpa_dev_buf_t src_buf_handle,
                                     uint32_t completion_requested)
```

- To write local memory to the remote side buffer:

```
void doca_dpa_dev_rdma_post_write(doca_dpa_dev_rdma_t rdma,
                                  doca_dpa_dev_mmap_t dst_mmap_handle,
                                  uint64_t dst_addr,
                                  doca_dpa_dev_mmap_t src_mmap_handle,
                                  uint64_t src_addr,
                                  size_t length,
                                  uint32_t completion_requested)
void doca_dpa_dev_rdma_post_buf_write(doca_dpa_dev_rdma_t rdma,
                                      doca_dpa_dev_buf_t dst_buf_handle,
                                      doca_dpa_dev_buf_t src_buf_handle,
                                      uint32_t completion_requested)
```

- To write local memory to the remote side buffer with an immediate data which can be retrieved when receiving a completion on this operation:

```
void doca_dpa_dev_rdma_post_write_imm(doca_dpa_dev_rdma_t rdma,
                                      doca_dpa_dev_mmap_t dst_mmap_handle,
                                      uint64_t dst_addr,
                                      doca_dpa_dev_mmap_t src_mmap_handle,
                                      uint64_t src_addr,
                                      size_t length,
                                      uint32_t immediate,
                                      uint32_t completion_requested)
void doca_dpa_dev_rdma_post_buf_write_imm(doca_dpa_dev_rdma_t rdma,
                                          doca_dpa_dev_buf_t dst_buf_handle,
                                          doca_dpa_dev_buf_t src_buf_handle,
                                          uint32_t immediate,
                                          uint32_t completion_requested)
// use the following API to retrieve immediate data on completion
uint32_t doca_dpa_dev_get_completion_immediate(doca_dpa_dev_completion_element_t comp_element)
```

- To send local memory:

```
void doca_dpa_dev_rdma_post_send(doca_dpa_dev_rdma_t rdma,
                                 doca_dpa_dev_mmap_t mmap_handle,
                                 uint64_t addr,
                                 size_t length,
                                 uint32_t completion_requested)
void doca_dpa_dev_rdma_post_buf_send(doca_dpa_dev_rdma_t rdma,
                                     doca_dpa_dev_buf_t send_buf_handle,
                                     uint32_t completion_requested)
```

- To send local memory with an immediate data which can be retrieved when receiving a completion on this operation:

```
void doca_dpa_dev_rdma_post_send_imm(doca_dpa_dev_rdma_t rdma,
                                     doca_dpa_dev_mmap_t mmap_handle,
                                     uint64_t addr,
                                     size_t length,
                                     uint32_t immediate,
                                     uint32_t completion_requested)
void doca_dpa_dev_rdma_post_buf_send_imm(doca_dpa_dev_rdma_t rdma,
                                         doca_dpa_dev_buf_t send_buf_handle,
                                         uint32_t immediate,
                                         uint32_t completion_requested)
// use the following API to retrieve immediate data on completion
uint32_t doca_dpa_dev_get_completion_immediate(doca_dpa_dev_completion_element_t comp_element)
```

- To handle posting RDMA receive operation, use the following APIs:
    - To post RDMA receive operation:

```
void doca_dpa_dev_rdma_post_receive(doca_dpa_dev_rdma_t rdma, doca_dpa_dev_mmap_t mmap_handle,
uint64_t addr, size_t length)
void doca_dpa_dev_rdma_post_buf_receive(doca_dpa_dev_rdma_t rdma, doca_dpa_dev_buf_t
receive_buf_handle)
```

  - Acknowledge that post receive operations are done (data has been received on associated data buffers). This acknowledgment enables DPA RDMA to repost `#num_acked` new post receive operations.

```
void doca_dpa_dev_rdma_receive_ack(doca_dpa_dev_rdma_t rdma, uint32_t num_acked)
```

- To perform an atomic add operation on the remote side buffer:

```
void doca_dpa_dev_rdma_post_atomic_fetch_add(doca_dpa_dev_rdma_t rdma,
                                             doca_dpa_dev_mmap_t dst_mmap_handle,
                                             uint64_t dst_addr,
                                             uint64_t value,
                                             uint32_t completion_requested)
void doca_dpa_dev_rdma_post_buf_atomic_fetch_add(doca_dpa_dev_rdma_t rdma,
                                                 doca_dpa_dev_buf_t dst_buf_handle,
                                                 uint64_t value,
                                                 uint32_t completion_requested)
```

- To signal a remote event:

```
doca_dpa_dev_rdma_signal_<add|set>(doca_dpa_dev_rdma_t rdma, doca_dpa_dev_sync_event_remote_t
remote_sync_event, uint64_t count)
```

> The following API is only relevant for a kernel used in `kernel_launch` APIs. This API is not relevant for DPA RDMA which is attached to a DPA Completion Context.

As all DPA RDMA operations are non-blocking, the following API is provided to kernel launch developers to wait until all previous RDMA operations are done (blocking call) to drain the RDMA DPA handle:

```
doca_dpa_dev_rdma_synchronize(doca_dpa_dev_rdma_t rdma)
```

When this call returns, all previous non-blocking operations on the DPA RDMA have completed (i.e., sent to the remote RDMA). It is expected that the `doca_dpa_dev_rdma_synchronize()` call would use the same thread as the handle calls.
Since DPA RDMAs are non-thread safe, each DPA RDMA must be accessed by a single thread at any given time. If user launches a kernel that should be executed by more than one thread and this kernel includes RDMA communication, it is expected that a user will use array of RDMAs so that each RDMA will be accessed by single thread (each thread can access it's RDMA instance by using `doca_dpa_dev_thread_rank()` as its index in the array of RDMA handles).
When using the Remote Event Exchange API, `void doca_dpa_dev_rdma_signal_<add|set>(..., doca_dpa_dev_event_remote_t event_handle, ...)` , within your kernel, note that `event` is a remote event. That is, an event created on the remote node and exported to a remote node (`doca_dpa_event_dev_remote_export(event_handle)`).

Multiple RDMA Contexts

To support attaching multiple DOCA RDMA contexts to a single DPA Completion Context, DOCA offers the following APIs.

- RDMA `user_data` which is set using the host API:

  ```
  doca_error_t doca_ctx_set_user_data(struct doca_ctx *ctx, union doca_data user_data)
  ```

  And can be retrieved in device using the completion API:

  ```
  uint32_t doca_dpa_dev_get_completion_user_data(doca_dpa_dev_completion_element_t comp_element)
  ```

  `user_data` should be used to distinguish which DOCA RDMA context has triggered this completion.
- RDMA `work request index` using device API for an RDMA completion:

  ```
  uint32_t doca_dpa_dev_rdma_completion_get_wr_index(doca_dpa_dev_completion_element_t comp_element)
  ```

  `work request index` should be used to get operation index of DOCA RDMA context which triggered this completion.

Example (Host-side Pseudo Code)

```
// create and start DPA Thread
doca_dpa_thread_create(&dpa_thread);
doca_dpa_thread_set_func_arg(dpa_thread);
doca_dpa_thread_start(dpa_thread);

// create and start DPA Completion Context which is attached to DPA Thread
doca_dpa_completion_create(&dpa_comp);
doca_dpa_completion_set_thread(dpa_comp, dpa_thread);
doca_dpa_completion_start(dpa_comp);

doca_dpa_thread_run(dpa_thread);

// create and start DPA RDMA context which is attached to DPA Completion Context
doca_rdma_create(doca_dev, &rdma);

doca_rdma_dpa_completion_attach(rdma, dpa_comp);

doca_rdma_ctx = doca_rdma_as_ctx(rdma);
doca_ctx_set_datapath_on_dpa(doca_rdma_ctx, doca_dpa);

doca_ctx_set_user_data(doca_rdma_ctx, user_data);

doca_ctx_start(doca_rdma_ctx);

doca_rdma_get_dpa_handle(rdma, &handle);
```

ⓘ Each completion on an RDMA operation triggers `dpa_thread`.

ⓘ Use output parameter `handle` in relevant RDMA device APIs in the thread kernel.

14.4.3.6.4.11 Data Structures

Hash Table

DOCA DPA provides an API to create a hash table on DPA. This data structure is managed on DPA using relevant device APIs.

Host-side API

- To create a hash table on DPA:

```
doca_error_t doca_dpa_hash_table_create(struct doca_dpa *dpa, unsigned int num_entries, struct
doca_dpa_hash_table **ht)
```

- To destroy a hash table:

```
doca_error_t doca_dpa_hash_table_destroy(struct doca_dpa_hash_table *ht)
```

- To obtain a DPA handle:

```
doca_error_t doca_dpa_hash_table_get_dpa_handle(struct doca_dpa_hash_table *ht, doca_dpa_dev_hash_table_t
*handle)
```

Use output parameter `handle` in relevant device APIs in the thread kernel.

Device-side API

- To add a new entry to the hash table:

```
void doca_dpa_dev_hash_table_add(doca_dpa_dev_hash_table_t ht_handle, uint32_t key, uint64_t value)
```

> Adding a new key when the hash table is full causes anomalous behavior.

- To remove an entry from the hash table:

```
void doca_dpa_dev_hash_table_remove(doca_dpa_dev_hash_table_t ht_handle, uint32_t key)
```

- To return the value to which the specified key is mapped in the hash table:

```
int doca_dpa_dev_hash_table_find(doca_dpa_dev_hash_table_t ht_handle, uint32_t key, uint64_t *value)
```

14.4.3.6.4.12 RPC and Kernel Launch

RPC

Host-side API

A blocking one-time call from the host application to execute a kernel on DPA.

> RPC is mainly used for control path.

The RPC's return value is reported back to the host application.

```
doca_error_t doca_dpa_rpc(struct doca_dpa *dpa, doca_dpa_func_t *func, uint64_t *retval, … /* func arguments */)
```

Example

- Device-side - DPA device `func` must be annotated with `__dpa_rpc__` annotation, such as:

```
__dpa_rpc__ uint64_t hello_rpc(int arg)
{
    ...
}
```

- Host-side:

```
extern doca_dpa_func_t hello_rpc;

uint64_t retval;
doca_dpa_rpc(dpa, &hello_rpc, &retval, 10);
```

Kernel Launch

DOCA DPA provides an API which enables full control for launching and monitoring kernels.

Since DOCA DPA libraries are not thread-safe, it is up to the programmer to make sure the kernel is written to allow it to run in a multi-threaded environment. For example, to program a kernel that uses RDMAs with 16 concurrent threads, the user should pass an array of 16 RDMAs to the kernel so that each thread can access its RDMA using its rank (`doca_dpa_dev_thread_rank()`) as an index to the array.

Host-side API

```
doca_dpa_kernel_launch_update_<add|set>(struct doca_dpa *dpa, struct doca_sync_event *wait_event, uint64_t
wait_threshold, struct doca_sync_event *comp_event, uint64_t comp_count, unsigned int num_threads, doca_dpa_func_t
*func, ... /* args */)
```

- This function asks DOCA DPA to run `func` in DPA by `num_threads` and give it the supplied list of arguments (variadic list of arguments).
- This function is asynchronous so when it returns, it does not mean that `func` started/ended its execution.
- To add control or flow/ordering to these asynchronous kernels, two optional parameters for launching kernels are available:
  - `wait_event` - the kernel does not start its execution until the event is signaled (if NULL, the kernel starts once DOCA DPA has an available EU to run on it) which means that DOCA DPA would not run the kernel until the event's counter is bigger than `wait_threshold`.

    > Please note that the valid values for `wait_threshold` and `wait_event` counter and are [0-254]. Values out of this range might cause anomalous behavior.

  - `comp_event` - once the last thread running the kernel is done, DOCA DPA updates this event (either sets or adds to its current counter value with `comp_count`).
- DOCA DPA takes care of packing (on host/Target BlueField) and unpacking (in DPA) the kernel parameters.
- `func` must be prefixed with the `__dpa_global__` macro for DPACC to compile it as a kernel (and add it to DPA executable binary) and not as part of host application binary.
- The programmer must declare `func` in their application also by adding the line `extern doca_dpa_func_t func`.

Device-side API

> The following APIs are only relevant for a kernel used in `kernel_launch` APIs. These APIs are not relevant in `doca_dpa_thread` kernel.

- To retrieve the running thread's rank for a given kernel on the DPA. If, for example, a kernel is launched to run with 16 threads, each thread running this kernel is assigned a rank ranging from 0 to 15 within this kernel. This is helpful for making sure each thread in the kernel only accesses data relevant for its execution to avoid data-races:

```
unsigned int doca_dpa_dev_thread_rank()
```

- To return the number of threads running current kernel:

```
unsigned int doca_dpa_dev_num_threads()
```

- To yield the thread which runs the kernel:

```
void doca_dpa_dev_yield(void)
```

Examples

Linear Execution Example

Kernel Code

```
#include "doca_dpa_dev.h"
#include "doca_dpa_dev_sync_event.h"

__dpa_global__ void
linear_kernel(doca_dpa_dev_sync_event_t wait_ev, doca_dpa_dev_sync_event_t comp_ev)
{
    if (wait_ev)
        doca_dpa_dev_sync_event_wait_gt(wait_ev, wait_th = 0);

    doca_dpa_dev_sync_event_update_add(comp_ev, comp_count = 1);
}
```

Host Application Pseudo Code

```
#include <doca_dev.h>
#include <doca_error.h>
#include <doca_sync_event.h>
#include <doca_dpa.h>

int main(int argc, char **argv)
{

    /*
        A
        |
        B
        |
        C
    */

    /* Open DOCA device */
    open_doca_dev(&doca_dev);
    /* Create doca dpa conext */
    doca_dpa_create(doca_dev, dpa_linear_app, &dpa_ctx, 0);

    /* Create event A - subscriber is DPA and publisher is CPU */
    doca_sync_event_create(&ev_a);
    doca_sync_event_add_publisher_location_cpu(ev_a, doca_dev);
    doca_sync_event_add_subscriber_location_dpa(ev_a, dpa_ctx);
    doca_sync_event_start(ev_a);

    /* Create event B - subscriber and publisher are DPA */
    doca_sync_event_create(&ev_b);
    doca_sync_event_add_publisher_location_dpa(ev_b, dpa_ctx);
    doca_sync_event_add_subscriber_location_dpa(ev_b, dpa_ctx);
    doca_sync_event_start(ev_b);
    /* Create event C - subscriber and publisher are DPA */
    doca_sync_event_create(&ev_c);
    doca_sync_event_add_publisher_location_dpa(ev_c, dpa_ctx);
    doca_sync_event_add_subscriber_location_dpa(ev_c, dpa_ctx);
    doca_sync_event_start(ev_c);

    /* Create completion event for last kernel - subscriber is CPU and publisher is DPA */
    doca_sync_event_create(&comp_ev);
    doca_sync_event_add_publisher_location_dpa(comp_ev, dpa_ctx);
    doca_sync_event_add_subscriber_location_cpu(comp_ev, doca_dev);
    doca_sync_event_start(comp_ev);

    /* Export kernel events and acquire their handles */
    doca_sync_event_get_dpa_handle(ev_b, dpa_ctx, &ev_b_handle);
    doca_sync_event_get_dpa_handle(ev_c, dpa_ctx, &ev_c_handle);
    doca_sync_event_get_dpa_handle(comp_ev, dpa_ctx, &comp_ev_handle);

    /* Launch kernels */
    doca_dpa_kernel_launch_update_add(wait_ev = ev_a, wait_threshold = 1, num_threads = 1, &linear_kernel,
kernel_args: NULL, ev_b_handle);
    doca_dpa_kernel_launch_update_add(wait_ev = NULL,  num_threads = 1, &linear_kernel, kernel_args: ev_b_handle,
ev_c_handle);
    doca_dpa_kernel_launch_update_add(wait_ev = NULL, &linear_kernel, num_threads = 1, kernel_args: ev_c_handle,
comp_ev_handle);

    /* Update host event to trigger kernels to start executing in a linear manner */
    doca_sync_event_update_set(ev_a, 1)

    /* Wait for completion of last kernel */
    doca_sync_event_wait_gt(comp_ev, 0);

    /* Tear Down... */
    teardown_resources();
}
```

Diamond Execution Example

Kernel Code

```
#include "doca_dpa_dev.h"
#include "doca_dpa_dev_sync_event.h"

__dpa_global__ void
diamond_kernel(doca_dpa_dev_sync_event_t wait_ev, uint64_t wait_th, doca_dpa_dev_sync_event_t comp_ev1,
doca_dpa_dev_sync_event_t comp_ev2)
{
    if (wait_ev)
        doca_dpa_dev_sync_event_wait_gt(wait_ev, wait_th);

    doca_dpa_dev_sync_event_update_add(comp_ev1, comp_count = 1);
    if (comp_ev2)    // can be 0 (NULL)
        doca_dpa_dev_sync_event_update_add(comp_ev2, comp_count = 1);
}
```

Host Application Pseudo Code

```
#include <doca_dev.h>
#include <doca_error.h>
#include <doca_sync_event.h>
#include <doca_dpa.h>

int main(int argc, char **argv)
{
    /*
         A
        / \
       C   B
      /   /
     D   /
      \ /
       E
    */

    /* Open DOCA device */
    open_doca_dev(&doca_dev);
    /* Create doca dpa conext */
    doca_dpa_create(doca_dev, dpa_diamond_app, &dpa_ctx, 0);

    /* Create root event A that will signal from the host the rest to start */
    doca_sync_event_create(&ev_a);
    // set publisher to CPU, subscriber to DPA and start event

    /* Create events B,C,D,E */
    doca_sync_event_create(&ev_b);
    doca_sync_event_create(&ev_c);
    doca_sync_event_create(&ev_d);
    doca_sync_event_create(&ev_e);
    // for events B,C,D,E, set publisher & subscriber to DPA and start event

    /* Create completion event for last kernel */
    doca_sync_event_create(&comp_ev);
    // set publisher to DPA, subscriber to CPU and start event

    /* Export kernel events and acquire their handles */
    doca_sync_event_get_dpa_handle(&ev_b_handle, &ev_c_handle, &ev_d_handle, &ev_e_handle, &comp_ev_handle);

    /* wait threshold for each kernel is the number of parent nodes */
    constexpr uint64_t wait_threshold_one_parent {1};
    constexpr uint64_t wait_threshold_two_parent {2};

    /* launch diamond kernels */
    doca_dpa_kernel_launch_update_set(wait_ev = ev_a, wait_threshold = 1, num_threads = 1, &diamond_kernel,
kernel_args: NULL, 0, ev_b_handle, ev_c_handle);
    doca_dpa_kernel_launch_update_set(wait_ev = NULL, num_threads = 1, &diamond_kernel, kernel_args: ev_b_handle,
wait_threshold_one_parent, ev_e_handle, NULL);
    doca_dpa_kernel_launch_update_set(wait_ev = NULL, num_threads = 1, &diamond_kernel, kernel_args: ev_c_handle,
wait_threshold_one_parent, ev_d_handle, NULL);
    doca_dpa_kernel_launch_update_set(wait_ev = NULL, num_threads = 1, &diamond_kernel, kernel_args: ev_d_handle,
wait_threshold_one_parent, ev_e_handle, NULL);
    doca_dpa_kernel_launch_update_set(wait_ev = NULL, num_threads = 1, &diamond_kernel, kernel_args: ev_e_handle,
wait_threshold_two_parent, comp_ev_handle, NULL);

    /* Update host event to trigger kernels to start executing in a diamond manner */
    doca_sync_event_update_set(ev_a, 1);
    /* Wait for completion of last kernel */
    doca_sync_event_wait_gt(comp_ev, 0);

    /* Tear Down... */
    teardown_resources();
}
```

Performance Optimizations

- The time interval between a kernel launch call from the host and the start of its execution on the DPA is significantly optimized when the host application calls `doca_dpa_kernel_launch_update_<add|set>()` repeatedly to execute with the same number of DPA threads. So, if the application calls `doca_dpa_kernel_launch_update_<add|set>(..., num_threads = x)`, the next call with `num_threads = x` would have a shorter latency (as low as ~5-7 microseconds) for the start of the kernel's execution.
- Applications calling for kernel launch with a wait event (i.e., the completion event of a previous kernel) also have significantly lower latency in the time between the host launching the kernel and the start of the execution of the kernel on the DPA. So, if the application calls `doca_dpa_kernel_launch_update_<add|set>( ..., completion event = m_ev, ...)` and then `doca_dpa_kernel_launch_update_<add|set>( wait event = m_ev, ...)`, the latter kernel launch call would have shorter latency (as low as ~3 microseconds) for the start of the kernel's execution.

Limitations

- The order in which kernels are launched is important. If an application launches K1 and then K2, K1 must not depend on K2's completion (e.g., wait on its wait event that K2 should update).
  Not following this guideline leads to unpredictable results (at runtime) for the application and might require restarting the DOCA DPA context (i.e., destroying, reinitializing, and rerunning the workload).
- DPA threads are an actual hardware resource and are, therefore, limited in number to 256 (including internal allocations and allocations explicitly requested by the user as part of the kernel launch API)
  - DOCA DPA does not check these limits. It is up to the application to adhere to this number and track thread allocation across different DPA contexts.
  - Each `doca_dpa_dev_rdma_t` consumes one thread.
- The DPA has an internal watchdog timer to make sure threads do not block indefinitely. Kernel execution time must be finite and not exceed the time returned. by `doca_dpa_get_kernel_max_run_time`.

- The `num_threads` parameter in the `doca_dpa_kernel_launch` call cannot exceed the maximum allowed number of threads to run a kernel returned. by `doca_dpa_get_max_threads_per_kernel`.

14.4.3.6.4.13 Logging and Tracing

DOCA DPA provides a set of debugging APIs to allow diagnosing and troubleshooting any issues on the device, as well as accessing real-time information from the running application.

Logging in the data path has significant impact on an application's performance. While the tracer provided by the library is of high-frequency and is designed to prevent significant impact on the application's performance.

Therefore its recommended to use:

- Logging in the control path
- Tracing in the data path

The user is able to control the log/trace file path and device log verbosity.

Host-side API

- To set/get the trace file path:

```
doca_error_t doca_dpa_trace_file_set_path(struct doca_dpa *dpa, const char *file_path)
doca_error_t doca_dpa_trace_file_get_path(struct doca_dpa *dpa, char *file_path, uint32_t *file_path_len);
```

- To set/get the log file path:

```
doca_error_t doca_dpa_log_file_set_path(struct doca_dpa *dpa, const char *file_path)
doca_error_t doca_dpa_log_file_get_path(struct doca_dpa *dpa, char *file_path, uint32_t *file_path_len)
```

- To set/get device log verbosity:

```
doca_error_t doca_dpa_set_log_level(struct doca_dpa *dpa, doca_dpa_dev_log_level_t log_level)
doca_error_t doca_dpa_get_log_level(struct doca_dpa *dpa, doca_dpa_dev_log_level_t *log_level)
```

Device-side API

- Log to host:

```
typedef enum doca_dpa_dev_log_level {
    DOCA_DPA_DEV_LOG_LEVEL_DISABLE = 10, /**< Disable log messages */
    DOCA_DPA_DEV_LOG_LEVEL_CRIT = 20,    /**< Critical log level */
    DOCA_DPA_DEV_LOG_LEVEL_ERROR = 30,   /**< Error log level */
    DOCA_DPA_DEV_LOG_LEVEL_WARNING = 40, /**< Warning log level */
    DOCA_DPA_DEV_LOG_LEVEL_INFO = 50,    /**< Info log level */
    DOCA_DPA_DEV_LOG_LEVEL_DEBUG = 60,   /**< Debug log level */
} doca_dpa_dev_log_level_t;

void doca_dpa_dev_log(doca_dpa_dev_log_level_t log_level, const char *format, ...)
```

- Log macros:

```
DOCA_DPA_DEV_LOG_CRIT(...)
DOCA_DPA_DEV_LOG_ERR(...)
DOCA_DPA_DEV_LOG_WARN(...)
DOCA_DPA_DEV_LOG_INFO(...)
DOCA_DPA_DEV_LOG_DBG(...)
```

- To create a trace message entry with arguments:

```
void doca_dpa_dev_trace(uint64_t arg1, uint64_t arg2, uint64_t arg3, uint64_t arg4, uint64_t arg5)
```

- To flush the trace message buffer to host:

```
void doca_dpa_dev_trace_flush(void)
```

14.4.3.6.4.14 Error Handling

DPA context can enter an error state caused by the device flow. The application can check this error state by calling the following host API:

```
doca_error_t doca_dpa_peek_at_last_error(const struct doca_dpa *dpa)
```

If a fatal error core dump and crash occur, data is written to the file path `/tmp/doca_dpa_fatal` or to the file path set by the API `doca_dpa_log_file_set_path()` , with the suffixes `.PID.core` and `.PID.crash` respectively, where PID is the process ID. The data written to the file would include a memory snapshot at the time of the crash, which would contain information instrumental in pinpointing the cause of a crash (e.g., the program's state, variable values, and the call stack).

ⓘ Creating core dump files can be done after the DPA application has crashed.

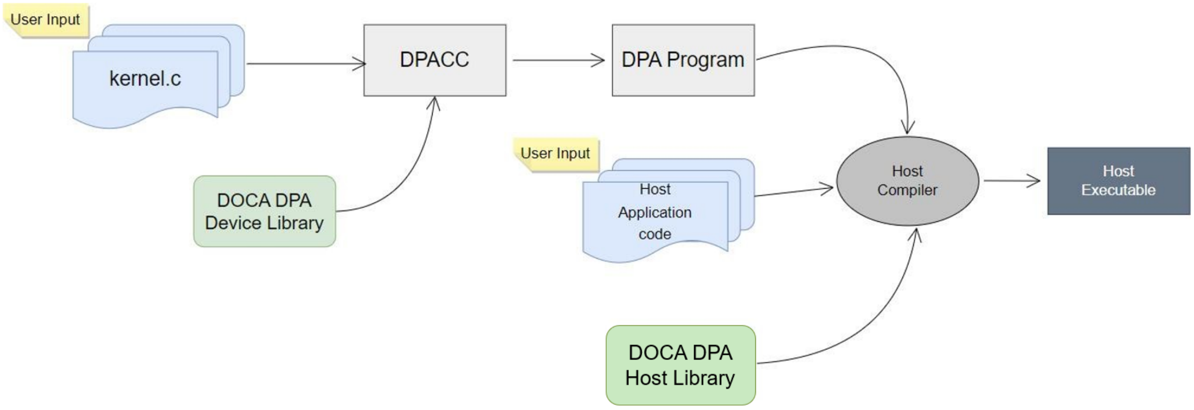
ⓘ This call does not reset the error state.

⚠ If an error occurred, DPA context enters a fatal state and must be destroyed by the user.

14.4.3.6.5 Hello World Example

14.4.3.6.5.1 Procedure Outline

1. Write DPA device code (i.e., kernels or `.c` files).
2. Use DPACC to build a DPA program (i.e., a host library which contains an embedded device executable). Input for DPACC:
   a. Kernels from step 1.
   b. DOCA DPA device library.
3. Build host executable using a host compiler. Input for the host compiler:
   a. DPA program.
   b. User host application `.c` / `.cpp` files.
4. Run host executable.



14.4.3.6.5.2 Procedure Steps

The following code snippets show a basic DPA code that eventually prints "Hello World" to `stdout` .

This is achieved using:

- A DPA Thread which prints the string and signals a DOCA Sync Event to indicate completion to host application
- A DPA RPC to notify DPA Thread

The steps are elaborated in the following subsections.

Prepare Kernels Code

```
#include "doca_dpa_dev.h"
#include "doca_dpa_dev_sync_event.h"

__dpa_global__ void hello_world_thread_kernel(uint64_t arg)
{
    DOCA_DPA_DEV_LOG_INFO("Hello World From DPA Thread!\n");
    doca_dpa_dev_sync_event_update_set(arg, 1);
    doca_dpa_dev_thread_finish();
}

__dpa_rpc__ uint64_t hello_world_thread_notify_rpc(doca_dpa_dev_notification_completion_t comp_handle)
{
    DOCA_DPA_DEV_LOG_INFO("Notifying DPA Thread From RPC\n");
    doca_dpa_dev_thread_notify(comp_handle);
    return 0;
}
```

Prepare Host Application Code

```
#include <stdio.h>
#include <unistd.h>
#include <doca_dev.h>
#include <doca_error.h>
#include <doca_sync_event.h>
#include <doca_dpa.h>

/**
 * A struct that includes all needed info on registered kernels and is initialized during linkage by DPACC.
 * Variable name should be the token passed to DPACC with --app-name parameter.
 */
extern struct doca_dpa_app *dpa_hello_world_app;

/**
 * kernel declaration that the user must declare for each kernel and DPACC is responsible to initialize.
 * Only then, user can use this variable in relevant host APIs
 */
doca_dpa_func_t hello_world_thread_kernel;
doca_dpa_func_t hello_world_thread_notify_rpc;

int main(int argc, char **argv)
{
    struct doca_dev *doca_dev = NULL;
    struct doca_dpa *dpa_ctx = NULL;
    struct doca_sync_event *cpu_se = NULL;
    doca_dpa_dev_sync_event_t cpu_se_handle = 0;
    struct doca_dpa_thread *dpa_thread = NULL;
    struct doca_dpa_notification_completion *notify_comp = NULL;
    doca_dpa_dev_notification_completion_t notify_comp_handle = 0;
    uint64_t retval = 0;

    printf("\n----> Open DOCA Device\n");
    /* Open appropriate DOCA device doca_dev */

    printf("\n----> Initialize DOCA DPA Context\n");
    doca_dpa_create(doca_dev, &dpa_ctx);
    doca_dpa_set_app(dpa_ctx, dpa_hello_world_app);
    doca_dpa_start(dpa_ctx);

    printf("\n----> Initialize DOCA Sync Event\n");
    doca_sync_event_create(&cpu_se);
    doca_sync_event_add_publisher_location_dpa(cpu_se, dpa_ctx);
    doca_sync_event_add_subscriber_location_cpu(cpu_se, doca_dev);
    doca_sync_event_start(cpu_se);
    doca_sync_event_get_dpa_handle(cpu_se, dpa_ctx, &cpu_se_handle);

    printf("\n----> Initialize DOCA DPA Thread\n");
    doca_dpa_thread_create(dpa_ctx, &dpa_thread);
    doca_dpa_thread_set_func_arg(dpa_thread, &hello_world_thread_kernel, cpu_se_handle);
    doca_dpa_thread_start(dpa_thread);

    printf("\n----> Initialize DOCA DPA Notification Completion\n");
    doca_dpa_notification_completion_create(dpa_ctx, dpa_thread, &notify_comp);
    doca_dpa_notification_completion_start(notify_comp);
    doca_dpa_notification_completion_get_dpa_handle(notify_comp, &notify_comp_handle);

    printf("\n----> Run DOCA DPA Thread\n");
    doca_dpa_thread_run(dpa_thread);

    printf("\n----> Trigger DPA RPC\n");
    doca_dpa_rpc(dpa_ctx, &hello_world_thread_notify_rpc, &retval, notify_comp_handle);

    printf("\n----> Waiting For hello_world_thread_kernel To Finish\n");
    doca_sync_event_wait_gt(cpu_se, 0, 0xFFFFFFFFFFFFFFFF);

    printf("\n----> Destroy DOCA DPA Notification Completion\n");
    doca_dpa_notification_completion_destroy(notify_comp);

    printf("\n----> Destroy DOCA DPA Thread\n");
    doca_dpa_thread_destroy(dpa_thread);

    printf("\n----> Destroy DOCA DPA event\n");
    doca_sync_event_destroy(cpu_se);

    printf("\n----> Destroy DOCA DPA context\n");
    doca_dpa_destroy(dpa_ctx);

    printf("\n----> Destroy DOCA device\n");
    doca_dev_close(doca_dev);

    printf("\n----> DONE!\n");
    return 0;
}
```

Build DPA Program

```
/opt/mellanox/doca/tools/dpacc \
    kernel.c \
    -o dpa_program.a \
    -hostcc=gcc \
    -hostcc-options="-Wno-deprecated-declarations -Werror -Wall -Wextra -W" \
    --devicecc-options="-D__linux__ -Wno-deprecated-declarations -Werror -Wall -Wextra -W" \
    --app-name="dpa_hello_world_app" \
    -ldpa \
    -I/opt/mellanox/doca/include/
```

Build Host Application

```
gcc hello_world.c -o hello_world \
    dpa_program.a \
    -I/opt/mellanox/doca/include/ \
    -DDOCA_ALLOW_EXPERIMENTAL_API \
    -L/opt/mellanox/doca/lib/x86_64-linux-gnu/ -ldoca_dpa -ldoca_common \
    -L/opt/mellanox/flexio/lib/ -lflexio \
    -lstdc++ -libverbs -lmlx5
```

Execution

```
$  ./hello_world

----> Open DOCA Device

----> Initialize DOCA DPA Context
```

```
----> Initialize DOCA Sync Event

----> Initialize DOCA DPA Thread

----> Initialize DOCA DPA Notification Completion

----> Run DOCA DPA Thread

----> Trigger DPA RPC
/ 10/[DOCA][DPA DEVICE][INF] Notifying DPA Thread From RPC
/  8/[DOCA][DPA DEVICE][INF] Hello World From DPA Thread!

----> Waiting For hello_world_thread_kernel To Finish

----> Destroy DOCA DPA Notification Completion

----> Destroy DOCA DPA Thread

----> Destroy DOCA DPA event

----> Destroy DOCA DPA context

----> Destroy DOCA device

----> DONE!
```

14.4.3.6.6 Samples

This section provides DPA sample implementation on top of the BlueField-3 networking platform.

> All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

To run DPA samples:

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample:

   ```
   cd /opt/mellanox/doca/samples/doca_dpa/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build
   ```

   > The binary `doca_<sample_name>` is created under `/tmp/build/`.

3. Sample (e.g., `dpa_initiator_target`) usage:

   ```
   Usage: doca_dpa_initiator_target [DOCA Flags] [Program Flags]

   DOCA Flags:
     -h, --help                              Print a help synopsis
     -v, --version                           Print program version information
     -l, --log-level                         Set the (numeric) log level for the program <10=DISABLE, 20=CRITI
   CAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
     --sdk-log-level                         Set the SDK (numeric) log level for the program <10=DISABLE, 20=C
   RITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
     -j, --json <path>                       Parse all command flags from an input json file

   Program Flags:
     -d, --device <device name>           device name that supports DPA (optional). If not provided then a
   random device will be chosen
   ```

4. For additional information per sample, use the `-h` option:

   ```
   /tmp/build/doca_<sample_name> -h
   ```

14.4.3.6.6.1 Basic Initiator Target

This sample illustrates how to trigger DPA Thread using DPA Completion Context attached to DOCA RDMA.

This sample consists of initiator and target endpoints.

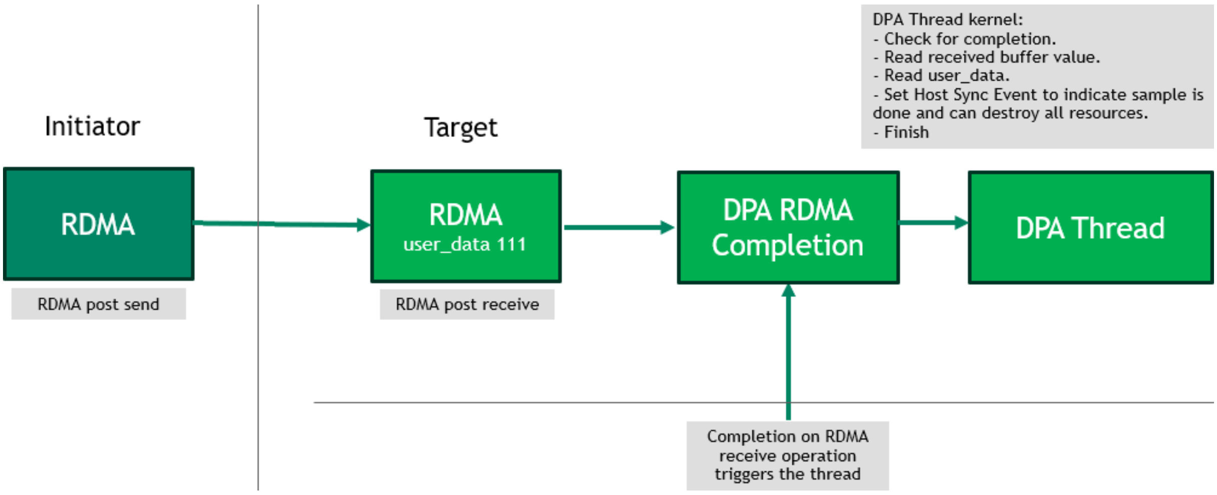
In the initiator endpoint, a DOCA RDMA executes RDMA post send operation using DPA RPC.

In the target endpoint, a DOCA RDMA, attached to DPA Completion Context, executes RDMA post receive operation using DPA RPC.

Completion on the post receive operation triggers DPA Thread which prints completion info and sets DOCA Sync Event to release the host application that waits on that event before destroying all resources and finish.

The sample logic includes:

1. Allocating DOCA DPA & DOCA RDMA resources for both initiator and target endpoints.
2. Target: Attaching DOCA RDMA to DPA Completion Context which is attached to DPA Thread.
3. Run DPA Thread.
4. Target: DPA RPC to execute RDMA post receive operation.
5. Initiator: DPA RPC to execute RDMA post send operation.
6. The completion on the post receive operation triggers DPA Thread.
7. Waiting on completion event to be set from DPA Thread.
8. Destroying all resources.


DPA Thread kernel:
- Check for completion.
- Read received buffer value.
- Read user_data.
- Set Host Sync Event to indicate sample is done and can destroy all resources.
- Finish
Initiator
Target
RDMA
RDMA post send
RDMA
user_data 111
RDMA post receive
DPA RDMA Completion
DPA Thread
Completion on RDMA receive operation triggers the thread

Reference:

- `/opt/mellanox/doca/samples/doca_dpa/dpa_basic_initiator_target/dpa_basic_initiator_target_main.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_basic_initiator_target/host/dpa_basic_initiator_target_sample.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_basic_initiator_target/device/dpa_basic_initiator_target_kernels_dev.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_basic_initiator_target/meson.build`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.h`

- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.c`
- `/opt/mellanox/doca/samples/doca_dpa/build_dpacc_samples.sh`

14.4.3.6.6.2 Advanced Initiator Target

This sample illustrates how to trigger DPA threads using both DPA Notification Completion and DPA Completion Context which is attached to multiple DOCA RDMAs.

This sample consists of initiator and target endpoints.

In the initiator endpoint, two DOCA RDMAs execute an RDMA post send operation using DPA RPC in the following order:

1. RDMA #1 executes the RDMA post send operation on buffer with value 1.
2. RDMA #2 executes the RDMA post send operation on buffer with value 2.
3. RDMA #1 executes the RDMA post send operation on buffer with value 3.
4. RDMA #2 executes the RDMA post send operation on buffer with value 4.

In the target endpoint, two DOCA RDMAs, RDMA #1 with user data `111` and RDMA #2 with user data `222`.

Target RDMAs are attached to a single DPA Completion Context which is attached to DPA Thread #1.

Target RDMAs execute the initial RDMA post receive operation using DPA RPC.

Completions on the post receive operations trigger DPA Thread #1 which:

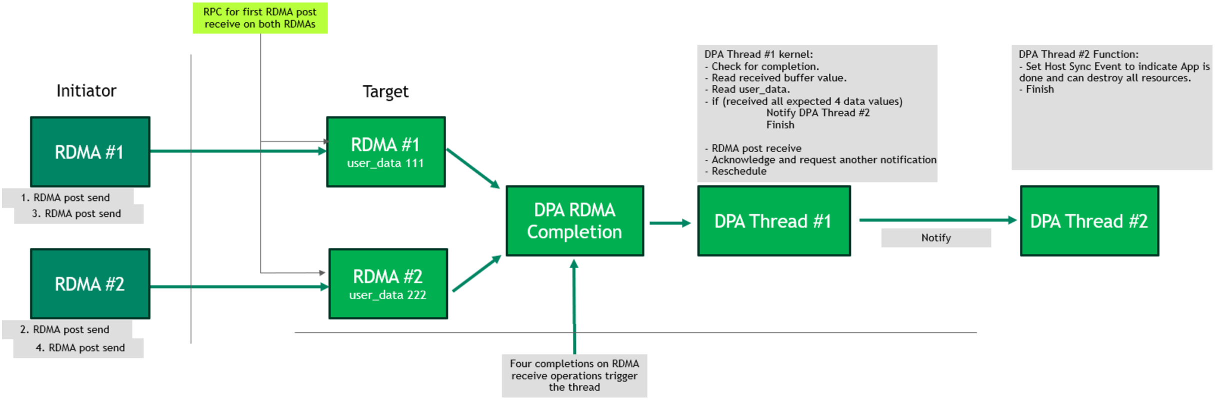
1. Prints completion info including user data.
2. Updates a local data base with the receive buffer value.
3. Repost RDMA receive operation.
4. Ack, request completion and reschedule.

Once target DPA Thread #1 receives all expected values "1, 2, 3, 4", it notify DPA Thread #2 and finish.

Once DPA Thread #2 is triggered, it sets DOCA Sync Event to release the host application that waits on that event before destroying all resources and finishing.

The sample logic includes:

1. Allocating DOCA DPA and DOCA RDMA resources for both initiator and target endpoints.
2. Target: Attaching both DOCA RDMAs to DPA Completion Context which is attached to DPA Thread #1.
3. Target: Attaching DPA Notification Completion to DPA Thread #2.
4. Run DPA threads.
5. Target: DPA RPC to execute the initial RDMA post receive operation.
6. Initiator: DPA RPC to execute all RDMA post send operations.
7. Completions on the post receive operations (4 completions) trigger DPA Thread #1.
8. Once all expected values are received, DPA Thread #1 notifies DPA Thread #2 and finishes.
9. Waiting on completion event to be set from DPA Thread #2.
10. Destroying all resources.



Reference:

- `/opt/mellanox/doca/samples/doca_dpa/dpa_initiator_target/dpa_initiator_target_main.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_initiator_target/host/dpa_initiator_target_sample.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_initiator_target/device/dpa_initiator_target_kernels_dev.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_initiator_target/meson.build`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.h`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.c`
- `/opt/mellanox/doca/samples/doca_dpa/build_dpacc_samples.sh`

14.4.3.6.6.3 Ping Pong

This sample illustrates the functionality of the following DPA objects:

- DPA Thread
- DPA Completion Context
- DOCA RDMA

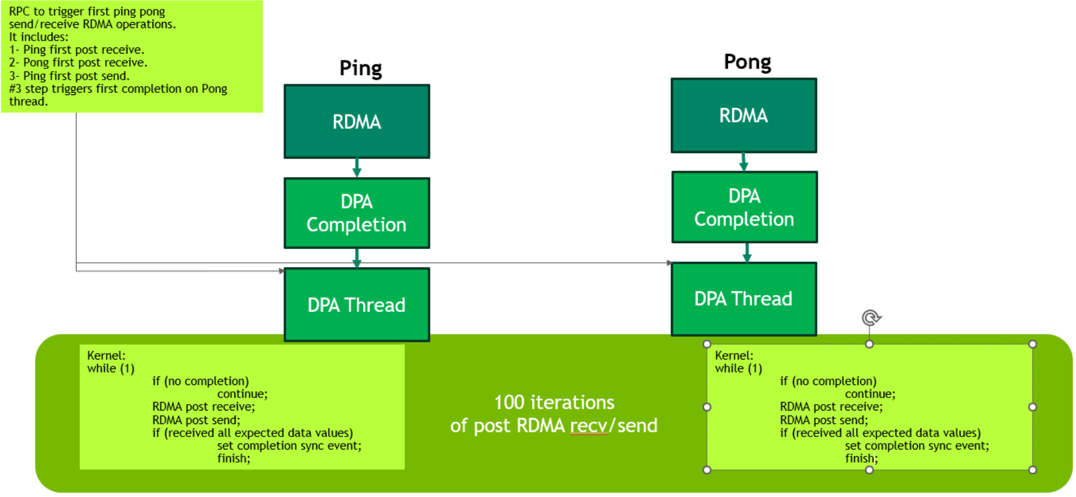
This sample consists of ping and pong endpoints which run for 100 iterations. On each iteration, DPA threads (ping and pong) post RDMA receive and send operations for data buffers with values [0-99].

Once all expected values are received on each DPA thread, it sets a DOCA Sync Event to release the host application waiting on that event before destroying all resources and finishes.

To trigger DPA threads, the sample uses a DPA RPC.

The sample logic includes:

1. Allocating DOCA DPA and DOCA RDMA resources.
2. Attaching DOCA RDMA to DPA completion context which is attached to DPA thread.
3. Run DPA threads.
4. DPA RPC to trigger DPA threads.
5. 100 ping pong iterations of RDMA post receive and send operations.
6. Waiting on completion events to be set from DPA threads.
7. Destroying all resources.



RPC to trigger first ping pong send/receive RDMA operations.
It includes:
1- Ping first post receive.
2- Pong first post receive.
3- Ping first post send.
#3 step triggers first completion on Pong thread.

```
Kernel:
while (1)
          if (no completion)
                    continue;
          RDMA post receive;
          RDMA post send;
          if (received all expected data values)
                    set completion sync event;
                    finish;
```

```
Kernel:
while (1)
          if (no completion)
                    continue;
          RDMA post receive;
          RDMA post send;
          if (received all expected data values)
                    set completion sync event;
                    finish;
```

Reference:

- `/opt/mellanox/doca/samples/doca_dpa/dpa_ping_pong/dpa_ping_pong_main.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_ping_pong/host/dpa_ping_pong_sample.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_ping_pong/device/dpa_ping_pong_kernels_dev.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_ping_pong/meson.build`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.h`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.c`
- `/opt/mellanox/doca/samples/doca_dpa/build_dpacc_samples.sh`

14.4.3.6.6.4 Kernel Launch

This sample illustrates how to launch a DOCA DPA kernel with wait and completion DOCA sync events.

The sample logic includes:

1. Allocating DOCA DPA resources.
2. Initializing wait and completion DOCA sync events for the DOCA DPA kernel.
3. Running `hello_world` DOCA DPA kernel that waits on the wait event.
4. Running a separate thread that triggers the wait event.
5. `hello_world` DOCA DPA kernel prints "Hello from kernel".
6. Waiting for the completion event of the kernel.
7. Destroying the events and resources.

Reference:

- `/opt/mellanox/doca/samples/doca_dpa/dpa_wait_kernel_launch/dpa_wait_kernel_launch_main.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_wait_kernel_launch/host/dpa_wait_kernel_launch_sample.c`

- `/opt/mellanox/doca/samples/doca_dpa/dpa_wait_kernel_launch/device/dpa_wait_kernel_launch_kernels_dev.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_wait_kernel_launch/meson.build`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.h`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.c`
- `/opt/mellanox/doca/samples/doca_dpa/build_dpacc_samples.sh`

14.4.3.7 DOCA PCC

This guide provides an overview and configuration instructions for DOCA Programmable Congestion Control (PCC) API.

14.4.3.7.1 Introduction

The DOCA PCC library provides a high-level programming interface that allows users to implement their own customized congestion control (CC) algorithm, facilitating efficient management of network congestion in their applications.

The DOCA PCC library provides an API to:

- Configure probe packets to send and receive
- Get the CC event/packet and access its fields
- Set a rate limit for a flow
- Maintain a context for each flow
- Initiate and configure CC algorithms
- Obtain request packets arriving from the network and setup response packets in return

This library uses the NVIDIA® BlueField®-3 Platform hardware acceleration for CC management, while providing an API that simplifies hardware complexity, allowing users to focus on the functionality of the CC algorithm.

14.4.3.7.2 Prerequisites

DOCA PCC-based applications can run either on the host machine or on the NVIDIA® BlueField®-3 Platform (or later) target.

> Currently, DOCA PCC is only supported for ETHERNET link type.

To enable DOCA PCC RP:

1. Run the following on the host/VM:

```
mlxconfig -d <mlx_device> -y s USER_PROGRAMMABLE_CC=1
```

2. Perform graceful shutdown then power cycle the host.

To enable DOCA PCC NP:

1. Run the following on the host/VM

```
mlxconfig -d <mlx_device> -y s PCC_INT_EN=0
```

2. Perform graceful shutdown then power cycle the host.

> Configuring `PCC_INT_EN` to 1 blocks the creation of DOCA PCC NP context and enables legacy NP solution. In addition, it only supports DOCA PCC RP context to set Congestion Control Message After Drop (CCMAD) probe packet format.
>
> If IFA2.0 support is needed, user needs to enable DOCA PCC RP and DOCA PCC NP on all nodes of the cluster.

> If running from the host in NIC mode, users must have PRIVILIGED permission to configure the above parameters. To check privileging level, run:
>
> ```
> mlxprivhost -d <mlx_device> q
> ```

The DPACC tool is used to compile and link user algorithm and device code with the DOCA PCC device library to get applications that can be loaded from the host program.

DPACC is bundled as part of the DOCA SDK installation package. For more information on DPACC, refer to NVIDIA DOCA DPACC Compiler.

14.4.3.7.3 Changes From Previous Releases

14.4.3.7.3.1 Changes in 2.8.0

Added

- `doca_pcc_dump_debug(const struct doca_pcc *pcc)`
- `doca_pcc_enable_debug(const struct doca_pcc *pcc, bool enable)`

Changed

- `DOCA_PCC_DEV_MAX_NUM_PARAMS_PER_ALGO` (0x1E) → (0x26)
- `DOCA_PCC_DEV_MAX_NUM_COUNTERS_PER_ALGO` (0xF) → (0x3F)
- `struct mlnx_cc_event_general_attr_t` - `port_num` split to two fields (see diff below)

```
struct mlnx_cc_event_general_attr_t {  /* Little Endian */
     uint32_t ev_type:8;     /* event type */
     uint32_t ev_subtype:8;  /* event subtype */
-    uint32_t port_num:8;    /* port id */
+    uint32_t port_num:4;    /* port id */
+    uint32_t reserved:4;
     uint32_t flags:8;    /* event flags */
 }
```

- `struct mlnx_cc_event_t` - added 12B reserved

```
struct mlnx_cc_event_t {     /* Little Endian */
+    uint32_t reserved[3];                      /* reserved */
struct mlnx_cc_event_general_attr_t ev_attr;    /* event general attributes */
```

```
uint32_t flow_tag;                                /* unique flow id */
uint32_t sn;                                      /* serial number */
uint32_t timestamp;                               /* event timestamp */
union mlnx_cc_event_spec_attr_t ev_spec_attr;     /* attributes which are different for different events */
};
```

14.4.3.7.4 Dependencies

The library requires firmware version 32.38.1000 and higher.

14.4.3.7.5 Architecture

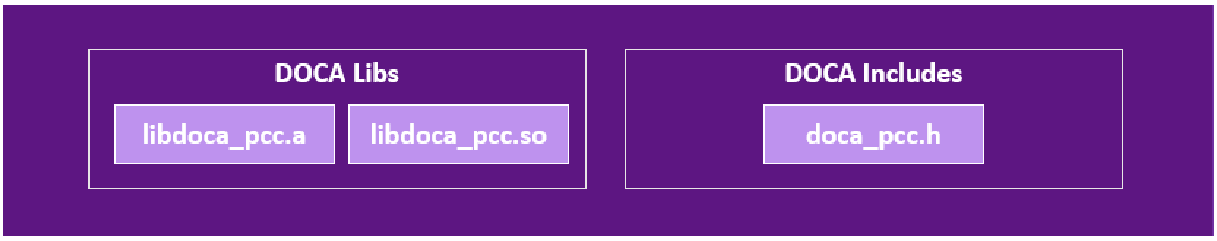
DOCA PCC comprises three main components which are part of the DOCA SDK installation package:

14.4.3.7.5.1 Host Library

The host library offers a unified interface for managing the DOCA PCC context configuration.

As part of the control path, the host library integrates passively within the application, orchestrating congestion control activities without directly handling data transmission.

Host/device library and header files:



DOCA Libs

libdoca_pcc.a

libdoca_pcc.so

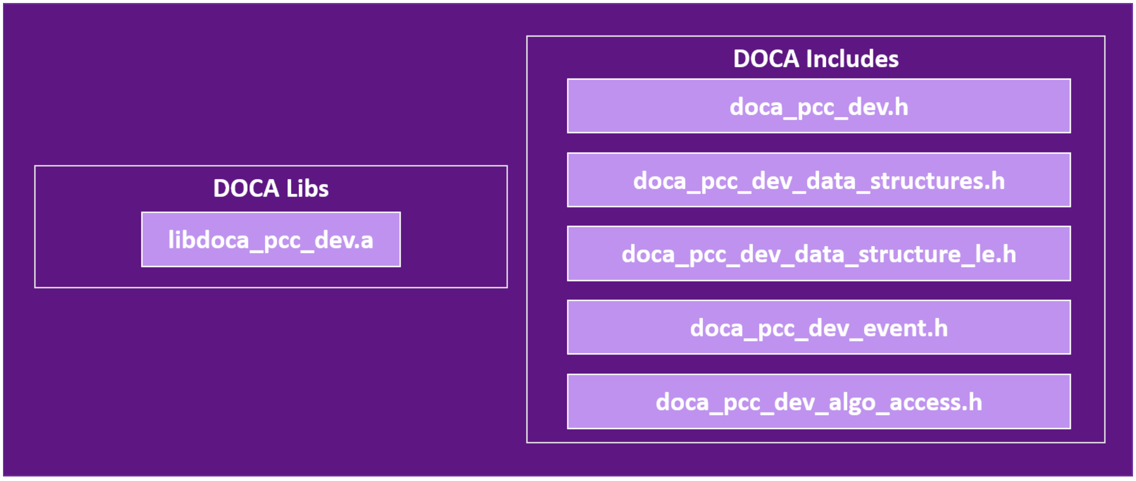
DOCA Includes

doca_pcc.h

14.4.3.7.5.2 Device Libraries

The DOCA PCC context assumes one of two roles:

- Reaction point (RP): Monitors network conditions actively, dynamically adjusting data transmission rates to alleviate congestion promptly. RP context is global per NIC.
  Device library and header files:



DOCA Libs

libdoca_pcc_dev.a

DOCA Includes

doca_pcc_dev.h

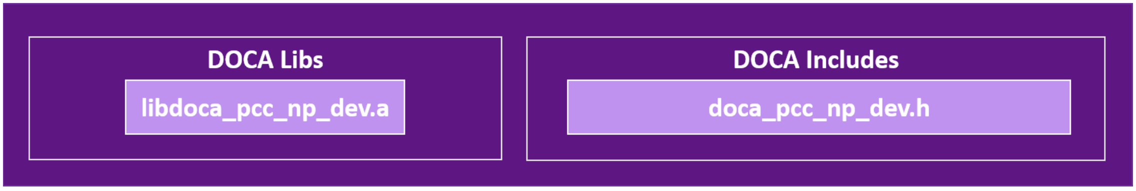
doca_pcc_dev_data_structures.h

doca_pcc_dev_data_structure_le.h

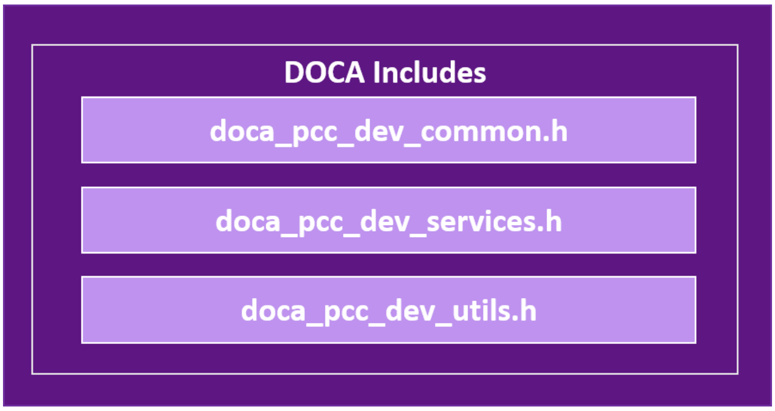
doca_pcc_dev_event.h

doca_pcc_dev_algo_access.h

- Notification point (NP): Passively receives congestion notifications from external sources, processing them intelligently to facilitate informed decisions within the application. NP context is global per e-switch owner.
  Device library and header files:



Both RP and NP device libraries share common headers:



Currently, the device library and the user algorithm are implemented and managed over the BlueField's data-path accelerator (DPA) subsystem.

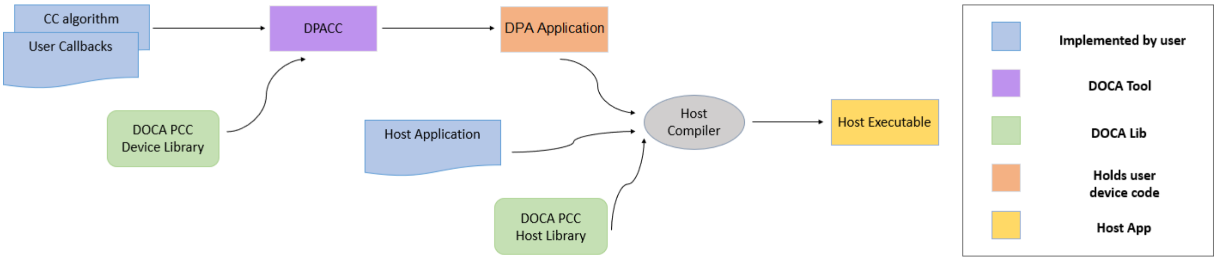
For more info on DPA, refer to DPA Subsystem.

14.4.3.7.5.3 Development Flow

DOCA enables developers to program the congestion control algorithm into the system using the DOCA PCC library.

The following are the required steps to start programming:

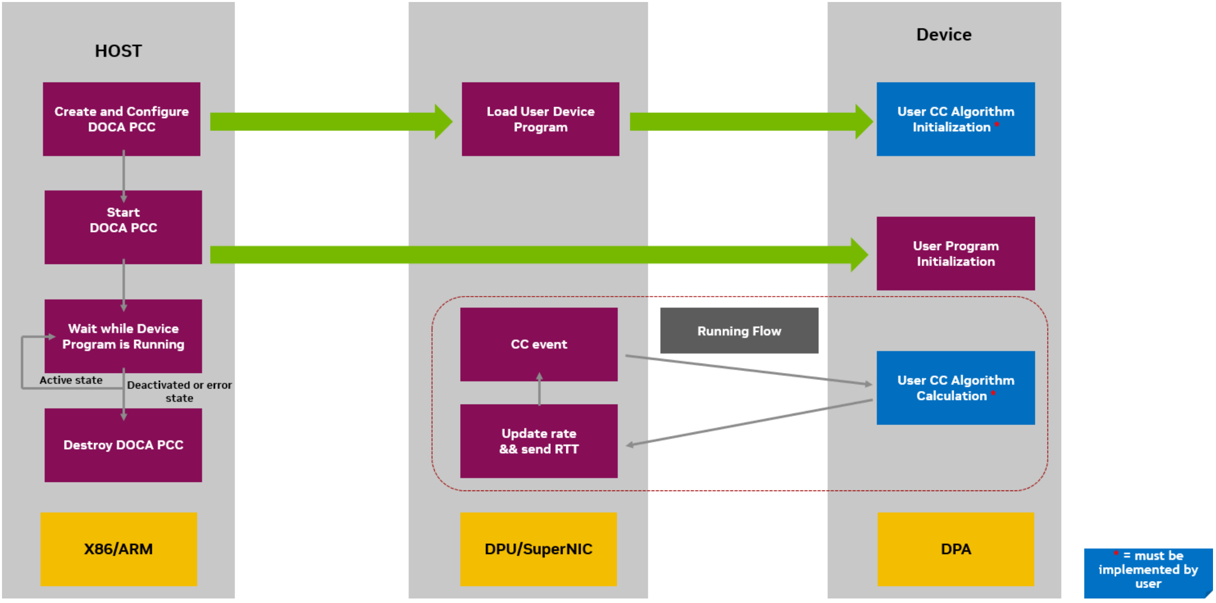
1. Implement CC algorithms and probe packet handling using the API provided by the device header files.
2. Implement the user callbacks defined by the library for DataPath:
   - For RP: `doca_pcc_dev_user_init()` , `doca_pcc_dev_user_set_algo_params()` , `doca_pcc_dev_user_algo()` .
   - For NP: `doca_pcc_dev_np_user_packet_handler()`
3. Use DPACC to build a DPA application (i.e., a host library which contains an embedded device executable). Input for DPACC are the files containing the implementation of the previous steps.
4. Build host executable using a host compiler. Inputs for the host compiler are the DPA application generated in the previous step and the user application host source files.
5. In the host executable, create and start a DOCA PCC context which is set with the DPA application containing the device code.



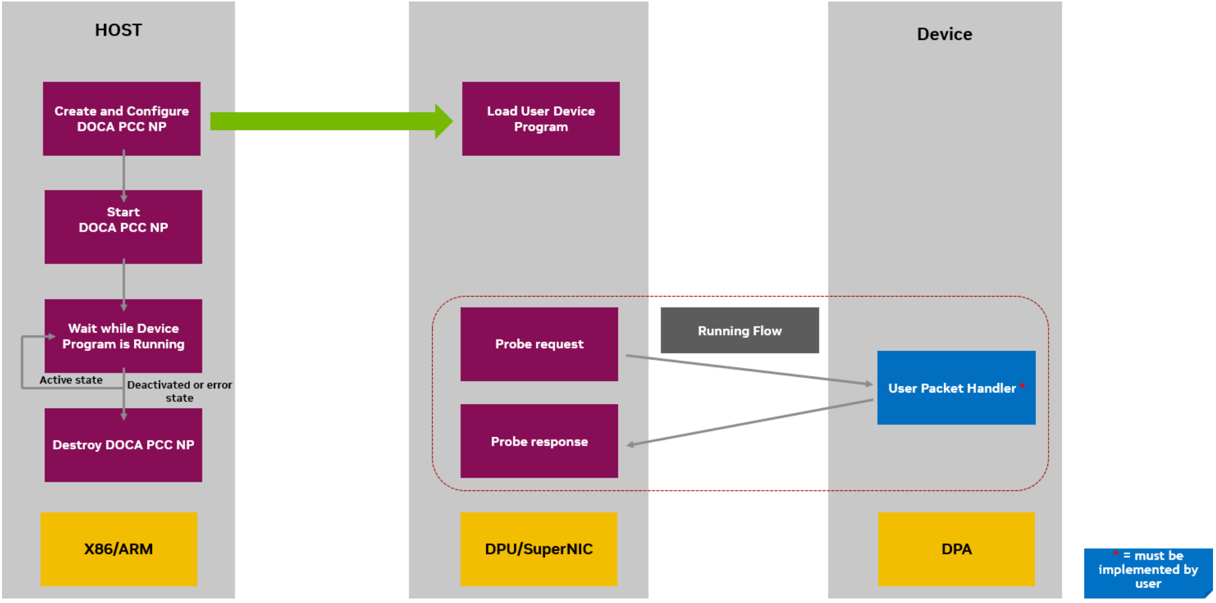
For a more descriptive example, refer to *NVIDIA DOCA PCC Application Guide*.

14.4.3.7.5.4 System Design

DOCA PCC flow for implementing an RP program:


HOST
Device
Create and Configure DOCA PCC
Load User Device Program
User CC Algorithm Initialization *
Start DOCA PCC
User Program Initialization
Wait while Device Program is Running
Active state
Deactivated or error state
CC event
Running Flow
User CC Algorithm Calculation *
Update rate && send RTT
Destroy DOCA PCC
X86/ARM
DPU/SuperNIC
DPA
* = must be implemented by user

DOCA PCC flow for implementing an NP program:



14.4.3.7.6 API

For the library API reference, refer to PCC API documentation in the NVIDIA DOCA Library APIs.

The following sections provide additional details about the library API.

14.4.3.7.6.1 Host API

The host library API consists of calls to set the PCC context attributes and observe availability of the process.

Selecting and Opening DOCA Device

To perform PCC operations, a device must be selected. To select a device, users may iterate over all DOCA devices using `doca_devinfo_list_create()` and check whether the device supports the desired PCC role either via `doca_devinfo_get_is_pcc_supported()` for RP, or `doca_pcc_np_cap_is_supported()` for NP.

Setting Up and Starting DOCA PCC Context

After selecting a DOCA device, a PCC context can be created.

As described in the Architecture section, The DOCA PCC library provides APIs to leverage Reaction Points (RP) and Notification Points (NP) to implement programmable congestion control strategies.

Call `doca_pcc_create()` to create a DOCA PCC RP context, and `doca_pcc_np_create()` to create a DOCA PCC NP context.

Afterwards, the following attributes must be set for the PCC context:

- Context app - the name of the DPA application compiled using DPACC, consisting of the device algorithm and code. This is set using the call `doca_pcc_set_app()`.
- Context threads - the affinity of DPA threads to be used to handle CC events. This is set using the call `doca_pcc_set_thread_affinity()`. The number of threads to be used must be constrained between the minimum and maximum number of threads allowed to run the PCC process (see `doca_pcc_get_min_num_threads()` and `doca_pcc_get_max_num_threads()`). The availability and usage of the threads for PCC is dependent on the complexity of the CC algorithm, link rate, and other potential DPA users.

  > ⚠ Users can manage DPA threads in the system using EU pre-configuration with the `dpaeumgmt` tool. For more information, refer to NVIDIA DOCA DPA Execution Unit Management Tool.

After setting up the context attributes, the context can be started using `doca_pcc_start()`. Starting the context initiates the CC algorithm supplied by the user.

Configuring Probe Packets

The DOCA PCC library provides APIs to configure the probe packet settings to tailor congestion control behaviors according to specific network conditions.

The probe packet serves to probe the network for congestion and gather essential feedback for congestion control algorithms.

The DOCA PCC Library supports the following probe packet types:

- CCMAD - Provides information about the network's round-trip time so the algorithm can detect and adapt to congestion proactively
- IFA1 - In-band Flow Analyzer 1 packets provide in-band congestion feedback for proactive congestion control
- IFA2 - In-band Flow Analyzer 2 packets offer an alternative method for in-band congestion feedback, optimized for specific network environments

Configuring Dedicated Fields for Different Probe Types

The DOCA PCC library provides APIs to configure specific fields in different supported probe packet types.

- IFA1 - support to configure probe marker
- IFA2 - support to configure gns and hop limit

Configuring Remote NP Handler

To enable Reaction Point contexts to interact with remote Notification Point contexts, the DOCA PCC library provides an API to set the expected remote handler type.

When the DOCA PCC RP process expects CCMAD probe packet responses from a DOCA PCC NP process, it should set it as so using the API `doca_pcc_rp_set_ccmad_remote_sw_handler()`. If not set, the DOCA PCC RP process expects that no remote DOCA PCC NP process is activated, and that responses are handled by the remote node's hardware. Note that if using other probe types than CCMAD, probe packet responses are always expected to be generated from a remote DOCA Notification Point process.

Debuggability

The DOCA PCC library provides a set of debugging APIs to allow the user to diagnose and troubleshoot any issues on the device, as well as accessing real-time information from the running application:

- `doca_pcc_set_dev_coredump_file()` - API to set a filename to write crash data and core dump into should a fatal error occur on the device side of the application. The data written into the file would include a memory snapshot at the time of the crash, which would contain information instrumental in pinpointing the cause of a crash (e.g., the program's state, variable values, and the call stack).
- `doca_pcc_set_trace_message()` - API to enable tracing on the device side of the application by setting trace message formats that can be printed from the device. The tracer provided by the library is of high-frequency and is designed to not have significant impact on the application's performance. This API can help the user to monitor and gain insight into the behavior of the running device algorithm, identify performance bottlenecks, and diagnose issues, without incurring any notable performance degradation.
- `doca_pcc_set_print_buffer_size()` - API to set the buffer size to be printed by the print API provided by the device library.

Host - Device Mailbox

The DOCA PCC library provides a set of APIs for sending and receiving messages through a mailbox. This service allows communication between the host and device:

- `doca_pcc_set_mailbox()` - API to set the mailbox attributes for the process.
- `doca_pcc_mailbox_get_request_buffer()` and `doca_pcc_mailbox_get_response_buffer()` - API to get the buffers with which the communication will be handled. User can set the request he wants to send to the device, and get a response back.
- `doca_pcc_mailbox_send()` - API to send the mailbox request to the device. This is a blocking call which invokes a callback on the device `doca_pcc_dev_user_mailbox_handle()` which user can handle.

High Availability

The DOCA PCC library provides high availability, allowing fast recovery should the running PCC process malfunction. High availability can be achieved by running multiple PCC processes in parallel.

When calling `doca_pcc_start()` , the library registers the process with the BlueField firmware such that the first PCC process to be registered becomes the ACTIVE PCC process (i.e., actually runs on DPA and handles CC events).

The other processes operate in STANDBY mode. If the ACTIVE process stops processing events or hits an error, the firmware replaces it with one of the standby processes, making it ACTIVE.

The defunct process should call `doca_pcc_destroy()` to free its resources.

The state of the process may be observed periodically using `doca_pcc_get_process_state()` . A change in the state of the process returns the call `doca_pcc_wait()` .

The following values describe the state of the PCC process at any point:

```
typedef enum {
    DOCA_PCC_PS_ACTIVE = 0,
    /**< The process handles CC events (only one process is active at a given time) */
    DOCA_PCC_PS_STANDBY = 1,
    /**< The process is in standby mode (another process is already ACTIVE)*/
    DOCA_PCC_PS_DEACTIVATED = 2,
    /**< The process was deactivated by NIC FW and should be destroyed */
    DOCA_PCC_PS_ERROR = 3,
    /**< The process is in error state and should be destroyed */
} doca_pcc_process_state_t;
```

14.4.3.7.6.2 Device API

The device library API consists of calls to setup the CC algorithm to handle CC events arriving on hardware.

Counter Sampling

The device libraries APIs provide an API to sample the NIC bytes counters. These counters help monitor the amount of data transmitted and received through the NIC.

The user can prepare the list of counters to read using `doca_pcc_dev_nic_counters_config()` and sample the new counters values with the call `doca_pcc_dev_nic_counters_sample()` .

Algorithm Access

The Reaction Point (RP) device library API provides a set of functions to initiate and identify the different CC algorithms.

The DOCA PCC library is designed to support more than one PCC algorithm. The library comes with a default algorithm which can be used fully or partially by the user using `doca_pcc_dev_default_internal_algo()` , alongside other CC algorithms supplied by the user. This can be useful for fast comparative runs between the different algorithms. Each algorithm can run on a different device port using `doca_pcc_dev_init_algo_slot()` .

The algorithm can supply its own identifier, initiate its parameter (using `doca_pcc_dev_algo_init_param()`), counter (using `doca_pcc_dev_algo_init_counter()`), and metadata base (using `doca_pcc_dev_algo_init_metadata()`).

Events

The RP device library API provides a set of optimized CC event access functions. These functions serve as helpers to build the CC algorithm and to provide runtime data to analyze and inspect CC events arriving on hardware.

Utilities

The device library APIs provide a set of optimized utility macros that are set to support programming the CC algorithm. Such utilities are composed of fixed-point operations, memory space fences, and more.

User Callbacks

The device libraries API consists of specific user callbacks used by the library to initiate and run the CC algorithm and handle input and output packets. These callbacks must be implemented by the user and, to be part of the DPA application, compiled by DPACC to provide to the DOCA PCC context.

The set of callbacks to be implemented for RP:

- `doca_pcc_dev_user_init()` - called on PCC process load and should initialize the data of all user algorithms
- `doca_pcc_dev_user_algo()` - entry point to the user algorithm handling code
- `doca_pcc_dev_user_set_algo_params()` - called when the parameter change is set externally

The set of callbacks to be implemented for NP:

- `doca_pcc_dev_np_user_packet_handler()` - called on probe packets arrival

14.4.4 DOCA DMA

This guide provides instructions on building and developing applications that require copying memory using Direct Memory Access (DMA).

14.4.4.1 Introduction

DOCA DMA provides an API to copy data between DOCA buffers using hardware acceleration, supporting both local and remote memory regions.

The library provides an API for executing DMA operations on DOCA buffers, where these buffers reside either in local memory (i.e., within the same host) or host memory accessible by the DPU. See DOCA Core for more information about the memory subsystem.

Using DOCA DMA, complex memory copy operations can be easily executed in an optimized, hardware-accelerated manner.

This document is intended for software developers wishing to accelerate their application's memory I/O operations and access memory that is not local to the host.

14.4.4.2 Prerequisites

This library follows the architecture of a DOCA Core Context, it is recommended read the following sections before:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

14.4.4.3 Library Changes From Previous Releases

14.4.4.3.1 Changes in 2.8.0

The following subsection(s) detail the `doca_comch` library updates in version 2.8.0.

14.4.4.3.1.1 API Additions

- `doca_error_t doca_dma_get_gpu_handle(struct doca_dma *dma, struct doca_gpu_dma **gpu_dma)`
    - Provides the option to export DMA to GPU and use GPUNetIO for DMA datapath on the GPU

14.4.4.4 Environment

DOCA DMA-based applications can run either on the host machine or on the NVIDIA® BlueField® DPU target.

Copying from Host to DPU and vice versa only works with a DPU configured running in DPU mode as described in NVIDIA BlueField Modes of Operation.

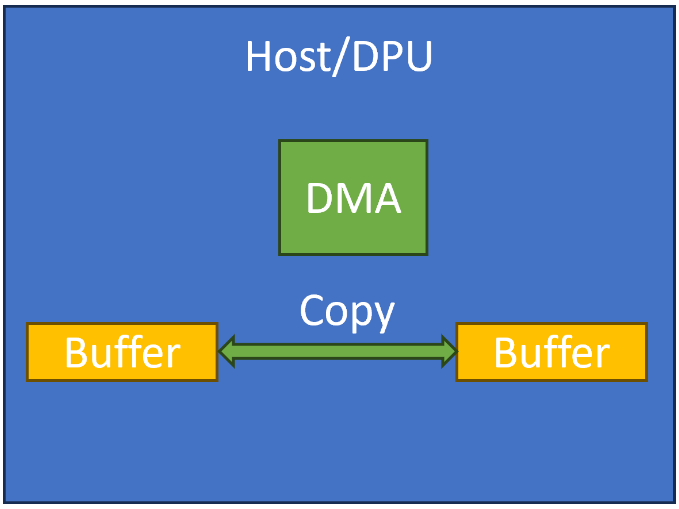
14.4.4.5 Architecture

DOCA DMA is a DOCA Context as defined by DOCA Core. See DOCA Core Context for more information.

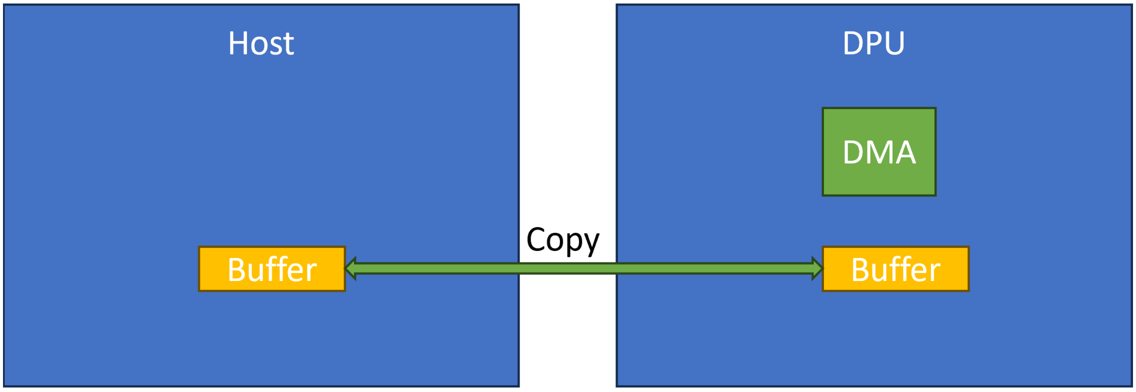
DOCA DMA leverages DOCA Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

DMA can be used to copy data as follows:

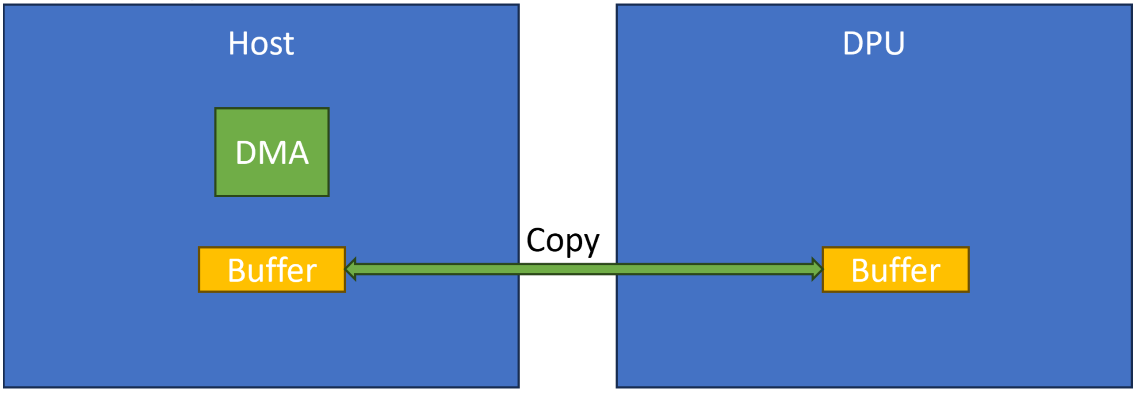
- Copying from local memory to local memory:



- Using DPU to copy memory between host and DPU:



- Using host to copy memory between host and DPU:



14.4.4.5.1 Objects

14.4.4.5.1.1 Device and Device Representor

The DMA library needs a DOCA device to operate. The device is used to access memory and perform the actual copy. See DOCA Core Device Discovery.

For same BlueField DPU, it does not matter which device is used (PF/VF/SF), as all these devices utilize the same hardware component. If there are multiple DPUs, then it is possible to create a DMA instance per DPU, providing each instance with a device from a different DPU.

To access memory that is not local (from the host to the DPU or vice versa), the DPU side of the application must select a device with an appropriate representor. See DOCA Core Device Representor Discovery.

The device must stay valid for as long as the DMA instance is not destroyed.

14.4.4.5.1.2 Memory Buffers

The memory copy task requires two DOCA buffers containing the destination and the source. Depending on the allocation pattern of the buffers, refer to the table in the "Inventory Types" section. To find what kind of memory is supported, refer to the table in section "Buffer Support".

Buffers must not be modified or read during the memory copy operation.

14.4.4.6 Configuration Phase

To start using the library, users must go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context, to allow execution of tasks and retrieval of events.

14.4.4.6.1 Configurations

The context can be configured to match the application use case.

To find if a configuration is supported, or what the min/max value for it is, refer to section "Device Support".

14.4.4.6.1.1 Mandatory Configurations

These configurations are mandatory and must be set by the application before attempting to start the context:

- At least one task/event type must be configured. See configuration of tasks and/or events in sections "Tasks" and "Events" respectively for information.
- A device with appropriate support must be provided upon creation

14.4.4.6.2 Device Support

DOCA DMA requires a device to operate. To picking a device, refer to "DOCA Core Device Discovery".

As device capabilities may change (see DOCA Core Device Support), it is recommended to select your device using the following method:

- `doca_dma_cap_task_memcpy_is_supported`

Some devices can allow different capabilities as follows:

- The maximum number of tasks
- The maximum buffer size

14.4.4.6.3 Buffer Support

Tasks support buffers with the following features:

Buffer Type	Source Buffer	Destination Buffer
Local mmap buffer	Yes	Yes
mmap from PCIe export buffer	Yes	Yes
mmap From RDMA export buffer	No	No
Linked list buffer	Yes	No

14.4.4.7 Execution Phase

This section describes execution on CPU using DOCA Core Progress Engine.

14.4.4.7.1 Tasks

DOCA DMA exposes asynchronous tasks that leverage the DPU hardware according to the DOCA Core architecture. See DOCA Core Task.

14.4.4.7.1.1 Memory Copy Task

The memory copy task allows copying memory from one location to another. Using buffers as described in Buffer Support.

Task Configuration

Description	API to set the configuration	API to query support
Enable the task	`doca_dma_task_memcpy_set_conf`	`doca_dma_cap_task_memcpy_is_supported`
Number of tasks	`doca_dma_task_memcpy_set_conf`	`doca_dma_cap_get_max_num_tasks`
Maximal buffer size	-	`doca_dma_cap_task_memcpy_get_max_buf_size`
Maximum buffer list size	-	`doca_dma_cap_task_memcpy_get_max_buf_list_len`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer that points to the memory to be copied	Only the data residing in the data segment is copied
Destination buffer	Buffer that points to where memory is copied	The data is copied to the tail segment extending the data segment

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task is completed successfully:

- The data is copied form source to destination
- The destination buffer data segment is extended to include the copied data

Task Completion Failure

If the task fails midway:

- The context may enter stopping state, if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, then the source and destination should not be read/written to
- Source and destination must not overlap
- Other limitations are described in DOCA Core Task

14.4.4.7.2 Events

DOCA DMA exposes asynchronous events to notify on changes that happen unexpectedly, according to DOCA Core architecture.

The only event DMA exposes is common events as described in DOCA Core Event.

14.4.4.8 State Machine

The DOCA DMA library follows the Context state machine as described in DOCA Core Context State Machine.

The following section describes how to move states and what is allowed in each state.

14.4.4.8.1 Idle

In this state it is expected that application:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to section "Configurations"
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop after making sure all tasks have been freed
Stopping	Call progress until all tasks are completed and freed

14.4.4.8.2 Starting

This state cannot be reached.

14.4.4.8.3 Running

In this state it is expected that application:

- Allocates and submits tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocating a previously configured task
- Submitting a task
- Calling stop

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

14.4.4.8.4 Stopping

In this state it is expected that application:

- Calls progress to complete all inflight tasks (tasks complete with failure)
- Frees any completed tasks

Allowed operations:

- Call progress

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and fatal error occurs
Running	Call stop without freeing all tasks

14.4.4.9 Alternative Datapath Options

DOCA DMA allows data path to be run on the CPU or GPU.

ⓘ For the CPU data path, see Execution Phase.

14.4.4.9.1 GPU Datapath

DOCA offers the DOCA GPUNetIO library which provides a programming model for offloading the orchestration of the communication to a GPU CUDA kernel.

The user may run a DMA operation on the GPU data path by configuring the DOCA DMA context used by the application in the following manner:

1. Obtain DOCA CTX by calling `doca_dma_as_ctx()` .
2. Set the datapath of the context to GPU by calling `doca_ctx_set_datapath_on_gpu()` . For additional information, refer to DOCA Core Alternative Data Path.
3. Finish context configuration and start the context by calling `doca_ctx_start()` . For additional information, refer to DOCA Core Context.

After configuring the datapath, the user can obtain a GPU handle for the DOCA RDMA context by calling `doca_dma_get_gpu_handle()` . The GPU handle must be passed to a GPU CUDA kernel so the DOCA GPUNetIO CUDA device functions can execute datapath operations. For additional information, refer to section "GPU Functions - RDMA" under DOCA GPUNetIO library documentation.

14.4.4.10 DOCA DMA Samples

This section describes DOCA DMA samples based on the DOCA DMA library.

The samples illustrate how to use the DOCA DMA API to do the following:

- Copy contents of a local buffer to another buffer
- Use DPU to copy contents of buffer on the host to a local buffer

ⓘ All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

14.4.4.10.1 Running the Samples

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample:

```
cd /opt/mellanox/doca/samples/doca_dma/dma_local_copy
meson /tmp/build
ninja -C /tmp/build
```

   The binary `doca_dma_local_copy` is created under `/tmp/build/` .
3. Sample (e.g., `doca_dma_local_copy`) usage:

```
Usage: doca_<sample_name> [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                              Print a help synopsis
  -v, --version                           Print program version information
  -l, --log-level                         Set the (numeric) log level for the program
<10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                       Parse all command flags from an input json file

Program Flags:
  -p, --pci_addr <PCI-ADDRESS>            PCI device address
  -t, --text                              Text to DMA copy
```

4. For additional information per sample, use the `-h` option:

```
/tmp/build/<sample_name> -h
```

14.4.4.10.2 Samples

14.4.4.10.2.1 DMA Local Copy

This sample illustrates how to locally copy memory with DMA from one buffer to another on the DPU. This sample should be run on the DPU.

The sample logic includes:

1. Locating DOCA device.
2. Initializing needed DOCA core structures.
3. Populating DOCA memory map with two relevant buffers.
4. Allocating element in DOCA buffer inventory for each buffer.
5. Initializing DOCA DMA memory copy task object.
6. Submitting DMA task.
7. Handling task completion once it is done.
8. Checking task result.
9. Destroying all DMA and DOCA core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_dma/dma_local_copy/dma_local_copy_sample.c`
- `/opt/mellanox/doca/samples/doca_dma/dma_local_copy/dma_local_copy_main.c`
- `/opt/mellanox/doca/samples/doca_dma/dma_local_copy/meson.build`

14.4.4.10.2.2 DMA Copy DPU

> This sample should run only after DMA Copy Host is run and the required configuration files (descriptor and buffer) have been copied to the DPU.

This sample illustrates how to copy memory (which contains user defined text) with DMA from the x86 host into the DPU. This sample should be run on the DPU.

The sample logic includes:

1. Locating DOCA device.
2. Initializing needed DOCA core structures.
3. Reading configuration files and saving their content into local buffers.
4. Allocating the local destination buffer in which the host text is to be saved.

5. Populating DOCA memory map with destination buffer.
6. Creating the remote memory map with the export descriptor file.
7. Creating memory map to the remote buffer.
8. Allocating element in DOCA buffer inventory for each buffer.
9. Initializing DOCA DMA memory copy task object.
10. Submitting DMA task.
11. Handling task completion once it is done.
12. Checking DMA task result.
13. If the DMA task ends successfully, printing the text that has been copied to log.
14. Printing to log that the host-side sample can be closed.
15. Destroying all DMA and DOCA core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_dma/dma_copy_dpu/dma_copy_dpu_sample.c`
- `/opt/mellanox/doca/samples/doca_dma/dma_copy_dpu/dma_copy_dpu_main.c`
- `/opt/mellanox/doca/samples/doca_dma/dma_copy_dpu/meson.build`

14.4.4.10.2.3 DMA Copy Host

> ⚠ This sample should be run first. It is user responsibility to transfer the two configuration files (descriptor and buffer) to the DPU and provide their path to the DMA Copy DPU sample.

This sample illustrates how to allow memory copy with DMA from the x86 host into the DPU. This sample should be run on the host.

The sample logic includes:

1. Locating DOCA device.
2. Initializing needed DOCA core structures.
3. Populating DOCA memory map with source buffer.
4. Exporting memory map.
5. Saving export descriptor and local DMA buffer information into files. These files should be transferred to the DPU before running the DPU sample.
6. Waiting until DPU DMA sample has finished.
7. Destroying all DMA and DOCA core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_dma/dma_copy_host/dma_copy_host_sample.c`
- `/opt/mellanox/doca/samples/doca_dma/dma_copy_host/dma_copy_host_main.c`
- `/opt/mellanox/doca/samples/doca_dma/dma_copy_host/meson.build`

14.4.5 DOCA Comch

> ⓘ DOCA Comm Channel API will be deprecated in the next DOCA release (2.9.0).

DOCA Comch API introduces features such as high-performance data path over the consumer-producer API, as well as working with DOCA progress engine and other standard DOCA Core objects.

⚠ DOCA Comch does not support event-triggered completions.

14.4.5.1 DOCA Comch – New

This guide provides instructions on building and developing applications that require communication channels between the x86 host and the BlueField Arm cores.

14.4.5.1.1 Introduction

DOCA Comch provides a communication channel between client applications on the host and servers on the BlueField Arm.

Benefits of using DOCA Comch:

- Security – the communication channel is isolated from the network
- Network independent – the state of the communication channel does not depend on the state and configuration of the network
- Ease of use

DOCA Comch provides two different data path APIs:

- Basic DOCA Comch send/receive for control messages
- High bandwidth, low latency, zero-copy, multi-producer, multi-consumer API

The following table summarizes the differences between the two data path APIs:

Features	Basic Send/Receive	Fast Path (using doca_comch_consumer/ doca_comch_producer)
Zero-copy	No	Yes
Takes network bandwidth	Yes	No
Isolated from network	Yes	Yes
Max msg size	Fixed	1GB or more (depends on hardware cap)
Multi-threaded	Safe for a single thread	Allows creation of consumer/producers per thread.
Multi-consumer	No	Yes
Multi-producer	Yes – allows multiple clients per server	Yes – allow multiple producers/ consumers per connection
Requires `doca_mmap` and `doca_buf`	No	Yes

14.4.5.1.2 Prerequisites

This library follows the architecture of a DOCA Core Context, it is recommended to read the following sections before:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem (fast path only)

14.4.5.1.3 Changes From Previous Release

14.4.5.1.3.1 Modified

Function name and return type changes

- `doca_error_t doca_comch_server_get_device_rep(const struct doca_comch_server *comch_server, struct doca_dev_rep **rep)`
  - DOCA 2.7 version:
    - `doca_error_t doca_comch_server_get_device_repr(const struct doca_comch_server *comch_server, struct doca_dev_rep **repr)`
- `doca_comch_server_event_connection_status_changed_register(server, server_connection_status_callback, server_connection_status_callback)`
  - DOCA 2.7 version:
    - `doca_comch_server_event_connection_register(server, server_connection_status_callback, server_connection_status_callback)`
- `doca_error_t doca_comch_consumer_set_dev_max_num_recv(struct doca_comch_consumer *consumer, uint32_t dev_num_recv)`
  - DOCA 2.7 version:
    - `doca_comch_consumer_set_dev_num_recv(struct doca_comch_consumer *consumer, uint32_t dev_num_recv)`
- `doca_error_t doca_comch_producer_set_dev_max_num_send(struct doca_comch_producer *producer, uint32_t dev_num_send)`
  - DOCA 2.7 version:
    - doca_comch_producer_set_dev_num_send(struct doca_comch_producer *producer, uint32_t dev_num_send)
- `doca_error_t doca_comch_consumer_completion_get_max_num_consumers(const struct doca_comch_consumer_completion *consumer_comp, uint32_t *max_num_consumers)`
  - DOCA 2.7 version:
    - `doca_comch_consumer_completion_get_max_num_consumers(struct doca_comch_consumer_completion *consumer_comp, uint32_t *max_num_consumers)`
- `doca_error_t doca_comch_consumer_completion_get_max_num_consumers(const struct doca_comch_consumer_completion *consumer_comp, uint32_t *max_num_consumers)`
  - DOCA 2.7 version:
    - `doca_comch_consumer_completion_get_max_num_consumers(struct doca_comch_consumer_completion *consumer_comp, uint32_t *max_num_consumers)`
- `doca_error_t doca_comch_consumer_completion_get_max_num_recv(const struct doca_comch_consumer_completion *consumer_comp, uint32_t *max_num_recv)`
  - DOCA 2.7 version:
    - `doca_comch_consumer_completion_get_max_num_recv(struct doca_comch_consumer_completion *consumer_comp, uint32_t *max_num_recv)`

Adding const to getter API functions

- `doca_comch_consumer_task_post_recv_get_buf(const struct doca_comch_consumer_task_post_recv *task)`
  - DOCA 2.7 version:
    - `doca_comch_consumer_task_post_recv_get_buf(struct doca_comch_consumer_task_post_recv *task)`
- `doca_comch_consumer_task_post_recv_get_producer_id(const struct doca_comch_consumer_task_post_recv *task)`
  - DOCA 2.7 version:

        - `doca_comch_consumer_task_post_recv_get_producer_id(struct doca_comch_consumer_task_post_recv *task)`
- `const uint8_t *doca_comch_consumer_task_post_recv_get_imm_data(const struct doca_comch_consumer_task_post_recv *task)`
    - DOCA 2.7 version:
        - `uint8_t *doca_comch_consumer_task_post_recv_get_imm_data(struct doca_comch_consumer_task_post_recv *task)`
- `doca_comch_consumer_task_post_recv_get_imm_data_len(const struct doca_comch_consumer_task_post_recv *task)`
    - DOCA 2.7 version:
        - `doca_comch_consumer_task_post_recv_get_imm_data_len(struct doca_comch_consumer_task_post_recv *task)`
- `doca_comch_producer_task_send_get_buf(const struct doca_comch_producer_task_send *task)`
    - DOCA 2.7 version:
        - `doca_comch_producer_task_send_get_buf(struct doca_comch_producer_task_send *task)`
- `doca_comch_producer_task_send_get_consumer_id(const struct doca_comch_producer_task_send *task)`
    - DOCA 2.7 version:
        - `doca_comch_producer_task_send_get_consumer_id(struct doca_comch_producer_task_send *task)`
- `doca_comch_producer_task_send_get_imm_data(const struct doca_comch_producer_task_send *task)`
    - DOCA 2.7 version:
        - `doca_comch_producer_task_send_get_imm_data(struct doca_comch_producer_task_send *task)`
- `doca_comch_producer_task_send_get_imm_data_len(const struct doca_comch_producer_task_send *task)`
    - DOCA 2.7 version:
        - `doca_comch_producer_task_send_get_imm_data_len(struct doca_comch_producer_task_send *task)`

14.4.5.1.4 Environment

DOCA Comch based applications can run either on the host machine or on the NVIDIA BlueField Arm.

Sending messages between the host and BlueField Arm can only be run with a BlueField configured with a mode as described in NVIDIA BlueField Modes of Operation.

For basic DOCA Comch send and receive, the following configuration is required:

- `doca_comch_server` context must run on the BlueField Arm cores
- `doca_comch_client` context must run on the host machine

> ⚠ Producer and consumer objects can run on both the host and BlueField Arm cores. However, there must be a valid client/server connection already established on the channel.

14.4.5.1.5 Architecture

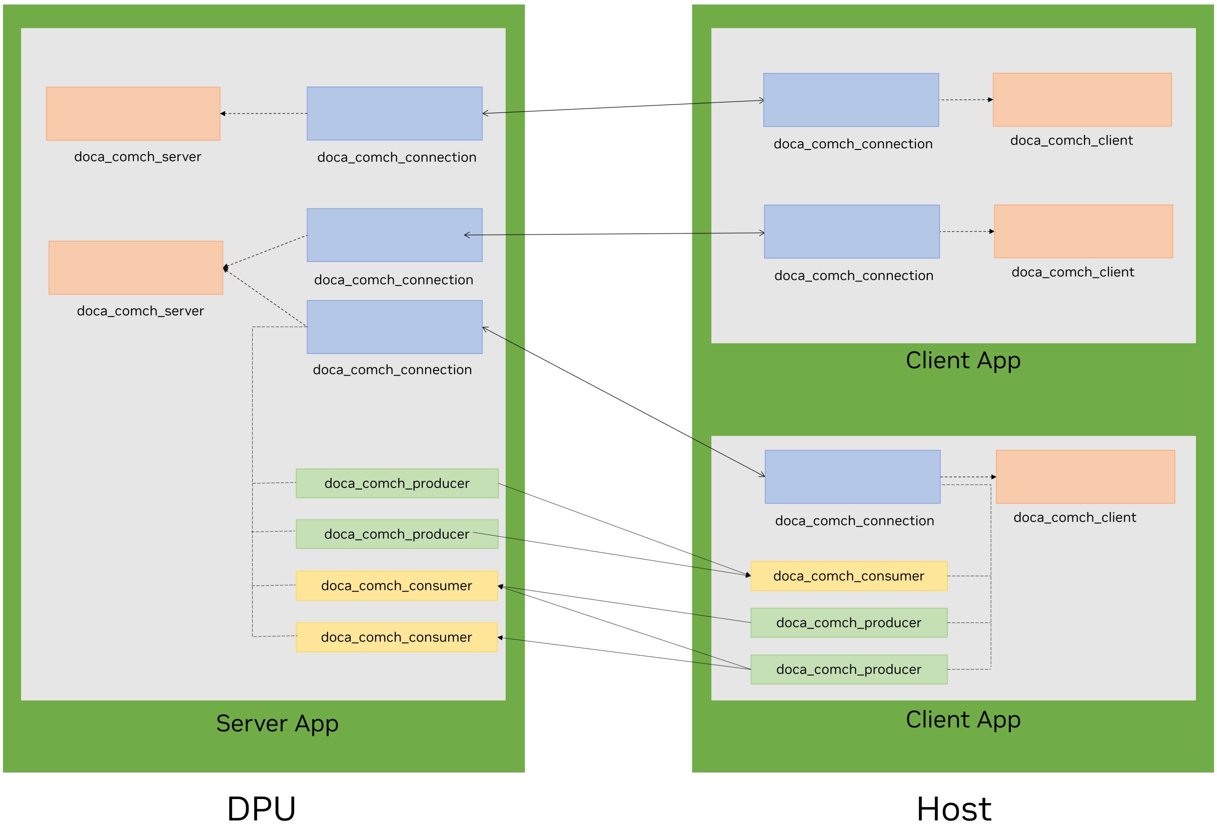
DOCA Comch is comprised of four DOCA Core Contexts. All DOCA Comch contexts leverage DOCA Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

A `doca_comch_server` context runs on the BlueField Arm and listens for incoming connections from the host side. Such host side connections are initiated by a `doca_comch_client` context.

Servers can receive connections from multiple clients in parallel, however, a client can only connect with one server. An established 1-to-1 connection between a client and a server is represented by a `doca_comch_connection`.

Once an established connection exists between a client and a server, the `doca_comch_producer` and `doca_comch_consumer` contexts can be used to run fast path channels.

The following diagram provides examples of the contexts use:



14.4.5.1.5.1 Objects

	Description	Location	Scope
`doca_comch_server`	Allows applications on the BlueField Arm cores to listen on a specific server name and accept new incoming connection from the host	BlueField Arm only	Per host PCIe function (`doca_dev` + `doca_dev_rep`)
`doca_comch_client`	Allows client applications to connect to a specific server name on the BlueField Arm cores	Host only	Per host PCIe function (`doca_dev`)
`doca_comch_connection`	A connection handle created on the client side or the server side when a new connection is established. This handle is used to send/receive messages or to create `doca_comch_consumer`s and `doca_comch_producer`s.	BlueField Arm and host	Per client server pair
`doca_comch_producer`	A handle for a FIFO-like send queue that provides a zero-copy API to send messages to a specific `doca_comch_consumer` on the same `doca_comch_connection`. Multiple `doca_comch_producer`s can be created per `doca_comch_connection`.	BlueField Arm and host	Per `doca_comch_connection`
`doca_comch_consumer`	A handle for a FIFO-like receive queue that provides a zero-copy API to receive messages from a `doca_comch_producer`	BlueField Arm and host	Per `doca_comch_connection`

14.4.5.1.5.2 Security Considerations

- DOCA Comch guarantees:
    - The client is connected to the server by providing the exact server name on the client side
    - Only clients on the PF/VF/SF represented by the `doca_dev_rep` provided upon server creation can connect to the server
    - The connection requests and data path are isolated from the network
- DOCA Comch does not provide security at the application level:
    - It is up to the user to implement application-level security and verify the identity of the client application
    - A server handles applications from a single PF/VF/SF. If a server application detects a compromised client application, the server app should consider all clients (from that PF/VF/SF) compromised.

14.4.5.1.5.3 Initialization Flow

doca_comch_server Initialization Flow

1. A `doca_comch_server` is created on a specific `doca_dev` and a specific `doca_dev_rep`.
2. A `doca_comch_server` must have a unique name per `doca_dev/doca_dev_rep` (i.e., two servers on the same `doca_dev` and `doca_dev_rep` cannot have the same name).
3. Once `doca_ctx_start()` is called, the `doca_comch_server` can start receiving new connection requests.
4. For the `doca_comch_server` to process new connection requests and messages, the user must periodically call `doca_pe_progress()`.
5. When a new connection request arrives, `doca_comch_server` calls the connection request handler function and passes a `doca_comch_connection` object.

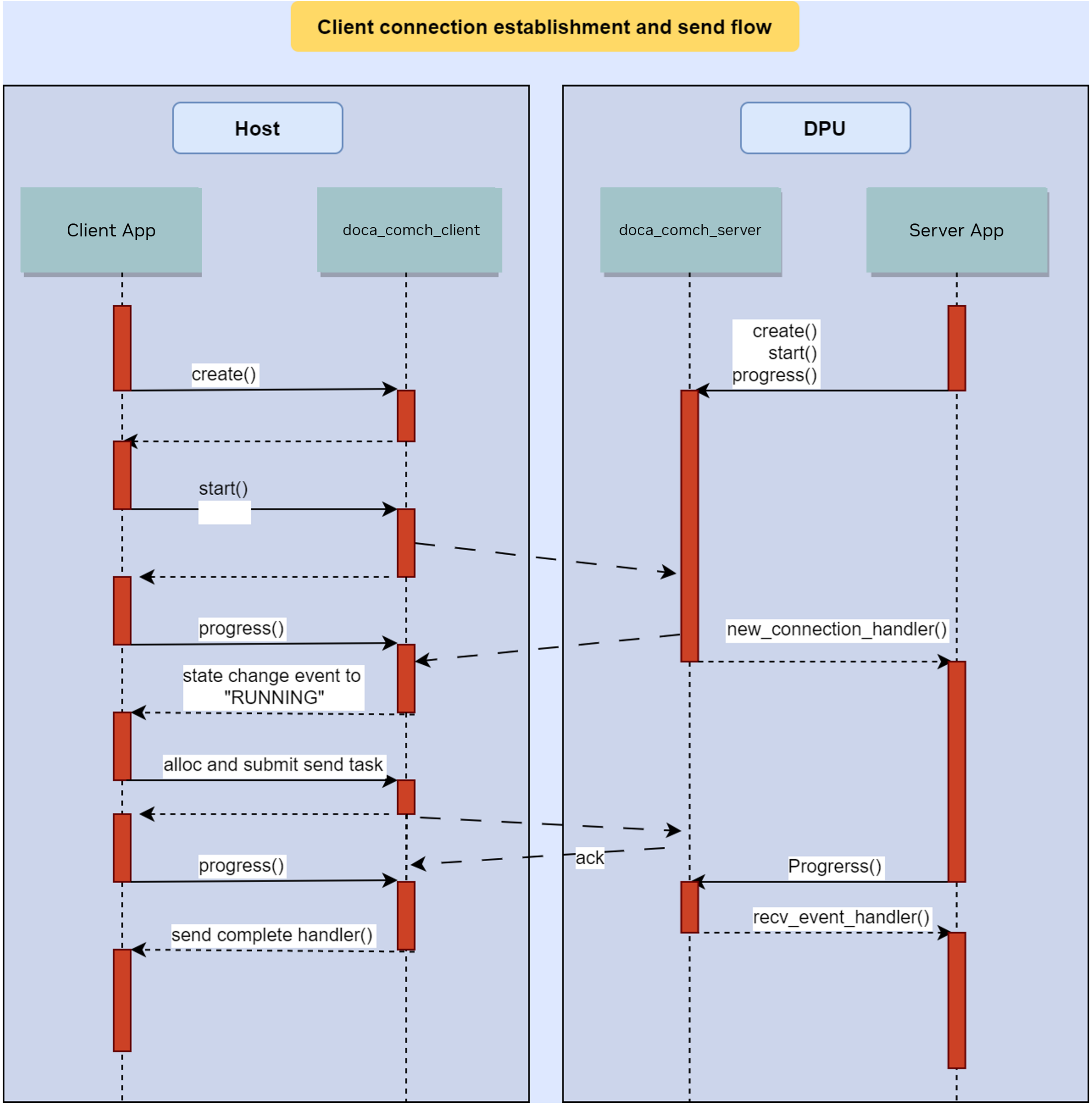
The server can now send and receive messages on the connection represented by `doca_comch_connection`.

doca_comch_client Initialization Flow

1. A `doca_comch_client` is created on a specific `doca_dev` is targeting a specific `doca_comch_server`.
2. Once `doca_ctx_start()` is called, `doca_comch_client` asynchronously tries to connect to the server.
3. To establish the connection and receive messages, the user must periodically call `doca_pe_progress()`.
4. When the connection is established, `doca_comch_client` calls the state change callback indicating state change to "RUNNING".

The client can now send a receive messages.

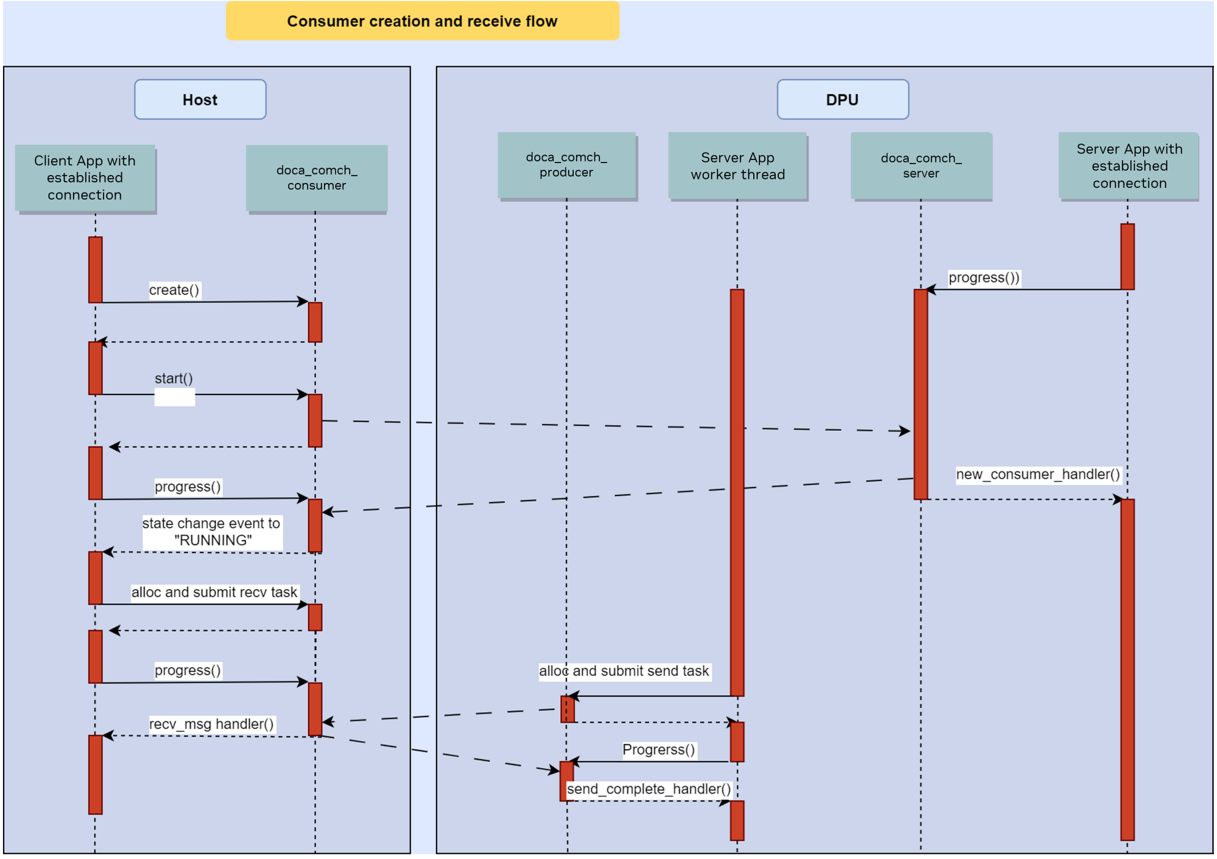
The following diagram describes the initialization of a basic client/server connection on DOCA Comch:



doca_comch_consumer Initialization Flow

1. A `doca_comch_consumer` is created on a specific `doca_comch_connection`.
2. `doca_pe_progress()` must be periodically called on the client/server PE to allow registration of the consumer.
3. After the `doca_comch_consumer` moves to "RUNNING" state:
   a. `doca_comch_consumer` notifies its existence to the peer (invoking a new consumer event).
   b. The application can start posting receive tasks.
   c. A `doca_comch_producer` on the peer side can start sending messages to that consumer.

The initialization flow is described in the following diagram:



14.4.5.1.5.4 Teardown Flow

The teardown flow must be executed in the following order, otherwise errors may occur.

Disconnecting Specific Connection

The proper disconnection process for a specific connection consists of the following steps:

1. Stop all consumers and producers linked to the connection.
2. Server/client:
   a. For server, a connection can be disconnected using `doca_comch_server_disconnect()` . If there are any active producers/consumers linked to the connection, the disconnect would fail. A disconnection notifies the client and initiates teardown on that side too.
   b. For client, since there is only one connection at any given time, the connection can be disconnected by calling `doca_ctx_stop()` . If there are any active producers/ consumers, the command would fail. Stopping the client context notifies the server of the disconnection and causes a disconnection of the connection on it.

Tearing Down DOCA Comch

The proper teardown for a DOCA Comch context consists of the following:

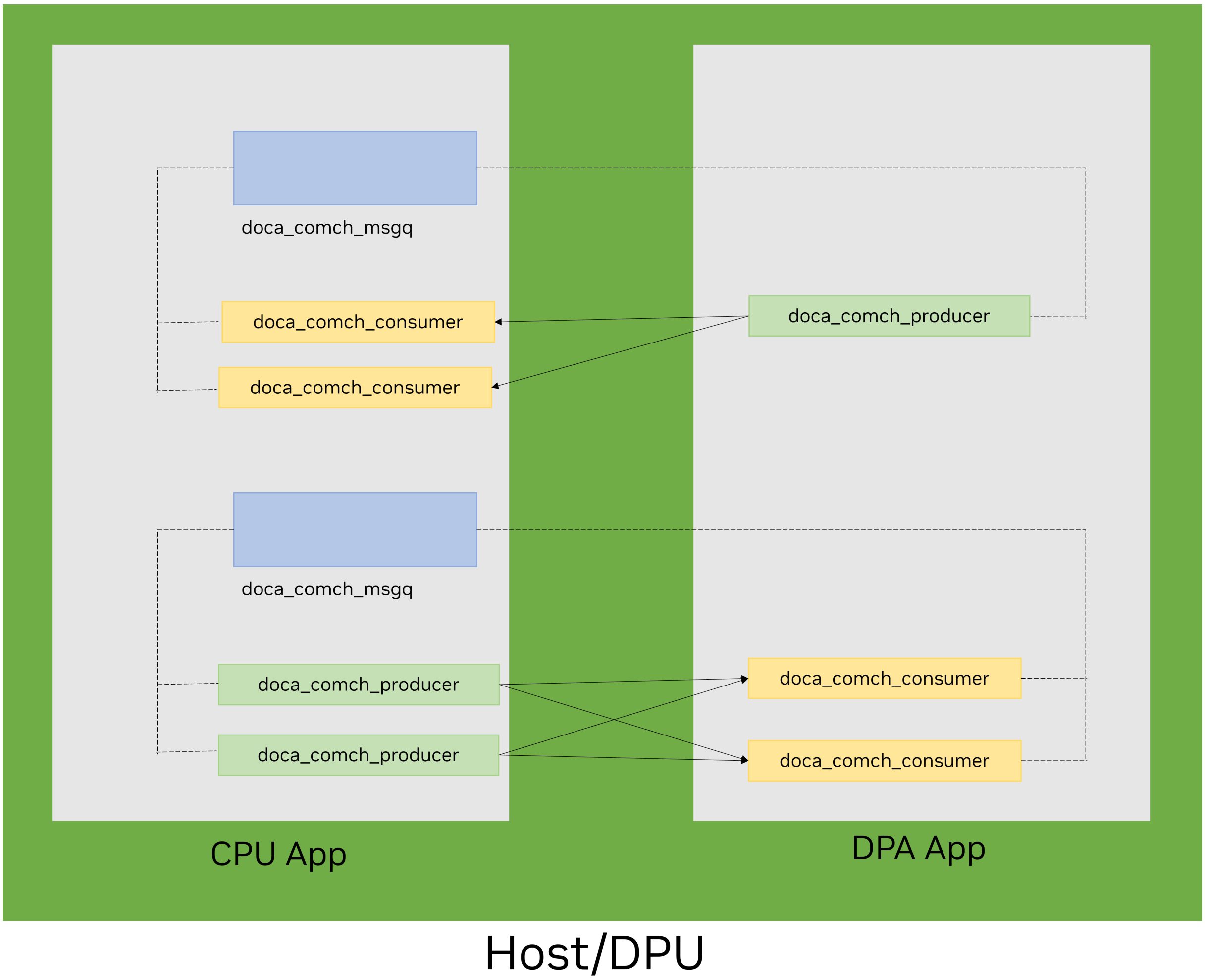
1. Stop all consumers and producers linked to the context.
2. Call `doca_ctx_stop()` . If there are any active connections, they would all be disconnected. If there are any active consumers/producers, the command would fail. Disconnecting/

stopping the context informs all active peers of the disconnection, and causes teardown (on clients) or disconnection (on server). Calling `doca_ctx_stop()` successfully moves the context to "stopping" state.

3. After moving to stopping state, `doca_pe_progress()` must be called until the context moves to idle state.

14.4.5.1.5.5 MsgQ (DPA Communication)

DOCA Comch MsgQ leverages the existing consumer/producer model to allow communication between host/BlueField and DPA.



Since communication between the host/BlueField and DPA is local, there is no need to create a server, client, or connection. Instead the user can create a MsgQ and use it to create producers and consumers directly.

When creating a consumer/producer using the MsgQ, it becomes possible to use them in the DPA application as well as the CPU application:

- The CPU application can utilize existing consumer/producer APIs for communication
- The DPA application has a different set of APIs that are usable within a DPA application

Communication Direction

Every instance of a MsgQ can only support a single communication direction as follows:

- Communication from host/BlueField to DPA
  - This direction may be specified using `doca_comch_msgq_set_dpa_consumer`
  - Consumers created from this MsgQ are referred to as DPA consumers, while producers are CPU producers
- Communication from DPA to host/BlueField
  - This direction may be specified using `doca_comch_msgq_set_dpa_producer`
  - Consumers created from this MsgQ are referred to as CPU consumers, while producers are DPA producers

To support bidirectional communication in an application, the user has to create 2 MsgQ instances, as shown in the above diagram.

14.4.5.1.6 Configuration Phase

To start using the library, users must go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context to allow execution of tasks and retrieval of events.

14.4.5.1.6.1 Configurations

The context can be configured to match the application use case.

To find out if a certain configuration is supported, or what the min/max value for it is, refer to Device Support.

Mandatory Configurations

These configurations are mandatory and must be set by the application before attempting to start the context:

- For a basic send/receive client or server:
  - A send task callback
  - A receive event callback
  - A device with appropriate support must be provided on creation
  - A valid server name must be provided on creation (for clients this is the server to connect to)
  - A connection event callback (server only)
- For fast path producer or consumer:
  - A device with appropriate support must be provided on creation
  - An established client to server connection must be provided on creation
  - A `doca_mmap` with PCIe read/write permissions of where data should be received must be provided on creation (consumer only)
  - A post receive task callback (consumer only)
  - A send task callback (producer only)
  - A new consumer callback (triggered upon creation/destruction of a remove consumer)
- For MsgQ fast path producer or consumer:
  - A started MsgQ must be provided on creation

- A DPA instance must be provided (DPA consumer/producer only)
- A DPA consumer completion context must be connected (DPA consumer only)
- A DPA completion context must be attached (DPA producer only)
- A post receive task callback (CPU consumer only)
- The number of receive operations (DPA consumer only)
- A send task callback (CPU producer only)
- The number of send operations (DPA producer only)

Optional Configurations

The following configurations are optional, if they are not set then a default value will be used:

For basic send/receive client:

- `doca_comch_(server|client)_set_max_msg_size` - set the maximum size of message that can be sent. If set, it must be matching between server and client.
- `doca_comch_(server|client)_set_recv_queue_size` - set the size of the queue to receive new messages on

For fast path consumers:

- `doca_comch_consumer_set_imm_data_len` - set the length of immediate data that a consumer can receive.

14.4.5.1.6.2 Device Support

DOCA Comch requires a device to operate. For instructions on picking a device, see DOCA Core Device Discovery.

As device capabilities are subject to change (see DOCA Core Device Support), it is recommended to select a device using the following methods:

- For basic client and server:
  - `doca_comch_cap_server_is_supported`
  - `doca_comch_cap_client_is_supported`
- For extended fast path functionality:
  - `doca_comch_producer_cap_is_supported`
  - `doca_comch_consumer_cap_is_supported`

Some devices can allow different capabilities as follows:

- The maximum length server name
- The maximum message size
- The maximum receive queue length
- The maximum clients that can connect to a server
- The maximum number of send tasks or post receive tasks
- The maximum buffer length for fast path
- The maximum immediate data supported by a fast path consumer

14.4.5.1.6.3 Buffer Support

Basic send and receive between a client and server does not use DOCA buffers and so has no restrictions on buffer type.

- For producers, supplied buffers need only be from a local mmap
- For consumers, post receive buffers are required to be from a PCIe export mmap

> ⚠ Chained buffers are not supported in DOCA Comch.

14.4.5.1.7 Execution Phase

This section describes execution on CPU using DOCA Core Progress Engine. For additional execution environments, refer to section "Alternative Datapath Options".

14.4.5.1.7.1 Tasks

DOCA Comch exposes asynchronous tasks that leverage the BlueField hardware according to DOCA Core architecture.

Control Channel Send Task

This task allows the sending of messages between connected client and server objects.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Number of tasks	`doca_comch_server_task_send_set_conf` `doca_comch_client_task_send_set_conf`	`doca_comch_cap_get_max_send_tasks`
Maximal message size	`doca_comch_server_set_max_msg_size` `doca_comch_client_set_max_msg_size`	`doca_comch_server_get_max_msg_size` `doca_comch_client_get_max_msg_size`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Peer	Established client/server connection	-
Message	Data string to send to remote client/server	The is no requirement for the message to be in DOCA mmap registered memory
Length	Number of bytes in the message	Must not exceed configured max size

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully:

- The message is delivered to the connections remote client/server
- A receive event is triggered on the remote side

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The message is not delivered to the remote side

Task Limitations

- The operation is not atomic
- Once the task has been submitted, then the message should not be updated
- Other limitations are described in DOCA Core Task

Consumer Post Receive Task

This task allows consumer objects to publish buffers which are available for remote producers to write to.

> ⚠ A Post Receive task may have a NULL buffer if it only wishes to receive immediate data.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_comch_consumer_task_post_recv_set_conf`	`doca_comch_consumer_cap_is_supported`
Number of tasks	`doca_comch_consumer_task_post_recv_set_conf`	`doca_comch_consumer_cap_get_max_num_tasks`
Maximal buffer size	-	`doca_comch_consumer_cap_get_max_buf_size`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Buffer	Buffer that the consumer can receive data on	Data is appended to the tail of the buffer ⓘ Buffers `doca_mmap` must have `DOCA_ACCESS_FLAG_PCI_READ_WRITE` flag set.

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

The task only completes once a producer has written to the advertised buffer (or immediate data, or both), not when the post receive has completed.

Upon successful completion, the buffer contains the data written by the producer and its length is updated appropriately.

Task Completion Failure

Task failure occurs if a buffer has not been successfully posted to receive data.

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- Producers are not aware of the buffer so would not write to it

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the buffer should not be read/written to
- Buffer must come from memory with PCIe read/write access
- Chained buffer lists are not supported
- MsgQ consumer does not support providing `doca_buf`, and can only receive immediate data
- Other limitations are described in DOCA Core Task

Producer Send Task

This task allows producer objects to copy buffers for use by remote consumers.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_comch_producer_task_send_set_conf`	`doca_comch_producer_cap_is_supported`
Number of tasks	`doca_comch_producer_task_send_set_conf`	`doca_comch_producer_cap_get_max_num_tasks`
Maximal buffer Size	-	`doca_comch_producer_cap_get_max_buf_size`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Buffer	Buffer that should be copied to a consumer	Only the data residing in the data segment is copied
Immediate data	Short byte array to add to the post receive completion entry	This is not a zero copy operation but does improve latency for small payloads

Name	Description	Notes
Immediate data length	Length of data immediate data pointed to	Maximum length is determined/set by individual consumers
Consumer ID	Identifier for the target consumer to write to	Active consumers and their IDs are advertised through consumer events

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task is completed successfully:

- The data is copied form the buffer to the next free buffer posted by the given consumer
- Consumers process buffers from a given consumer in the order they are sent

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination memory may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the buffer should not be read/written to
- The buffer length should not be greater than consumer post receive buffers (an invalid value is returned otherwise)
- MsgQ producer does not support providing `doca_buf`, and can only send immediate data
- All limitations described in DOCA Core Task

14.4.5.1.7.2 Events

DOCA Comch exposes asynchronous events to notify about changes that happen out of the blue, according to the DOCA Core architecture. See DOCA Core Event.

Common events as described in DOCA Core Event.

Control Channel Receive Event

This event triggers whenever a remote client/server has sent a message to the local client/server object.

Event Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	`doca_comch_server_event_msg_recv_register` `doca_comch_client_event_msg_recv_register`	-

Event Trigger Condition

The event is triggered when a remote message is received on any currently active connection associated with the client or server.

Event Output

Upon event detection, the registered callback is triggered, passing the following parameters:

- A pointer to the message data

  > ⓘ The data is only valid in the context of the callback.

- The length in bytes of the message
- The active connection on which the message was received

Connection Status Changed Event (Server Only)

This event provides asynchronous updates on the state of any connections associated with a server.

> ⚠ A client object can only connect to a single server, so its connection state can be tracked through its `doca_ctx` state and the generic `doca_ctx_set_state_changed_cb` function.

Event Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	`doca_comch_server_event_connection_status_changed_register`	-

Event Trigger Condition

The event is triggered when a new connection is either established or a current connection disconnected on a server.

Event Output

Separate callbacks are registered for connection or disconnection events with the appropriate one triggered based on the specific event.

Both callbacks contain a Boolean indicating if the connection or disconnection was successful.

Consumer Event

This event indicates that a new consumer object has been created or an existing consumer object has been destroyed.

Event Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	`doca_comch_server_event_consumer_register` `doca_comch_client_event_consumer_register`	-

Event Trigger Condition

The event is triggered whenever a new consumer is created or a current consumer destroyed on the remote side of an established DOCA Comch connection.

Event Output

The event hits a separate callback for either the creation or destruction of a consumer.

Callback parameters include:

- The established DOCA Comch connection on which the consumer is connected (on the remote side)
- The ID of the consumer (a unique value per Comch connection)

14.4.5.1.8 State Machine

The DOCA Comch library follows the Context state machine described in DOCA Core Context State Machine.

The following section describes how to move to the state and what is allowed in each state.

14.4.5.1.8.1 Idle

In this state it is expected that the application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to Configurations
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop after making sure all tasks have been freed
Stopping	Call progress until all tasks are completed and freed

14.4.5.1.8.2 **Starting**

In this state it is expected that the application will:

- Call progress to allow transition to next state (e.g., when a connection attempt completes)

Allowed operations:

- Call progress

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

14.4.5.1.8.3 Running

In this state, it is expected that the application:

- Allocates and submit tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocate a previously configured task
- Submit an allocated task
- Call stop

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration
Starting	Call progress until context state transitions

14.4.5.1.8.4 Stopping

In this state, it is expected that the application will:

- Free any completed tasks

Allowed operations:

- Allocate previously configured task
- Submit an allocated task
- Call stop

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and fatal error occurs
Running	Call stop without freeing all tasks

14.4.5.1.9 Alternative Datapath Options

DOCA Comch can be run on as part of DPA data path, using the MsgQ.

14.4.5.1.9.1 DPA

Using the MsgQ it is possible to create consumer/producer on the DPA. They follow the definition described in DOCA Core DPA.

Since these objects can be used in DPA, they have DPA APIs that can be used to perform the data path operations expanded on in the following subsections.

Consumer Ack

The `doca_dpa_dev_comch_consumer_ack` API prepares the DPA consumer to receive a number of immediate messages from CPU producers.

Configuration

Description	API to Set the Configuration	API to Query Support
Queue Size	`doca_comch_consumer_set_dev_max_num_recv`	-

Input

Name	Description	Notes
Number of Messages	A number describing how many additional immediate messages this consumer can receive	Must not exceed the queue size

Completion

Whenever a message is received from the CPU producer a completion element is generatedand can be polled using `doca_dpa_dev_comch_consumer_get_completion`.

Using the generated completion, it is possible to get the following outputs:

Name	Description	Notes
Immediate Message	A pointer to the immediate message that the CPU producer sent	The message lifetime is the same as the completion element lifetime. That is, once the completion is acked using `doca_dpa_dev_comch_consumer_completion_ack`, the pointer is no longer valid. To retain the message past the completion lifetime, the user must copy the contents of the message.
Immediate Message Length	The length in bytes of the immediate message that the CPU producer sent	
Producer ID	The ID of the CPU producer that sent the message	User can find the IDs of each producer by using `doca_comch_producer_get_id`

Limitations

- The maximal immediate message size is 32 bytes

Producer Post Send Immediate Only

The `doca_dpa_dev_comch_producer_post_send_imm_only` API sends an immediate message to the CPU consumer. Once the message arrives at the CPU consumer side, the CPU consumer receive task completes.

The CPU producer must have posted a receive task prior to this. The user can verify if the consumer can receive the message using `doca_dpa_dev_comch_producer_is_consumer_empty`. Note, however, that this may add overhead.

Configuration

Description	API to Set the Configuration	API to Query Support
Queue Size	`doca_comch_producer_set_dev_max_num_send`	-

Input

Name	Description	Notes
Immediate Message	Short byte array to be sent to the CPU consumer	This is not a zero copy operation but does improve latency for small payloads
Immediate Message Length	Length of the message the immediate message points to	The maximum length is 32 bytes
Consumer ID	Identifier for the target CPU consumer to write to	User can find the IDs of each consumer by using `doca_comch_consumer_get_id`
Completion Requested	Flag indicating whether to generate a completion once the send is completed	This refers to the DPA producer completion which is separate from the completion the CPU consumer receives • 0 - no completion • 1 - otherwise

Completion

Once the message arrives to the CPU consumer, a completion element is generated, indicating that the send is complete (this is separate from the completion the CPU consumer receives) and can be polled using `doca_dpa_dev_get_completion`.

Using the generated completion, it is possible to get the following outputs:

Name	Description	Notes
Producer User Data	Producer user data provided during configuration of the producer	User data previously set using `doca_ctx_set_user_data` when configuring this producer. User data which is returned belongs to the DPA producer this completion has been generated for, and can be used to identify the specific producer.

Limitations

- The maximal immediate message size is 32 bytes

Producer DMA Copy

The `doca_dpa_dev_comch_producer_dma_copy` API performs a DMA copy operation and, once the copy operation is done, sends an immediate message to the CPU consumer. Once the message arrives at the CPU consumer side, the CPU consumer receive task completes.

The CPU producer must have posted a receive task prior to this. The user can verify if the consumer can receive the message using `doca_dpa_dev_comch_producer_is_consumer_empty`. Note, however, that this may add overhead.

Configuration

Description	API to Set the Configuration	API to Query Support
Queue Size	`doca_comch_producer_set_dev_num_recv`	-

Input

Name	Description	Notes
Destination Mmap	Mmap representing the memory to be used as the destination of the copy operation	This mmap must have `LOCAL_READ_WRITE` access enabled
Destination Address	The address to be used as the destination of the copy operation	The address and copy length must be within the range of the destination mmap's memory range
Source Mmap	Mmap representing the memory to be used as the source of the copy operation	This mmap must have `LOCAL_READ` access enabled
Source Address	The address to be used as the source of the copy operation	The address and copy length must be within the range of the source mmap's memory range
Length	The length of the copy operation	Source and destination addresses must not overlap
Immediate Message	Short byte array to be sent to the CPU consumer once the copy operation is done	This is not a zero copy operation but does improve latency for small payloads
Immediate Message Length	Length of the message the immediate message points to	The maximum length is 32 bytes
Consumer ID	Identifier for the target CPU consumer to write to	User can find the IDs of each consumer using `doca_comch_consumer_get_id`
Completion Requested	Flag indicating whether to generate a completion once the send is completed	This refers to the DPA producer completion which is separate from the completion the CPU consumer receives • 0 - no completion • 1 - otherwise

Completion

Once copy is complete and the message arrives to the CPU consumer, a completion element is generated, indicating that the copy is complete (this is separate from the completion the CPU consumer receives) and can be polled using `doca_dpa_dev_get_completion`.

Using the generated completion, it is possible to get the following outputs:

Name	Description	Notes
Producer User Data	Producer user data provided during configuration of the producer	The user data set using `doca_ctx_set_user_data` when configuring this producer. The user data which is returned belongs to the DPA producer this completion has been generated for, and can be used to identify the specific producer.

Limitations

- The maximal immediate message size is 32 bytes

14.4.5.1.10 DOCA Comch Samples

This section describes DOCA Comch samples based on the DOCA Comch library.

The samples illustrate how to use the DOCA Comch API to do the following:

- Set up a client/server between host and BlueField Arm cores and use it to send text messages
- Configure fast path producers and consumers, and send messages between them

> ⓘ All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

14.4.5.1.10.1 Running the Samples

1. Refer to the following documents:
    - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
    - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample:

    ```
    cd /opt/mellanox/doca/samples/doca_comch/<sample_name>
    meson /tmp/build
    ninja -C /tmp/build
    ```

    The binary `doca_<sample_name>` is created under `/tmp/build/`.
3. All DOCA Comch samples accept the same input arguments:

Sample	Argument	Description
doca_comch_ctrl_path_server doca_comch_ctrl_path_client doca_comch_data_path_high_speed_server doca_comch_data_path_high_speed_client	-p , --pci-addr	DOCA Comch device PCIe address
	-r , --rep-pci	DOCA Comch device representor PCIe address (required only on BlueField Arm)
	-t , --text	Text to be sent to the other side of channel (overwrites default)

4. For additional information per sample, use the -h option:

```
/tmp/build/<sample_name> -h
```

14.4.5.1.10.2 Samples

DOCA Comch Control Path Client/Server

⚠ doca_comch_ctrl_path_server must be run on the BlueField Arm side and started before doca_comch_ctrl_path_client is started on the host.

This sample sets up a client server connection between the host and BlueField Arm cores.

The connection is used to pass two messages, the first sent by the client when the connection is established and the second by the server on receipt of the client's message.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the core DOCA structures.
3. Initializing and configuring client/server contexts.
4. Registering tasks and events for sending/receiving messages and tracking connection changes.
5. Allocating and submitting tasks for sending control path messages.
6. Handling event completions for receiving messages.
7. Stopping and destroying client/server objects.

References:

- /opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_client/comch_ctrl_path_client_main.c
- /opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_client/comch_ctrl_path_client_sample.c
- /opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_server/comch_ctrl_path_server_main.c
- /opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_server/comch_ctrl_path_server_sample.c
- /opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_common.c
- /opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_common.h

⚠ `doca_comch_data_path_high_speed_server` should be run on the BlueField Arm cores and should be started before `doca_comch_data_path_high_speed_client` is started on the host.

This sample sets up a client server connection between host and BlueField Arm.

The connection is used to create a producer and consumer on both sides and pass a message across the two fastpath connections.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the core DOCA structures.
3. Initializing and configuring client/server contexts.
4. Initializing and configuring producer/consumer contexts on top of an established connection.
5. Submitting post receive tasks for population by producers.
6. Submitting send tasks from producers to write to consumers.
7. Stopping and destroying producer/consumer objects.
8. Stopping and destroying client/server objects.

References:

- `/opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speed_client/comch_data_path_high_speed_client_main.c`
- `/opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speedclient/comch_data_path_high_speed_client_sample.c`
- `/opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speedserver/comch_data_path_high_speed_server_main.c`
- `/opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speedserver/comch_data_path_high_speed_server_sample.c`
- `/opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speed_common.c`
- `/opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speed_common.h`

14.4.5.2 DOCA Comm Channel - Deprecated

This guide provides instructions on how to use the DOCA Comm Channel API.

14.4.5.2.1 Introduction

The DOCA Comm Channel (CC) provides a secure, network-independent communication channel between the host and the DPU.

The communication channel allows the host to control services on the DPU or to activate certain offloads.

The DOCA Comm Channel is reliable, message-based, and connecting multiple clients to a single service. The API allows communication between a client using any PF/VF/SF on the host to a service on the DPU.

14.4.5.2.2 Prerequisites

The CC service can only run on the DPU while the client can only run on a host connected to the DPU.

Refer to NVIDIA DOCA Release Notes for the supported versions of firmware, OS, and MLNX_OFED.

14.4.5.2.3 API

14.4.5.2.3.1 Objects

struct doca_comm_channel_ep_t

Represents a Comm Channel endpoint either on the client or service side. The endpoint is needed for every other Comm Channel API function.

struct doca_comm_channel_addr_t

Also referred to as `peer_address`, represents a connection and can be used to identify the source of a received message. It is required to send a message using `doca_comm_channel_ep_sendto()`.

14.4.5.2.3.2 Query Device Capabilities

Querying the device capabilities allows users to know the derived Comm Channel limitation (see section Limitations for more information), and to set the properties of an endpoint accordingly.

The capabilities under this section, apart from maximal service name length, may vary between different devices. To select the device you wish to establish a connection upon, you may query each of the devices for its capabilities.

doca_comm_channel_get_max_service_name_len()

As each connection requires a name, users must know the maximal length of the name and may use this function to query it. This length includes the null-terminating character, and any name longer than this length is not accepted when trying to establish a connection with Comm Channel.

```
doca_error_t doca_comm_channel_get_max_service_name_len(uint32_t *max_service_name_len);
```

- `max_service_name_len [out]` - pointer to a parameter that will hold the max service name length on success.
- Returns - `doca_error_t` value. `DOCA_SUCCESS` if successful, or an error value upon failure. Possible error values are documented in the header file.

doca_comm_channel_get_max_message_size()

Each connection has an upper limit for the messages size. This function returns the maximal value that can be set for this property, for a given device. This limitation is important when trying to set the max message size for an endpoint with doca_comm_channel_ep_set_max_msg_size().

```
doca_error_t doca_comm_channel_get_max_message_size(struct doca_devinfo *devinfo, uint32_t *max_message_size);
```

- `devinfo [in]` - pointer to a `doca_devinfo` which should be queried for this capability.

- `max_message_size [out]` - pointer to a parameter that on success holds the maximal value that can be set for max message size when communicating on the provided `devinfo`.
- Returns - `doca_error_t` value. `DOCA_SUCCESS` if successful, or an error value upon failure. Possible error values are documented in the header file.

doca_comm_channel_get_max_send_queue_size()

Returns the maximum send queue size that can be set for a given device. This value describes the maximum possible amount of outgoing in-flight messages for a connection. This limitation is important when trying to set the max message size for an endpoint with doca_comm_channel_ep_set_send_queue_size().

```
doca_error_t doca_comm_channel_get_max_send_queue_size(struct doca_devinfo *devinfo, uint32_t
*max_send_queue_size);
```

- `devinfo [in]` - pointer to a `doca_devinfo` which should be queried for this capability.
- `max_send_queue_size [out]` - pointer to a parameter that on success, holds the maximal value that can be set for the send queue size when communicating upon the given `devinfo`.
- Returns - `doca_error_t` value. `DOCA_SUCCESS` if successful, or an error value upon failure. Possible error values are documented in the header file.

doca_comm_channel_get_max_recv_queue_size()

Returns the maximum receive queue size that can be set for a given device. This value describes the maximum possible amount of incoming in-flight messages for a connection. This limitation is important when trying to set the max message size for an endpoint with doca_comm_channel_ep_set_recv_queue_size().

```
doca_error_t doca_comm_channel_get_max_recv_queue_size(struct doca_devinfo *devinfo, uint32_t
*max_recv_queue_size);
```

- `devinfo [in]` - pointer to a `doca_devinfo` which should be queried for this capability.
- `max_ recv_queue_size [out]` - pointer to a parameter that on success holds the maximal value that can be set for the receive queue size when communicating upon the given `devinfo`.
- Returns - `doca_error_t` value. `DOCA_SUCCESS` if successful, or an error value upon failure. Possible error values are documented in the header file.

doca_comm_channel_get_service_max_num_connections()

Returns the maximum amount of connections a service on the DPU can maintain for a given device. If the maximum amount returned is zero, the number of connections is unlimited.

```
doca_error_t doca_comm_channel_get_service_max_num_connections(struct doca_devinfo *devinfo, uint32_t
*max_num_connections);
```

- `devinfo [in]` - pointer to a `doca_devinfo` which should be queried for this capability.
- `max_num_connections [out]` - pointer to a parameter that on success will hold the maximal number of connections the DPU can maintain when communicating upon the given `devinfo`.

- Returns - `doca_error_t` value. `DOCA_SUCCESS` if successful, or an error value upon failure. Possible error values are documented in the header file.

14.4.5.2.3.3 Creating and Configuring an Endpoint

doca_comm_channel_ep_create()

This function is used to create and initialize the endpoint used for all Comm Channel functions.

```
doca_error_t doca_comm_channel_ep_create(struct doca_comm_channel_ep_t **ep);
```

- `ep [out]` - pointer to the created endpoint object.
- Returns - `doca_error_t` value. `DOCA_SUCCESS` if successful, or an error value upon failure. Possible error values are documented in the header file.

doca_comm_channel_ep_set_*() and doca_comm_channel_ep_get_*()

Use `doca_comm_channel_ep_set_*()` functions to set the properties of the endpoint, and corresponding `doca_comm_channel_ep_get_*()` functions to retrieve the current properties of the endpoint.

Mandatory Properties

To use the endpoint, the following properties must be set before calling `doca_comm_channel_ep_listen()` and `doca_comm_channel_ep_connect()`.

doca_comm_channel_ep_set_device()

This function sets the local device through which the communication should be established.

```
doca_error_t doca_comm_channel_ep_set_device(struct doca_comm_channel_ep_t *local_ep, struct doca_dev *device);
```

- `local_ep [in]` - pointer to the endpoint for which the property should be set.
- `device [in]` - the `doca_dev` object which should be used for communication.
- Returns - `doca_error_t` value. `DOCA_SUCCESS` if successful, or an error value upon failure. Possible error values are documented in the header file.

doca_comm_channel_ep_set_device_rep()

This function sets the device representor through which the communication should be established on the service side.

```
doca_error_t doca_comm_channel_ep_set_device_rep(struct doca_comm_channel_ep_t *local_ep, struct doca_dev_rep
*device_rep);
```

- `local_ep [in]` - a pointer to the endpoint for which the property should be set.
- `device_rep [in]` - the `doca_dev_rep` object which should be used for communication.
- Returns - `doca_error_t` value. `DOCA_SUCCESS` if successful, or an error value upon failure. Possible error values are documented in the header file.

Optional Properties

The following properties have a default value and may be set as long as the EP is not yet active.

doca_comm_channel_ep_set_max_msg_size()

This function sets an upper limit to the size of the messages the application wishes to handle in this EP while communicating with a given endpoint. The actual `max_msg_size` may be increased by this function. If this property was not set by the user, a default value is used and may be queried using `doca_comm_channel_ep_get_max_msg_size()` function.

```
doca_error_t doca_comm_channel_ep_set_max_msg_size(struct doca_comm_channel_ep_t *local_ep, uint16_t max_msg_size);
```

- `local_ep [in]` - a pointer to the endpoint for which the property should be set.
- `max_msg_size [in]` - the preferred maximal message size.
- Returns - `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - `DOCA_ERROR_INVALID_VALUE` if a null pointer to the endpoint has been given or if `max_msg_size` is equal to 0 or above the maximal value possible for this property.

doca_comm_channel_ep_set_send_queue_size()

This function sets the send queue size used when communicating with a given endpoint. The actual `send_queue_size` may be increased by this function. If this property has not been set by the user, a default value is used which may be queried using the `doca_comm_channel_ep_get_send_queue_size()` function.

```
doca_error_t doca_comm_channel_ep_set_send_queue_size(struct doca_comm_channel_ep_t *local_ep, uint16_t
send_queue_size);
```

- `local_ep [in]` - pointer to the endpoint for which the property should be set.
- `send_queue_size [in]` - the preferred send queue size.
- Returns - `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - `DOCA_ERROR_INVALID_VALUE` if a null pointer to the endpoint has been given or if `send_queue_size` is equal to 0 or above the maximal value possible for this property.
  - The rest of the error values that may be returned are documented in the header file.

doca_comm_channel_ep_set_recv_queue_size()

This function sets the receive queue size used when communicating with a given endpoint. The actual `recv_queue_size` may be increased by this function. If this property has not been set by the user, a default value is used which may be queried using `doca_comm_channel_ep_get_recv_queue_size()` function.

```
doca_error_t doca_comm_channel_ep_set_recv_queue_size(struct doca_comm_channel_ep_t *local_ep, uint16_t
rcv_queue_size);
```

- `local_ep [in]` - pointer to the endpoint for which the property should be set.
- `rcv_queue_size [in]` - the preferred receive queue size.
- Returns - `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.

    - `DOCA_ERROR_INVALID_VALUE` if a null pointer to the endpoint has been given or if `rcv_queue_size` is equal to 0 or above the maximal value possible for this property.
    - The rest of the error values that may be returned are documented in the header file.

14.4.5.2.3.4 Establishing Connections over Endpoints

The Comm Channel connection is established between endpoints, one on the host and the other on the DPU.

For a client, each connection requires its own EP. On the DPU side, all of the clients with the same service name on a specific representor are connected to a single EP, through which the connections are managed.

The following functions are relevant for the endpoint.

doca_comm_channel_ep_listen()

Used to listen on service endpoint, this function can only be called on the DPU. The service listens on the DOCA device representor provided using `doca_comm_channel_ep_set_device_rep()` . Calling listen allows clients to connect to the service.

```
doca_error_t doca_comm_channel_ep_listen(struct doca_comm_channel_ep_t *local_ep, const char *name);
```

- `local_ep [in]` - pointer to an endpoint to listen on.
- `name [in]` - the name for the service to listen on. Clients must provide the same name to connect to the service.
- Returns - `doca_error_t` value:
    - `DOCA_SUCCESS` if successful.
    - `DOCA_ERROR_BAD_STATE` if mandatory properties (`doca_dev` and `doca_dev_rep`) have not been set.
    - `DOCA_ERROR_NOT_PERMITTED` if called on the host and not on the DPU.
    - The rest of the error values that may be returned are documented in the header file.

doca_comm_channel_ep_connect()

Used to create a connection between a client and a service. This function can only be called on the host.

```
doca_error_t doca_comm_channel_ep_connect(struct doca_comm_channel_ep_t *local_ep,
                                           const char *name, struct doca_comm_channel_addr_t **peer_addr);
```

- `local_ep [in]` - a pointer to an endpoint to connect from.
- `name [in]` - the name of the service that the client connects to. Must be the same name the service listens on.
- `peer_addr [out]` - Contains the pointer to the new connection.
- Returns - `doca_error_t` value:
    - `DOCA_SUCCESS` if successful.
    - `DOCA_ERROR_BAD_STATE` if a mandatory property (`doca_dev`) has not been set.
    - `DOCA_ERROR_NOT_PERMITTED` if called on the DPU and not on the host.
    - The rest of the error values that may be returned are documented in the header file.

14.4.5.2.3.5 Message Event Channel

Getting notifications for messages sent and received through an EP is managed by the event channel, using the functions listed here.

doca_comm_channel_ep_get_event_channel()

After a connection is established through the EP, this function extracts send/receive handles which can be used to get an interrupt when a new event happens using `epoll()` or a similar function.

- A send event happens when at least one in-flight message processing ends.
- A receive event happens when a new incoming message is received.

Users may decide to extract only one of the handles and send a NULL parameter for the other.

The event channels are owned by the endpoint and they are released when `doca_comm_channel_ep_destroy()` is called.

```
doca_error_t doca_comm_channel_ep_get_event_channel(struct doca_comm_channel_ep_t *local_ep,
                                                     doca_event_channel_t *send_event_channel, doca_event_channel_t
*recv_event_channel);
```

- `local_ep [in]` - pointer to the endpoint for which a handle should be returned.
- `send_event_channel [out]` - pointer that holds a handle for sent messages if successful.
- `recv_event_channel [out]` - pointer that holds a handle for received messages if successful.
- Returns - `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - `DOCA_ERROR_BAD_STATE` if no connection has been established (i.e., `doca_comm_channel_ep_listen()` or `doca_comm_channel_ep_connect()` has not been called beforehand).
  - The rest of the error values that may be returned are documented in the header file.

doca_comm_channel_ep_event_handle_arm_send()

After an interrupt caused by an event on the handle for sent messages, the handle should be re-armed to enable interrupts on it:

```
doca_error_t doca_comm_channel_ep_event_handle_arm_send(struct doca_comm_channel_ep_t *local_ep);
```

- `local_ep [in]` - pointer to the endpoint from which the handle has been extracted.
- Returns - `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - The rest of the error values that may be returned are documented in the header file.

doca_comm_channel_ep_event_handle_arm_recv()

After an interrupt caused by an event on the handle for received messages, the handle should be re-armed to enable interrupts on it:

```
doca_error_t doca_comm_channel_ep_event_handle_arm_recv(struct doca_comm_channel_ep_t *local_ep);
```

- `local_ep [in]` - pointer to the endpoint from which the handle has been extracted.
- Returns - `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - The rest of the error values that may be returned are documented in the header file.

14.4.5.2.3.6 doca_comm_channel_ep_sendto()

Used to send a message from one side to the other. The function runs in non-blocking mode. Refer to section "Usage" for more details.

```
doca_error_t doca_comm_channel_ep_sendto(struct doca_comm_channel_ep_t *local_ep, const void *msg
                                         size_t len, int flags, struct doca_comm_channel_addr_t *peer_addr);
```

- `local_ep [in]` - pointer to an endpoint to send the message from.
- `msg [in]` - pointer to the buffer that contains the data to be sent.
- `len [in]` - length of data to be sent.
- `flags [in]` - currently, only DOCA_CC_MSG_FLAG_NONE is a valid flag.
- `peer_addr [in]` - Peer address to send the message to (see also struct doca_comm_channel_addr_t) that has been returned by `doca_comm_channel_ep_connect()` or `doca_comm_channel_rp_recvfrom()`.
- Returns - `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - `DOCA_ERROR_AGAIN` if the send queue is full and this function should be called again.
  - `DOCA_ERROR_CONNECTION_RESET` if the provided `peer_addr` experienced an error and must be disconnected.
  - The rest of the error values that may be returned are documented in the header file.

14.4.5.2.3.7 doca_comm_channel_ep_recvfrom()

Used to receive a packet of data on either the service or the host. The function runs in non-blocking mode. Refer to Usage for more details.

```
doca_error_t doca_comm_channel_ep_recvfrom(struct doca_comm_channel_ep_t *local_ep, void *msg,
                                           size_t *len, int flags, struct doca_comm_channel_addr_t **peer_addr);
```

- `local_ep [in]` - pointer to an endpoint to receive the message on.
- `msg [out]` - pointer to a buffer that message should be written to.
- `len [in\out]` - the input is the length of the given message buffer (`msg`). The output is the actual length of the received message.
- `flags [in]` - `DOCA_CC_MSG_FLAG_NONE`.
- `peer_addr [out]` - handle to `peer_addr` that represents the connection the message arrived from
- Returns - `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - `DOCA_ERROR_AGAIN` if no message is received.
  - `DOCA_ERROR_CONNECTION_RESET` if the message received is from a `peer_addr` that has an error.

  - The rest of the error values that may be returned are documented in the header file.

14.4.5.2.3.8 Information Regarding Each Connection

Each connection established over the EP is represented by a `doca_comm_channel_addr_t` structure, which can also be referred to as a `peer_addr`. This structure is returned by either `doca_comm_channel_ep_connect()` when a connection is established or by `doca_comm_channel_ep_recvfrom()` to identify the connection from which the message has been received.

doca_comm_channel_peer_addr_set_user_data() and doca_comm_channel_peer_addr_get_user_data()

Using `doca_comm_channel_peer_addr_set_user_data()`, users may give each connection a context, similar to an ID, to identify it later, using `doca_comm_channel_peer_addr_get_user_data()`. If a context is not set for a `peer_addr`, it is given the default value "0".

```
doca_error_t doca_comm_channel_ep_recvfrom(struct doca_comm_channel_ep_t *local_ep, void *msg,
                                           size_t *len, int flags, struct doca_comm_channel_addr_t **peer_addr);
```

- `peer_addr [in]` - pointer to `doca_comm_channel_addr_t` structure representing the connection.
- `user_context [in]` - context that should be set for the connection.
- Returns - `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - DOCA_ERROR_INVALID_VALUE if `peer_address` is `NULL`.

```
doca_error_t doca_comm_channel_peer_addr_get_user_data(struct doca_comm_channel_addr_t *peer_addr, uint64_t
*user_context);
```

- `peer_addr [in]` - pointer to `doca_comm_channel_addr_t` structure representing the connection.
- `user_context [out]` - pointer to a parameter that will hold the context on success.
- Returns - `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - `DOCA_ERROR_INVALID_VALUE` if the parameters is `NULL`.

Querying Statistics for Connection

Using the `peer_addr`, users may gather and query the following statistics:

- The number of messages sent.
- The number of bytes sent.
- The number of messages received.
- The number of bytes received.
- The number of outgoing messages yet to be sent.

doca_comm_channel_peer_addr_update_info()

Takes a snapshot with the current statistics of the connection. This function should be called prior to any statistics querying function. It is also used to check the connection status. See Connection Flow for more.

```
doca_error_t doca_comm_channel_peer_addr_update_info(struct doca_comm_channel_addr_t *peer_addr);
```

- `peer_addr [in]` - pointer to `doca_comm_channel_addr_t` structure representing the connection.
- Returns - `doca_error_t` value:
    - `DOCA_SUCCESS` if successful.
    - `DOCA_ERROR_CONNECTION_INPROGRESS` if the connection has yet to be established.
    - `DOCA_ERROR_CONNECTION_ABORTED` if the connection is in an error state.
    - The rest of the error values that may be returned are documented in the header file.

doca_comm_channel_peer_addr_get_send_messages()

This function returns the total number of messages sent to a given `peer_addr` as measured when `doca_comm_channel_peer_addr_update_info()` has been last called.

```
doca_error_t doca_comm_channel_peer_addr_get_send_messages(const struct doca_comm_channel_addr_t *peer_addr,
uint64_t *send_messages);
```

- `peer_addr [in]` - pointer to `doca_comm_channel_addr_t` structure representing the connection.
- `send_messages [out]` - pointer to a parameter that holds the number of messages sent through the `peer_addr` on success.
- Returns - `doca_error_t` value:
    - `DOCA_SUCCESS` if successful.
    - The rest of the error values that may be returned are documented in the header file.

doca_comm_channel_peer_addr_get_send_bytes()

This function returns the total number of bytes sent to a given `peer_addr` as measured when `doca_comm_channel_peer_addr_update_info()` has been last called.

```
doca_error_t doca_comm_channel_peer_addr_get_send_bytes(const struct doca_comm_channel_addr_t *peer_addr, uint64_t
*send_bytes);
```

- `peer_addr [in]` - pointer to `doca_comm_channel_addr_t` structure representing the connection.
- `send_bytes [out]` - pointer to a parameter that holds the number of bytes sent through the `peer_addr` on success.
- Returns - `doca_error_t` value:
    - `DOCA_SUCCESS` if successful.
    - The rest of the error values that may be returned are documented in the header file.

doca_comm_channel_peer_addr_get_recv_messages()

This function return the total number of messages received from a given `peer_addr` as measured when `doca_comm_channel_peer_addr_update_info()` has been last called.

```
doca_error_t doca_comm_channel_peer_addr_get_recv_messages(const struct doca_comm_channel_addr_t *peer_addr,
uint64_t *recv_messages);
```

- `peer_addr [in]` - pointer to `doca_comm_channel_addr_t` structure representing the connection.
- `recv_messages [out]` - pointer to a parameter that holds the number of messages received from the `peer_addr` on success.
- Returns - `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - The rest of the error values that may be returned are documented in the header file.

doca_comm_channel_peer_addr_get_recv_bytes()

This function will return the total number of bytes received from a given `peer_addr` as measured when `doca_comm_channel_peer_addr_update_info()` has been last called.

```
doca_error_t doca_comm_channel_peer_addr_get_recv_bytes(const struct doca_comm_channel_addr_t *peer_addr, uint64_t
*recv_bytes);
```

- `peer_addr [in]` - pointer to `doca_comm_channel_addr_t` structure representing the connection.
- `recv_bytes [out]` - pointer to a parameter that holds the number of bytes sent through the `peer_addr` on success.
- Returns - `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - The rest of the error values that may be returned are documented in the header file.

doca_comm_channel_peer_addr_get_send_in_flight_messages()

This function returns the number of messages still in transmission to a specific `peer_addr` as measured when `doca_comm_channel_peer_addr_update_info()` has been last called. This function can be used to check if all messages are sent before disconnecting.

```
doca_error_t doca_comm_channel_peer_addr_get_send_in_flight_messages(const struct doca_comm_channel_addr_t
*peer_addr,
                                                                      uint64_t *send_in_flight_messages);
```

- `peer_addr [in]` - pointer to `doca_comm_channel_addr_t` structure representing the connection.
- `send_in_flight_messages [out]` - pointer to a parameter that holds the number of in-flight messages to the `peer_addr` on success.
- Returns - `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - The rest of the error values that may be returned are documented in the header file.

14.4.5.2.3.9 Service State and Events

The service state and events API provides information about the state of the service including current connected clients, pending connections, and service state. All the functions in this section are relevant and can be run on the service side only.

doca_comm_channel_ep_get_service_event_channel()

After a service is created and starts listening, this function extracts a handle which can be used to get an interrupt when a new service event happens using `epoll()` or a similar function.

The currently supported events are service failure, new client connection, and client disconnection. After an event is triggered, the application can call doca_comm_channel_ep_update_service_state_info() and the following getter functions to query the service state and connections.

The service event channel is armed automatically when calling doca_comm_channel_ep_update_service_state_info().

```
doca_error_t doca_comm_channel_ep_get_service_event_channel(struct doca_comm_channel_ep_t *local_ep,
doca_event_channel_t *service_event_channel);
```

- `local_ep [in]` - pointer to the service endpoint that should be queried.
- `service_event_channel [out]` - event handle for service events.
- Returns - `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - The rest of the error values that may be returned are documented in the header file.

doca_comm_channel_ep_update_service_state_info()

> This function should be called prior to calling service status get functions.

Takes a snapshot of the current state of the service. The return value may indicate the service state. If the service is in error state, then it is non-recoverable and the endpoint must be destroyed.

> Calling this function invalidates any array received using `doca_comm_channel_ep_get_peer_addr_list()`.

```
doca_error_t doca_comm_channel_ep_update_service_state_info(struct doca_comm_channel_ep_t *local_ep);
```

- `local_ep [in]` - pointer to the service endpoint that should be queried.
- Returns - `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - `DOCA_ERROR_CONNECTION_RESET` if the service is in error state and cannot be recovered.
  - The rest of the error values that may be returned are documented in the header file.

doca_comm_channel_ep_get_peer_addr_list()

This function returns the list of connected `peer_addr` s as present when `doca_comm_channel_ep_update_service_state_info()` was last called.

> This list includes only active `peer_addr` s which have not been disconnected from the client side or the service side.

The output array is only valid until `doca_comm_channel_ep_update_service_state_info()` is called again.

```
doca_error_t doca_comm_channel_ep_get_peer_addr_list(const struct doca_comm_channel_ep_t *local_ep,
                                                     struct doca_comm_channel_addr_t ***peer_addr_array,
                                                     uint32_t *peer_addr_array_len);
```

- `local_ep [in]` - pointer to the service endpoint that should be queried.
- `peer_addr_array [out]` - pointer to array of peer addresses.
- `peer_addr_array_len [out]` - the number of entries in `peer_addr_array`.
- Returns - `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - The rest of the error values that may be returned are documented in the header file.

doca_comm_channel_ep_get_pending_connections()

This function returns the list of pending connections as present when `doca_comm_channel_ep_update_service_state_info()` was last called. Pending connections are connections that were initiated by the client side but not complete from the service side.

> If a pending connection exists, the application is expected to call `doca_comm_channel_ep_recvfrom()` to complete the connection. See section "Connection Flow" for more.

```
doca_error_t doca_comm_channel_ep_get_pending_connections(const struct doca_comm_channel_ep_t *local_ep,
                                                          uint32_t *pending_connections);
```

- `local_ep [in]` - pointer to the service endpoint that should be queried.
- `pending_connections [out]` - the number of pending connections.
- Returns - `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - The rest of the error values that may be returned are documented in the header file.

14.4.5.2.3.10 doca_comm_channel_ep_disconnect()

Disconnects an endpoint from a specific `peer_address`. The disconnection is one-sided and the other side is unaware of it. New connections can be created afterwards. Refer to "Usage" for more details.

```
doca_error_t doca_comm_channel_ep_disconnect(struct doca_comm_channel_ep_t *local_ep, struct
doca_comm_channel_addr_t *peer_addr);
```

- `local_ep [in]` - pointer to the endpoint that should be disconnected.

- `peer_addr [in]` - the connection from which the endpoint should be disconnected.
- Returns - `doca_error_t` value:
    - `DOCA_SUCCESS` if successful.
    - `DOCA_ERROR_NOT_CONNECTED` if there is no connection between the endpoint and the peer address.

14.4.5.2.3.11 doca_comm_channel_ep_destroy()

Disconnects all connections of the endpoint, destroys the endpoint object, and frees all related resources.

```
doca_error_t doca_comm_channel_ep_destroy(struct doca_comm_channel_ep_t *ep);
```

- `local_ep [in]` - pointer to the endpoint that should be destroyed.
- Returns - `doca_error_t` value:
    - `DOCA_SUCCESS` if successful.
    - The rest of the error values that may be returned are documented in the header file.

14.4.5.2.4 Limitations

14.4.5.2.4.1 Endpoint Properties

The maximal values of all endpoint properties can be queried using the proper get functions (see section "Query Device Capabilities"). The `max_message_size`, `send_queue_size`, and `recv_queue_size` attributes may be increased internally. The updated property value can be queried with the proper get functions.

See the following table and section "doca_comm_channel_ep_set_*() and doca_comm_channel_ep_get_*()" for more details.

Property	Get Function
Max message size	`doca_comm_channel_get_max_message_size()`
Send queue size	`doca_comm_channel_get_max_send_queue_size()`
Receive queue size	`doca_comm_channel_get_max_recv_queue_size()`
Service name length	`doca_comm_channel_get_max_service_name_len()`

14.4.5.2.4.2 Multi-client

A single service on the DPU can serve multiple clients but a client can only connect to a single service.

The maximal number of clients connected to a single service can be queried using `doca_comm_channel_get_service_max_num_connections()`.

14.4.5.2.4.3 Multiple Services

Multiple endpoints can be created on the same DPU but different services listening on the same representor must have different names. Services listening on different representors can have the same name.

14.4.5.2.4.4 Threads

The DOCA Comm Channel is not thread-safe. Using a single endpoint over multiple threads is possible only with the use of locks to prevent parallel usage of the same resources. Different endpoints can be used over different threads with no restriction as each endpoint has its own resources.

14.4.5.2.5 Usage

14.4.5.2.5.1 Objects

While working with DOCA Comm Channel, the user must maintain two types of objects:

- `struct doca_comm_channel_ep_t` (referred to as "endpoint")
- `struct doca_comm_channel_addr_t` (referred to as "peer_address")

Endpoint

The endpoint object represents the endpoint of the Comm Channel, either on the client or service side. The endpoint is created by calling the `doca_comm_channel_ep_create()` function. It is required for every other Comm Channel function.

Peer_address

The `peer_address` structure represents a connection. It is created when a new connection is made (i.e., client calls `doca_comm_channel_ep_connect()` or a service receives a connection through `doca_comm_channel_ep_recvfrom()`). Refer to section "Connection Flow" for more details on connections.

The `peer_address` structure can be used to identify the source of a received message and is necessary to send a message using `doca_comm_channel_ep_sendto()`. `peer_address` has an identifier, `user_data`, which can be set by the user using `doca_comm_channel_peer_addr_user_data_set()` and retrieved using `doca_comm_channel_peer_addr_user_data_get()`. The default value for `user_data` is 0. The `user_data` field can be used to identify the `peer_address` object.

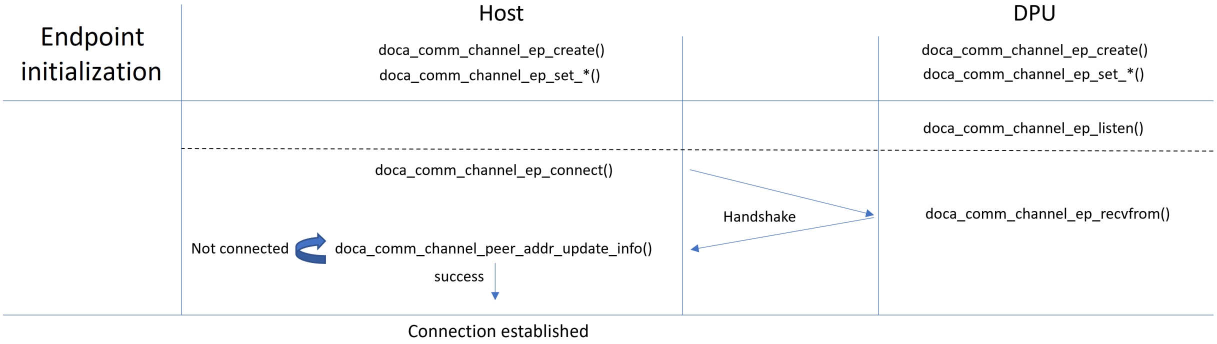
14.4.5.2.5.2 Endpoint Initialization

To start using the DOCA Comm Channel, the user must create an endpoint object using the `doca_comm_channel_ep_create()` function. After creating the endpoint object, the user must set the mandatory endpoint properties: `doca_dev` for client and service, `doca_dev_rep` for service only. The user may also set the optional endpoint properties.

For further information about endpoint initialization, refer to section "Establishing Connection over Endpoint".

14.4.5.2.5.3 Connection Flow

The following diagram illustrates the process of establishing a connection between the host and a service.


Host
DPU
Endpoint initialization
doca_comm_channel_ep_create()
doca_comm_channel_ep_set_*()
doca_comm_channel_ep_create()
doca_comm_channel_ep_set_*()
doca_comm_channel_ep_listen()
doca_comm_channel_ep_connect()
Handshake
doca_comm_channel_ep_recvfrom()
Not connected
doca_comm_channel_peer_addr_update_info()
success
Connection established

1. After initializing the endpoint on the service side, one should call `doca_comm_channel_ep_listen()` with a legal service name (see "Limitations") to start listening.
2. After the service starts listening and the client endpoint is created, the client calls `doca_comm_channel_ep_connect()` with the same service name used for listening.

As part of the connect function, the client starts a handshake protocol with the server, which then waits until the service completes the handshake. If connect is called before the service is listening or the handshake process fails, then the connect function fails.

From the connect function, the client receives a `peer_addr` object representing the new connection to the service:

1. To check whether the connection is complete or not, the client must call `doca_comm_channel_peer_addr_update_info()` with the new `peer_addr`. Depending on the function return code, the client would know whether the connection is complete (`DOCA_SUCCESS`), rejected (`DOCA_ERROR_CONNECTION_ABORTED`) or still in progress (`DOCA_ERROR_CONNECTION_INPROGRESS`).
2. The service receiving new connections is done using `doca_comm_channel_ep_recvfrom()`. No indication is given that a new connection is made. The server keeps waiting to receive packets. If the handshake fails or is done for an existing client, then the receive function fails.

For more information, see section "doca_comm_channel_ep_listen()".

14.4.5.2.5.4 Data Transfer Flow

After a connection is established between client and service, both sides can send and receive data using the `doca_comm_channel_ep_sendto()` and `doca_comm_channel_ep_recvfrom()` functions, respectively.

If multiple clients are connected to the same service, then the `doca_comm_channel_ep_recvfrom()` function reads the messages in the order of their arrival, regardless of their source.

To send a message, the endpoint must obtain the target's `peer_address` object. This restriction necessitates the client to start the communication (not including the handshake), by sending the first message, for the server to obtain the client's `peer_address` object and send data back.

The `doca_comm_channel_ep_sendto()` function adds the message to an internal send queue where it is processed asynchronously. This means that even if the `doca_comm_channel_ep_sendto()` function returns with `DOCA_SUCCESS`, the message itself may fail to send (e.g., if the other side has been disconnected). If a message fails to send, the relevant `peer_address` moves to `error_state`. See section "Connection Errors" for more.

For more information, see section "doca_comm_channel_ep_sendto()".

14.4.5.2.5.5 Event Channel and Event Handling

When trying to send or receive messages, the application may face a situation where the resources are not ready—send queue full or no new messages received. In this case, the Comm Channel returns `DOCA_ERROR_AGAIN` for the call. This return value indicates that the function must be called again later in order to complete. To know when to call the send/receive function again, the application can use two approaches:

- Active polling – that is, to use a loop to call the send/receive functions immediately or after a certain time until the `DOCA_SUCCESS` return code is received.
- Using CC event channel to know when to call the send/receive function again.
  The CC event channel is a mechanism that enables getting an event when a new CC event happens. It is divided to send and receive event channels which can be retrieved using `doca_comm_channel_ep_get_event_channel()`. After retrieving the event channels, the application can use `poll` in Linux or `GetQueuedCompletionStatus` in Windows to sleep and wait for events.
  When first using the event channels and after each event is received using the event channel, it must be armed using `doca_comm_channel_ep_event_handle_arm_send()` or `doca_comm_channel_ep_event_handle_arm_recv()` to receive more events.
  For more information, see section "Event Channel".

14.4.5.2.5.6 Connection Errors

In certain cases, for example if a remote peer disconnects and the local endpoint tries sending a message, a `peer_addr` can move to error state. In such cases, no new messages can be sent to or received from the certain `peer_addr`.

The Comm Channel indicates a `peer_addr` is in an error state by returning `DOCA_ERROR_CONNECTION_RESET` on `doca_comm_channel_ep_sendto()` if trying to send a message to an errored `peer_addr` or on `doca_comm_channel_ep_recvfrom()` when receiving a message from a `peer_addr` marked as errored, or when calling `doca_comm_channel_peer_addr_update_info()`.

When a `peer_addr` is in an error state, it is the application's responsibility to disconnect the said `peer_addr` using `doca_comm_channel_ep_disconnect()` .

14.4.5.2.5.7 Connection Statistics

The `peer_addr` object provides a statistics mechanism. To get the updated statistics, the application should call `doca_comm_channel_peer_addr_update_info()` which saves a snapshot of the current statistics.

After calling the update function, the application can query the following statistics which return the data from that snapshot:

Statistic Function	Returns
`doca_comm_channel_peer_addr_get_send_messages()`	Number of messages sent to the specific `peer_addr`
`doca_comm_channel_peer_addr_get_send_bytes()`	Number of bytes sent to the specific `peer_addr`
`doca_comm_channel_peer_addr_get_recv_messages()`	Number of messages received from the specific `peer_addr`
`doca_comm_channel_peer_addr_get_recv_bytes()`	Number of bytes received from the specific `peer_addr`
`doca_comm_channel_peer_addr_get_send_in_flight_messages()`	Number of messages sent to the specific `peer_addr` and without returning a confirmation yet

The in-flight messages can be used to make sure all messages have been successfully sent before disconnecting or destroying the endpoint.

For more information, see section "Querying Statistics for Connection".

14.4.5.2.5.8 Service State and Connections

DOCA Comm Channel provides an API, `doca_comm_channel_ep_update_service_state_info()` , to query for the service state and connections which an application can call.

The service state is returned as the return value from the update function:

- If the return value is `DOCA_SUCCESS` the service state is operational.
- If the return value is `DOCA_ERROR_CONNECTION_RESET` the service is down and cannot be recovered, and the endpoint should be destroyed.

After calling the update function, the application can query the following functions which return the connection data from that snapshot:

Information Function	Returns
`doca_comm_channel_ep_get_peer_addr_list()`	Returns the list of connected `peer_addrs`
`doca_comm_channel_ep_get_pending_connections()`	Number of pending connections waiting for the service. If there are pending connections, `doca_comm_channel_ep_recvfrom()` should be called to handle them.

14.4.5.2.5.9 Disconnection Flow

Disconnection can occur specifically by using `doca_comm_channel_ep_disconnect()` or when destroying the whole endpoint.

Disconnection is one-sided, which means that the other side is unaware of the channel being closed and experiences errors when sending data. It is up to the application to synchronize the connection teardown.

Disconnection of a `peer_addr` destroys all of the resources related to it.

It is possible to perform another handshake and establish a new channel connection after disconnection.

For more information, see section "doca_comm_channel_ep_disconnect()".

14.4.5.2.5.10 Endpoint Destruction

When calling `doca_comm_channel_ep_destroy()`, all resources related to the endpoint are freed immediately which means that if there are any messages in the send queue that have not been sent yet, they are aborted.

To make sure all messages have been successfully sent before disconnection, the application can use the `doca_comm_channel_peer_addr_get_send_in_flight_messages()` statistics function. See section "Connection Statistics" for more information.

14.4.5.2.6 DOCA Comm Channel Samples

This section provides Comm Channel sample implementation on top of the BlueField DPU.

14.4.5.2.6.1 Running the Sample

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample:

```
cd /opt/mellanox/doca/samples/doca_comm_channel/<sample_name>
meson /tmp/build
ninja -C /tmp/build
```

   The binary `doca_<sample_name>` is created under `/tmp/build/`.

3. Sample (e.g., `comm_channel_server`) usage:

```
Usage: doca_comm_channel_server [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
```

```
  -j, --json <path>                 Parse all command flags from an input json file

Program Flags:
  -p, --pci-addr                    DOCA Comm Channel device PCI address
  -r, --rep-pci                     DOCA Comm Channel device representor PCI address (needed only on DPU)
  -t, --text                        Text to be sent to the other side of channel
```

> ⚠ The flag `--rep-pci` is relevant only on the DPU.

4. For additional information per sample, use the `-h` option:

```
/tmp/build/doca_<sample_name> -h
```

14.4.5.2.6.2 Samples

Comm Channel Server

> ⚠ This sample should be run before "Comm Channel Client".

This sample illustrate how to create a simple server on the DPU to communicate with a client on the host.

The sample logic includes:

1. Creating Comm Channel endpoint.
2. Parsing PCIe address.
3. Opening Comm Channel DOCA device based on the PCIe address.
4. Opening Comm Channel DOCA device representor based on the PCIe address.
5. Setting Comm Channel endpoint properties.
6. Listening for new connections.
7. Waiting until new message arrives.
8. Sending the entered text message as a response.
9. Closing connection and freeing resources.

Reference:

- `/opt/mellanox/doca/samples/doca_comm_channel/comm_channel_server/comm_channel_server_sample.c`
- `/opt/mellanox/doca/samples/doca_comm_channel/comm_channel_server/comm_channel_server_main.c`
- `/opt/mellanox/doca/samples/doca_comm_channel/comm_channel_server/meson.build`

Comm Channel Client

> ⚠ This sample should be run after "Comm Channel Server".

This sample illustrates how to create a simple client on the host to communicate with a server on the DPU.

The sample logic includes:

1. Creating Comm Channel endpoint.
2. Parsing PCIe address.
3. Opening Comm Channel DOCA device based on the PCIe address.
4. Setting Comm Channel endpoint properties.
5. Connecting current endpoint to server side.
6. Sending the entered text message.
7. Receiving server response.
8. Closing connection and freeing resources.

Reference:

- `/opt/mellanox/doca/samples/doca_comm_channel/comm_channel_client/comm_channel_client_sample.c`
- `/opt/mellanox/doca/samples/doca_comm_channel/comm_channel_client/comm_channel_client_main.c`
- `/opt/mellanox/doca/samples/doca_comm_channel/comm_channel_client/meson.build`

14.4.6 DOCA UROM

This guide provides an overview and configuration instructions for DOCA Unified Resources and Offload Manager (UROM) API.

14.4.6.1 Introduction

> ⚠ This library is currently supported at alpha level only.

The DOCA Unified Resource and Offload Manager (UROM) offers a framework for offloading a portion of parallel computing tasks, such as those related to HPC or AI workloads and frameworks, from the host to the NVIDIA DPUs. This framework includes the UROM service which is responsible for resource discovery, coordination between the host and DPU, and the management of UROM workers that execute parallel computing tasks.

When an application utilizes the UROM framework for offloading, it consists of two main components: the host part and the UROM worker on the DPU. The host part is responsible for interacting with the DOCA UROM API and operates as part of the application with the aim of offloading tasks to the DPU. This component establishes a connection with the UROM service and initiates an offload request. In response to the offload request, the UROM service provides network identifiers for the workers, which are spawned by the UROM service. If the UROM service is running as a Kubernetes POD, the workers are spawned within the POD. Each worker is responsible for executing either a single offload or multiple offloads, depending on the requirements of the host application.

14.4.6.2 Prerequisites

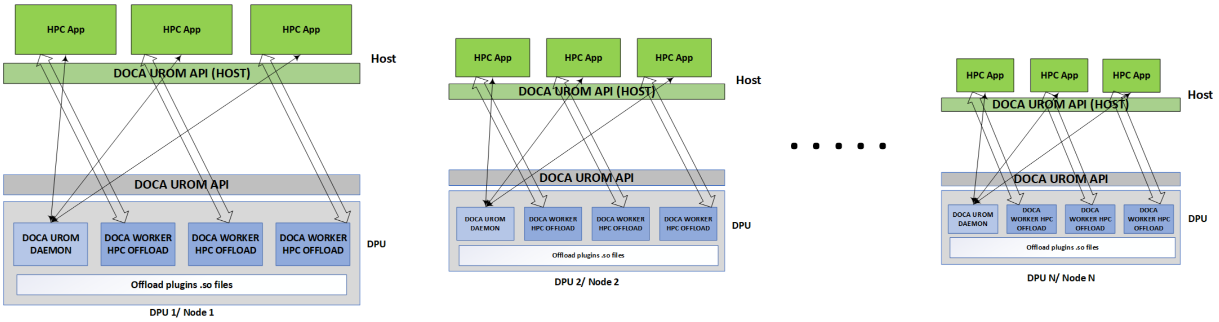
UCX is required for the communication channel between the host and DPU parts of DOCA UROM based on TCP socket transport. This is a mechanism to transfer commands from the host to the UROM service on the DPU and receive responses from the DPU.

By default, UCX scans all available devices on the machine and selects the best ones based on performance characteristics. The environment variable `UCX_NET_DEVICES=<dev1>,<dev2>,...` would restrict UCX to using only the specified devices. For example, `UCX_NET_DEVICES=eth2` uses the Ethernet device `eth2` for TCP socket transport.

For more information about UCX, refer to DOCA UCX Programming Guide.

14.4.6.3 Architecture

14.4.6.3.1 UROM Deployment



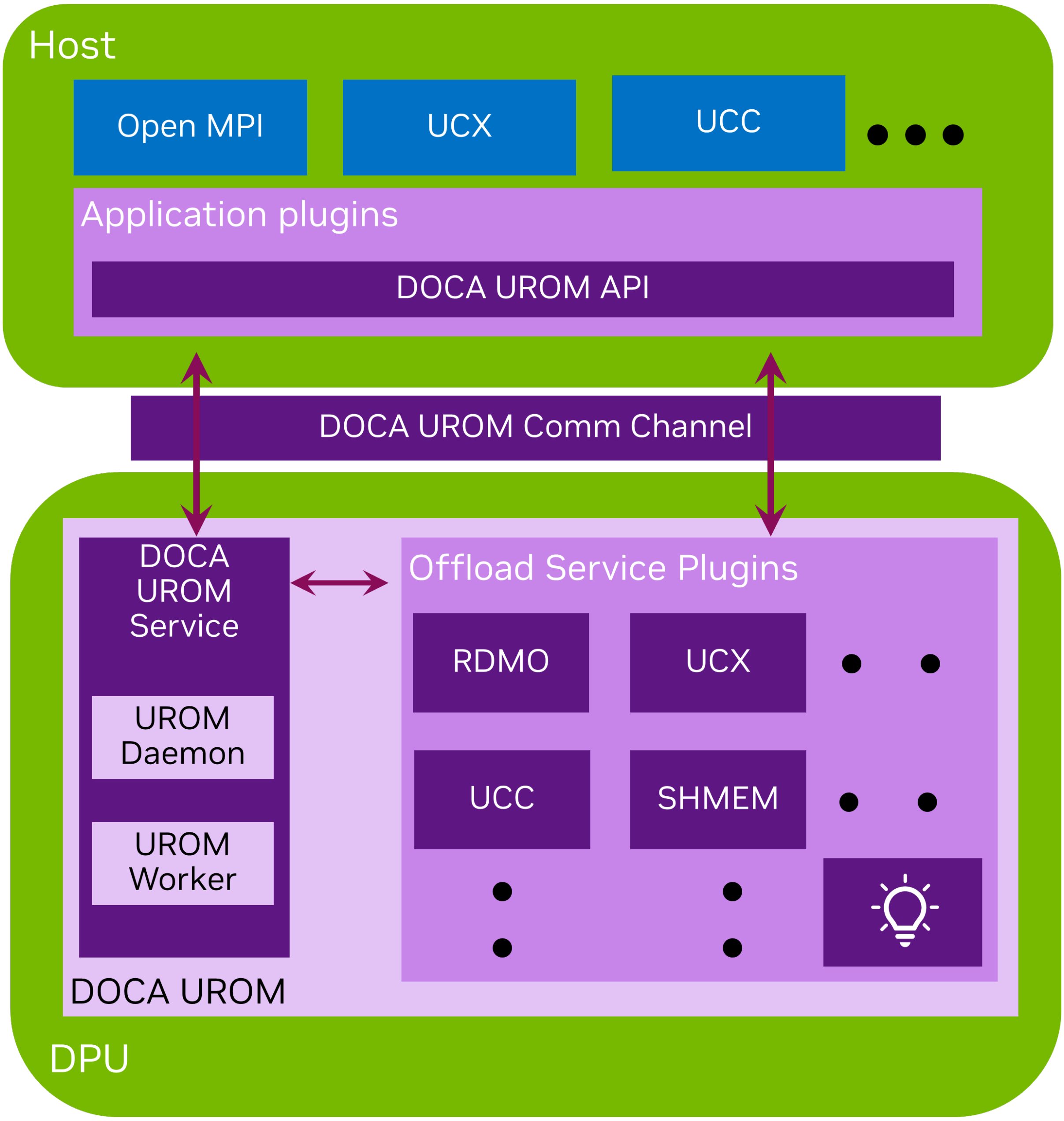
The diagram illustrates a standard UROM deployment where each DPU is required to host both a service process instance and a group of worker processes.

The typical usage of UROM services involves the following steps:

1. Every process in the parallel application discovers the UROM service.
2. UROM handles authentication and provides service details.
3. The host application receives the available offloading plugins on the local DPU through UROM service.
4. The host application picks the desired plugin info and triggers UROM worker plugin instances on the DPU through the UROM service.
5. The application delegates specific tasks to the UROM workers.
6. UROM workers execute these tasks and return the results.

14.4.6.3.2 UROM Framework

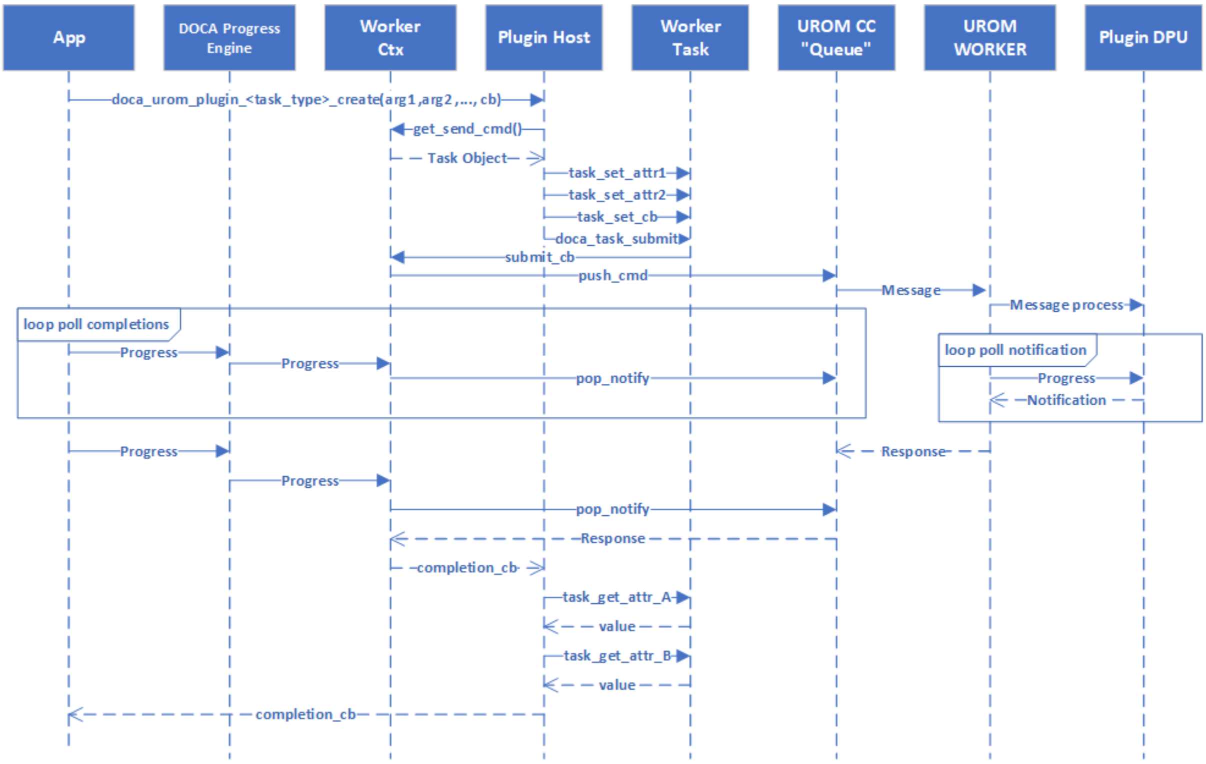
This diagram shows a high-level overview of the DOCA UROM framework.



A UROM offload plugin is where developers of AI/HPC offloads implement their own offloading logic while using DOCA UROM as the transport layer and resource manager. Each plugin defines commands to execute logic on the DPU and notifications that are returned to the host application. Each type of supported offload corresponds to a distinct type of DOCA UROM plugin. For example, a developer may need a UCC plugin to offload UCC functionality to the DPU. Each plugin implements a DPU-side plugin API and exposes a corresponding host-side interface.

A UROM daemon loads the plugin DPU version (`.so` file) in runtime as part of the discovery of local plugins.

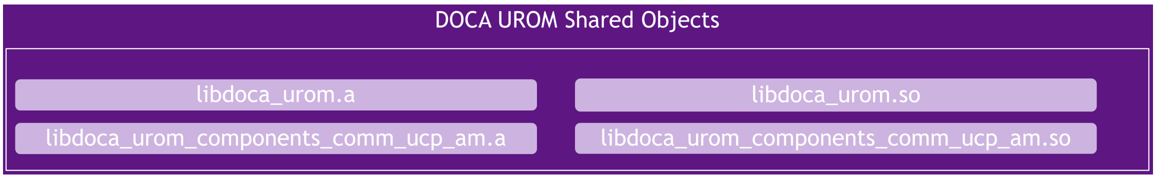
14.4.6.3.2.1 Plugin Task Offloading Flow



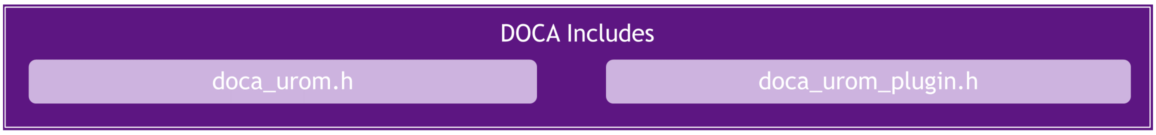
14.4.6.3.3 UROM Installation

DOCA UROM is an integral part of the DOCA SDK installation package. Depending on your system architecture and enabled offload plugins, UROM is comprised by several components, which can be categorized into two main parts: those on the host and those on the DPU.

- DOCA UROM library components:
  - `libdoca_urom` shared object - contains the DOCA UROM API
  - `libdoca_urom_components_comm_ucp_am` - includes the UROM communication channel interface API


DOCA UROM Shared Objects
libdoca_urom.a
libdoca_urom.so
libdoca_urom_components_comm_ucp_am.a
libdoca_urom_components_comm_ucp_am.so

- DOCA UROM headers:

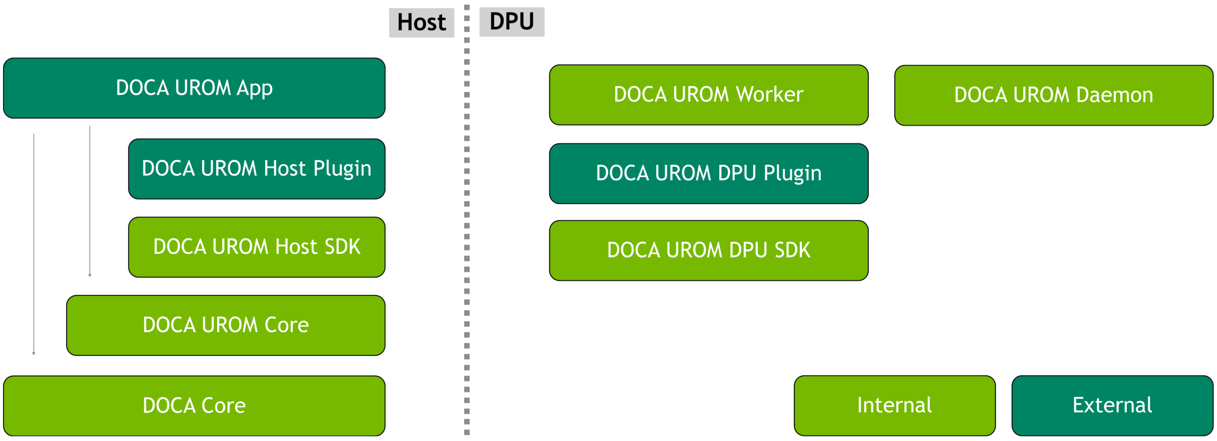


The header files include definitions for DOCA UROM as described in the following:

- DOCA UROM host interface (`doca_urom.h`) - this header includes three essential components: contexts, tasks, and plugins.
  - Service context (`doca_urom_service`) - this context serves as an abstraction of the UROM service process. Tasks posted within this context include the authentication, spawning, and termination of workers on the DPU.
  - Worker context (`doca_urom_worker`) - this context abstracts the DPU UROM worker, which operates on behalf of host application plugins (offload). Tasks posted within this context involve relaying commands from the host application to the worker on behalf of a specific offload plugin, such as offloaded functionality for communication operations.
  - Domain context (`doca_urom_domain`) - this context encapsulates a group of workers belonging to the same host application. This concept is similar to the MPI (message passing interface) communicator in the MPI programming model or PyTorch's process groups. Plugins are not required to use the UROM Domain.
- DOCA UROM plugin interface (`doca_urom_plugin.h`) - this header includes the main structure and definitions that the user can use to build both the host and DPU components of their own offloading plugins
  - UROM plugin interface structure (`urom_plugin_iface`) - this interface includes a set of operations to be executed by the UROM worker
  - UROM worker command structure (`urom_worker_cmd`) - this structure defines the worker instance command format
  - UROM worker notification structure (`urom_worker_notify`) - this structure defines the worker instance notification format

The following diagram shows various software components of DOCA UROM:

- DOCA Core – involves DOCA device discovery, DOCA progress engine, DOCA context, etc.
- DOCA UROM Core – includes the UROM library functionality
- DOCA UROM Host SDK – UROM API for the host application to use
- DOCA UROM DPU SDK – UROM API for the NVIDIA® BlueField® networking platform (DPU or SuperNIC) to use
- DOCA UROM Host Plugin – user plugin host version
- DOCA UROM DPU Plugin – user plugin DPU version
- DOCA UROM App – user UROM host application
- DOCA UROM Worker – the offload functionality component that executes the offloading logic
- DOCA UROM Daemon – is responsible for resource discovery, coordination between the host and DPU, managing the workers on BlueField



14.4.6.4 API

ⓘ More information is available on DOCA UROM API in the NVIDIA DOCA Library APIs.

⚠ The pkg-config (`*.pc` file) for the UROM library is `doca-urom` .

The following sections provide additional details about the library API.

14.4.6.4.1 DOCA_UROM_SERVICE_FILE

This environment variable sets the path to the UROM service file. When creating the UROM service object (see `doca_urom_service_create`), UROM performs a look-up using this file, the hostname where an application is running, and the PCIe address of the associated DOCA device to identify the network address, and network devices associated with the UROM service.

This file contains one entry per line describing the location of each UROM service that may be used by UROM. The format of each line must be as follows:

```
<app_hostname> <service_type> <dev_hostname> <dev_pci_addr> <net,devs>
```

Example:

```
app_host1 dpu dpu_host1 03:00.0 dev1:1,dev2:1
```

Fields are described in the following table:

Field	Description
`app_hostname`	Network hostname (or IP address) for the node that this line applies to
`service_type`	The UROM service type. Valid type is `dpu` (used for all DOCA devices).

Field	Description
`dev_hostname`	Network hostname (or IP address) for the associated DOCA device
`dev_pci_addr`	PCIe address of the associated DOCA device. This must match the PCIe address provided by DOCA.
`net,devs`	Comma-separated list of network devices shared between the host and DOCA device

14.4.6.4.2 doca_urom_service

An opaque structure that represents a DOCA UROM service.

```
struct doca_urom_service;
```

14.4.6.4.3 doca_urom_service_plugin_info

DOCA UROM plugin info structure. UROM generates this structure for each plugin on the local DPU where the UROM service is running and the service returns an array of available plugins to the host application to pick which plugins to use.

```
struct doca_urom_service_plugin_info {
    uint64_t id;
    uint64_t version;
    char plugin_name[DOCA_UROM_PLUGIN_NAME_MAX_LEN];
};
```

- `id` - Unique ID to send commands to the plugin, UROM generates this ID
- `version` - Plugin DPU version to verify that the plugin host interface has the same version
- `plugin_name` - The `.so` plugin file name without "`.so`". The name is used to find the desired plugin.

14.4.6.4.4 doca_urom_service_get_workers_by_gid_task

An opaque structure representing a DOCA service gets workers by group ID task.

```
struct doca_urom_service_get_workers_by_gid_task;
```

14.4.6.4.5 doca_urom_service_create

Before performing any UROM service operation (spawn worker, destroy worker, etc.), it is essential to create a `doca_urom_service` object. A service object is created in state `DOCA_CTX_STATE_IDLE`. After creation, the user may configure the service using setter methods (e.g., `doca_urom_service_set_dev()`).

Before use, a service object must be transitioned to state `DOCA_CTX_STATE_RUNNING` using the `doca_ctx_start()` interface. A typical invocation looks like `doca_ctx_start(doca_urom_service_as_ctx(service_ctx))`.

```
doca_error_t doca_urom_service_create(struct doca_urom_service **service_ctx);
```

- `service_ctx [in/out]` - `doca_urom_service` object to be created
- Returns - `DOCA_SUCCESS` on success, error code otherwise

> Multiple application processes could create different service objects that represent/connect to the same worker on the DPU.

14.4.6.4.6 doca_urom_service_destroy

Destroy a `doca_urom_service` object.

```
doca_error_t doca_urom_service_destroy(struct doca_urom_service *service_ctx);
```

- `service_ctx[in]` - `doca_urom_service` object to be destroyed. It is created by `doca_urom_service_create()` .
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.7 doca_urom_service_set_max_comm_msg_size

Set the maximum size for a message in the UROM communication channel. The default message size is 4096B.

> It is important to ensure that the combined size of the plugins' commands and notifications and the UROM structure's size do not exceed this maximum size.

Once the service state is running, users cannot update the maximum size for the message.

```
doca_error_t doca_urom_service_set_max_comm_msg_size(struct doca_urom_service *service_ctx, size_t msg_size);
```

- `service_ctx[in]` - a pointer to `doca_urom_service` object to set new message size
- `msg_size[in]` - new message size to set
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.8 doca_urom_service_as_ctx

Convert a `doca_urom_service` object into a DOCA object.

```
struct doca_ctx *doca_urom_service_as_ctx(struct doca_urom_service *service_ctx);
```

- `service_ctx[in]` - a pointer to `doca_urom_service` object
- Returns - a pointer to the `doca_ctx` object on success, `NULL` otherwise

14.4.6.4.9 doca_urom_service_get_plugins_list

Retrieve the list of supported plugins on the UROM service.

```
doca_error_t doca_urom_service_get_plugins_list(struct doca_urom_service *service_ctx, const struct
doca_urom_service_plugin_info **plugins, size_t *plugins_count);
```

- `service_ctx[in]` - a pointer to `doca_urom_service` object
- `plugins[out]` - an array of pointers to `doca_urom_service_plugin_info` object
- `plugins_count[out]` - number of plugins
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.10 doca_urom_service_get_cpuset

Get the allowed CPU set for the UROM service on BlueField, which can be used when spawning workers to set processor affinity.

```
doca_error_t doca_urom_service_get_cpuset(struct doca_urom_service *service_ctx, doca_cpu_set_t *cpuset);
```

- `service_ctx[in]` - a pointer to `doca_urom_service` object
- `cpuset[out]` - set of allowed CPUs
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.11 doca_urom_service_get_workers_by_gid_task_allocate_init

Allocate a get-workers-by-GID service task and set task attributes.

```
doca_error_t doca_urom_service_get_workers_by_gid_task_allocate_init(struct doca_urom_service *service_ctx,
                                                                     uint32_t gid,

doca_urom_service_get_workers_by_gid_task_completion_cb_t cb,
                                                                     struct

doca_urom_service_get_workers_by_gid_task **task);
```

- `service_ctx[in]` - a pointer to `doca_urom_service` object
- `gid[in]` - group ID to set
- `cb[in]` - user task completion callback
- `task[out]` - a new get-workers-by-GID service task
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.12 doca_urom_service_get_workers_by_gid_task_release

Release a get-workers-by-GID service task and task resources.

```
doca_error_t doca_urom_service_get_workers_by_gid_task_release(struct doca_urom_service_get_workers_by_gid_task
*task);
```

- `task[in]` - service task to release
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.13 doca_urom_service_get_workers_by_gid_task_as_task

Convert a `doca_urom_service_get_workers_by_gid_task` object into a DOCA task object.

After creating a service task and configuring it using setter methods (e.g., `doca_urom_service_get_workers_by_gid_task_set_gid()`) or as part of task allocation, the user should submit the task by calling `doca_task_submit`.

A typical invocation looks like `doca_task_submit(doca_urom_service_get_workers_by_gid_task_as_task(task))`.

```
struct doca_task *doca_urom_service_get_workers_by_gid_task_as_task(struct
doca_urom_service_get_workers_by_gid_task *task);
```

- `task[in]` - get-workers-by-GID service task
- Returns - a pointer to the `doca_task` object on success, `NULL` otherwise

14.4.6.4.14 doca_urom_service_get_workers_by_gid_task_get_workers_count

Get the number of workers returned for the requested GID.

```
size_t doca_urom_service_get_workers_by_gid_task_get_workers_count(struct doca_urom_service_get_workers_by_gid_task
*task);
```

- `task[in]` - get-workers-by-GID service task
- Returns - workers ID's array size

14.4.6.4.15 doca_urom_service_get_workers_by_gid_task_get_worker_ids

Get service get workers task IDs array.

```
const uint64_t *doca_urom_service_get_workers_by_gid_task_get_worker_ids(struct
doca_urom_service_get_workers_by_gid_task *task);
```

- `task[in]` - get-workers-by-GID service task
- Returns - workers ID's array, `NULL` otherwise

14.4.6.4.16 doca_urom_worker

An opaque structure representing a DOCA UROM worker context.

```
struct doca_urom_worker;
```

14.4.6.4.17 doca_urom_worker_cmd_task

An opaque structure representing a DOCA UROM worker command task context.

```
struct doca_urom_worker_cmd_task;
```

14.4.6.4.18 doca_urom_worker_cmd_task_completion_cb_t

A worker command task completion callback type. It is called once the worker task is completed.

```
typedef void (*doca_urom_worker_cmd_task_completion_cb_t)(struct doca_urom_worker_cmd_task *task,
                                                          union doca_data task_user_data,
                                                          union doca_data ctx_user_data);
```

- `task[in]` - a pointer to worker command task
- `task_user_data[in]` - user task data
- `ctx_user_data[in]` - user worker context data

14.4.6.4.19 doca_urom_worker_create

This method creates a UROM worker context.

A worker is created in a `DOCA_CTX_STATE_IDLE` state. After creation, a user may configure the worker using setter methods (e.g., `doca_urom_worker_set_service()`). Before use, a worker must be transitioned to state `DOCA_CTX_STATE_RUNNING` using the `doca_ctx_start()` interface. A typical invocation looks like `doca_ctx_start(doca_urom_worker_as_ctx(worker_ctx))` .

```
doca_error_t doca_urom_worker_create(struct doca_urom_worker **worker_ctx);
```

- `worker_ctx [in/out]` - `doca_urom_worker` object to be created
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.20 doca_urom_worker_destroy

Destroys a UROM worker context.

```
doca_error_t doca_urom_worker_destroy(struct doca_urom_worker *worker_ctx);
```

- `worker_ctx [in]` - `doca_urom_worker` object to be destroyed. It is created by `doca_urom_worker_create()` .
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.21 doca_urom_worker_set_service

Attaches a UROM service to the worker context. The worker is launched on the DOCA device managed by the provided service context.

```
doca_error_t doca_urom_worker_set_service(struct doca_urom_worker *worker_ctx, struct doca_urom_service
*service_ctx);
```

- `service_ctx [in]` - service context to set
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.22 doca_urom_worker_set_id

This method sets the worker context ID to be used to identify the worker. Worker IDs enable an application to establish multiple connections to the same worker process running on a DOCA device.

Worker ID must be unique to a UROM service.

- If `DOCA_UROM_WORKER_ID_ANY` is specified, the service assigns a unique ID for the newly created worker.
- If a specific ID is used, the service looks for an existing worker with matching ID. If one exists, the service establishes a new connection to the existing worker. If a matching worker does not exist, a new worker is created with the specified worker ID.

```
doca_error_t doca_urom_worker_set_id(struct doca_urom_worker *worker_ctx, uint64_t worker_id);
```

- `worker_ctx [in]` - `doca_urom_worker` object
- `worker_id [in]` - worker ID
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.23 doca_urom_worker_set_gid

Set worker group ID. This ID must be set before starting the worker context.

Through service get workers by GID task, the application can have the list of workers' IDs which are running on DOCA device and that belong to the same group ID.

```
doca_error_t doca_urom_worker_set_gid(struct doca_urom_worker *worker_ctx, uint32_t gid);
```

- `worker_ctx [in]` - `doca_urom_worker` object
- `gid [in]` - worker group ID
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.24 doca_urom_worker_set_plugins

Adds a plugin mask for the supported plugins by the UROM worker on the DPU. The application can use up to 62 plugins.

```
doca_error_t doca_urom_worker_set_plugins(struct doca_urom_worker *worker_ctx, uint64_t plugins);
```

- `worker_ctx[in]` - `doca_urom_worker` object
- `plugins[in]` - an ORing set of worker plugin IDs
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.25 doca_urom_worker_set_env

Set worker environment variables when spawning worker on DPU side by DOCA UROM service. They must be set before starting the worker context.

> ⓘ This call fails if the worker already spawned on the DPU.

```
doca_error_t doca_urom_worker_set_env(struct doca_urom_worker *worker_ctx, char *const env[], size_t count);
```

- `worker_ctx [in]` - `doca_urom_worker` object
- `env [in]` - an array of environment variables

- `count [in]` - array size
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.26 doca_urom_worker_as_ctx

Convert a `doca_urom_worker` object into a DOCA object.

```
struct doca_ctx *doca_urom_worker_as_ctx(struct doca_urom_worker *worker_ctx);
```

- `worker_ctx[in]` - a pointer to `doca_urom_worker` object
- Returns - a pointer to the `doca_ctx` object on success, `NULL` otherwise

14.4.6.4.27 doca_urom_worker_cmd_task_allocate_init

Allocate worker command task and set task attributes.

```
doca_error_t doca_urom_worker_cmd_task_allocate_init(struct doca_urom_worker *worker_ctx, uint64_t plugin, struct
doca_urom_worker_cmd_task **task);
```

- `worker_ctx [in]` - a pointer to `doca_urom_worker` object
- `plugin [in]` - task plugin ID
- `task [out]` - set worker command new task
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.28 doca_urom_worker_cmd_task_release

Release worker command task.

```
doca_error_t doca_urom_worker_cmd_task_release(struct doca_urom_worker_cmd_task *task);
```

- `task[in]` - worker task to release
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.29 doca_urom_worker_cmd_task_set_plugin

Set worker command task plugin ID. The plugin ID is created by the UROM service and the plugin host interface should hold it to create UROM worker command tasks.

```
void doca_urom_worker_cmd_task_set_plugin(struct doca_urom_worker_cmd_task *task, uint64_t plugin);
```

- `task [in]` - worker task
- `plugin [in]` - task plugin to set

14.4.6.4.30 doca_urom_worker_cmd_task_set_cb

Set worker command task completion callback.

```
void doca_urom_worker_cmd_task_set_cb(struct doca_urom_worker_cmd_task *task,
doca_urom_worker_cmd_task_completion_cb_t cb);
```

- `task[in]` - worker task
- `plugin[in]` - task callback to set

14.4.6.4.31 doca_urom_worker_cmd_task_get_payload

Get worker command task payload. The plugin interface populates this buffer by plugin command structure. The payload size is the maximum message size in the DOCA UROM communication channel (the user can configure the size by calling `doca_urom_service_set_max_comm_msg_size()`). To update the payload buffer, the user should call `doca_buf_set_data()` .

```
struct doca_buf *doca_urom_worker_cmd_task_get_payload(struct doca_urom_worker_cmd_task *task);
```

- `task [in]` - worker task
- Returns - a `doca_buf` that represents the task's payload

14.4.6.4.32 doca_urom_worker_cmd_task_get_response

Get worker command task response. To get the response's buffer, the user should call `doca_buf_get_data()` .

```
struct doca_buf *doca_urom_worker_cmd_task_get_response(struct doca_urom_worker_cmd_task *task);
```

- `task [in]` - worker task
- Returns - a `doca_buf` that represents the task's response

14.4.6.4.33 doca_urom_worker_cmd_task_get_user_data

Get worker command user data to populate. The data refers to the reserved data inside the task that the user can get when calling the completion callback. The maximum data size is 32 bytes.

```
void *doca_urom_worker_cmd_task_get_user_data(struct doca_urom_worker_cmd_task *task);
```

- `task [in]` - worker task
- Returns - a pointer to user data memory

14.4.6.4.34 doca_urom_worker_cmd_task_as_task

Convert a `doca_urom_worker_cmd_task` object into a DOCA task object.

After creating a worker command task and configuring it using setter methods (e.g., `doca_urom_worker_cmd_task_set_plugin()`) or as part of task allocation, the user should submit the task by calling `doca_task_submit` .

A typical invocation looks like `doca_task_submit(doca_urom_worker_cmd_task_as_task(task))` .

```
struct doca_task *doca_urom_worker_cmd_task_as_task(struct doca_urom_worker_cmd_task *task);
```

- `task[in]` - worker command task
- Returns - a pointer to the `doca_task` object on success, `NULL` otherwise

14.4.6.4.35 doca_urom_domain

An opaque structure representing a DOCA UROM domain context.

```
struct doca_urom_domain;
```

14.4.6.4.36 doca_urom_domain_allgather_cb_t

A callback for a non-blocking all-gather operation.

```
typedef doca_error_t (*doca_urom_domain_allgather_cb_t)(void *sbuf, void *rbuf, size_t msglen, void *coll_info,
void **req);
```

- `sbuf [in]` - local buffer to send to other processes
- `rbuf [in]` - global buffer to include other process's source buffer
- `msglen [in]` - source buffer length
- `coll_info [in]` - collection info
- `req [in]` - allgather request data
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.37 doca_urom_domain_req_test_cb_t

A callback to test the status of a non-blocking allgather request.

```
typedef doca_error_t (*doca_urom_domain_req_test_cb_t)(void *req);
```

- `req [in]` - allgather request data to check status
- Returns - `DOCA_SUCCESS` on success, `DOCA_ERROR_IN_PROGRESS` otherwise

14.4.6.4.38 doca_urom_domain_req_free_cb_t

A callback to free a non-blocking allgather request.

```
typedef doca_error_t (*doca_urom_domain_req_free_cb_t)(void *req);
```

- `req [in]` - allgather request data to release.
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.39 doca_urom_domain_oob_coll

Out-of-band communication descriptor for domain creation.

```
struct doca_urom_domain_oob_coll {
    doca_urom_domain_allgather_cb_t allgather;
    doca_urom_domain_req_test_cb_t req_test;
    doca_urom_domain_req_free_cb_t req_free;
    void *coll_info;
    uint32_t n_oob_indexes;
    uint32_t oob_index;
};
```

- `allgather` - non-blocking allgather callback
- `req_test` - request test callback
- `req_free` - request free callback
- `coll_info` - context or metadata required by the OOB collective
- `n_oob_indexes` - number of endpoints participating in the OOB operation (e.g., number of client processes representing domain workers)
- `oob_index` - an integer value that represents the position of the calling processes in the given OOB operation. The data specified by `src_buf` is placed at the offset "`oob_index`*size" in the `recv_buf`.

  > ⚠ `oob_index` must be unique at every calling process and should be in the range [0: `n_oob_indexes`).

14.4.6.4.40 doca_urom_domain_create

Creates a UROM domain context. A domain is created in state `DOCA_CTX_STATE_IDLE`. After creation, a user may configure the domain using setter methods (e.g., `doca_urom_domain_set_workers()`). Before use, a domain must be transitioned to state `DOCA_CTX_STATE_RUNNING` using the `doca_ctx_start()` interface. A typical invocation looks like `doca_ctx_start(doca_urom_domain_as_ctx(worker_ctx))`.

```
doca_error_t doca_urom_domain_create(struct doca_urom_domain **domain_ctx);
```

- `domain_ctx [in/out]` - `doca_urom_domain` object to be created
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.41 doca_urom_domain_destroy

Destroys a UROM domain context.

```
doca_error_t doca_urom_domain_destroy(struct doca_urom_domain *domain_ctx);
```

- `domain_ctx [in]` - `doca_urom_domain` object to be destroyed; it is created by `doca_urom_domain_create()`
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.42 doca_urom_domain_set_workers

Sets the list of workers in the domain.

```
doca_error_t doca_urom_domain_set_workers(struct doca_urom_domain *domain_ctx, uint64_t *domain_worker_ids, struct
doca_urom_worker **workers, size_t workers_cnt);
```

- `domain_ctx [in]` - `doca_urom_domain` object

- `domain_worker_ids [in]` - list of domain worker IDs
- `workers [in]` - an array of UROM worker contexts that should be part of the domain
- `workers_cnt [in]` - the number of workers in the given array
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.43 doca_urom_domain_add_buffer

Attaches local buffer attributes to the domain. It should be called after calling `doca_urom_domain_set_buffers_count()`.

The local buffer will be shared with all workers belonging to the domain.

```
doca_error_t doca_urom_domain_add_buffer(struct doca_urom_domain *domain_ctx, void *buffer, size_t buf_len, void
 *memh, size_t memh_len, void *mkey, size_t mkey_len);
```

- `domain_ctx [in]` - `doca_urom_domain` object
- `buffer [in]` - buffer ready for remote access which is given to the domain
- `buf_len [in]` - buffer length
- `memh [in]` - memory handle for the exported buffer. (should be packed)
- `memh_len [in]` - memory handle size
- `mkey [in]` - memory key for the exported buffer. (should be packed)
- `mkey_len [in]` - memory key size
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.44 doca_urom_domain_set_oob

Sets OOB communication info to be used for domain initialization.

```
doca_error_t doca_urom_domain_set_oob(struct doca_urom_domain *domain_ctx, struct doca_urom_domain_oob_coll *oob);
```

- `domain_ctx [in]` - `doca_urom_domain` object
- `oob [in]` - OOB communication info to set
- Returns - `DOCA_SUCCESS` on success, error code otherwise

14.4.6.4.45 doca_urom_domain_as_ctx

Convert a `doca_urom_domain` object into a DOCA object.

```
struct doca_ctx *doca_urom_domain_as_ctx(struct doca_urom_domain *domain_ctx);
```

- `domain_ctx[in]` - a pointer to `doca_urom_domain` object
- Returns - a pointer to the `doca_ctx` object on success, `NULL` otherwise

14.4.6.5 Execution Model

DOCA UROM uses the DOCA Core Progress Engine as an execution model for service and worker contexts and tasks progress. For more details about it please refer to this guide.

14.4.6.6 UROM Building Blocks

This section explains the general concepts behind the fundamental building blocks to use when creating a DOCA UROM application and offloading functionality.

14.4.6.6.1 Program Flow

14.4.6.6.1.1 DPU

Launching DOCA UROM Service

DOCA UROM service should be run before running the application on the host to offload commands to BlueField. For more information, refer to the NVIDIA DOCA UROM Service Guide.

14.4.6.6.1.2 Host

Initializing UROM Service Context

1. Create service context: Establish a service context within the control plane alongside the progress engine.
2. Set service context attributes: Specific attributes of the service context are configured. The required attribute is `doca_dev` .
3. Start the service context: The service context is initiated by invoking the `doca_ctx_start` f unction.
    a. Discover BlueField availability: The UROM library identifies the available BlueField device.
    b. Connect to UROM service: The library establishes a connection to the UROM service. The connection process is synchronized, meaning that the host process and the BlueField service process are blocked until the connection is established.
    c. Perform lookup using UROM service file: A lookup operation is executed using the UROM service file. The path to this file should be specified in the `DOCA_UROM_SERVICE_FILE` environment variable. More information can be found in `doca_urom.h` .
4. Switch context state to `DOCA_CTX_STATE_RUNNING` : The context state transitions to `DOCA_CTX_STATE_RUNNING` at this point.
5. Service context waits for worker bootstrap requests: The service context is now in a state where it awaits and handles worker bootstrap requests.

```
/* Create DOCA UROM service instance */
doca_urom_service_create(&service);

/* Connect service context to DOCA progress engine */
doca_pe_connect_ctx(pe, doca_urom_service_as_ctx(service));

/* Set service attributes */
doca_urom_service_set_max_workers(service, nb_workers)
doca_urom_service_set_dev(service, dev);

/* Start service context */
doca_ctx_start(doca_urom_service_as_ctx(service));

/* Handling workers bootstrap requests */
do {
    doca_pe_progress(pe);
} while (!are_all_workers_started);
```

Picking UROM Worker Offload Functionality

Once the service context state is running, the application can call `doca_urom_service_get_plugins_list()` to get the available plugins on the local BlueField device where the UROM service is running.

The UROM service generates an identifier for each plugin and the application is responsible for forwarding this ID to the host plugin interface for sending commands and receiving notifications by calling `urom_<plugin_name>_init(<plugin_id>, <plugin_version>)`.

```
const char *plugin_name = "worker_rdmo";
struct doca_urom_service *service;
const struct doca_urom_service_plugin_info *plugins, *rdmo_info;

/* Create and Start UROM service context. */
..

/* Get worker plugins list. */
doca_urom_service_get_plugins_list(*service, &plugins, &plugins_count);

/* Check if RDMO plugin exists. */
for (i = 0; i < plugins_count; i++) {
        if (strcmp(plugin_name, plugins[i].plugin_name) == 0) {
            rdmo_info = &plugins[i];
            break;
        }
}

/* Attach RDMO plugin ID and DPU plugin version for compatibility check */
urom_rdmo_init(rdmo_info->id, rdmo_info->version);
```

Initializing UROM Worker Context

1. Create a service context and connect the worker context to DOCA Progress Engine (PE).
2. Set worker context attributes (in the example below worker plugin is RDMO).
3. Start worker context, submitting internally spawns worker requests on the service context.
4. Worker context state changes to `DOCA_CTX_STATE_STARTING` (this process is asynchronous).
5. Wait until the worker context state changes to `DOCA_CTX_STATE_RUNNING`:
    a. When calling `doca_pe_progress`, check for a response from the service context that the spawning worker on BlueField is done.
    b. If the worker is spawned on BlueField, connect to it and change the status to running.

```
const struct doca_urom_service_plugin_info *rdmo_info;

/* Create DOCA UROM worker instance */
doca_urom_worker_create(&worker);

/* Connect worker context to DOCA progress engine */
doca_pe_connect_ctx(pe, doca_urom_worker_as_ctx(worker));

/* Set worker attributes */
doca_urom_worker_set_service(worker, service);
doca_urom_worker_set_id(worker, worker_id);
doca_urom_worker_set_max_inflight_tasks(worker, nb_tasks);
doca_urom_worker_set_plugins(worker, rdmo_info->id);
doca_urom_worker_set_cpuset(worker, cpuset);

/* Start UROM worker context */
doca_ctx_start(doca_urom_worker_as_ctx(worker));

/* Progress until worker state changes to running or error happened */
do {
    doca_pe_progress(pe);
    result = doca_ctx_get_state(doca_urom_worker_as_ctx(worker), &state);
} while (state == DOCA_CTX_STATE_STARTING);
```

Offloading Plugin Task

Once the worker context state is `DOCA_CTX_STATE_RUNNING`, the worker is ready to execute offload tasks. The example below is for offloading an RDMO command.

1. Prepare RDMO task arguments (e.g., completion callback).
2. Call the task function from the plugin host interface.

3. Poll for completion by calling `doca_pe_progress`.
4. Get completion notification through the user callback.

```
    int ret;
    struct doca_urom_worker *worker;
    struct rdmo_result res = {0};
    union doca_data cookie = {0};
    size_t server_worker_addr_len;
    ucp_address_t *server_worker_addr;

    cookie.ptr = &res;
    res.result = DOCA_SUCCESS;

    ucp_worker_create(*ucp_context, &worker_params, server_ucp_worker);
    ucp_worker_get_address(*server_ucp_worker, &server_worker_addr, &server_worker_addr_len);

    /* Create and submit RDMO client init task */
    urom_rdmo_task_client_init(worker, cookie, 0, server_worker_addr, server_worker_addr_len,
urom_rdmo_client_init_finished);

    /* Wait for completion */
    do {
        ret = doca_pe_progress(pe);
        ucp_worker_progress(*server_ucp_worker);
    } while (ret == 0 && res.result == DOCA_SUCCESS);

    /* Check task result */
    if (res.result != DOCA_SUCCESS)
        DOCA_LOG_ERR("Client init task finished with error");
```

Initializing UROM Domain Context

1. Create a domain context on the control plane PE.
2. Set domain context attributes.
3. Start the domain context by calling `doca_ctx_start`.
    a. Exchange memory descriptors between all workers.
4. Wait until the domain context state is running.

```
/* Create DOCA UROM domain instance */
doca_urom_domain_create(&domain);

/* Connect domain context to DOCA progress engine */
doca_pe_connect_ctx(pe, doca_urom_domain_as_ctx(domain));;

/* Set domain attributes */
doca_urom_domain_set_oob(domain, oob);
doca_urom_domain_set_workers(domain, worker_ids, workers, nb_workers);
doca_urom_domain_set_buffers_count(domain, nb_buffers);
for each buffer:
    doca_urom_domain_add_buffer(domain);

/* Start domain context */
doca_ctx_start(doca_urom_domain_as_ctx(domain));

/* Loop till domain state changes to running */
do {
    doca_pe_progress(pe);
    result = doca_ctx_get_state(doca_urom_domain_as_ctx(domain), &state);
} while (state == DOCA_CTX_STATE_STARTING && result == DOCA_SUCCESS);
```

Destroying UROM Domain Context

1. Request the domain context to stop by calling `doca_ctx_stop` .
2. Clean up resources by destroying the domain context.

```
/* Request domain context stop */
doca_ctx_stop(doca_urom_domain_as_ctx(domain));

/* Destroy domain context */
doca_urom_domain_destroy(domain);
```

Destroying UROM Worker Context

1. Request the worker context stop by calling `doca_ctx_stop` and posting the destroy command on the service context.
2. Wait until a completion for the destroy command is received.
    a. Change worker state to idle.
3. Clean up resources.

```
/* Stop worker context */
doca_ctx_stop(doca_urom_worker_as_ctx(worker));

/* Progress till receiving a completion */
do {
    doca_pe_progress(pe);
    doca_ctx_get_state(doca_urom_worker_as_ctx(worker), &state);
} while (state != DOCA_CTX_STATE_IDLE);

/* Destroy worker context */
doca_urom_worker_destroy(worker);
```

Destroying UROM Service Context

1. Wait for the completion of the UROM worker context commands.
2. Once all UROM workers have been successfully destroyed, initiate service context stop by invoking `doca_ctx_stop`.
3. Disconnect from the UROM service.
4. Perform resource cleanup.

```
/* Handling workers teardown requests*/
do {
    doca_pe_progress(pe);
} while (!are_all_workers_exited);

/* Stop service context */
doca_ctx_stop(doca_urom_service_as_ctx(service));

/* Destroy service context */
doca_urom_service_destroy(service);
```

14.4.6.6.2 Plugin Development

14.4.6.6.2.1 Developing Offload Plugin on DPU

1. Implement `struct urom_plugin_iface` methods.
    a. The `open()` method initializes the plugin connection state and may create an endpoint to perform communication with other processes/workers.

```
static doca_error_t urom_worker_rdmo_open(struct urom_worker_ctx *ctx)
{
    ucp_context_h ucp_context;
    ucp_worker_h ucp_worker;
    struct urom_worker_rdmo *rdmo_worker;

    rdmo_worker = calloc(1, sizeof(*rdmo_worker));
    if (rdmo_worker == NULL)
        return DOCA_ERROR_NO_MEMORY;

    ctx->plugin_ctx = rdmo_worker;

    /* UCX transport layer initialization */
    .
    .
    .

    /* Create UCX worker Endpoint */
    ucp_worker_create(ucp_context, &worker_params, &ucp_worker);
    ucp_worker_get_address(ucp_worker, &rdmo_worker->ucp_data.worker_address, &rdmo_worker-
>ucp_data.ucp_addrlen);

    /* Resources  initialization */
    rdmo_worker->clients = kh_init(client);
    rdmo_worker->eps = kh_init(ep);

    /* Init completions list, UROM worker checks completed requests by calling progress() method */
    ucs_list_head_init(&rdmo_worker->completed_reqs);

    return DOCA_SUCCESS;
}
```

b. The `addr()` method returns the address of the plugin endpoint generated during `open()` if it exists (e.g., UCX endpoint to communicate with other UROM workers).

c. The `worker_cmd()` method is used to parse and start work on incoming commands to the plugin.

```
static doca_error_t urom_worker_rdmo_worker_cmd(struct urom_worker_ctx *ctx, ucs_list_link_t
*cmd_list)
{
    struct urom_worker_rdmo_cmd *rdmo_cmd;
    struct urom_worker_cmd_desc *cmd_desc;
    struct urom_worker_rdmo *rdmo_worker = (struct urom_worker_rdmo *)ctx->plugin_ctx;

    while (!ucs_list_is_empty(cmd_list)) {
        /* Get new RDMO command from the list */
        cmd_desc = ucs_list_extract_head(cmd_list, struct urom_worker_cmd_desc, entry);

        /* Unpack and deserialize RDMO command */
        urom_worker_rdmo_cmd_unpack(&cmd_desc->worker_cmd, cmd_desc->worker_cmd.len, &cmd);

        rdmo_cmd = (struct urom_worker_rdmo_cmd *)cmd->plugin_cmd;
        /* Handle command according to it's type */
        switch (rdmo_cmd->type) {
        case UROM_WORKER_CMD_RDMO_CLIENT_INIT:
            /* Handle RDMO client init command */
            status = urom_worker_rdmo_client_init_cmd(rdmo_worker, cmd_desc);
            break;
        case UROM_WORKER_CMD_RDMO_RQ_CREATE:
            /* Handle RDMO RQ create command */
            status = urom_worker_rdmo_rq_create_cmd(rdmo_worker, cmd_desc);
            break;
         .
         .
         .
        default:
            DOCA_LOG_INFO("Invalid RDMO command type: %lu", rdmo_cmd->type);
            status = DOCA_ERROR_INVALID_VALUE;
            break;
        }
        free(cmd_desc);
        if (status != DOCA_SUCCESS)
            return status;
    }
    return status;
}
```

d. The `progress()` method is used to give CPU time to the plugin code to advance asynchronous tasks.

```
static doca_error_t urom_worker_rdmo_progress(struct urom_worker_ctx *ctx, ucs_list_link_t
*notif_list)
{
    struct urom_worker_notif_desc *nd;
    struct urom_worker_rdmo *rdmo_worker = (struct urom_worker_rdmo *)ctx->plugin_ctx;

    /* RDMO UCP worker progress */
    ucp_worker_progress(rdmo_worker->ucp_data.ucp_worker);

    /* Check if completion list is empty */
    if (ucs_list_is_empty(&rdmo_worker->completed_reqs))
        return DOCA_ERROR_EMPTY;

    /* Pop completed commands from the list */
    while (!ucs_list_is_empty(&rdmo_worker->completed_reqs)) {
        nd = ucs_list_extract_head(&rdmo_worker->completed_reqs, struct urom_worker_notif_desc,
entry);
        ucs_list_add_tail(notif_list, &nd->entry);
    }
    return DOCA_SUCCESS;
}
```

e. The `notif_pack()` method is used to serialize notifications before they are sent back to the host.

2. Implement and expose the following symbols:

   a. `doca_error_t urom_plugin_get_version(uint64_t *version);`
      Returns a compile-time constant value stored within the `.so` file and is used to verify that the host and DPU plugin versions are compatible.
   b. `doca_error_t urom_plugin_get_iface(struct urom_plugin_iface *iface);`
      Get the `urom_plugin_iface` struct with methods implemented by the plugin.
3. Compile the user plugin as an `.so` file and place it where the UROM service can access it. For more details, refer to section "Plugin Discovery and Reporting" under the NVIDIA DOCA UROM Service Guide.

14.4.6.6.2.2 Creating Plugin Host Task

1. Allocate and init worker command task.
2. Populate payload buffer by task command.
3. Pack and serialize the command.
4. Set user data.
5. Submit the task.

```
doca_error_t urom_rdmo_task_client_init(struct doca_urom_worker *worker_ctx, union doca_data cookie,
uint32_t id, void *addr, uint64_t addr_len, urom_rdmo_client_init_finished cb)
{
    doca_error_t result;
    size_t pack_len = 0;
    struct doca_buf *payload;
    struct doca_urom_worker_cmd_task *task;
    struct doca_rdmo_task_data *task_data;
    struct urom_worker_rdmo_cmd *rdmo_cmd;

    /* Allocate task */
    doca_urom_worker_cmd_task_allocate_init(worker_ctx, rdmo_id, &task);

    /* Get payload buffer */
    payload = doca_urom_worker_cmd_task_get_payload(task);
    doca_buf_get_data(payload, (void **)&rdmo_cmd);
    doca_buf_get_data_len(payload, &pack_len);

    /* Populate commands attributes */
    rdmo_cmd->type = UROM_WORKER_CMD_RDMO_CLIENT_INIT;
    rdmo_cmd->client_init.id = id;
    rdmo_cmd->client_init.addr = addr;
    rdmo_cmd->client_init.addr_len = addr_len;

    /* Pack and serialize the command */
    urom_worker_rdmo_cmd_pack(rdmo_cmd, &pack_len, (void *)rdmo_cmd);

    /* Update payload data size */
    doca_buf_set_data(payload, rdmo_cmd, pack_len);

    /* Set user data */
    task_data = (struct doca_rdmo_task_data *)doca_urom_worker_cmd_task_get_user_data(task);
    task_data->client_init_cb = cb;
    task_data->cookie = cookie;

    /* Set task plugin callback */
    doca_urom_worker_cmd_task_set_cb(task, urom_rdmo_client_init_completed);

    /* Submit task */
    doca_task_submit(doca_urom_worker_cmd_task_as_task(task));

    return DOCA_SUCCESS;
}
```

14.4.6.7 DOCA UROM Samples

This section provides DOCA UROM library sample implementations on top of BlueField.

The samples illustrate how to use the DOCA UROM API to do the following:

- Define and create a UROM plugin host and DPU versions for offloading HPC/AI tasks
- Build host applications that use the plugin to execute jobs on BlueField by the DOCA UROM service and workers

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

14.4.6.7.1 Sample Prerequisite

Sample	Type	Prerequisite
Sandbox	Plugin	A plugin which offloads the UCX tagged send/receive API
Graph	Plugin	The plugin uses UCX data structures and UCX endpoint
UROM Ping Pong	Program	The sample uses the Open MPI package as a launcher framework to launch two processes in parallel

14.4.6.7.2 Running the Sample

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples
2. To build a given sample:

```
cd /opt/mellanox/doca/samples/doca_urom/<sample_name>
meson /tmp/build
ninja -C /tmp/build
```

The binary `doca_<sample_name>` is created under `/tmp/build/`.

3. UROM Sample arguments:

Sample	Argument	Description
UROM multi-workers bootstrap	`-d, --device <IB device name>`	IB device name
UROM Ping Pong	`-d, --device <IB device name>`	IB device name
	`-m, --message`	Specify ping pong message

4. For additional information per sample, use the `-h` option:

```
/tmp/build/doca_<sample_name> -h
```

14.4.6.7.3 UROM Plugin Samples

DOCA UROM plugin samples have two components. The first one is the host component which is linked with UROM host programs. The second is the DPU component which is compiled as an `.so` file and is loaded at runtime by the DOCA UROM service (daemon, workers).

To build a given plugin:

```
cd /opt/mellanox/doca/samples/doca_urom/plugins/worker_<plugin_name>
meson /tmp/build
ninja -C /tmp/build
```

> The binary `worker_<sample_name>.so` file is created under `/tmp/build/`.

14.4.6.7.3.1 Graph

This plugin provides a simple example for creating a UROM plugin interface. It exposes only a single command loopback, sending a specific value in the command, and expects to receive the same value in the notification from UROM worker.

References:

- `/opt/mellanox/doca/samples/doca_urom/plugins/worker_graph/meson.build`
- `/opt/mellanox/doca/samples/doca_urom/plugins/worker_graph/urom_graph.h`
- `/opt/mellanox/doca/samples/doca_urom/plugins/worker_graph/worker_graph.c`
- `/opt/mellanox/doca/samples/doca_urom/plugins/worker_graph/worker_graph.h`

14.4.6.7.3.2 Sandbox

This plugin provides a set of commands for using the offloaded ping pong communication operation.

References:

- `/opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox/meson.build`
- `/opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox/urom_sandbox.h`
- `/opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox/worker_sandbox.c`
- `/opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox/worker_sandbox.h`

14.4.6.7.4 UROM Program Samples

DOCA UROM program samples can run only on the host side and require at least one DOCA UROM service instance to be running on BlueField.

The environment variable should be set `DOCA_UROM_SERVICE_FILE` to the path to the UROM service file.

14.4.6.7.4.1 UROM Multi-worker Bootstrap

This sample illustrates how to properly initialize DOCA UROM interfaces and use the API to spawn multiple workers on the same application process.

The sample initiates four threads as UROM workers to execute concurrently, alongside the main thread operating as a UROM service. It divides the workers into two groups based on their IDs, with odd-numbered workers in one group and even-numbered workers in the other.

Each worker executes the data loopback command by using the Graph plugin, sends a specific value, and expects to receive the same value in the notification.

The sample logic includes:

1. Opening DOCA IB device.
2. Initializing needed DOCA core structures.
3. Creating and starting UROM service context.
4. Initiating the Graph plugin host interface by attaching the generated plugin ID.
5. Launching 4 threads and for each of them:
    a. Creating and starting UROM worker context.
    b. Once the worker context switches to running, sending the loopback graph command to wait until receiving a notification.
    c. Verifying the received data.
    d. Waiting until an interrupt signal is received.
6. The main thread checking for pending jobs of spawning workers (4 jobs, one per thread).
7. Waiting until an interrupt signal is received.
8. The main thread checking for pending jobs of destroying workers (4 jobs, one per thread) for exiting.
9. Cleaning up and exiting.

References:

- `/opt/mellanox/doca/samples/doca_urom/urom_multi_workers_bootstrap/urom_multi_workers_bootstrap_sample.c`
- `/opt/mellanox/doca/samples/doca_urom/urom_multi_workers_bootstrap/urom_multi_workers_bootstrap_main.c`
- `/opt/mellanox/doca/samples/doca_urom/urom_multi_workers_bootstrap/meson.build`
- `/opt/mellanox/doca/samples/doca_urom/urom_common.c`
- `/opt/mellanox/doca/samples/doca_urom/urom_common.h`

14.4.6.7.4.2 UROM Ping Pong

This sample illustrates how to properly initialize the DOCA UROM interfaces and use its API to create two different workers and run ping pong between them by using Sandbox plugin-based UCX.

The sample is using Open MPI to launch two different processes, one process as server and the second one as client, the flow is decided according to process rank.

The sample logic per process includes:

1. Initializing MPI.
2. Opening DOCA IB device.
3. Creating and starting UROM service context.
4. Initiating the Sandbox plugin host interface by attaching the generated plugin id.
5. Creating and starting UROM worker context.
6. Creating and starting domain context.

7. Through domain context, the sample processes exchange the worker's details to communicate between them on the BlueField side for ping pong flow.
8. Starting ping pong flow between the processes, each process offloading the commands to its worker on the BlueField side.
9. Verifying that ping pong is finished successfully.
10. Destroying the domain context.
11. Destroying the worker context.
12. Destroying the service context.

References:

- `/opt/mellanox/doca/samples/doca_urom/urom_ping_pong/urom_ping_pong_sample.c`
- `/opt/mellanox/doca/samples/doca_urom/urom_ping_pong/urom_ping_pong_main.c`
- `/opt/mellanox/doca/samples/doca_urom/urom_ping_pong/meson.build`
- `/opt/mellanox/doca/samples/doca_urom/urom_common.c`
- `/opt/mellanox/doca/samples/doca_urom/urom_common.h`

14.4.7 DOCA RDMA

This guide provides an overview and configuration instructions for the DOCA RDMA API.

14.4.7.1 Introduction

> This library is currently supported at beta level only.

DOCA RDMA enables direct access to the memory of remote machines, without interrupting the processing of their CPUs or operating systems. Avoiding CPU interruptions reduces context switching for I/O operations, leading to lower latency and higher bandwidth compared to traditional network communication methods.

DOCA RDMA library provides an API to execute the various RDMA operations.

This document is intended for software developers wishing to improve their applications by utilizing RDMA operations.

> RDMA operations should be executed over a secure channel in a production deployment, given the sensitivity that arises from the nature of the protocol.

14.4.7.2 Prerequisites

This library follows the architecture of a DOCA Core Context, it is recommended read the following sections before proceeding:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

14.4.7.3 Environment

DOCA RDMA-based applications can run either on the host machine or on the NVIDIA® BlueField® networking platform (DPU or SuperNIC).

14.4.7.4 Architecture

DOCA RDMA is a DOCA Context as defined by DOCA Core. See NVIDIA DOCA Core Context for more information.

DOCA RDMA consists of two connected sides, passing data between one another. This includes the option for one side to access the remote side's memory if the granted permissions allow it.

The connection between the two sides can either be based on InfiniBand (IB) or based on Ethernet using RoCE. Currently, only reliable connection (RC) transport type is supported.

DOCA RDMA leverages the Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

The supported operations that may be executed between the two sides, using DOCA RDMA, are:

- Receive
- Send
- Send with immediate
- Write
- Write with immediate
- Read
- Atomic compare and swap
- Atomic fetch and add
- Get remote DOCA Sync Event
- Set remote DOCA Sync Event
- Add remote DOCA Sync Event

14.4.7.4.1 Objects

14.4.7.4.1.1 Device

The RDMA library requires a DOCA device to operate. This device is used to utilize the connection between the peers in RDMA, access memory, and perform the different operations.

> ⚠ The device must stay valid until the RDMA instance is destroyed.

14.4.7.4.1.2 Memory Map

Executing any DOCA RDMA operation in which data is passed between the peers requires creating a memory map (mmap) on each side.

- The mmap's permissions must include the relevant RDMA permission, according to the required RDMA operations. Tasks fail in case of insufficient permissions.

> Refer to section "Permissions" for more information.

- To allow the peer to execute RDMA operations, the mmap must be exported, using `doca_mmap_export_rdma()` , and passed to the peer (i.e., the side requesting the RDMA operation) where the remote mmap is created and used to access the memory.

14.4.7.4.1.3 Buffer Inventory and Buffers

Executing any DOCA RDMA operation, in which data is passed between the peers, requires using buffers, and thus requires a buffer inventory as well.

Each operation calls for a different set-up for the buffers in use, this is explicitly explained in the "Tasks" section.

14.4.7.5 Configuration Phase

To start using the library you need to first go through a configuration phase as described in DOCA Core Context Configuration Phase

This section describes how to configure and start the context, to allow execution of tasks and retrieval of events.

14.4.7.5.1 Configurations

The context can be configured to match the application use case.

14.4.7.5.1.1 Mandatory Configurations

These configurations are mandatory and must be set by the application before attempting to start the context:

Task Configurations

At least one task/event type must be configured. See configuration of Tasks and/or Events.

Permissions

Different tasks require different permission to be set for both the RDMA and the mmap in use.

The following table summarizes the necessary RDMA and mmap permissions for each RDMA operation:

DOCA RDMA task Types	Minimal Permissions				Should Export MMAP? [1]
	The Side Submitting the Task		The Peer		
	RDMA	MMAP	RDMA	MMAP	
Read **Get Remote Sync Event**	-	Local read write	RDMA read	Local read write | RDMA read	Yes

Write **Write with Immediate** **Set Remote Sync Event**	-	Local read write	RDMA write	Local read write | RDMA write	Yes
Atomic Compare and Swap **Atomic Fetch and Add** **Add Remote Sync Event**	-	Local read write	RDMA atomic	Local read write | RDMA atomic	Yes
Send **Send with Immediate**	-	Local read write	-	Local read write	No
Receive	Depending on the received task	Local read write	Not relevant		

1. Refers to the peer. A side that only submits tasks is never required to export an mmap.

14.4.7.5.1.2 Optional Configurations

If these configurations are not set, a default value is used.

Users may edit the default properties of the RDMA instance using the `doca_rdma_set_<property>()` . The user may also query the default/set properties using `doca_rdma_cap_get_<property>(struct doca_rdma *, …)` functions.

> The number of tasks that can be submitted in bulk is dependent on the properties `max_send_buf_list_len` and `send_queue_size` .

Refer to Library Capability for querying valid property values when configuring the library context.

14.4.7.5.2 Device Support

DOCA RDMA requires a device to operate. For picking a device, see DOCA Core Device Discovery.

As device capabilities may change in the future, it is recommended to query each `doca_devinfo` for its capabilities relevant to RDMA operations, using `doca_rdma_cap_*(struct doca_devinfo *, …)` functions, and check whether the device is suitable for the required RDMA task types, using `doca_rdma_task_<task_type>_is_supported()` .

BlueField-2 and higher devices are supported:

- On the host, any `doca_dev` is supported
- On the BlueField Platform, applications must provide the library with SFs as a `doca_dev` . See NVIDIA OpenvSwitch Acceleration (OVS in DOCA) and BlueField DPU Scalable Function to see how to create SFs and connect them to the appropriate ports.

> ⓘ An exception to this is when running RDMA on the DPA datapath, which currently only supports PFs.

14.4.7.5.3 Buffer Support

The DOCA RDMA library utilizes different buffer types, depending on the task and the buffer's purpose:

- Local mmap buffer
- Mmap from RDMA export buffer
- Mmap from PCIe export buffers

  > ⓘ This type of buffer can be used in an equivalent manner to local mmap buffers.

- Linked list buffer

For task-specific information, refer to section "Tasks".

14.4.7.5.4 Establishing RDMA Connections

To establish the communication between the peers and allow the execution of different DOCA RDMA tasks, the RDMA instances must be connected.

> ⚠ This should be executed after `doca_ctx_start()` is called.

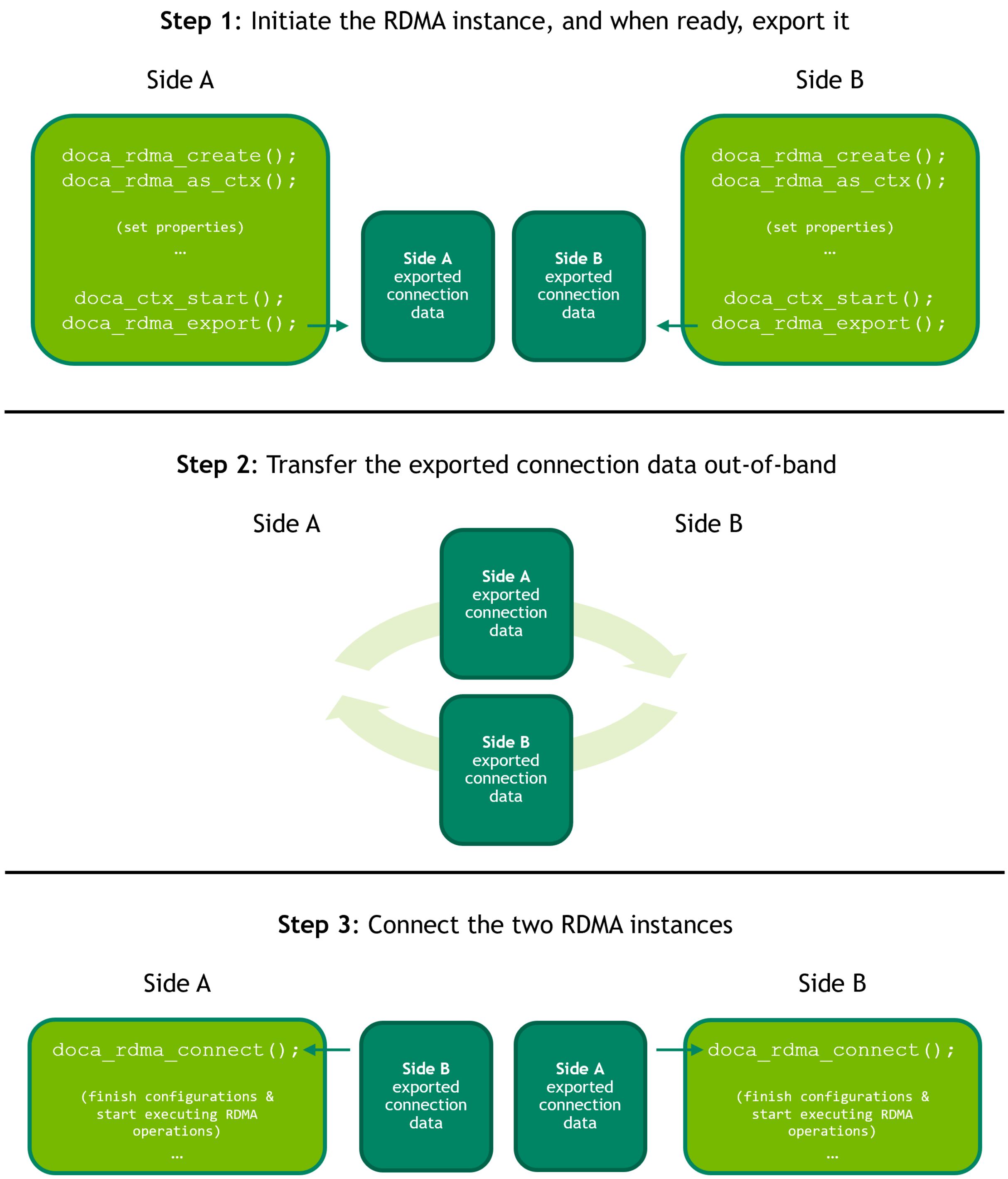
> ⓘ Refer to section "State Machine" for more information.

There are two methods to establish RDMA connections as detailed in the following subsections.

14.4.7.5.4.1 Exporting and Connecting RDMA

Connecting the RDMA instances can be done by exporting each RDMA instance to the remote side to a blob by using `doca_rdma_export()`, transferring the blob to the opposite side, out-of-band (OOB), and providing it as input to the `doca_rdma_connect()` function on that side.

All in all, the configuration flow should be as presented in the following image:



Step 1: Initiate the RDMA instance, and when ready, export it

Side A
Side B
Side A exported connection data
Side B exported connection data

```
doca_rdma_create();
doca_rdma_as_ctx();

(set properties)
…

doca_ctx_start();
doca_rdma_export();
```

```
doca_rdma_create();
doca_rdma_as_ctx();

(set properties)
…

doca_ctx_start();
doca_rdma_export();
```

Step 2: Transfer the exported connection data out-of-band

Side A
Side B
Side A exported connection data
Side B exported connection data

Step 3: Connect the two RDMA instances

Side A
Side B
Side B exported connection data
Side A exported connection data

```
doca_rdma_connect();

(finish configurations &
start executing RDMA
operations)
…
```

```
doca_rdma_connect();

(finish configurations &
start executing RDMA
operations)
…
```

The exported data contains sensitive information. Make sure to pass this data through a secure channel!

14.4.7.5.4.2 Connecting Using RDMA CM Connection Flow

⚠ This connection method is not currently available for DPA/GPU data paths.

The RDMA CM (communication manager) flow uses the server/client scheme where one of the RDMA instances acts as a server for the second RDMA instance (client). The process for both RDMA instances is non-blocking, event driven, and governed by the progress engine (PE). The connection process is reported to both instances by callbacks which should be set with `doca_rdma_set_connection_state_callbacks()`.

There are four state callbacks:

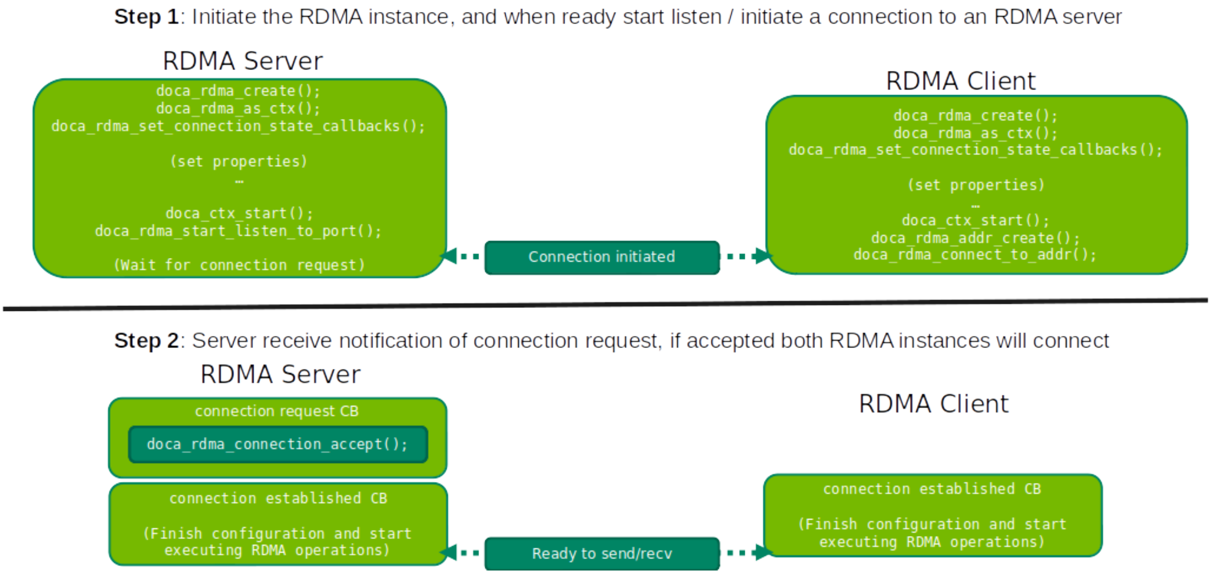
- Connection request callback - This function is called by `doca_pe_progress()` when a connection request is received by an RDMA instance acting as server
- Connection established callback - This function is called by `doca_pe_progress()` when a connection is successfully established between server/client RDMA instances
- Connection failure callback - This function is called by `doca_pe_progress()` when a connection fails to be established between server/client RDMA instances
- Connection disconnection callback - This function is called by `doca_pe_progress()` when a connection disconnects either server/client RDMA instances

A typical connection flow would be as follows:

1. Prior to initiating a connection, the RDMA instance acting as server (i.e., RDMA server) must start active listening for a connection from a remote RDMA peer (using RDMA CM) to a specific port using `doca_rdma_start_listen_to_port()`. An RDMA server can stop listening for a connection from a remote RDMA peer (using RDMA CM) by using `doca_rdma_stop_listen_to_port()`.
2. The RDMA CM instance acting as client (i.e., RDMA client) can now perform an RDMA connection to the RDMA server. As first step it must create an address object by using `doca_rdma_addr_create()`. The parameters to this function correspond to the RDMA server details required to perform a connection. This object can be destroyed by using `doca_rdma_addr_destroy()`, and retrieve it from a connection with `doca_rdma_connection_get_addr()`.
3. The RDMA client can set the connection user data to include in each connection using `doca_rdma_connection_set_user_data()`, and retrieve it from a connection using `doca_rdma_connection_get_user_data()`.
4. The RDMA client can now perform a connection to the RDMA server using `doca_rdma_connect_to_addr()`. Depending on the network topology and configuration, the connection can be established with IPv4, IPv6, or GID.
5. The RDMA server receives a notification with a connection request through the previously set connection request callback function. The RDMA server can decide to accept the connection with `doca_rdma_connection_accept()` or reject the connection with `doca_rdma_connection_reject()`.
    - If the RDMA server rejects the connection or the connection cannot be successfully established, the RDMA server and RDMA client receive a notification through the connection failure callback function.

- If the RDMA server accepts the connection and the connection can be successfully established, the RDMA server and RDMA client receive a notification through the connection established callback function.

6. After the RDMA operation is complete, either side can perform the disconnection process using `doca_rdma_connection_disconnect()`. The RDMA instance that did not initiate the disconnection process receives a notification through the disconnection callback function.



Step 1: Initiate the RDMA instance, and when ready start listen / initiate a connection to an RDMA server

RDMA Server

```
doca_rdma_create();
doca_rdma_as_ctx();
doca_rdma_set_connection_state_callbacks();

(set properties)
…

doca_ctx_start();
doca_rdma_start_listen_to_port();

(Wait for connection request)
```

RDMA Client

```
doca_rdma_create();
doca_rdma_as_ctx();
doca_rdma_set_connection_state_callbacks();

(set properties)
…
doca_ctx_start();
doca_rdma_addr_create();
doca_rdma_connect_to_addr();
```

Step 2: Server receive notification of connection request, if accepted both RDMA instances will connect

RDMA Server

```
connection request CB
doca_rdma_connection_accept();
```

```
connection established CB

(Finish configuration and start
executing RDMA operations)
```

RDMA Client

```
connection established CB

(Finish configuration and start
executing RDMA operations)
```

> The connection process involves resolving the RDMA server connection address. This process is limited to 5 seconds by default, but it can be set using `doca_rdma_set_connection_request_timeout()` and retrieved using `doca_rdma_get_connection_request_timeout()`.

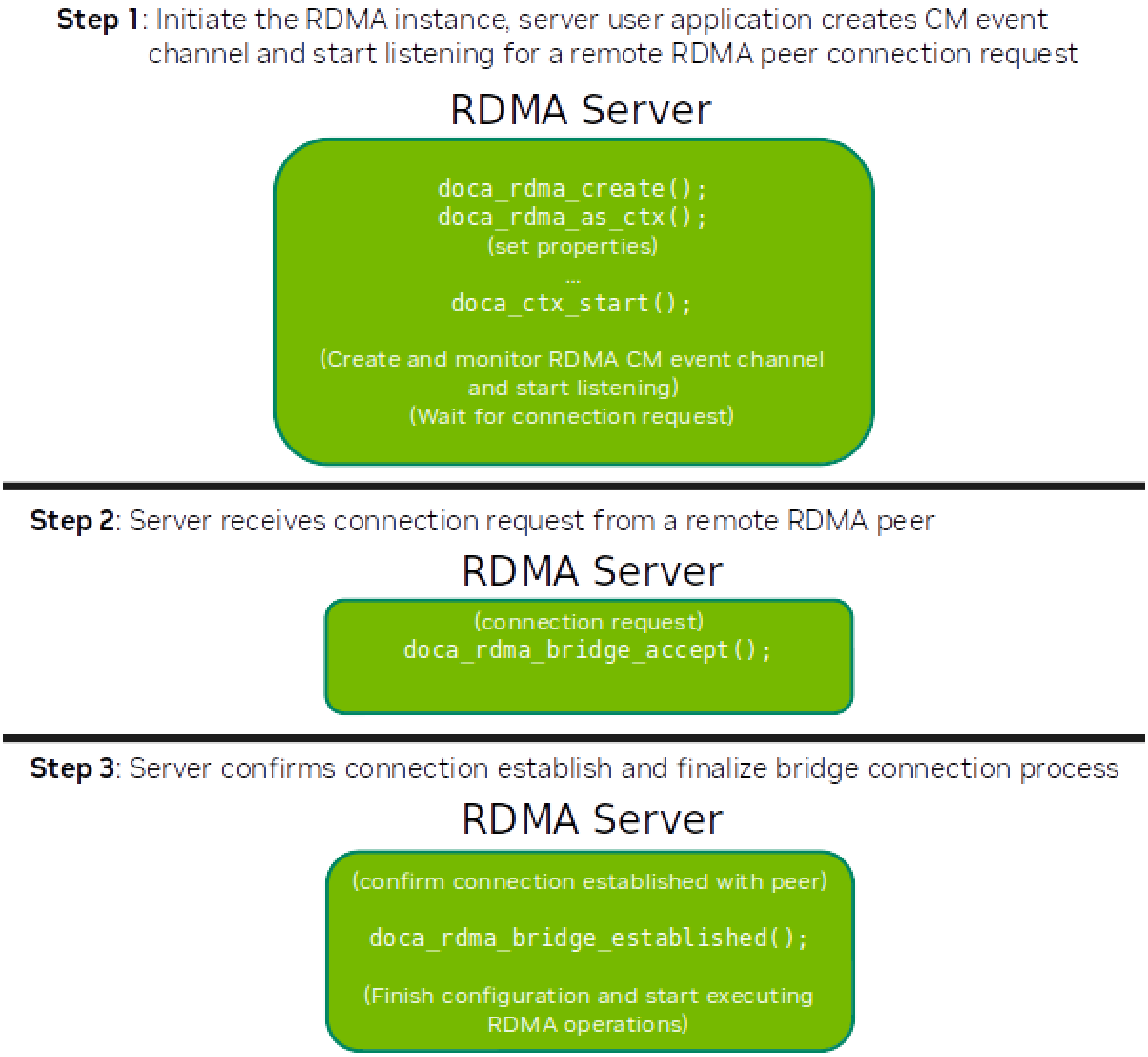
14.4.7.5.4.3 Using Bridge Functions to Accept CM Connection

DOCA RDMA offers connection functionality for user RDMA CM applications acting as a server that maintains a CM event channel and performs the listen process itself (i.e., not using DOCA RDMA connection flow functions).

The functionality must be executed as follows:

1. Server user application, using RDMA CM, must create an RDMA CM event channel, start active listening for a connection from a remote RDMA peer, and monitor the created CM event channel. These functions are performed without the use of the DOCA RDMA connection flow functions explained in section "Connecting Using RDMA CM Connection Flow".
2. Once the server user application received a connection request from a remote RDMA peer acting as client (using RDMA CM), it can call `doca_rdma_bridge_accept()`. This method acts as a bridge to accept a connection request from an application that performs the listen process by itself. The previously explained `doca_rdma_connection_accept()` cannot be used for this connection step as the user application needs to provide the RDMA CM id to accept the connection.

3. After the server side calls `doca_rdma_bridge_accept()` and confirms the client connection is successfully established, it should call `doca_rdma_bridge_established()` to finish the connection process from the server side. Only after a connection is established can DOCA RDMA tasks be allocated and submitted.



Step 1: Initiate the RDMA instance, server user application creates CM event channel and start listening for a remote RDMA peer connection request

```
doca_rdma_create();
doca_rdma_as_ctx();
```

(set properties)

...

```
doca_ctx_start();
```

(Create and monitor RDMA CM event channel and start listening)
(Wait for connection request)

Step 2: Server receives connection request from a remote RDMA peer

(connection request)

```
doca_rdma_bridge_accept();
```

Step 3: Server confirms connection establish and finalize bridge connection process

(confirm connection established with peer)

```
doca_rdma_bridge_established();
```

(Finish configuration and start executing RDMA operations)

14.4.7.6 Execution Phase

This section describes execution on CPU using DOCA Core PE. For additional execution environments refer to section "Alternative Datapath Options".

14.4.7.6.1 Tasks

DOCA RDMA exposes asynchronous tasks that leverage the DPU hardware according to the DOCA Core architecture. See DOCA Core Task.

⚠ Most DOCA RDMA operations are not atomic and therefore it is imperative that the application handle synchronization appropriately. Moreover, successful completion of a write task, with or without immediate, does not guarantee data has been fully written to the remote address.

⚠ All buffers used in DOCA RDMA tasks must remain valid until the task result is retrieved.

14.4.7.6.1.1 Receive Task

This task should be submitted prior to an expected submission of a send/send with immediate/write with immediate task on the remote side.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_receive_set_conf`	`doca_rdma_cap_task_receive_is_supported`
Number of tasks	`doca_rdma_task_receive_set_conf`	-
Destination buffer list length	`doca_rdma_task_receive_set_dst_buf_list_len`	`doca_rdma_cap_task_receive_get_max_dst_buf_list_len`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Destination buffer	Buffer pointing to a local memory address. The data is written to the buffer upon successful completion of the task.	• Linked list buffers are supported • The given destination buffer/list of buffers (given in `dst_buf`) must have a total length sufficient for the expected message size or the task would fail • The destination buffer is not mandatory and may be NULL when the requested DOCA RDMA task on the remote side is "write with immediate" or when the remote side is sending an empty message, with or without immediate (these tasks are presented later on in the "Tasks" section) • For the DOCA RDMA receive task, the length of each buffer is considered as the length from the end of the data section until the end of the buffer, as this is the available memory that can be written to in each buffer. The data length is increased in each buffer if data is written to it once the task is successfully completed.

Task Output

Common output as described in DOCA Core Task.

Name	Description	Notes
Result length	The length of data received by the task	Valid only on successful completion of the task
Result opcode	The opcode of the operation executed by the peer and received by the task	Valid only after task completion, irrespective of success
Result immediate data	The immediate data received by the task	- Valid only on successful completion of the task - Valid only when an immediate value was received (i.e. when the result opcode is `DOCA_RDMA_OPCODE_RECV_SEND_WITH_IMM` or `DOCA_RDMA_OPCODE_RECV_WRITE_WITH_IMM`) - may be retrieved using `doca_rdma_task_receive_get_result_opcode()`)

Task Completion Success

After the task completes successfully, the following happens:

- The received data is copied to the tail segment extending the original data segment
- The data length is increased by the received data length

Task Completion Failure

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user
- If a non-fatal error occurs, the task status is updated. Some buffers may be updated and some may remain unchanged.

Task Limitations

- The operation is not atomic and therefore it is imperative that the application handle synchronization appropriately
- The destination buffer must remain valid until task is completed
- The total length of the message must not exceed the `max_message_size` device capability
- The buffer list length must not exceed the `dst_buf_list_len` property of the DOCA RDMA receive task
- Other limitations are described in DOCA Core Task

14.4.7.6.1.2 Send Task

This task should be submitted to transfer a message to the remote side, and while the remote side is expecting a message and had submitted a receive task beforehand.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_send_set_conf`	`doca_rdma_cap_task_send_is_supported`
Number of tasks	`doca_rdma_task_send_set_conf`	-

Description	API to Set the Configuration	API to Query Support
Source buffer list length	`doca_rdma_set_max_send_buf_list_len` [2]	`doca_rdma_cap_get_max_send_buf_list_len`

[2] This configuration affects other tasks as well. ↩

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer pointing to a local memory address and holds the data to be sent to the remote peer	• Linked list buffers are supported • The total length of the given source buffer/list of buffers (in `src_buf`) may not exceed the expected message size on the remote side or the task fails • The source buffer is not mandatory and may be NULL when wishing to send an empty message • For the DOCA RDMA send task, the length of each buffer is considered as its data length

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- On successful completion of the task, the data in the source buffer will be sent to the remote side.
- It doesn't indicate that the data is received by the remote side.

Task Completion Failure

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user
- If a non-fatal error occurs, the task status is updated

Task Limitations

- The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.
- The source buffer must remain valid until the task completes
- The total length of the message must not exceed the `max_message_size` device capability
- The buffer list length must not exceed the `max_send_buf_list_len` property of the DOCA RDMA instance
- Other limitations are described in DOCA Core Task

14.4.7.6.1.3 Send With Immediate Task

This task should be submitted to transfer a message to the remote side with immediate data (a 32-bit value sent to the remote side, out-of-band), and while the remote side is expecting a message and had submitted a receive task beforehand.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_send_imm_set_conf`	`doca_rdma_cap_task_send_imm_is_supported`
Number of tasks	`doca_rdma_task_send_imm_set_conf`	-
Source buffer list length	`doca_rdma_set_max_send_buf_list_len` [3]	`doca_rdma_cap_get_max_send_buf_list_len`

[3] This configuration affects other tasks as well. ↩

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer pointing to a local memory address and holding the data to be sent to the remote peer	• Linked list buffers are supported. • The total length of the given source buffer/list of buffers (in `src_buf`) may not exceed the expected message size on the remote side or the task fails. • The source buffer is not mandatory and may be NULL when wishing to send an empty message (may be relevant when wishing to keep a connection alive) • For the DOCA RDMA send task, the length of each buffer is considered as its data length
Immediate data	32-bit value sent to the remote side, out-of-band	• The `immediate_data` field should be in Big-Endian format. This value is received by the remote side only once a receive task is completed successfully.

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The data in the source buffer is sent to the remote side
- It does not indicate that the data is received by the remote side

Task Completion Failure

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user
- If a non-fatal error occurs, the task status is updated

Task Limitations

- The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.
- The source buffer must remain valid until the task completes
- The total length of the message must not exceed the `max_message_size` device capability
- The buffer list length must not exceed the `max_send_buf_list_len` property of the DOCA RDMA instance
- Other limitations are described in DOCA Core Task

14.4.7.6.1.4 Read Task

This task should be submitted when wishing to read data from remote memory (i.e., the memory on the remote side of the connection).

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_read_set_conf`	`doca_rdma_cap_task_read_is_supported`
Number of tasks	`doca_rdma_task_read_set_conf`	-
Destination buffer list length	`doca_rdma_set_max_send_buf_list_len` [4]	`doca_rdma_cap_get_max_send_buf_list_len`

4. This configuration affects other tasks as well. ↩

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Points to a remote memory address and holds the data to be read	• Linked list buffers are not supported • The source buffer (`src_buf`) is not mandatory and may be NULL when wishing to read zero bytes (may be relevant when wishing to keep a connection alive) • The data is read only from the data section of the source buffer • The length of the source buffer is considered its data length. The length of data read from the source buffer depends on its data length yet can not exceed the total length of the given destination buffer/list of buffers. That is, the actual length read depends on the minimal length between the source and destination.
Destination buffer	Points to a local memory address. The data is written to the buffer upon successful completion of the task	• Linked list buffers are supported • The length of each destination buffer is considered as the length from the end of the data section until the end of the buffer, as this is the available memory that can be written to in each buffer • May be NULL if the source buffer has been set to NULL

Task Output

Common output as described in DOCA Core Task.

Name	Description	Notes
Result length	The length of data read by the task	Valid only on successful completion of the task

Task Completion Success

After the task completes successfully, the following happens:

- The read data is appended after the data section in the destination buffer, as it was prior to the task submission
- The data length is increased by the read data length

Task Completion Failure

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user
- If a non-fatal error occurs, the task status is updated. Some destination buffers may be updated and some may remain unchanged.

Task Limitations

- The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.
- The task buffers must remain valid until task is completed
- The given source buffer length must not exceed the `max_message_size` device capability

- The destination buffer list length must not exceed the `max_send_buf_list_len` property of the DOCA RDMA instance
- Other limitations are described in DOCA Core Task

14.4.7.6.1.5 Write Task

This task should be submitted when wishing to write data to remote memory (i.e., the memory on the remote side of the connection).

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_write_set_conf`	`doca_rdma_cap_task_write_is_supported`
Number of tasks	`doca_rdma_task_write_set_conf`	-
Source buffer list length	`doca_rdma_set_max_send_buf_list_len` [5]	`doca_rdma_cap_get_max_send_buf_list_len`

5. This configuration affects other tasks as well. ↩

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer pointing to a local memory address and holding the data to be written to the remote peer.	• Linked list buffers are supported • The source buffer should point to a local memory address from which the data should be read. The data is read only from the data section of the source buffer. • The source buffer (`src_buf`) is not mandatory and may be NULL when wishing to write zero bytes (may be relevant when wishing to keep a connection alive) • The length of the buffer is considered as its data length

Name	Description	Notes
Destination buffer	Points to a remote memory address. The data is written to the buffer upon successful completion of the task.	• Linked list buffers are not supported • The destination buffer (dst_buf) should point to a remote memory address • The length of the buffer is considered as its data length • The length of the destination buffer is considered as the length from the end of the data section until the end of the buffer, as this is the available memory that can be written to • The length of data written to the destination buffer depends on the total length of the given source buffer/list of buffers • May be NULL if the source buffer was set to NULL

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The written data is appended after the data section in the destination buffer, as it was prior to the task submission.
- The data length is increased by the written data length

Task Completion Failure

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user
- If a non-fatal error occurs, the task status is updated. Some destination buffers may be updated and some may remain unchanged.

Task Limitations

- The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.
- The task buffers must remain valid until task is completed
- The total length of the given source buffer/list of buffers must be not exceed the `max_message_size` device capability
- The source buffer list length must not exceed the `max_send_buf_list_len` property of the DOCA RDMA instance
- Other limitations are described in DOCA Core Task

14.4.7.6.1.6 Write With Immediate Task

This task should be submitted when wishing to write data to remote memory (i.e., the memory on the remote side of the connection).

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_write_imm_set_conf`	`doca_rdma_cap_task_write_imm_is_supported`
Number of tasks	`doca_rdma_task_write_imm_set_conf`	-
Source buffer list length	`doca_rdma_set_max_send_buf_list_len` [6]	`doca_rdma_cap_get_max_send_buf_list_len`

[6] This configuration affects other tasks as well. ↩

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer pointing to a local memory address and holding the data to be written to the remote peer	• Linked list buffers are supported • The source buffer should point to a local memory address from which the data should be read. The data is read only from the data section of the source buffer. • The source buffer (`src_buf`) is not mandatory and may be NULL when wishing to write zero bytes • The length of the buffer is considered as its data length
Destination buffer	Points to a remote memory address. The data is written to the buffer upon successful completion of the task.	• Linked list buffers are not supported • The destination buffer (`dst_buf`) should point to a remote memory address • The length of the buffer is considered as its data length • The length of the destination buffer is considered as the length from the end of the data section until the end of the buffer, as this is the available memory that can be written to • The length of data written to the destination buffer depends on the total length of the given source buffer/list of buffers • May be NULL if the source buffer was set to NULL
Immediate data	32-bit value sent to the remote side, out-of-band	• Should be in a Big-Endian format • Value is received by the remote side only once a receive task completes successfully

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

A write with immediate task succeeds only if the remote side is expecting the immediate and had submitted a receive task beforehand.

After the task completes successfully, the following happens:

- The written data is appended after the data section in the destination buffer, as it was prior to the task submission
- The data length is increased by the written data length.

Task Completion Failure

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user
- If a non-fatal error occurs, the task status is updated. Some destination buffers may be updated and some may remain unchanged.

Task Limitations

- The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.
- The tasks buffers must remain valid until task is completed
- The total length of the given source buffer/list of buffers must be not exceed the `max_message_size` device capability
- The source buffer list length must not exceed the `max_send_buf_list_len` property of the DOCA RDMA instance
- Other limitations are described in DOCA Core Task

14.4.7.6.1.7 Atomic Compare and Swap Task

This task should be submitted when wishing to execute an 8-byte atomic read-modify-write operation on the remote memory (i.e., the memory on the remote side of the connection), in which the remote value is retrieved and updated if it is equal to a given value.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_atomic_cmp_swp_set_conf`	`doca_rdma_cap_task_atomic_cmp_swp_is_supported`
Number of tasks	`doca_rdma_task_atomic_cmp_swp_set_conf`	-

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Destination buffer	Buffer pointing to a remote memory address	- Linked list buffers are not supported - The destination buffer's data section must begin in a memory address aligned to 8-bytes - Only the first 8-bytes following the data address are considered for atomic operations
Compare data	64-bit value to be compared with the value in the destination buffer	
Swap data	64-bit value to be swapped with the value in the destination buffer	- The value in the destination buffer is only swapped if the compared data value is equal to the value in the destination buffer. Otherwise, the destination buffer remains unchanged.
Result buffer	Buffer pointing to a local memory address. The original value of the destination buffer (before executing the atomic operation) is written to the buffer upon success.	- Linked list buffers are not supported - The result is written to the first 8-bytes following the data address

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- If the compared values are equal, the value in the destination is swapped with the 64-bit value in the task's swap data field (`swap_data`)
- If the compared values are not equal, the value in the destination value remains unchanged
- The original value of the destination buffer (before executing the atomic operation) is written to the result buffer

Task Completion Failure

If the task fails midway:

- The context is stopped, and the task should be freed by the user

Task Limitations

- Task buffers must remain valid until task is completed
- Other limitations are described in DOCA Core Task

14.4.7.6.1.8 Atomic Fetch and Add Task

This task should be submitted when wishing to execute an 8-byte atomic read-modify-write operation on the remote memory (i.e., the memory on the remote side of the connection), in which the remote value is retrieved and increased by a given value.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_atomic_fetch_add_set_conf`	`doca_rdma_cap_task_atomic_fetch_add_is_supported`
Number of tasks	`doca_rdma_task_atomic_fetch_add_set_conf`	-

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Destination buffer	Buffer that points to a remote memory address	• Linked list buffers are not supported • The destination buffer's data section must begin in a memory address aligned to 8-bytes • Only the first 8-bytes following the data address are considered for atomic operations
Add data	64-bit value to be added to the value in the destination buffer	
Result buffer	Buffer pointing to a local memory address. The original value of the destination buffer (before executing the atomic operation) is written to the buffer upon success.	• Linked list buffers are not supported • The result is written to the first 8-bytes following the data address

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The value in the destination is increased by the 64-bit value in the task's add data field
- The original value of the destination buffer (before executing the atomic operation) is written to the result buffer

Task Completion Failure

If the task fails midway:

- The context is stopped, and the task should be freed by the user

Task Limitations

- Task buffers must remain valid until task is completed
- Other limitations are described in DOCA Core Task

14.4.7.6.1.9 Get Remote Sync Event Task

This task should be submitted when wishing to get the value of a remote sync event.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_remote_net_sync_event_get_set_conf`	`doca_rdma_cap_task_remote_net_sync_event_get_is_supported`
Number of tasks	`doca_rdma_task_remote_net_sync_event_get_set_conf`	-
Destination buffer list length	`doca_rdma_set_max_send_buf_list_len` [7]	`doca_rdma_cap_get_max_send_buf_list_len`

[7] This configuration affects other tasks as well. ↩

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Sync Event	The remote DOCA Sync Event to get its value	
Destination buffer	Points to a local memory address. The Sync Event value is written to the buffer upon successful completion of the task.	• Linked list buffers are supported • The length of the each buffer is considered as the length from the end of the data section until the end of the buffer, as this is the available memory that can be written to in each buffer

Task Output

Common output as described in DOCA Core Task.

Name	Description	Notes
Result length	The length of data received by the task	Valid only on successful completion of the task

Task Completion Success

After the task completes successfully, the following happens:

- The remote Sync Event value is appended after the data section in the destination buffer, as it was prior to the task submission
- The data length is increased by the retrieved data length

Task Completion Failure

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user
- If a non-fatal error occurs, the task status is updated. Some destination buffers may be updated and some may remain unchanged.

Task Limitations

- The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.
- The destination buffer must remain valid until the task is completed
- The destination buffer list length must not exceed the `max_send_buf_list_len` property of the DOCA RDMA instance
- Other limitations are described in DOCA Core Task

14.4.7.6.1.10 Set Remote Sync Event Task

This task should be submitted when wishing to set a remote sync event to a given value.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_remote_net_sync_event_notify_set_set_conf`	`doca_rdma_cap_task_remote_net_sync_event_notify_set_is_supported`
Number of tasks	`doca_rdma_task_remote_net_sync_event_notify_set_set_conf`	-
Source buffer list length	`doca_rdma_set_max_send_buf_list_len` [8]	`doca_rdma_cap_get_max_send_buf_list_len`

[8] This configuration affects other tasks as well. ↩

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Points to a local memory address from which the Sync Event should be retrieved	• Linked list buffers are supported • The data is retrieved only from the buffer data section, until 8-bytes • The length of the source buffer is considered its data length. The length of data retrieved from the source buffer will not exceed the Sync Event value length (8-bytes). Thus, the actual length retrieved depends on the minimal length between the source buffer and Sync Event value length.
Sync Event	The remote DOCA Sync Event to get its value	

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The remote sync event value is set to the data in the source buffer

Task Completion Failure

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user
- If a non-fatal error occurs, the task status is updated, and the Sync Event value is undefined

Task Limitations

- The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.
- The source buffer must remain valid until the task completes
- The source buffer list length must not exceed the `max_send_buf_list_len` property of the DOCA RDMA instance
- Other limitations are described in DOCA Core Task

14.4.7.6.1.11 Add Remote Sync Event Task

This task should be submitted when wishing to atomically increase a remote sync event by a given value.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_remote_net_sync_event_notify_add_set_conf`	`doca_rdma_cap_task_remote_net_sync_event_notify_add_is_supported`
Number of tasks	`doca_rdma_task_remote_net_sync_event_notify_add_set_conf`	-

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Sync event	A remote Sync Event	
Add data	64-bit value that is added to the Sync Event value	
Result buffer	Buffer pointing to a local memory address. The original Sync Event value of the destination buffer (before executing the atomic operation) is written to the buffer upon success.	• Linked list buffers are not supported • The result is written to the first 8-bytes following the data address

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The value of the remote sync event is increased by the 64-bit value in the task's add data field
- The original value of the remote sync event (before executing the operation) is written to the result buffer

Task Completion Failure

If the task fails midway:

- The context is stopped, and the task should be freed by the user

Task Limitations

- Result buffer must remain valid until task is completed
- Other limitations are described in DOCA Core Task

14.4.7.6.2 Events

DOCA RDMA exposes asynchronous events to notify about changes that happen unexpectedly, according to DOCA Core architecture.

The only event DOCA RDMA exposes is common events as described in DOCA Core Event.

14.4.7.7 State Machine

The DOCA RDMA library follows the Context state machine as described in DOCA Core Context State Machine.

The following section describes how to move states and what is allowed in each state.

14.4.7.7.1 Idle

In this state, it is expected that application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to section "Configurations"
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
N/A	Create the context
Running	Call stop after making sure all tasks have been freed
Stopping	Call progress until all tasks are completed and freed

14.4.7.7.2 Starting

This state cannot be reached.

14.4.7.7.3 Running

In this state, it is expected that application:

1. Connects the RDMA instances on both peers. Refer to section "Establishing RDMA Connections" for more information.
2. Performs an RDMA instance disconnection process if the connection was established using the RDMA CM flow. Refer to section "Connecting Using RDMA CM Connection Flow" for more information.
3. Performs a new connection of the RDMA instances on both peers after an RDMA instance disconnection process if the connection was established using the RDMA CM flow. Refer to section "Connecting Using RDMA CM Connection Flow" for more information.
4. Accepts and indicates an established RDMA connection if the listening and CM channel monitoring was done by the user application. Refer to section "Connecting Using RDMA CM Connection Flow" for more information.
5. Allocates and submits tasks.
6. Calls progress to complete tasks and/or receive events.

Allowed operations:

- Performing a connection between 2 peers
- Allocating previously configured task
- Submitting an allocated task
- Calling stop

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

14.4.7.7.4 Stopping

In this state, it is expected that application:

1. Calls progress to complete all inflight tasks (tasks complete with failure)
2. Frees any completed tasks
3. Performs an RDMA instance disconnection process if the connection was established using the RDMA CM flow. Refer to section "Connecting Using RDMA CM Connection Flow" for more information.

Allowed operations:

- Call progress

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and fatal error occurs
Running	Call stop without freeing all tasks

14.4.7.8 Alternative Datapath Options

DOCA RDMA allows data path to be run on DPA or GPU.

14.4.7.8.1 DPA Datapath

DOCA offers the DOCA DPA library which provides a programming model for offloading communication-centric user code to run on the DPA processor on the BlueField DPU. For additional information on the DOCA DPA library.

⚠ DOCA RDMA on DPA datapath supports local networks only (i.e., cross-network or routing is not supported).

The user can choose to run an RDMA operation on the DPA datapath by configuring the DOCA RDMA context used by the application in the following manner:

1. Obtain DOCA CTX by calling `doca_rdma_as_ctx()` .
2. Set the datapath of the context to DPA by calling `doca_ctx_set_datapath_on_dpa()` . For additional information, refer to DOCA Core Alternative Data Path.
3. Finish context configuration and start the context by calling `doca_ctx_start()` . For additional information, refer to DOCA Context.

After configuring the datapath, the user can obtain a DPA handle for the DOCA RDMA context by calling `doca_rdma_get_dpa_handle()` .

The DPA handle can be used by the DOCA DPA library for datapath operations. For additional information, refer to DOCA DPA Communication Model.

14.4.7.8.2 GPU Datapath

DOCA offers the DOCA GPUNetIO library which provides a programming model for offloading the orchestration of the communication to a GPU CUDA kernel. For additional information on the DOCA GPUNetIO library.

The user can choose to run an RDMA operation on the GPU datapath by configuring the DOCA RDMA context used by the application in the following manner:

1. Obtain DOCA CTX by calling `doca_rdma_as_ctx()` .
2. Set the datapath of the context to GPU by calling `doca_ctx_set_datapath_on_gpu()` . For additional information, refer to DOCA Core Alternative Data Path.
3. Finish context configuration and start the context by calling `doca_ctx_start()` . For additional information, refer to DOCA Core Context.

After configuring the datapath, the user can obtain a GPU handle for the DOCA RDMA context by calling `doca_rdma_get_gpu_handle()` .

The GPU handle must be passed to a GPU CUDA kernel so the DOCA GPUNetIO CUDA device functions can execute datapath operations. For additional information, refer to DOCA GPUNetIO device functions.

14.4.7.9 DOCA RDMA Samples

These samples illustrate how to use the DOCA RDMA API to execute DOCA RDMA operations.

> ⓘ All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

14.4.7.9.1 Running the Samples

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample:

   ```
   cd /opt/mellanox/doca/samples/doca_rdma/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build
   ```

   > ⓘ The binary `doca_<sample_name>` is created under `/tmp/build/`.

3. Sample usage:
   - Common arguments

Argument	Description
`-d`, `--device`	IB device name (optional). If not provided, a random IB device is assigned.
`-ld`, `--local-descriptor-path`	Local descriptor file path that includes the local connection information to be copied to the remote program
`-re`, `--remote-descriptor-path`	Remote descriptor file path that includes the remote connection information to be copied from the remote program
`-m`, `--mmap-descriptor-path`	Remote descriptor file path that includes the remote mmap connection information to be copied from the remote program
`-g`, `--gid-index`	GID index for DOCA RDMA (optional)

   - Sample-specific arguments

Sample	Argument	Description
RDMA Read Responder	`-r` , `--read-string`	String to read (optional). If not provided, "Hi DOCA RDMA!" is defined.
RDMA Send RDMA Send Immediate	`-s` , `--send-string`	
RDMA Write Requester RDMA Write Immediate Requester	`-w` , `--write-string`	

4. For additional information per sample, use the `-h` option:

```
/tmp/build/<sample_name> -h
```

14.4.7.9.2 Samples

Each sample presents a connection between two peers, transferring data from one to another, using a different RDMA operation in each sample. For more information on the available RDMA operations, refer to section "Tasks".

Each sample is comprised of two executables, each running on a peer.

The samples can run on either DPU or host, as long as the chosen peers have a connection between them.

> Prior to running the samples, ensure that the chosen devices, selected by the device name and the GID index, are set correctly and have a connection between one another. In each sample, it is the user's responsibility to copy the descriptors between the peers.

Most of the samples follow the following main basic steps:

1. Allocating resources:
    a. Locating and opening a device. The chosen device is one that supports the tasks relevant for the sample. If the sample requires no task, any device may be chosen.
    b. Creating a local MMAP and configuring it (including setting the MMAP memory range and relevant permissions)
    c. Creating a DOCA PE
    d. Creating an RDMA instance and configuring it (including setting the relevant permissions)
    e. Connecting the RDMA context to the PE
2. Sample-specific configurations:
    a. Configuring the tasks relevant to the sample, if any. Including:
        i. Setting the number of tasks for each task type.
        ii. Setting callback functions for each task type, with the following logic:
            1. Successful completion callback:
                a. Verifying the data received from the remote, if any, is valid.
                b. Printing the transferred data.
                c. Freeing the task and task-specific resources (such as source/destination buffers).

d. If an error occurs in steps a. and b., update the error that was encountered.

> ⚠ If the context is not in idle sate, only the first error in the flow is saved.

e. Decreasing the number of remaining tasks and stopping the context once it reaches 0.

2. Failed completion callback:
   a. Update the error that was encountered.

   > ⚠ If the context is not in idle sate, only the first error in the flow is saved.

   b. Freeing the task and task-specific resources (such as source/destination buffers).
   c. Decreasing the number of remaining tasks and stopping the context once it reaches 0.

b. Setting a state change callback function, with the following logic:

- Once the context moves to Starting state (can only be reached from Idle), export and connect the RDMA and, in some samples, export the local mmap or the sync event.

  > ⚠ During this step, the user is responsible for copying the descriptors between the two peers.

  > ⚠ The descriptors are to be read and used only by the peer, using the relevant DOCA functions (the descriptors contain encoded data).

- Once the context moves to Running state (can only be reached from Starting state in RDMA samples):
  - In some samples, only print a log and wait for the peer, or synchronize events
  - In other samples, prepare and submit a task:
    a. If needed, create an mmap from the received exported mmap descriptor, passed from the peer.
    b. Request the required buffers from the buffer inventory.
    c. Allocate and initiate the required task, together with setting the number of remaining tasks parameter as the task's user data.
    d. Submit the task.
- Once the context moves to Stopping state, print a relevant log.
- Once the context moves to Idle state:
  1. Print a relevant log.
  2. Send update that the main loop may be stopped.

3. Setting the program's resources as the context user data to be used in callbacks.
4. Creating a buffer inventory and starting it.

5. Starting the context.

   > ⓘ After starting the context, the state change callback function is called by the PE which executes the relevant steps.

   > ⓘ In a successful run, each section is executed in the order they are presented in section 2.b.

6. Progressing the PE until the context returns to Idle state and the main loop may be stooped, either because of a run in which all tasks have been completed, or due to a fatal error.
7. Cleaning up the resources.

14.4.7.9.2.1 RDMA Read

RDMA Read Requester

This sample illustrates how to read from a remote peer (the responder) using DOCA RDMA.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample are set to local read and write.
2. A read task is configured for this sample.
3. In this sample, data is read from the peer, verified to be valid, and printed in the successful task completion callback.
4. The local mmap is not exported as the peer does not intend to access it.
5. To read from the peer, a remote mmap is created from the peer's exported mmap.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_read_requester/rdma_read_requester_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_read_requester/rdma_read_requester_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_read_requester/meson.build`

RDMA Read Responder

This sample illustrates how to set up a remote peer for a DOCA RDMA read request.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for both the local mmap and the RDMA instance in this sample allow for RDMA read.
2. No tasks are configured for this sample, and thus no tasks are prepared and submitted, nor are there task completion callbacks.
3. The local mmap is exported to the remote memory to allow it to be used by the peer for RDMA read.
4. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_read_responder/rdma_read_responder_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_read_responder/rdma_read_responder_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_read_responder/meson.build`

14.4.7.9.2.2 RDMA Write

RDMA Write Requester

This sample illustrates how to write to a remote peer (the responder) using DOCA RDMA.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample are set to local read and write.
2. A write task is configured for this sample.
3. In this sample, data is written to the peer and printed in the successful task completion callback.
4. The local mmap is not exported as the peer does not intend to access it.
5. To write to the peer, a remote mmap is created from the peer's exported mmap.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_requester/rdma_write_requester_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_requester/rdma_write_requester_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_requester/meson.build`

RDMA Write Responder

This sample illustrates how to set up a remote peer for a DOCA RDMA write request.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for both the local mmap and the RDMA instance in this sample allow for RDMA write.
2. No tasks are configured for this sample, and thus no tasks are prepared and submitted, nor are there task completion callbacks. In this sample, the data written to the memory of the responder is printed once the context state is changed to Running, using the state change callback. This is done only after receiving input from the user, indicating that the requester had finished writing.
3. The local mmap is exported to the remote memory to allow it to be used by the peer for RDMA write.
4. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_responder/rdma_write_responder_sample.c`

- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_responder/rdma_write_responder_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_responder/meson.build`

14.4.7.9.2.3 RDMA Write Immediate

RDMA Write Immediate Requester

This sample illustrates how to write to a remote peer (the responder) using DOCA RDMA along with a 32-bit immediate value which is sent OOB.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample is set to local read and write.
2. A write with immediate task is configured for this sample.
3. In this sample, data is written to the peer and printed in the successful task completion callback.
4. The local mmap is not exported as the peer does not intend to access it.
5. To write to the peer, a remote mmap is created from the peer's exported mmap.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_requester/rdma_write_immediate_requester_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_requester/rdma_write_immediate_requester_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_requester/meson.build`

RDMA Write Immediate Responder

This sample illustrates how the set up a remote peer for a DOCA RDMA write request whilst receiving a 32-bit immediate value from the peer's OOB.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for both the local mmap and the RDMA instance in this sample allow for RDMA write.
2. A receive task is configured for this sample to retrieve the immediate value. Failing to submit a receive task prior to the write with immediate task results in a fatal failure.
3. In this sample, the successful task completion callback also includes:
    a. Checking the result opcode, to verify that the receive task has completed after receiving a write with immediate request.
    b. Verifying the data written to the memory of the responder is valid and printing it, along with the immediate data received.
4. The local mmap is exported to the remote memory, to allow it to be used by the peer for RDMA write.
5. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_responder/rdma_write_immediate_responder_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_responder/rdma_write_immediate_responder_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_responder/meson.build`

14.4.7.9.2.4 RDMA Send and Receive

RDMA Send

This sample illustrates how to send a message to a remote peer using DOCA RDMA.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample is set to local read and write.
2. A send task is configured for this sample.
3. In this sample, the data sent is printed during the task preparation, not in the successful task completion callback.
4. The local mmap is not exported as the peer does not intend to access it.
5. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_send/rdma_send_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_send/rdma_send_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_send/meson.build`

RDMA Receive

This sample illustrates how the remote peer can receive a message sent by the peer (the sender).

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample is set to local read and write.
2. A receive task is configured for this sample to retrieve the sent data. Failing to submit a receive task prior to the send task results in a fatal failure.
3. In this sample, data is received from the peer verified to be valid and printed in the successful task completion callback.
4. The local mmap is not exported as the peer does not intend to access it.
5. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_receive/rdma_receive_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_receive/rdma_receive_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_receive/meson.build`

14.4.7.9.2.5 RDMA Send and Receive with Immediate

RDMA Send with Immediate

This sample illustrates how to send a message to a remote peer using DOCA RDMA along with a 32-bit immediate value which is sent OOB.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample is set to local read and write.
2. A send with immediate task is configured for this sample.
3. In this sample, the data sent is printed during the task preparation, not in the successful task completion callback.
4. The local mmap is not exported as the peer does not intend to access it.
5. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_send_immediate/rdma_send_immediate_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_send_immediate/rdma_send_immediate_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_send_immediate/meson.build`

RDMA Receive with Immediate

This sample illustrates how the remote peer can receive a message sent by the peer (the sender) while also receiving a 32-bit immediate value from the peer's OOB.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample is set to local read and write.
2. A receive task is configured for this sample to retrieve the sent data and the immediate value. Failing to submit a receive task prior to the send with immediate task results in a fatal failure.
3. In this sample, the successful task completion callback also includes:
    a. Checking the result opcode, to verify that the receive task has completed after receiving a sent message with an immediate.
    b. Verifying the data received from the peer is valid and printing it along with the immediate data received.
4. In this sample, data is received from the peer verified to be valid and printed in the successful task completion callback.
5. The local mmap is not exported as the peer does not intend to access it.
6. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_receive_immediate/rdma_receive_immediate_sample.c`

- `/opt/mellanox/doca/samples/doca_rdma/rdma_receive_immediate/rdma_receive_immediate_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_receive_immediate/meson.build`

14.4.7.9.2.6 RDMA Remote Sync Event

This sample illustrates how to synchronize between local sync event and a remote sync event DOCA RDMA.

RDMA Remote Sync Event Requester

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample is set to local read and write.
2. A "remote net sync event notify set" task is configured for this sample.
    - For this task, the successful task completion callback has the following logic:
        i. Printing an info log saying the task was successfully completed and a specific successful completion log for the task.
        ii. Decreasing the number of remaining tasks. Once 0 is reached:
            1. Freeing the task and task-specific resources.
            2. Stopping the context.
    - For this task, the failed task completion callback stops the context even when the number of remaining tasks is different than 0 (since the synchronization between the peers would fail).
3. A "remote net sync event get" task is configured for this sample.
    - For this task, the successful task completion callback also includes:
        i. Resubmitting the task, until a value greater than or equal to the expected value is retrieved.
        ii. Once such value is retrieved, submitting a "remote net sync event notify set" task to signal sample completion, including:
            1. Updating the successful completion message accordingly.
            2. Increasing the number of submitted tasks.
            3. If an error was encountered, and the "remote net sync event notify set" task was not submitted, the task and task resources are freed.
    - For this task, the failed task completion callback also includes freeing the "remote net sync event notify set" task and task resources.
4. The local mmap is not exported as the peer does not intend to access it.
5. No remote mmap is created as there is no intention to access the remote memory in this sample.
6. To synchronize events with the peer, a sync event remote net is created from the peer's exported sync event.
7. Both tasks are prepared and submitted in the state change callback, once the context moves from starting to running.
8. The user data of the "remote net sync event get" task points to the "remote net sync event notify set" task.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_requester/rdma_sync_event_requester_sample.c`

- `/opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_requester/rdma_sync_event_requester_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_requester/meson.build`

RDMA Remote Sync Event Responder

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample is set to local read and write.
2. This sample includes creating a local sync event and exporting it to the remote memory to allow the peer to create a remote handle.
3. No tasks are configured for this sample, and thus no tasks are prepared and submitted, nor are there task completion callbacks. In this sample, the following steps are executed once the context moves from starting to running, using the state change callback:
   a. Waiting for the sync event to be signaled from the remote side.
   b. Notifying the sync event from the local side.
   c. Waiting for completion notification from the remote side.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_responder/rdma_sync_event_responder_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_responder/rdma_sync_event_responder_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_responder/meson.build`

14.4.8 DOCA Ethernet

This guide provides an overview and configuration instructions for the DOCA ETH API.

14.4.8.1 Introduction

> ⚠ The DOCA Ethernet library is supported at alpha level.

DOCA ETH comprises of two APIs, DOCA ETH RXQ and DOCA ETH TXQ. The control path is always handled on the host/DPU CPU side by the library. The datapath can be managed either on the CPU by the DOCA ETH library or on the GPU by the GPUNetIO library.

DOCA ETH RXQ is an RX queue. It defines a queue for receiving packets. It also supports receiving Ethernet packets on any memory mapped by `doca_mmap`.

The memory location to which packets are scattered is agnostic to the processor which manages the datapath (CPU/DPU/GPU). For example, the datapath may be managed on the CPU while packets are scattered to GPU memory.

DOCA ETH TXQ is an TX queue. It defines a queue for sending packets. It also supports sending Ethernet packets from any memory mapped by `doca_mmap`.

To free the CPU from managing the datapath, the user can choose to manage the datapath from the GPU. In this mode of operation, the library collects user configurations and creates a receive/send queue object on the GPU memory (using the DOCA GPU sub-device) and coordinates with the network card (NIC) to interact with the GPU processor.

14.4.8.2 Prerequisites

This library follows the architecture of a DOCA Core Context. It is recommended to read the following sections:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem
- DOCA Flow Programming Guide
- OpenvSwitch Offload
- BlueField DPU Scalable Function (for using SF on DPU)
- DOCA GPUNetIO (for GPU datapath)

14.4.8.3 Changes From Previous Releases

14.4.8.3.1 Changes in 2.8.0

The following subsection(s) detail the `doca_eth` library updates in version 2.8.0.

14.4.8.3.1.1 Added

- `doca_error_t doca_eth_rxq_set_notification_moderation(struct doca_eth_rxq *eth_rxq, uint16_t period_usec, uint16_t comp_count)`
- `doca_error_t doca_eth_txq_task_send_num_expand(struct doca_eth_txq *eth_txq, uint32_t task_send_num)`
- `doca_error_t doca_eth_txq_task_lso_send_num_expand(struct doca_eth_txq *eth_txq, uint32_t task_lso_send_num)`
- `doca_error_t doca_eth_txq_task_batch_send_num_expand(struct doca_eth_txq *eth_txq, uint16_t task_batches_num)`
- `doca_error_t doca_eth_txq_task_batch_lso_send_num_expand(struct doca_eth_txq *eth_txq, uint16_t task_batches_num)`

14.4.8.3.1.2 Changed

- `doca_eth_rxq_task_recv_allocate_init(struct doca_eth_rxq *eth_rxq, union doca_data user_data, struct doca_buf *pkt, struct doca_eth_rxq_task_recv **task_recv)`
  - `→ doca_eth_rxq_task_recv_allocate_init(struct doca_eth_rxq *eth_rxq, struct doca_buf *pkt, union doca_data user_data, struct doca_eth_rxq_task_recv **task_recv)`

14.4.8.4 Environment

DOCA ETH based applications can run either on the Linux host machine or on the NVIDIA® BlueField® DPU target. The following is required:

- Applications should run with root privileges
- To run DOCA ETH on the DPU, applications must supply the library with SFs as a `doca_dev`. See OpenvSwitch Offload and BlueField DPU Scalable Function to see how to create SFs and connect them to the appropriate ports.
- Applications need to use DOCA Flow to forward incoming traffic to DOCA ETH RXQ's queue. See DOCA Flow and DOCA ETH RXQ samples for reference.

> ⓘ Make sure the system has free huge pages for DOCA Flow.

14.4.8.5 Architecture

DOCA ETH is comprised of two parts: DOCA ETH RXQ and DOCA ETH TXQ.

14.4.8.5.1 DOCA ETH RXQ

14.4.8.5.1.1 Operating Modes

DOCA ETH RXQ can operate in the three modes, each exposing a slightly different control/datapath.

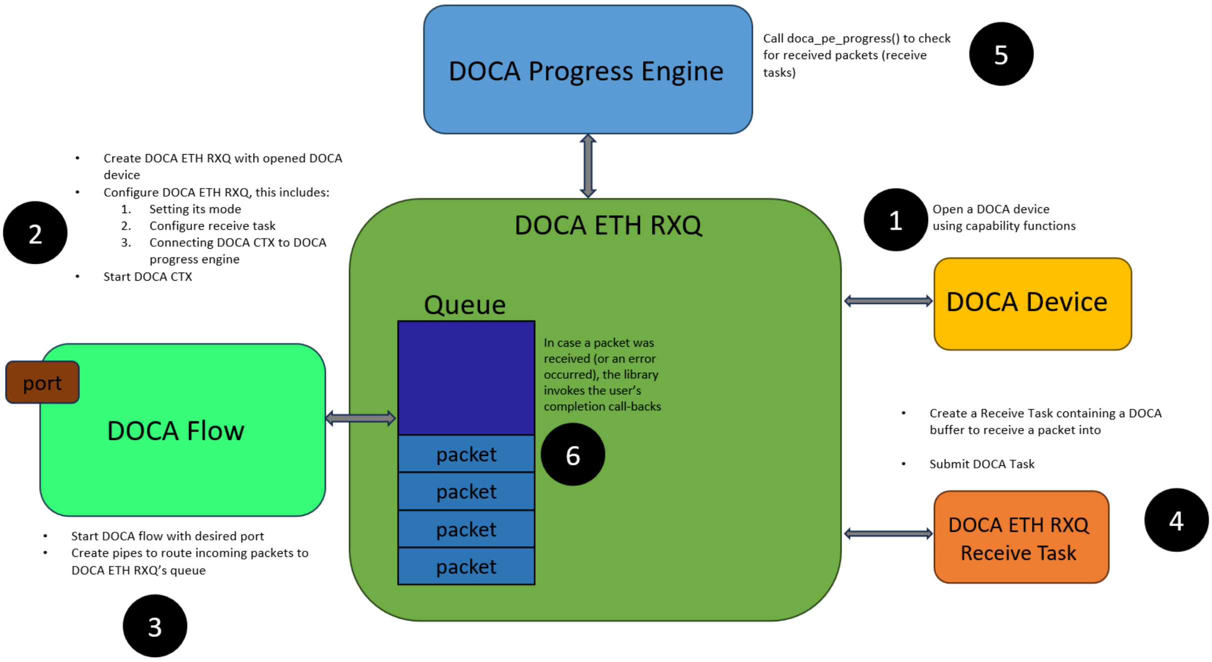
Regular Receive

> ⓘ This mode is supported only for CPU datapath.

In this mode, the received packet buffers are managed by the user. To receive a packet, the user should submit a receive task containing a `doca_buf` to write the packet into.

The application uses this mode if it wants to:

- Run on CPU
- Manage the memory of received packet and the packet's exact place in memory
- Forward the received packets to other DOCA libraries


DOCA Progress Engine
Call doca_pe_progress() to check for received packets (receive tasks)
5
Create DOCA ETH RXQ with opened DOCA device
Configure DOCA ETH RXQ, this includes:
1. Setting its mode
2. Configure receive task
3. Connecting DOCA CTX to DOCA progress engine
Start DOCA CTX
2
DOCA ETH RXQ
1
Open a DOCA device using capability functions
DOCA Device
Queue
In case a packet was received (or an error occurred), the library invokes the user's completion call-backs
port
DOCA Flow
packet
packet
packet
packet
6
Create a Receive Task containing a DOCA buffer to receive a packet into
Submit DOCA Task
DOCA ETH RXQ Receive Task
4
Start DOCA flow with desired port
Create pipes to route incoming packets to DOCA ETH RXQ's queue
3

Cyclic Receive

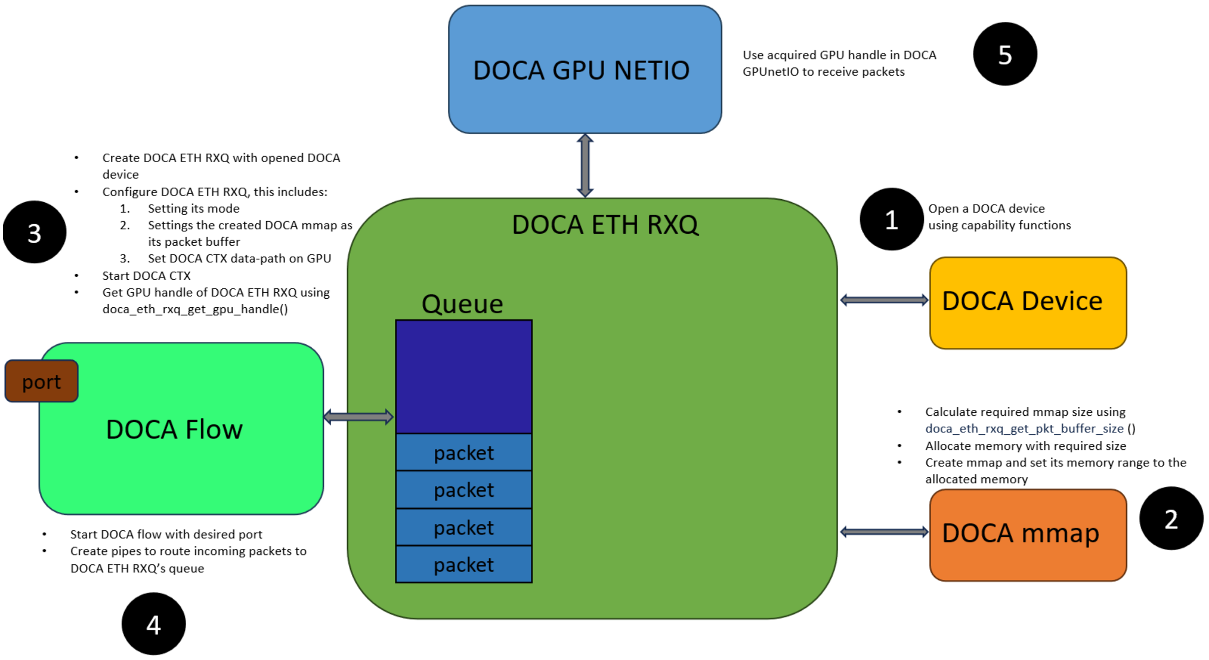
> ⓘ This mode is supported only for GPU datapath.

In this mode, the library scatters packets to the packet buffer (supplied by the user as `doca_mmap`) in a cyclic manner. Packets acquired by the user may be overwritten by the library if not processed fast enough by the application.

In this mode, the user must provide DOCA ETH RXQ with a packet buffer to be managed by the library (see `doca_eth_rxq_set_pkt_buf()`). The buffer should be large enough to avoid packet loss (see `doca_eth_rxq_estimate_packet_buf_size()`).

The application uses this mode if:

- It wants to run on GPU
- It has a deterministic packet processing time, where a packet is guaranteed to be processed before the library overwrites it with a new packet
- It wants best performance


DOCA GPU NETIO
Use acquired GPU handle in DOCA GPUnetIO to receive packets
5
Create DOCA ETH RXQ with opened DOCA device
Configure DOCA ETH RXQ, this includes:
1. Setting its mode
2. Settings the created DOCA mmap as its packet buffer
3. Set DOCA CTX data-path on GPU
Start DOCA CTX
Get GPU handle of DOCA ETH RXQ using doca_eth_rxq_get_gpu_handle()
3
DOCA ETH RXQ
1
Open a DOCA device using capability functions
Queue
DOCA Device
port
DOCA Flow
packet
packet
packet
packet
Calculate required mmap size using doca_eth_rxq_get_pkt_buffer_size ()
Allocate memory with required size
Create mmap and set its memory range to the allocated memory
DOCA mmap
2
Start DOCA flow with desired port
Create pipes to route incoming packets to DOCA ETH RXQ's queue
4

Managed Memory Pool Receive

This mode is supported only for CPU datapath.

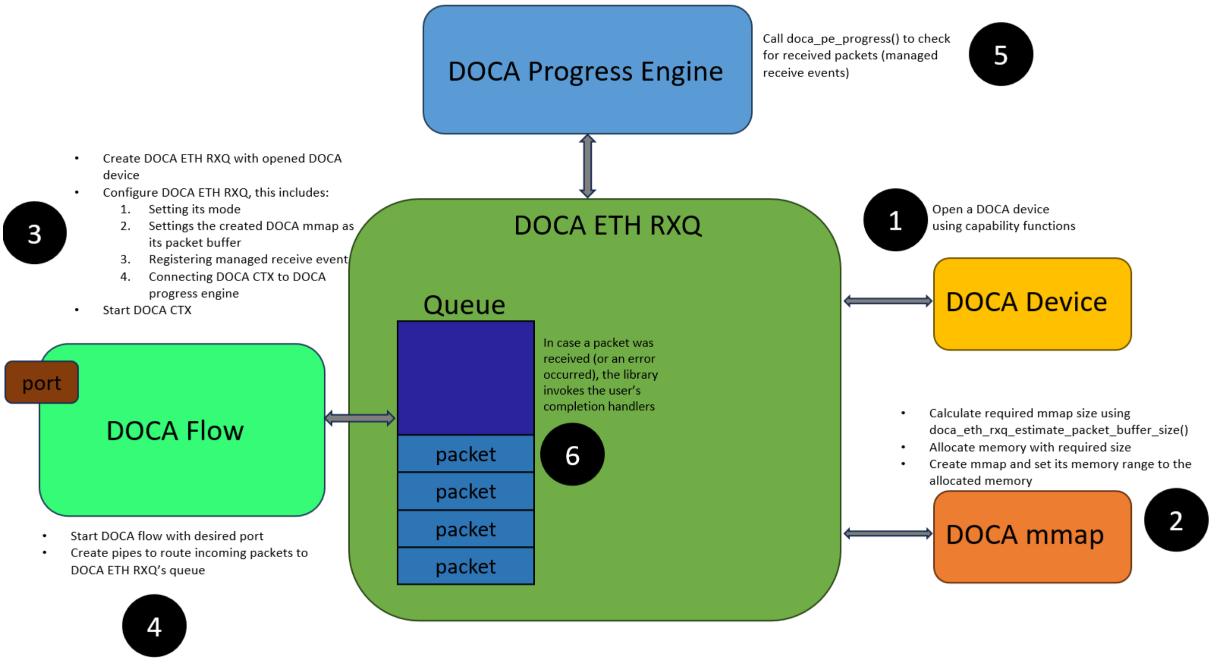
In this mode, the library uses various optimizations to manage the packet buffers. Packets acquired by the user cannot be overwritten by the library unless explicitly freed by the application. Thus, if the application does not release the packet buffers fast enough, the library would run out of memory and packets would start dropping.

Unlike Cyclic Receive mode, the user can pass the packet to other libraries in DOCA with the guarantee that the packet is not overwritten while being processed by those libraries.

In this mode, the user must provide DOCA ETH RXQ with a packet buffer to be managed by the library (see `doca_eth_rxq_set_pkt_buf()`). The buffer should be large enough to avoid packet loss (see `doca_eth_rxq_estimate_packet_buf_size()`).

The application uses this mode if:

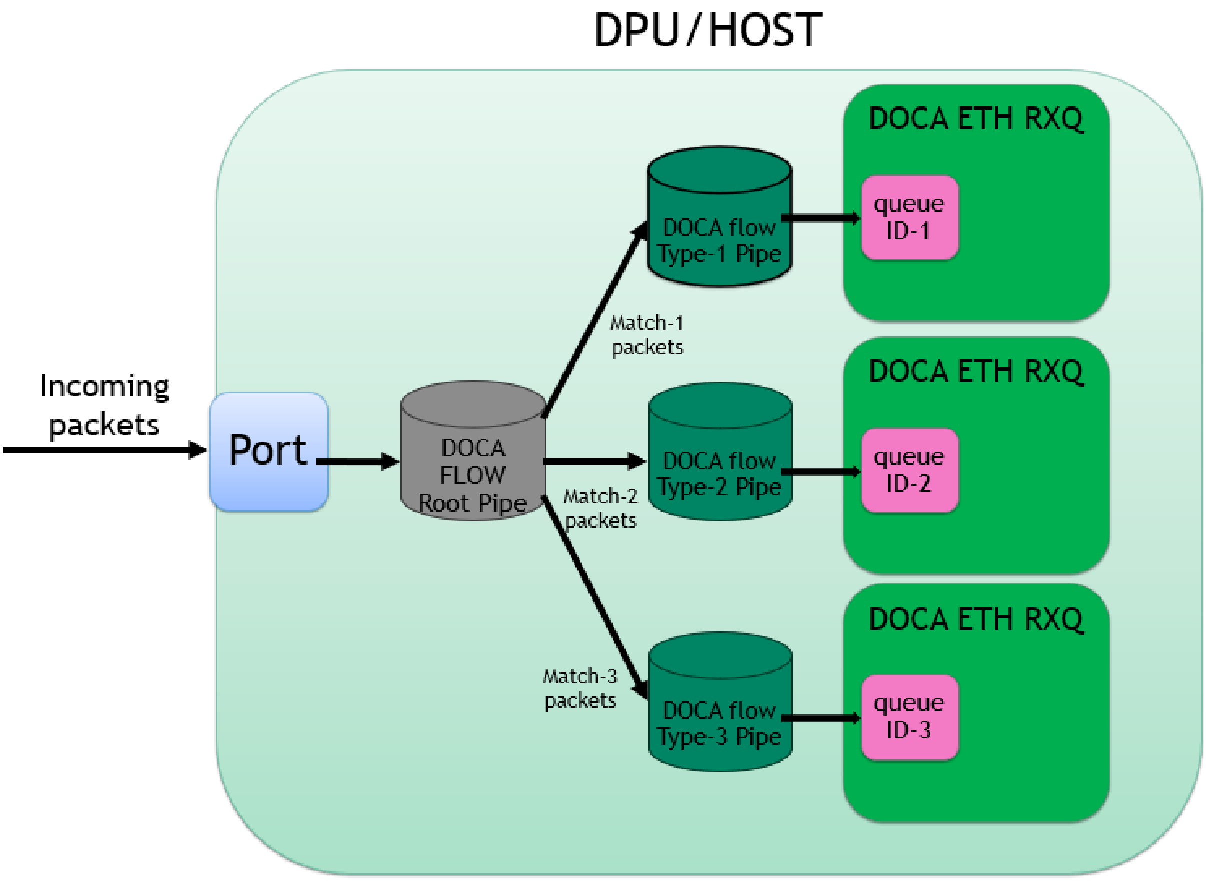
- It wants to run on CPU
- It has a deterministic packet processing time, where a packet is guaranteed to be processed before the library runs out of memory and packets start dropping
- It wants to forward the received packets to other DOCA libraries
- It wants best performance


DOCA Progress Engine
Call doca_pe_progress() to check for received packets (managed receive events)
5
- Create DOCA ETH RXQ with opened DOCA device
- Configure DOCA ETH RXQ, this includes:
  1. Setting its mode
  2. Settings the created DOCA mmap as its packet buffer
  3. Registering managed receive event
  4. Connecting DOCA CTX to DOCA progress engine
- Start DOCA CTX
3
DOCA ETH RXQ
1
Open a DOCA device using capability functions
DOCA Device
Queue
In case a packet was received (or an error occurred), the library invokes the user's completion handlers
port
DOCA Flow
packet
packet
packet
packet
6
- Calculate required mmap size using doca_eth_rxq_estimate_packet_buffer_size()
- Allocate memory with required size
- Create mmap and set its memory range to the allocated memory
DOCA mmap
2
- Start DOCA flow with desired port
- Create pipes to route incoming packets to DOCA ETH RXQ's queue
4

14.4.8.5.1.2 Working with DOCA Flow

In order to route incoming packets to the desired DOCA ETH RXQ, applications need to use DOCA Flow. Applications need to do the following:

- Create and start DOCA Flow on the appropriate port (device)
- Create pipes to route packets into
- Get the queue ID of the queue (inside DOCA ETH RXQ) using `doca_eth_rxq_get_flow_queue_id()`
- Add an entry to a pipe which routes packets into the RX queue (using the queue ID we obtained)



For more details see DOCA ETH RXQ samples and DOCA Flow.

14.4.8.5.2 DOCA ETH TXQ

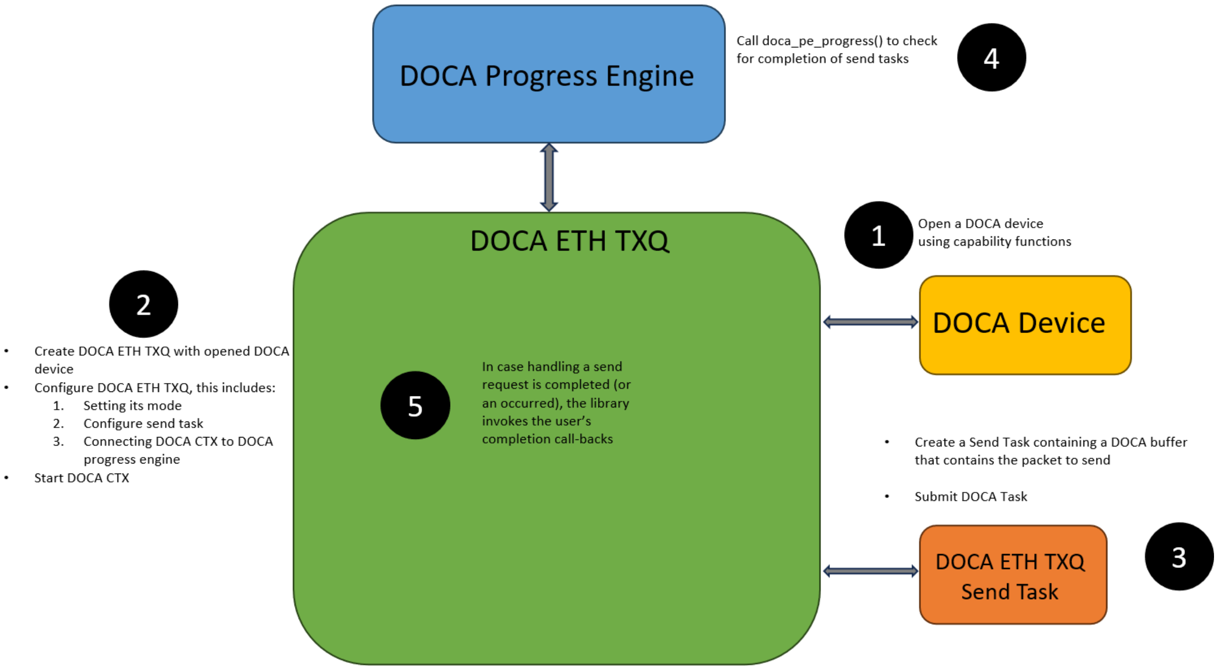
14.4.8.5.2.1 Operating Modes

DOCA ETH TXQ can only operate in one mode.

Regular Send

For the CPU datapath, the user should submit a send task containing a `doca_buf` of the packet to send.

For information regarding the datapath on the GPU, see DOCA GPUNetIO.


DOCA Progress Engine
Call doca_pe_progress() to check for completion of send tasks
4
DOCA ETH TXQ
1
Open a DOCA device using capability functions
DOCA Device
2
Create DOCA ETH TXQ with opened DOCA device
Configure DOCA ETH TXQ, this includes:
1. Setting its mode
2. Configure send task
3. Connecting DOCA CTX to DOCA progress engine
Start DOCA CTX
5
In case handling a send request is completed (or an occurred), the library invokes the user's completion call-backs
Create a Send Task containing a DOCA buffer that contains the packet to send
Submit DOCA Task
DOCA ETH TXQ Send Task
3

14.4.8.5.2.2 Offloads

DOCA ETH TXQ supports:

- Large Segment Offloading (LSO) - the hardware supports LSO on transmitted TCP packets over IPv4 and IPv6. LSO enables the software to prepare a large TCP message for sending with a header template (the application should provide this header to the library) which is updated automatically for every generated segment. The hardware segments the large TCP message into multiple TCP segments. Per each such segment, device updates the header template accordingly (see LSO Send Task).
- L3/L4 checksum offloading - the hardware supports calculation of checksum on transmitted packets and validation of received packet checksum. Checksum calculation is supported for TCP/UDP running over IPv4 and IPv6. (In case of tunneling, the hardware calculates the checksum of the outer header.) The hardware does not require any pseudo header checksum calculation, and the value placed in TCP/UDP checksum is ignored when performing the calculation. See `doca_eth_txq_set_l3_chksum_offload()` / `doca_eth_txq_set_l4_chksum_offload()` .

14.4.8.5.3 Objects

- `doca_mmap` - in Cyclic Receive and Managed Memory Pool Receive modes, the user must configure DOCA ETH RXQ with packet buffer to write the received packets into as a `doca_mmap` (see DOCA Core Memory Subsystem)
- `doca_buf` - in Regular Receive mode, the user must submit receive tasks that includes a buffer to write the received packet into as a `doca_buf` . Also, In Regular Send mode, the user must submit send tasks that include a buffer of the packet to send as a `doca_buf` (see DOCA Core Memory Subsystem).

14.4.8.6 Configurations Phase

To start using the library, the user must first first go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context to allow execution of tasks and retrieval of events.

> ⚠ DOCA ETH in GPU datapath does not need to be associated with a DOCA PE (since the datapath is not on the CPU).

14.4.8.6.1 Configurations

The context can be configured to match the application use case.

To find if a configuration is supported or the min/max value for it, refer to Device Support.

14.4.8.6.2 Mandatory Configurations

These configurations are mandatory and must be set by the application before attempting to start the context.

14.4.8.6.2.1 DOCA ETH RXQ

- At least one task/event/event_batch type must be configured. Refer to Tasks/Events/Event Batch for more information.
- Max packet size (the maximum size of packet that can be received) must be provided at creation time of the DOCA ETH RXQ context
- Max burst size (the maximum number of packets that the library can handle at the same time) must be provided at creation time of the DOCA ETH RXQ context
- A device with appropriate support must be provided upon creation
- When in Cyclic Receive or Managed Memory Pool Receive modes, a `doca_mmap` must be provided in-order write the received packets into (see `doca_eth_rxq_set_pkt_buf()`)
- In case of a GPU datapath, A DOCA GPU sub-device must be provided using `doca_ctx_set_datapath_on_gpu()`

14.4.8.6.2.2 DOCA ETH TXQ

- At least one task/task_batch type must be configured. Refer to Tasks/Task Batch for more information.
- Max burst size (the maximum number of packets that the library can handle at the same time) must be provided at creation time of the DOCA ETH TXQ context
- A device with appropriate support must be provided on creation
- In case of a GPU datapath, a DOCA GPU sub-device must be provided using `doca_ctx_set_datapath_on_gpu()`

14.4.8.6.3 Optional Configurations

The following configurations are optional. If they are not set, then a default value is used.

14.4.8.6.3.1 DOCA ETH RXQ

- RXQ mode - User can set the working mode using `doca_eth_rxq_set_type()`. The default type is Regular Receive.
- Max receive buffer list length - User can set the maximum length of buffer list/chain as a receive buffer using `doca_eth_rxq_set_max_recv_buf_list_len()`. The default value is 1.

14.4.8.6.3.2 DOCA ETH TXQ

- TXQ mode - User can set the working mode using `doca_eth_txq_set_type()`. The default type is Regular Send.
- Max send buffer list length - User can set the maximum length of buffer list/chain as a send buffer using `doca_eth_txq_set_max_send_buf_list_len()`. The default value is 1.
- L3/L4 offload checksum - User can enable/disable L3/L4 checksum offloading using `doca_eth_txq_set_l3_chksum_offload()` / `doca_eth_txq_set_l4_chksum_offload()`. They are disabled by default.
- MSS - User can set MSS (maximum segment size) value for LSO send task/task_batch using `doca_eth_txq_set_mss()`. The default value is 1500.
- Max LSO headers size - User can set the maximum LSO headers size for LSO send task/task_batch using `doca_eth_txq_set_max_lso_header_size()`. The default value is 74.

14.4.8.6.4 Device Support

DOCA ETH requires a device to operate. For picking a device, see DOCA Core Device Discovery.

To check if a device supports a specific mode, use the type capabilities functions (see `doca_eth_rxq_cap_is_type_supported()` and `doca_eth_txq_cap_is_type_supported()`).

Devices can allow the following capabilities:

- The maximum burst size
- The maximum buffer chain list (only for Regular Receive/Regular Send modes)
- The maximum packet size (only for DOCA ETH RXQ)
- L3/L4 checksum offloading capability (only for DOCA ETH TXQ)
- Maximum LSO message/header size (only for DOCA ETH TXQ)
- Wait-on-time offloading capability (only for DOCA ETH TXQ in GPU datapath)

14.4.8.6.5 Buffer Support

DOCA ETH support buffers (`doca_mmap` or `doca_buf`) with the following features:

Buffer Type	Send Task	LSO Send Task	Receive Task	Managed Receive Event
Local mmap buffer	Yes	Yes	Yes	Yes

Buffer Type	Send Task	LSO Send Task	Receive Task	Managed Receive Event
Mmap from PCIe export buffer	Yes	Yes	Yes	Yes
Mmap from RDMA export buffer	No	No	No	No
Linked list buffer	Yes	Yes	Yes	No

For buffer support in the case of GPU datapath, see DOCA GPUNetIO Programming Guide.

14.4.8.7 Execution Phase

This section describes execution on CPU (unless stated otherwise) using DOCA Core Progress Engine.

> ⚠ For information regarding GPU datapath, see DOCA GPUNetIO.

14.4.8.7.1 Tasks

DOCA ETH exposes asynchronous tasks that leverage the DPU hardware according to the DOCA Core architecture. See DOCA Core Task.

14.4.8.7.1.1 DOCA ETH RXQ

Receive Task

This task allows receiving packets from a `doca_dev`.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	calling `doca_eth_rxq_task_recv_set_conf()`	`doca_eth_rxq_cap_is_type_supported()` checking support for Regular Receive mode
Number of tasks	`task_recv_num` in `doca_eth_rxq_task_recv_set_conf()`	-
Max receive buffer list length	`doca_eth_rxq_set_max_recv_buf_list_len()` (default value is 1)	`doca_eth_rxq_cap_get_max_recv_buf_list_len()`
Maximal packet size	`max_packet_size` in `doca_eth_rxq_create()`	`doca_eth_rxq_cap_get_max_packet_size()`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Packet buffer	Buffer pointing to the memory where received packet are to be written	The received packet is written to the tail segment extending the data segment

Task Output

Common output as described in DOCA Core Task.

Additionally:

Name	Description	Notes
L3 checksum result	Value indicating whether the L3 checksum of the received packet is valid or not	Can be queried using `doca_eth_rxq_task_recv_get_l3_ok()`
L4 checksum result	Value indicating whether the L4 checksum of the received packet is valid or not	Can be queried using `doca_eth_rxq_task_recv_get_l4_ok()`

Task Completion Success

After the task is completed successfully the following will happen:

- The received packet is written to the packet buffer
- The packet buffer data segment is extended to include the received packet

Task Completion Failure

If the task fails midway:

- The context enters stopping state
- The packet buffer `doca_buf` object is not modified
- The packet buffer contents may be modified

Task Limitations

All limitations described in DOCA Core Task

Additionally:

- The operation is not atomic.
- Once the task has been submitted, then the packet buffer should not be read/written to.

14.4.8.7.1.2 DOCA ETH TXQ

Send Task

This task allows sending packets from a `doca_dev`.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	calling `doca_eth_txq_task_send_set_conf()`	`doca_eth_txq_cap_is_type_supported()` checking support for Regular Send mode
Number of tasks	`task_send_num` in `doca_eth_txq_task_send_set_conf()`	-
Max send buffer list length	`doca_eth_txq_set_max_send_buf_list_len()` (default value is 1)	`doca_eth_txq_cap_get_max_send_buf_list_len()`

L3/L4 offload checksum	`doca_eth_txq_set_l3_chksum_offload()`
	`doca_eth_txq_set_l4_chksum_offload()`
	Disabled by default.
		`doca_eth_txq_cap_is_l3_chksum_offload_supported()`
		`doca_eth_txq_cap_is_l4_chksum_offload_supported()`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Packet buffer	Buffer pointing to the packet to send	The sent packet is the memory in the data segment

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

The task finishing successfully does not guarantee that the packet has been transmitted onto the wire. It only signifies that the packet has successfully entered the device's TX hardware and that the packet buffer `doca_buf` is no longer in the library's ownership and it can be reused by the application.

Task Completion Failure

If the task fails midway:

- The context enters stopping state
- The packet buffer `doca_buf` object is not modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the packet buffer should not be written to
- Other limitations are described in DOCA Core Task

LSO Send Task

This task allows sending "large" packets (larger than MTU) from a `doca_dev` (hardware splits the packet into several packets smaller than the MTU and sends them).

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	calling `doca_eth_txq_task_lso_send_set_conf()`	`doca_eth_txq_cap_is_type_supported()` checking support for Regular Send mode
Number of tasks	`task_lso_send_num` in `doca_eth_txq_task_lso_send_set_conf()`	-
Max send buffer list length	`doca_eth_txq_set_max_send_buf_list_len()` (default value is 1)	`doca_eth_txq_cap_get_max_send_buf_list_len()`

L3/L4 offload checksum	`doca_eth_txq_set_l3_chksum_offload()`
	`doca_eth_txq_set_l4_chksum_offload()`
	(disabled by default)
		`doca_eth_txq_cap_is_l3_chksum_offload_supported()`
		`doca_eth_txq_cap_is_l4_chksum_offload_supported()`

MSS	`doca_eth_txq_set_mss()` (default value is 1500)	-
Max LSO headers size	`doca_eth_txq_set_max_lso_header_size()` (default value is 74)	`doca_eth_txq_cap_get_max_lso_header_size()`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Packet payload buffer	Buffer that points to the "large" packet's payload (does not include headers) to send	The sent packet is the memory in the data segment
Packet headers buffer	Gather list that when combined includes the "large" packet's headers to send	See `struct doca_gather_list`

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

The task finishing successfully does not guarantee that the packet has been transmitted onto the wire. It only means that the packet has successfully entered the device's TX hardware and that the packet payload buffer and the packet headers buffer is no longer in the library's ownership and it can be reused by the application.

Task Completion Failure

If the task fails midway:

- The context enters stopping state
- The packet payload buffer `doca_buf` object and the packet header buffer `doca_gather_list` are not modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the packet payload buffer and the packet headers buffer should not be written to
- Other limitations are described in DOCA Core Task

14.4.8.7.2 Events

DOCA ETH exposes asynchronous events to notify about changes that happen asynchronously, according to the DOCA Core architecture. See DOCA Core Event.

In addition to common events as described in DOCA Core Event, DOCA ETH exposes an extra events:

14.4.8.7.2.1 DOCA ETH RXQ

Managed Receive Event

This event allows receiving packets from a `doca_dev` (without requiring the application to manage the memory the packets are written to).

Event Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	`doca_eth_rxq_event_managed_recv_register()`	`doca_eth_rxq_cap_is_type_supported()` checking support for Managed Memory Pool Receive mode

Event Trigger Condition

The event is triggered every time a packet is received.

Event Success Handler

The success callback (provided in the event registration) is invoked and the user is expected to perform the following:

- Use the `pkt` parameter to process the received packet
- Use `event_user_data` to get the application context
- Query L3/L4 checksum results of the packet
- Free the `pkt` (a `doca_buf` object) and return it to the library

> ⚠ Not freeing the `pkt` may cause scenario where packets are lost.

Event Failure Handler

The failure callback (provided in the event registration) is invoked, and the following happens:

- The context enters stopping state
- The `pkt` parameter becomes NULL
- The `event_user_data` parameter contains the value provided by the application when registering the event

14.4.8.7.2.2 DOCA ETH TXQ

Error Send Packet

This event is relevant when running DOCA ETH on GPU datapath (see DOCA GPUNetIO). It allows detecting failure in sending packets.

Event Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	`doca_eth_txq_gpu_event_error_send_packet_register()`	Always supported

Event Trigger Condition

The event is triggered when sending a packet fails.

Event Handler

The callback (provided in the event registration) is invoked and the user can:

- Get the position (index) of the packet that TXQ failed to send

Notify Send Packet

This event is relevant when running DOCA ETH on GPU datapath (see DOCA GPUNetIO). It notifies user every time a packet is sent successfully.

Event Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	`doca_eth_txq_gpu_event_notify_send_packet_register()`	Always supported

Event Trigger Condition

The event is triggered when sending a packet fails.

Event Handler

The callback (provided in the event registration) is invoked and the user can:

- Get the position (index) of the packet was sent
- Timestamp of sending the packet

14.4.8.7.3 Task Batch

DOCA ETH exposes asynchronous task batches that leverage the BlueField Platform hardware according to the DOCA Core architecture.

14.4.8.7.3.1 DOCA ETH RXQ

There are no task batches in ETH RXQ at the moment.

14.4.8.7.3.2 DOCA ETH TXQ

Send Task Batch

This is an extended task batch for Send Task which allows batched sending of packets from a `doca_dev`.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task batch	calling `doca_eth_txq_task_batch_send_set_conf()`	`doca_eth_txq_cap_is_type_supported()` checking support for Regular Send mode
Number of task batches	`num_task_batches` in `doca_eth_txq_task_batch_send_set_conf()`	-
Max number of tasks per task batch	`max_tasks_number` in `doca_eth_txq_task_batch_send_set_conf()`	-
Max send buffer list length	`doca_eth_txq_set_max_send_buf_list_len()` (default value is 1)	`doca_eth_txq_cap_get_max_send_buf_list_len()`

L3/L4 offload checksum	`doca_eth_txq_set_l3_chksum_offload()`
	`doca_eth_txq_set_l4_chksum_offload()`
	Disabled by default.
		`doca_eth_txq_cap_is_l3_chksum_offload_supported()`
		`doca_eth_txq_cap_is_l4_chksum_offload_supported()`

Task Input

Name	Description	Notes
Tasks number	Number of send tasks "behind" the task batch	This number equals the number of packets to send
Batch user data	User data associated for the task batch	-
Packets array	Pointer to an array of buffers pointing at the packets to send per task	The sent packet is the memory in the data segment of each buffer
User data array	Pointer to an array of user data per task	-

Task Output

Name	Description
Status array	Pointer to an array of statuses per task of the finished task batch

Task Completion Success

A task batch is complete if all the send tasks finished successfully and all the packets entered the device's TX hardware. All packets in the "Packet array" are now in the ownership of the user.

Task Completion Failure

If a task batch fails, then one (or more) of the tasks associated with the task batch failed. The user can look at "Status array" to see which task/packet caused the failure.

Also, the following behavior is expected:

- The context enters stopping state
- The packet's `doca_buf` objects are not modified

Task Limitations

In addition to all the Send Task Limitations:

- Task batch completion occurs only when all the tasks are completed (no partial completion)

LSO Send Task Batch

This is an extended task batch for LSO Send Task which allows batched sending of LSO packets from a `doca_dev`.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task batch	Calling `doca_eth_txq_task_batch_lso_send_set_conf()`	`doca_eth_txq_cap_is_type_supported()` checking support for Regular Send mode
Number of task batches	`num_task_batches` in `doca_eth_txq_task_batch_lso_send_set_conf()`	-
Max number of tasks per task batch	`num_task_batches` in `doca_eth_txq_task_batch_lso_send_set_conf()`	-
Max send buffer list length	`doca_eth_txq_set_max_send_buf_list_len()` (default value is 1)	`doca_eth_txq_cap_get_max_send_buf_list_len()`

L3/L4 offload checksum	`doca_eth_txq_set_l3_chksum_offload()`
	`doca_eth_txq_set_l4_chksum_offload()`
	Disabled by default.
		`doca_eth_txq_cap_is_l3_chksum_offload_supported()`
		`doca_eth_txq_cap_is_l4_chksum_offload_supported()`

Description	API to Set the Configuration	API to Query Support
MSS	`doca_eth_txq_set_mss()` (default value is 1500)	-
Max LSO headers size	`doca_eth_txq_set_max_lso_header_size()` (default value is 74)	`doca_eth_txq_cap_get_max_lso_header_size()`

Task Input

Name	Description	Notes
Tasks number	Number of send tasks "behind" the task batch	This number equals the number of packets to send
Batch user data	User data associated for the task batch	-
Packets payload array	Pointer to an array of buffers pointing at the "large" packet's payload to send per task	The sent packet payload is the memory in the data segment of each buffer
Packets headers array	Pointer to an array of gather lists, each of which when combined assembles a "large" packet's headers to send per task	See `struct doca_gather_list`
User data array	Pointer to an array of user data per task	-

Task Output

Name	Description
Status array	Pointer to an array of status per task of the finished task batch

Task Completion Success

A task batch is complete if all the LSO send tasks finished successfully and all the packets entered the device's TX hardware. All packet payload in "Packets payload array" and packet headers in "Packets headers array" are now in the ownership of the user.

Task Completion Failure

If a task batch fails, then one (or more) of the tasks associated with the task batch failed, and the user can look at the "Status array" to try and figure out which task/packet caused the failure.

Also, the following behavior is expected:

- The context enters stopping state
- The packets payload `doca_buf` objects are not modified
- The packets headers `doca_gather_list` objects are not modified

Task Limitations

In addition to all the LSO Send Task Limitations:

- Task batch completion happens only when all the tasks are completed (no partial completion)

14.4.8.7.4 Event Batch

DOCA ETH exposes asynchronous event batches to notify about changes that happen asynchronously.

14.4.8.7.4.1 DOCA ETH RXQ

Managed Receive Event Batch

This is an extended event batch for Managed Receive Event which allows receiving packets from a `doca_dev` (without requiring the application to manage the memory the packets are written to).

Event Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event batch	Calling `doca_eth_rxq_event_batch_managed_recv_register()`	`doca_eth_rxq_cap_is_type_supported()` checking support for Managed Memory Pool Receive mode
Max events number: Equal to the maximum number of completed events per event batch completion	`events_number_max` in `doca_eth_rxq_event_batch_managed_recv_register()`	-
Min events number: Equal to the minimum number of completed events per event batch completion	`events_number_min` in `doca_eth_rxq_event_batch_managed_recv_register()`	-

Event Trigger Condition

The event batch is triggered every time a number of packets (number between "Min events number" and "Max events number") are received.

Event Batch Success Handler

The success callback (provided in the event of batch registration) is invoked and the user is expected to perform the following:

1. Identify the number of received packets by `events_number`.
2. Use the `pkt_array` parameter to process the received packets.
3. Use `event_batch_user_data` to get the application context.
4. Query the L3/L4 checksum results of the packets using `l3_ok_array` and `l4_ok_array`.
5. Free the buffers from `pkt_array` (a `doca_buf` object) and return it to the library. This can be done in two ways:
   - Iterating over the buffers in `pkt_array` and freeing them using `doca_buf_dec_refcount()`.
   - Freeing all the buffers in `pkt_array` together (gives better performance) using `doca_eth_rxq_event_batch_managed_recv_pkt_array_free()`.

Event Batch Failure Handler

The failure callback (provided in the event batch registration) is invoked, and the following happens:

- The context enters stopping state
- The `pkt_array` parameter is NULL
- The `l3_ok_array` parameter is NULL
- The `l4_ok_array` parameter is NULL
- The `event_batch_user_data` parameter contains the value provided by the application when registering the event

14.4.8.7.4.2 DOCA ETH TXQ

There are no event batches in ETH TXQ at the moment.

14.4.8.8 State Machine

The DOCA ETH library follows the Context state machine as described in DOCA Core Context State Machine.

The following section describes how to move to the state and what is allowed in each state.

14.4.8.8.1 Idle

In this state it is expected that application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to Configurations
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Creating the context
Running	Calling stop after: • All tasks are completed and freed • All `doca_buf` objects returned by Managed Receive Event callback are freed
Stopping	Calling progress until: • All tasks are completed and freed • All `doca_buf` objects returned by Managed Receive Event callback are freed

14.4.8.8.2 Starting

This state cannot be reached.

14.4.8.8.3 Running

In this state it is expected that application will do the following:

- Allocate and submit tasks
- Call progress to complete tasks and/or receive events

Allowed operations:

- Allocate previously configured task
- Submit a task
- Call `doca_eth_rxq_get_flow_queue_id()` to connect the RX queue to DOCA Flow
- Call stop

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

14.4.8.8.4 Stopping

In this state, it is expected that application:

- Calls progress to complete all inflight tasks (tasks complete with failure)
- Frees any completed tasks
- Frees `doca_buf` objects returned by Managed Receive Event callback

Allowed operations:

- Call progress

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and fatal error occurs
Running	Call stop without either: • Freeing all tasks • Freeing all `doca_buf` objects returned by Managed Receive Event callback

14.4.8.9 Alternative Datapath Options

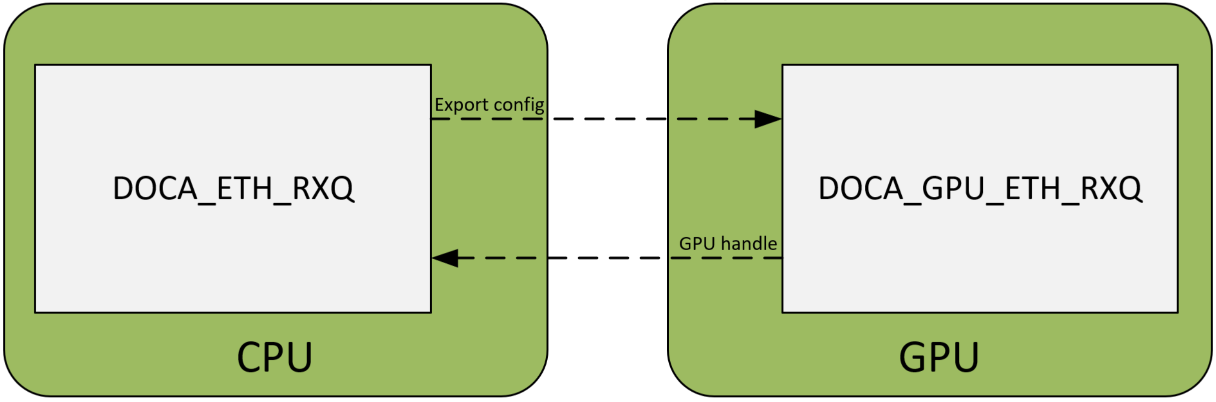
In addition to the CPU datapath (mentioned in Execution Phase), DOCA ETH supports running on GPU datapath. This allows applications to release the CPU from datapath management and allow low latency GPU processing of network traffic.

To export the handles, the application should call `doca_ctx_set_datapath_on_gpu()` before `doca_ctx_start()` to program the library to set up a GPU operated context.

To get the GPU context handle, the user should call `doca_rxq_get_gpu_handle()` which returns a pointer to a handle in the GPU memory space.

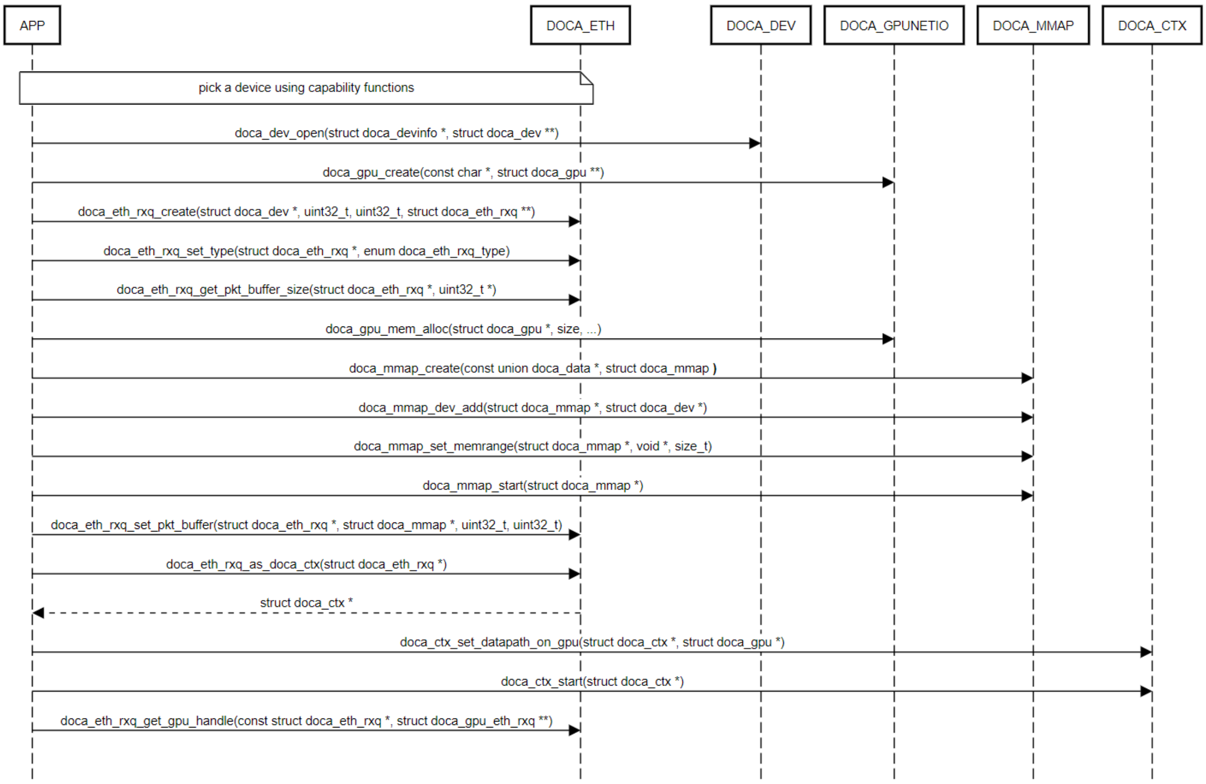
> ⚠ The datapath cannot be managed concurrently for the GPU and the CPU.

The DOCA ETH context is configured on the CPU and then exported to the GPU:



The following example shows the expected flow for a GPU-managed datapath with packets being scattered to GPU memory (for `doca_eth_rxq`):

1. Create a DOCA GPU device handler.
2. Create `doca_eth_rxq` and configure its parameters.
3. Set the datapath of the context to GPU.
4. Start the context.
5. Get a GPU handle of the context.


APP
DOCA_ETH
DOCA_DEV
DOCA_GPUNETIO
DOCA_MMAP
DOCA_CTX
pick a device using capability functions
doca_dev_open(struct doca_devinfo *, struct doca_dev **)
doca_gpu_create(const char *, struct doca_gpu **)
doca_eth_rxq_create(struct doca_dev *, uint32_t, uint32_t, struct doca_eth_rxq **)
doca_eth_rxq_set_type(struct doca_eth_rxq *, enum doca_eth_rxq_type)
doca_eth_rxq_get_pkt_buffer_size(struct doca_eth_rxq *, uint32_t *)
doca_gpu_mem_alloc(struct doca_gpu *, size, ...)
doca_mmap_create(const union doca_data *, struct doca_mmap)
doca_mmap_dev_add(struct doca_mmap *, struct doca_dev *)
doca_mmap_set_memrange(struct doca_mmap *, void *, size_t)
doca_mmap_start(struct doca_mmap *)
doca_eth_rxq_set_pkt_buffer(struct doca_eth_rxq *, struct doca_mmap *, uint32_t, uint32_t)
doca_eth_rxq_as_doca_ctx(struct doca_eth_rxq *)
struct doca_ctx *
doca_ctx_set_datapath_on_gpu(struct doca_ctx *, struct doca_gpu *)
doca_ctx_start(struct doca_ctx *)
doca_eth_rxq_get_gpu_handle(const struct doca_eth_rxq *, struct doca_gpu_eth_rxq **)

For more information regarding the GPU datapath see DOCA GPUNetIO.

14.4.8.10 DOCA ETH Samples

This section describes DOCA ETH samples based on the DOCA ETH library.

The samples illustrate how to use the DOCA ETH API to do the following:

- Send "regular" packets (smaller than MTU) using DOCA ETH TXQ
- Send "large" packets (larger than MTU) using DOCA ETH TXQ
- Receive packets using DOCA ETH RXQ in Regular Receive mode
- Receive packets using DOCA ETH RXQ in Managed Memory Pool Receive mode

> ⓘ All of the DOCA samples described in this section are governed under the BSD-3 software license agreement.

14.4.8.10.1 Running the Samples

1. Refer to the following documents:
    - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
    - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample (e.g., `eth_txq_send_ethernet_frames`):

    ```
    cd /opt/mellanox/doca/samples/doca_eth/eth_txq_send_ethernet_frames
    meson /tmp/build
    ninja -C /tmp/build
    ```

    The binary `eth_txq_send_ethernet_frames` is created under `/tmp/build/`.
3. Sample (e.g., `eth_txq_send_ethernet_frames`) usage:

    ```
    Usage: doca_eth_txq_send_ethernet_frames [DOCA Flags] [Program Flags]

    DOCA Flags:
      -h, --help                              Print a help synopsis
       -v, --version                           Print program version information
      -l, --log-level                         Set the (numeric) log level for the program
    <10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>

       --sdk-log-level                         Set the SDK (numeric) log level for the program <10=DISABLE,
    20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>

    -j, --json <path>                       Parse all command flags from an input json file

    Program Flags:
       -d, --device                      IB device name - default: mlx5_0
       -m, --mac-addr                    Destination MAC address to associate with the ethernet frames -
    default: FF:FF:FF:FF:FF:FF
    ```
4. For additional information per sample, use the `-h` option:

    ```
    /tmp/build/<sample_name> -h
    ```

14.4.8.10.2 Samples

> ⚠ The following samples are for the CPU datapath. For GPU datapath samples, see DOCA GPUNetIO.

14.4.8.10.2.1 ETH TXQ Send Ethernet Frames

This sample illustrates how to send a "regular" packet (smaller than MTU) using DOCA ETH TXQ.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the required DOCA Core structures.
3. Populating DOCA memory map with one buffer to the packet's data.
4. Writing the packet's content into the allocated buffer.
5. Allocating elements from DOCA buffer inventory for the buffer.
6. Initializing and configuring DOCA ETH TXQ context.
7. Starting the DOCA ETH TXQ context.
8. Allocating DOCA ETH TXQ send task.
9. Submitting DOCA ETH TXQ send task into progress engine.
10. Retrieving DOCA ETH TXQ send task from the progress engine.
11. Handling the completed task using the provided callback.
12. Stopping the DOCA ETH TXQ context.
13. Destroying DOCA ETH TXQ context.
14. Destroying all DOCA Core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_eth/eth_txq_send_ethernet_frames/eth_txq_send_ethernet_frames_sample.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_txq_send_ethernet_frames/eth_txq_send_ethernet_frames_main.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_txq_send_ethernet_frames/meson.build`

14.4.8.10.2.2 ETH TXQ LSO Send Ethernet Frames

This sample illustrates how to send a "large" packet (larger than MTU) using DOCA ETH TXQ.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the required DOCA Core structures.
3. Populating DOCA memory map with one buffer to the packet's payload.
4. Writing the packet's payload into the allocated buffer.
5. Allocating elements from DOCA Buffer inventory for the buffer.
6. Allocating DOCA gather list consisting of one node to the packet's headers.
7. Writing the packet's headers into the allocated gather list node.
8. Initializing and configuring DOCA ETH TXQ context.
9. Starting the DOCA ETH TXQ context.
10. Allocating DOCA ETH TXQ LSO send task.
11. Submitting DOCA ETH TXQ LSO send task into progress engine.
12. Retrieving DOCA ETH TXQ LSO send task from the progress engine.
13. Handling the completed task using the provided callback.
14. Stopping the DOCA ETH TXQ context.
15. Destroying DOCA ETH TXQ context.
16. Destroying all DOCA Core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_eth/eth_txq_lso_send_ethernet_frames/eth_txq_lso_send_ethernet_frames_sample.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_txq_lso_send_ethernet_frames/eth_txq_lso_send_ethernet_frames_main.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_txq_lso_send_ethernet_frames/meson.build`

14.4.8.10.2.3 ETH TXQ Batch Send Ethernet Frames

This sample illustrates how to send a batch of "regular" packets (smaller than MTU) using DOCA ETH TXQ.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the required DOCA Core structures.
3. Populating DOCA memory map with multiple buffers, each representing a packet's data.
4. Writing the packets' content into the allocated buffers.
5. Allocating elements from DOCA Buffer inventory for the buffers.
6. Initializing and configuring DOCA ETH TXQ context.
7. Starting the DOCA ETH TXQ context.
8. Allocating DOCA ETH TXQ send task batch.
9. Copying all buffers' pointers to task batch's `pkt_arry`.
10. Submitting DOCA ETH TXQ send task batch into the progress engine.
11. Retrieving DOCA ETH TXQ send task batch from the progress engine.
12. Handling the completed task batch using the provided callback.
13. Stopping the DOCA ETH TXQ context.
14. Destroying DOCA ETH TXQ context.
15. Destroying all DOCA Core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_eth/eth_txq_batch_send_ethernet_frames/eth_txq_batch_send_ethernet_frames_sample.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_txq_batch_send_ethernet_frames/eth_txq_batch_send_ethernet_frames_main.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_txq_batch_send_ethernet_frames/meson.build`

14.4.8.10.2.4 ETH TXQ Batch LSO Send Ethernet Frames

This sample illustrates how to send a batch of "large" packets (larger than MTU) using DOCA ETH TXQ.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the required DOCA Core structures.
3. Populating DOCA memory map with multiple buffers, each representing a packet's payload.
4. Writing the packets' payload into the allocated buffers.

5. Allocating elements from DOCA Buffer inventory for the buffers.
6. Allocating DOCA gather lists each consisting of one node for the packet's headers.
7. Writing the packets' headers into the allocated gather list nodes.
8. Initializing and configuring DOCA ETH TXQ context.
9. Starting the DOCA ETH TXQ context.
10. Allocating DOCA ETH TXQ LSO send task.
11. Copying all buffers' pointers to task batch's `pkt_payload_arry`.
12. Copying all gather lists' pointers to task batch's `headers_arry`.
13. Submitting DOCA ETH TXQ LSO send task batch into the progress engine.
14. Retrieving DOCA ETH TXQ LSO send task batch from the progress engine.
15. Handling the completed task batch using the provided callback.
16. Stopping the DOCA ETH TXQ context.
17. Destroying DOCA ETH TXQ context.
18. Destroying all DOCA Core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_eth/eth_txq_batch_lso_send_ethernet_frames/eth_txq_batch_lso_send_ethernet_frames_sample.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_txq_batch_lso_send_ethernet_frames/eth_txq_batch_lso_send_ethernet_frames_main.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_txq_batch_lso_send_ethernet_frames/meson.build`

14.4.8.10.2.5 ETH RXQ Regular Receive

This sample illustrates how to receive a packet using DOCA ETH RXQ in Regular Receive mode.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the required DOCA Core structures.
3. Populating DOCA memory map with one buffer to the packet's data.
4. Allocating element from DOCA Buffer inventory for each buffer.
5. Initializing DOCA Flow.
6. Initializing and configuring DOCA ETH RXQ context.
7. Starting the DOCA ETH RXQ context.
8. Starting DOCA Flow.
9. Creating a pipe connecting to DOCA ETH RXQ's RX queue.
10. Allocating DOCA ETH RXQ receive task.
11. Submitting DOCA ETH RXQ receive task into the progress engine.
12. Retrieving DOCA ETH RXQ receive task from the progress engine.
13. Handling the completed task using the provided callback.
14. Stopping DOCA Flow.
15. Stopping the DOCA ETH RXQ context.
16. Destroying DOCA ETH RXQ context.
17. Destroying DOCA Flow.
18. Destroying all DOCA Core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_regular_receive/eth_rxq_regular_receive_sample.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_regular_receive/eth_rxq_regular_receive_main.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_regular_receive/meson.build`

14.4.8.10.2.6 ETH RXQ Managed Receive

This sample illustrates how to receive packets using DOCA ETH RXQ in Managed Memory Pool Receive mode.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the required DOCA Core structures.
3. Calculating the required size of the buffer to receive the packets from DOCA ETH RXQ.
4. Populating DOCA memory map with a packets buffer.
5. Initializing DOCA Flow.
6. Initializing and configuring DOCA ETH RXQ context.
7. Registering DOCA ETH RXQ managed receive event.
8. Starting the DOCA ETH RXQ context.
9. Starting DOCA Flow.
10. Creating a pipe connecting to DOCA ETH RXQ's RX queue.
11. Retrieving DOCA ETH RXQ managed receive events from the progress engine.
12. Handling the completed events using the provided callback.
13. Stopping DOCA Flow.
14. Stopping the DOCA ETH RXQ context.
15. Destroying DOCA ETH RXQ context.
16. Destroying DOCA Flow.
17. Destroying all DOCA Core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_managed_mempool_receive/eth_rxq_managed_mempool_receive_sample.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_managed_mempool_receive/eth_rxq_managed_mempool_receive_main.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_managed_mempool_receive/meson.build`

14.4.8.10.2.7 ETH RXQ Batch Managed Receive

This sample illustrates how to receive batches of packets using DOCA ETH RXQ in Managed Memory Pool Receive mode.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the required DOCA Core structures.

3. Calculating the required size of the buffer to receive the packets from DOCA ETH RXQ.
4. Populating DOCA memory map with a packets buffer.
5. Initializing DOCA Flow.
6. Initializing and configuring DOCA ETH RXQ context.
7. Registering DOCA ETH RXQ managed receive event batch.
8. Starting the DOCA ETH RXQ context.
9. Starting DOCA Flow.
10. Creating a pipe connecting to DOCA ETH RXQ's RX queue.
11. Retrieving DOCA ETH RXQ managed receive event batches from the progress engine.
12. Handling the completed event batches using the provided callback.
13. Stopping DOCA Flow.
14. Stopping the DOCA ETH RXQ context.
15. Destroying DOCA ETH RXQ context.
16. Destroying DOCA Flow.
17. Destroying all DOCA Core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_batch_managed_mempool_receive/eth_rxq_batch_managed_mempool_receive_sample.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_batch_managed_mempool_receive/eth_rxq_batch_managed_mempool_receive_main.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_batch_managed_mempool_receive/meson.build`

14.4.9 DOCA GPUNetIO

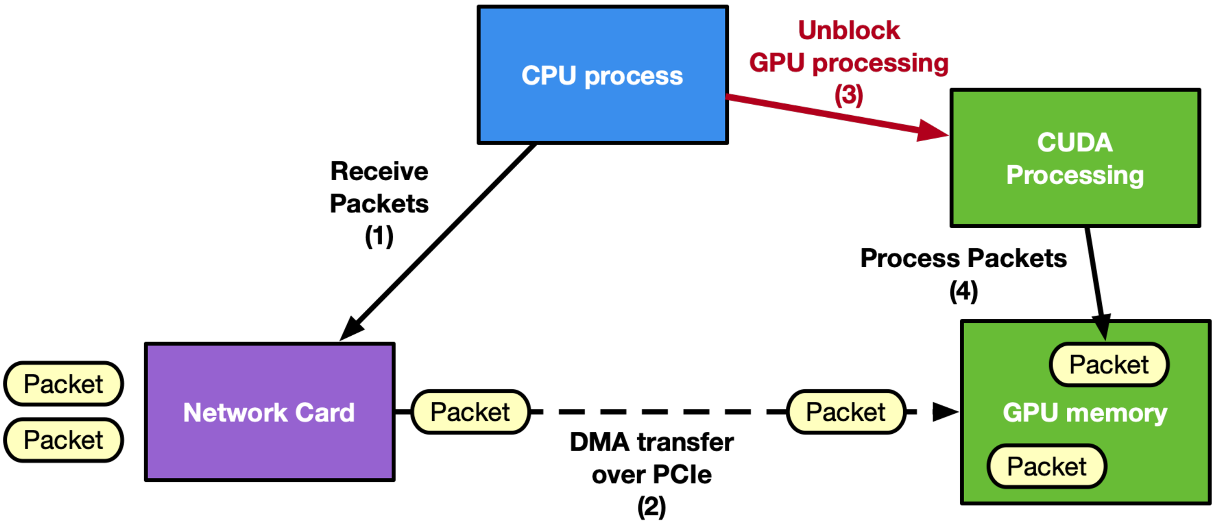
This document provides an overview and configuration instructions for DOCA GPUNetIO API.

14.4.9.1 Introduction

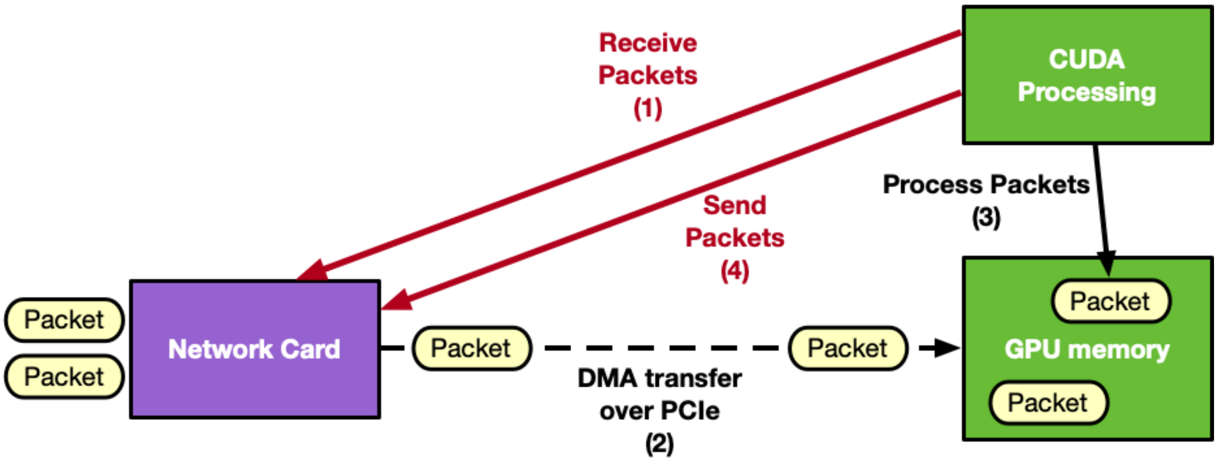
Real-time GPU processing of network packets is a technique useful for application domains involving signal processing, network security, information gathering, input reconstruction, and more. These applications involve the CPU in the critical path (CPU-centric approach) to coordinate the network card (NIC) for receiving packets in the GPU memory (GPUDirect RDMA) and notifying a packet-processing CUDA kernel waiting on the GPU for a new set of packets. In lower-power platforms, the CPU can easily become the bottleneck, masking GPU value. The aim is to maximize the zero-packet-loss throughput at the the lowest latency possible.

A CPU-centric approach may not be scalable when increasing the number of clients connected to the application as the time between two receive operations on the same queue (client) would increase with the number of queues. The new DOCA GPUNetIO library allows developers to orchestrate these kinds of applications while optimizing performance, combining GPUDirect RDMA for data-path acceleration, GDRCopy library to give the CPU direct access to GPU memory, and GPUDirect async kernel-initiated network (GDAKIN) communications to allow a CUDA kernel to directly control the NIC.

CPU-centric approach:



GPU-centric approach:



DOCA GPUNetIO enables GPU-centric solutions that remove the CPU from the critical path by providing the following features:

- GPUDirect async kernel-initiated technology – a GPU CUDA kernel can directly control other hardware components like the network card or NVIDIA® BlueField®'s DMA engine
  - GDAKIN communications – a GPU CUDA kernel can control network communications to send or receive data
    - GPU can control Ethernet communications
    - GPU can control RDMA communications (InfiniBand or RoCE are supported)
    - CPU intervention is not needed in the application critical path
  - DMA engine – a GPU CUDA kernel can trigger a memory copy using BlueField's DMA engine
- GPUDirect RDMA – use a contiguous GPU memory to send or receive RDMA data or Ethernet packets without CPU memory staging copies
- Semaphores – provide a standardized low-latency message passing protocol between two CUDA kernels or a CUDA kernel and a CPU thread
- Smart memory allocation – allocate aligned GPU memory buffers, possibly exposing them to direct CPU access

    - Combination of CUDA, DPDK gpudev library and GDRCopy library already embedded in the DPDK released with DOCA
- Accurate send scheduling - schedule Ethernet packets' send in the future according to a user-provided timestamp

Aerial 5G SDK, Morpheus, and Holoscan Advanced Network Operator are examples of NVIDIA applications actively using DOCA GPUNetIO.

For a deep dive into the technology and motivations, please refer to the NVIDIA blog posts Inline GPU Packet Processing with NVIDIA DOCA GPUNetIO and Unlocking GPU-Accelerated RDMA with NVIDIA DOCA GPUNetIO. Another NVIDIA blog post Realizing the Power of Real-time Network Processing with NVIDIA DOCA GPUNetIO has been published to provide more use-case examples where DOCA GPUNetIO has been useful to improve the execution.

> RDMA on DOCA GPUNetIO is currently supported at alpha level.

14.4.9.2 Changes From Previous Releases

14.4.9.2.1 Changes in 2.8

The following subsection(s) detail the `doca_gpunetio` library updates in version 2.8.0.

14.4.9.2.1.1 Added

- `__device__ doca_error_t doca_gpu_dev_buf_get_mkey(const struct doca_gpu_buf *buf, uint32_t *mkey)`
- `__device__ doca_error_t doca_gpu_dev_buf_create(uintptr_t addr, uint32_t mkey, struct doca_gpu_buf **buf)`
- `__device__ doca_error_t doca_gpu_dev_dma_memcpy(struct doca_gpu_dma *dma, struct doca_gpu_buf *src_buf, uint64_t src_offset, struct doca_gpu_buf *dst_buf, uint64_t dst_offset, size_t length)`
- `__device__ doca_error_t doca_gpu_dev_dma_commit(struct doca_gpu_dma *dma)`
- `__device__ doca_error_t doca_gpu_dev_rdma_wait_all(struct doca_gpu_dev_rdma *rdma, uint32_t *num_ops)`

Changed

- `struct doca_gpu_buf` - Added uint32_t `mkey` field after `size` field
- `struct doca_gpu_eth_rxq` - Added bool `need_flush`
- `__device__ doca_error_t doca_gpu_dev_rdma_recv_weak(struct doca_gpu_dev_rdma_r *rdma_r, size_t recv_length, uint64_t recv_offset, const uint32_t flags_bitmask, uint32_t position,` ~~`uint64_t *hdl`~~`);`
- `__device__ doca_error_t doca_gpu_dev_rdma_recv_strong(struct doca_gpu_dev_rdma_r *rdma_r, struct doca_gpu_buf *recv_buf, size_t recv_length, uint64_t recv_offset, const uint32_t flags_bitmask,` ~~`uint64_t *hdl`~~`);`

- `__device__ doca_error_t doca_gpu_dev_rdma_recv_wait_all(struct doca_gpu_dev_rdma_r *rdma_r)`
    - `→ __device__ doca_error_t doca_gpu_dev_rdma_recv_wait_all(struct doca_gpu_dev_rdma_r *rdma_r, uint64_t *hdl, const enum doca_gpu_dev_rdma_recv_wait_flags flags, enum doca_rdma_opcode *opcode, uint32_t *imm)`

14.4.9.3 System Configuration

DOCA GPUNetIO requires a properly configured environment which depends on whether the application should run on the x86 host or DPU Arm cores. The following subsections describe the required configuration in both scenarios, assuming DOCA, CUDA Toolkit and NVIDIA driver are installed on the system (x86 host or BlueField Arm) where the DOCA GPUNetIO is built and executed.

DOCA GPUNetIO is available for all DOCA for host and BFB packages downloadable here.

Assuming the DOCA package has been downloaded and installed on the system, to install all DOCA GPUNetIO components, run:

- For Ubuntu/Debian:

```
apt install doca-all doca-sdk-gpunetio libdoca-sdk-gpunetio-dev
```

- For RHEL:

```
yum install doca-all doca-sdk-gpunetio doca-sdk-gpunetio-devel
```

Internal hardware topology of the system should be GPUDirect-RDMA-friendly to maximize the internal throughput between the GPU and the NIC.

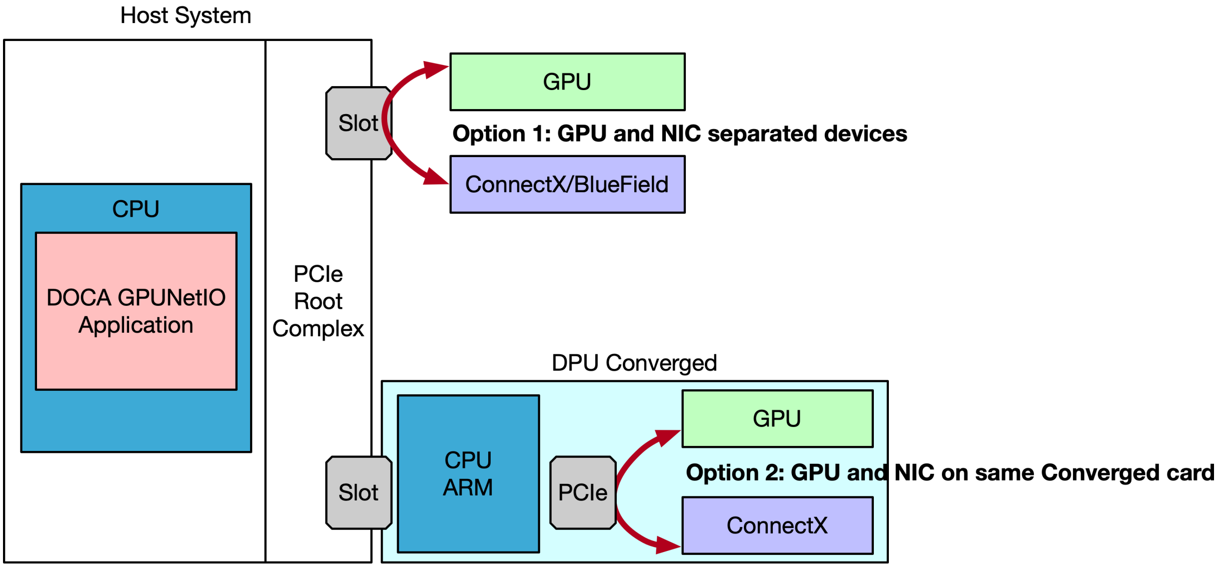
As DOCA GPUNetIO is present in both DOCA-for-Host and DOCA BFB (for BlueField Arm), a GPUNetIO application can be executed either on the host CPU or on the BlueField's Arm cores. The following subsections provide a description of both scenarios.

> **KVM**
>
> DOCA GPUNetIO has been tested on bare-metal and in docker but never in a virtualized environment. Using KVM is discouraged for now.

14.4.9.3.1 Application on Host CPU

Assuming the DOCA GPUNetIO application is running on the host x86 CPU cores, it is highly recommended to have a dedicated PCIe connection between the GPU and the NIC. This topology can be realized in two ways:

- Adding an additional PCIe switch to one of the PCIe root complex slots and attaching to this switch a GPU and a NVIDIA® ConnectX® adapter
- Connecting an NVIDIA® Converged Accelerator DPU to the PCIe root complex and setting it to NIC mode (i.e., exposing the GPU and NIC devices to the host)


Host System
CPU
DOCA GPUNetIO Application
PCIe Root Complex
Slot
GPU
Option 1: GPU and NIC separated devices
ConnectX/BlueField
DPU Converged
Slot
CPU ARM
PCIe
GPU
Option 2: GPU and NIC on same Converged card
ConnectX

You may check the topology of your system using `lspci -tvvv` or `nvidia-smi topo -m`.

14.4.9.3.1.1 Option 1: ConnectX Adapter in Ethernet Mode

> NVIDIA® ConnectX® firmware must be 22.36.1010 or later. It is highly recommended to only use NVIDIA adapter from ConnectX-6 Dx and later.

DOCA GPUNetIO allows a CUDA kernel to control the NIC when working with Ethernet protocol. For this reason, the ConnectX must be set to Ethernet mode.

To do that, follow these steps:

1. Start MST, check the status, and copy the MST device name:

```
# Start MST
mst start
mst status -v

MST modules:
------------
    MST PCI module is not loaded
    MST PCI configuration module loaded
PCI devices:
------------
DEVICE_TYPE             MST                           PCI       RDMA            NET
NUMA
ConnectX6DX(rev:0)      /dev/mst/mt4125_pciconf0.1    b5:00.1   mlx5_1          net-ens6f1                 0
ConnectX6DX(rev:0)      /dev/mst/mt4125_pciconf0      b5:00.0   mlx5_0          net-ens6f0                 0
```

2. Configure the NIC to Ethernet mode and enable Accurate Send Scheduling (if required on the send side):

   > The following example assumes that the adapter is dual-port. If single port, only P1 options apply.

```
mlxconfig -d <mst_device> s KEEP_ETH_LINK_UP_P1=1 KEEP_ETH_LINK_UP_P2=1 KEEP_IB_LINK_UP_P1=0
KEEP_IB_LINK_UP_P2=0
mlxconfig -d <mst_device> --yes set ACCURATE_TX_SCHEDULER=1 REAL_TIME_CLOCK_ENABLE=1
```

3. Perform cold reboot to apply the configuration changes:

```
ipmitool power cycle
```

14.4.9.3.1.2 Option 2: DPU Converged Accelerator in NIC mode

To expose and use the GPU and the NIC on the converged accelerator DPU to an application running on the Host x86, configure the DPU to operate in NIC mode.

To do that, follow these steps:

> Valid for both NVIDIA® BlueField®-2 and NVIDIA® BlueField®-3 converged accelerator DPUs.

1. Start MST, check the status, and copy the MST device name:

```
# Enable MST
sudo mst start
sudo mst status

MST devices:
------------
/dev/mst/mt41686_pciconf0         - PCI configuration cycles access.
                                    domain:bus:dev.fn=0000:b8:00.0 addr.reg=88 data.reg=92
cr_bar.gw_offset=-1
                                    Chip revision is: 01
```

2. Expose the GPU on the converged accelerator DPU to the host.
    - For BlueField-2, the `PCI_DOWNSTREAM_PORT_OWNER` offset must be set to 4:

    ```
    sudo mlxconfig -d <mst_device> --yes s PCI_DOWNSTREAM_PORT_OWNER[4]=0x0
    ```

    - For BlueField-3, the `PCI_DOWNSTREAM_PORT_OWNER` offset must be set to 8:

    ```
    sudo mlxconfig -d <mst_device> --yes s PCI_DOWNSTREAM_PORT_OWNER[8]=0x0
    ```

3. Set BlueField to Ethernet mode, enable Accurate Send Scheduling (if required on the send side), and set it to NIC mode:

```
sudo mlxconfig -d <mst_device> --yes set LINK_TYPE_P1=2 LINK_TYPE_P2=2 INTERNAL_CPU_MODEL=1
INTERNAL_CPU_PAGE_SUPPLIER=1 INTERNAL_CPU_ESWITCH_MANAGER=1 INTERNAL_CPU_IB_VPORT0=1
INTERNAL_CPU_OFFLOAD_ENGINE=DISABLED
sudo mlxconfig -d <mst_device> --yes set ACCURATE_TX_SCHEDULER=1 REAL_TIME_CLOCK_ENABLE=1
```

4. Perform cold reboot to apply the configuration changes:

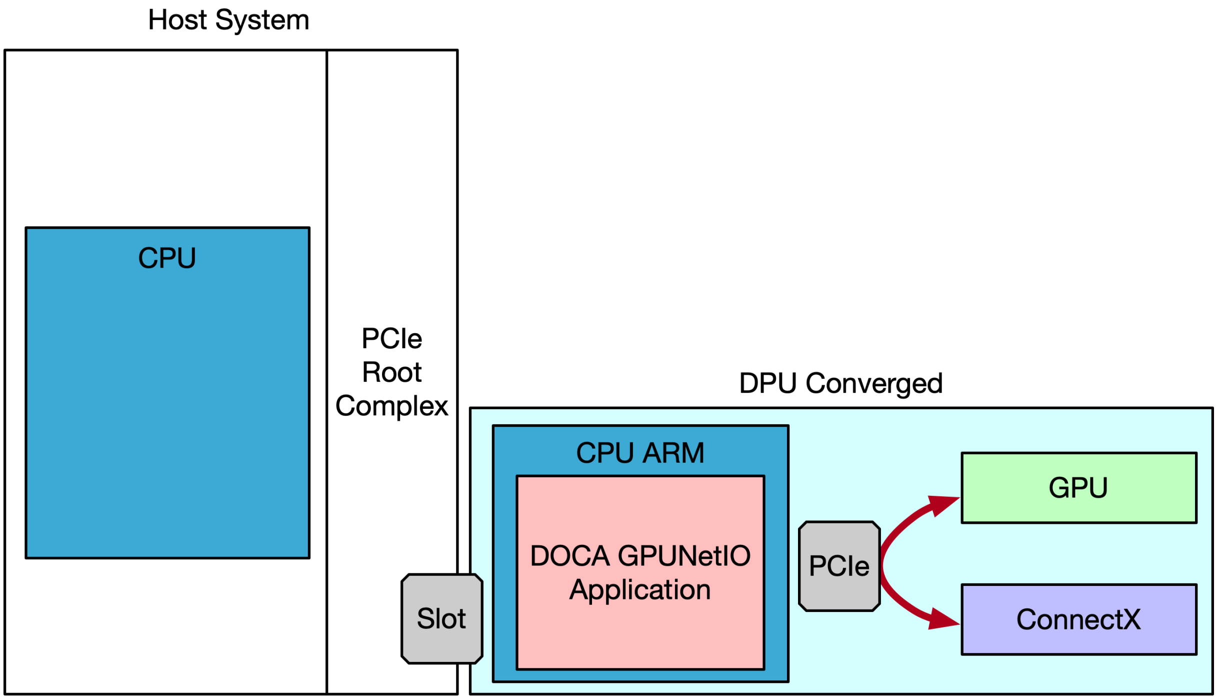
```
ipmitool power cycle
```

5. Verify configuration:

```
sudo mlxconfig -d <mst_device> q LINK_TYPE_P1 LINK_TYPE_P2 INTERNAL_CPU_MODEL INTERNAL_CPU_PAGE_SUPPLIER
INTERNAL_CPU_ESWITCH_MANAGER INTERNAL_CPU_IB_VPORT0 INTERNAL_CPU_OFFLOAD_ENGINE ACCURATE_TX_SCHEDULER
REAL_TIME_CLOCK_ENABLE
         LINK_TYPE_P1                                ETH(2)
         LINK_TYPE_P2                                ETH(2)
         INTERNAL_CPU_MODEL                          EMBEDDED_CPU(1)
         INTERNAL_CPU_PAGE_SUPPLIER                  EXT_HOST_PF(1)
         INTERNAL_CPU_ESWITCH_MANAGER                EXT_HOST_PF(1)
         INTERNAL_CPU_IB_VPORT0                      EXT_HOST_PF(1)
         INTERNAL_CPU_OFFLOAD_ENGINE                 DISABLED(1)
         ACCURATE_TX_SCHEDULER                       True(1)
         REAL_TIME_CLOCK_ENABLE                      True(1)
```

14.4.9.3.2 Application on BlueField Converged Arm CPU

In this scenario, the DOCA GPUNetIO is running on the CPU Arm cores of the BlueField using the GPU and NIC on the same BlueField.



The converged accelerator DPU must be set to CPU mode after flashing the right BFB image (refer to NVIDIA DOCA Installation Guide for Linux for details). From the x86 host, configure the DPU as detailed in the following steps:

> ⓘ Valid for both BlueField-2 and BlueField-3 converged accelerator DPUs.

1. Start MST, check the status, and copy the MST device name:

```
# Enable MST
sudo mst start
sudo mst status

MST devices:
------------
/dev/mst/mt41686_pciconf0         - PCI configuration cycles access.
                                    domain:bus:dev.fn=0000:b8:00.0 addr.reg=88 data.reg=92
cr_bar.gw_offset=-1
                                    Chip revision is: 01
```

2. Set the DPU as the GPU owner.
   a. For BlueField-2 the `PCI_DOWNSTREAM_PORT_OWNER` offset must be set to 4:

```
sudo mlxconfig -d <mst_device> --yes s PCI_DOWNSTREAM_PORT_OWNER[4]=0xF
```

   b. For BlueField-3 the `PCI_DOWNSTREAM_PORT_OWNER` offset must be set to 8:

```
sudo mlxconfig -d <mst_device> --yes s PCI_DOWNSTREAM_PORT_OWNER[8]=0xF
```

3. Set BlueField to Ethernet mode and enable Accurate Send Scheduling (if required on the send side):

```
sudo mlxconfig -d <mst_device> --yes set LINK_TYPE_P1=2 LINK_TYPE_P2=2 INTERNAL_CPU_MODEL=1
INTERNAL_CPU_PAGE_SUPPLIER=0 INTERNAL_CPU_ESWITCH_MANAGER=0 INTERNAL_CPU_IB_VPORT0=0
INTERNAL_CPU_OFFLOAD_ENGINE=ENABLED
sudo mlxconfig -d <mst_device> --yes set ACCURATE_TX_SCHEDULER=1 REAL_TIME_CLOCK_ENABLE=1
```

4. Perform cold reboot to apply the configuration changes:

```
ipmitool power cycle
```

5. Verify configuration:

```
mlxconfig -d <mst_device> q LINK_TYPE_P1 LINK_TYPE_P2 INTERNAL_CPU_MODEL INTERNAL_CPU_PAGE_SUPPLIER
INTERNAL_CPU_ESWITCH_MANAGER INTERNAL_CPU_IB_VPORT0 INTERNAL_CPU_OFFLOAD_ENGINE ACCURATE_TX_SCHEDULER
REAL_TIME_CLOCK_ENABLE
...
Configurations:                                        Next Boot
        LINK_TYPE_P1                                   ETH(2)
        LINK_TYPE_P2                                   ETH(2)
        INTERNAL_CPU_MODEL                             EMBEDDED_CPU(1)
        INTERNAL_CPU_PAGE_SUPPLIER                     ECPF(0)
        INTERNAL_CPU_ESWITCH_MANAGER                   ECPF(0)
        INTERNAL_CPU_IB_VPORT0                         ECPF(0)
        INTERNAL_CPU_OFFLOAD_ENGINE                    ENABLED(0)
        ACCURATE_TX_SCHEDULER                          True(1)
        REAL_TIME_CLOCK_ENABLE                         True(1)
```

At this point, it should be possible to SSH into BlueField to access the OS installed on it. Before installing DOCA GPUNetIO as previously described, CUDA Toolkit (and NVIDIA driver) must be installed.

14.4.9.3.3 PCIe Configuration

On some x86 systems, the Access Control Services (ACS) must be disabled to ensure direct communication between the NIC and GPU, whether they reside on the same converged accelerator DPU or on different PCIe slots in the system. The recommended solution is to disable ACS control via BIOS (e.g., Supermicro or HPE). Alternatively, it is also possible to disable it via command line, but it may not be as effective as the BIOS option. Assuming system topology Option 2, with a converged accelerator DPU as follows:

```
$ lspci -tvvv...+-[0000:b0]-+-00.0  Intel Corporation Device 09a2
 |           +-00.1  Intel Corporation Device 09a4
 |           +-00.2  Intel Corporation Device 09a3
 |           +-00.4  Intel Corporation Device 0998
 |           \-02.0-[b1-b6]----00.0-[b2-b6]--+-00.0-[b3]--+-00.0  Mellanox Technologies MT42822 BlueField-2
integrated ConnectX-6 Dx network controller
 |                                             |            +-00.1  Mellanox Technologies MT42822 BlueField-2
integrated ConnectX-6 Dx network controller
 |                                             |            \-00.2  Mellanox Technologies MT42822 BlueField-2 SoC
Management Interface
 |                                             \-01.0-[b4-b6]----00.0-[b5-b6]----08.0-[b6]----00.0  NVIDIA
Corporation Device 20b8
```

The PCIe switch address to consider is `b2:00.0` (entry point of the DPU). ACSCtl must have all negative values:

PCIe set

```
setpci -s b2:00.0 ECAP_ACS+6.w=0:fc
```

To verify that the setting has been applied correctly:

PCIe check

```
$ sudo lspci -s b2:00.0 -vvvv | grep -i ACSCtl
ACSCtl: SrcValid- TransBlk- ReqRedir- CmpltRedir- UpstreamFwd- EgressCtrl- DirectTrans-
```

Please refer to this page and this page for more information.

If the application still does not report any received packets, try to disable IOMMU. On some systems, it can be done from the BIOS looking for the the `VT-d` or `IOMMU` from the NorthBridge configuration and change that setting to `Disable` and save it. The system may also require adding `intel_iommu=off` or `amd_iommu=off` to the kernel options. That can be done through the grub command line as follows:

IOMMU

```
$ sudo vim /etc/default/grub
# GRUB_CMDLINE_LINUX_DEFAULT="iommu=off intel_iommu=off <more options>"
$ sudo update-grub
$ sudo reboot
```

14.4.9.3.4 Hugepages

A DOCA GPUNetIO application over Ethernet uses typically DOCA Flow to set flow steering rules to the Ethernet receive queues. Flow-based programs require an allocation of huge pages and it can be done temporarily as explained in the DOCA Flow or permanently via grub command line:

IOMMU

```
$ sudo vim /etc/default/grub
# GRUB_CMDLINE_LINUX_DEFAULT="default_hugepagesz=1G hugepagesz=1G hugepages=4 <more options>"
$ sudo update-grub
$ sudo reboot

# After rebooting, check huge pages info
$ grep -i huge /proc/meminfo
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
FileHugePages:         0 kB
HugePages_Total:       4
HugePages_Free:        4
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:    1048576 kB
Hugetlb:         4194304 kB
```

14.4.9.3.5 GPU Configuration

CUDA Toolkit 12.1 or newer must be installed on the host. It is also recommended to enable persistence mode to decrease initial application latency `nvidia-smi -pm 1`.

To allow the CPU to access the GPU memory directly without the need for CUDA API, DPDK and DOCA require the GDRCopy kernel module to be installed on the system:

GPU Configuration

```
# Run nvidia-peermem kernel module
sudo modprobe nvidia-peermem
```

```
# Install GDRCopy
sudo apt install -y check kmod
git clone https://github.com/NVIDIA/gdrcopy.git /opt/mellanox/gdrcopy
cd /opt/mellanox/gdrcopy
make
# Run gdrdrv kernel module
./insmod.sh

# Double check nvidia-peermem and gdrdrv module are running
$ lsmod | egrep gdrdrv
gdrdrv                  24576  0
nvidia               55726080  4 nvidia_uvm,nvidia_peermem,gdrdrv,nvidia_modeset

# Export library path
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/opt/mellanox/gdrcopy/src

# Ensure CUDA library path is in the env var
export PATH="/usr/local/cuda/bin:${PATH}"
export LD_LIBRARY_PATH="/usr/local/cuda/lib:/usr/local/cuda/lib64:${LD_LIBRARY_PATH}"
export CPATH="$(echo /usr/local/cuda/targets/{x86_64,sbsa}-linux/include | sed 's/ /:/'):${CPATH}"
```

14.4.9.3.5.1 BlueField-3 Specific Configuration

To run a DOCA GPUNetIO application on the BlueField Arm cores in a BlueField-3 converged card (section "Application on DPU Converged Arm CPU"), it is mandatory to set an NVIDIA driver option at the end of the driver configuration file:

Set NVIDIA driver option

```
cat <<EOF | sudo tee /etc/modprobe.d/nvidia.conf
options nvidia NVreg_RegistryDwords="RmDmaAdjustPeerMmioBF3=1;"
EOF
```

To make sure the option has been detected by the NVIDIA driver, run:

Check NVIDIA driver option

```
$ grep RegistryDwords /proc/driver/nvidia/params
RegistryDwords: "RmDmaAdjustPeerMmioBF3=1;"
RegistryDwordsPerDevice: ""
```

14.4.9.3.5.2 GPU Memory Mapping (nvidia-peermem vs. dmabuf)

To allow the NIC to send and receive packets using GPU memory, it is required to launch the NVIDIA kernel module `nvidia-peermem` (using `modprobe nvidia-peermem`). It is shipped by default with the CUDA Toolkit installation.

Mapping buffers through the `nvidia-peermem` module is the legacy mapping mode.

Alternatively, DOCA offers the ability to map GPU memory through the `dmabuf` providing a set high-level functions. Prerequisites are DOCA installed on a system with:

- Linux Kernel ≥ 6.2
- libibverbs ≥ 1.14.44
- CUDA Toolkit installed with the `-m=kernel-open` flag (which implies the NVIDIA driver in Open Source mode)

⚠ Installing DOCA on kernel 6.2 to enable the `dmabuf` is experimental.

An example can be found in the DOCA GPU Packet Processing application:

GPU Configuration

```
/* Get from CUDA the dmabuf file-descriptor for the GPU memory buffer */
result = doca_gpu_dmabuf_fd(gpu_dev, gpu_buffer_addr, gpu_buffer_size, &(dmabuf_fd));
if (result != DOCA_SUCCESS) {
    /* If it fails, create a DOCA mmap for the GPU memory buffer with the nvidia-peermem legacy method */
    doca_mmap_set_memrange(gpu_buffer_mmap, gpu_buffer_addr, gpu_buffer_size);
} else {
    /* If it succeeds, create a DOCA mmap for the GPU memory buffer using the dmabuf method */
    doca_mmap_set_dmabuf_memrange(gpu_buffer_mmap, dmabuf_fd, gpu_buffer_addr, 0, gpu_buffer_size);
}
```

If the function `doca_gpu_dmabuf_fd` fails, it probably means the NVIDIA driver is not installed with the open-source mode.

Later, when calling the `doca_mmap_start`, the DOCA library tries to map the GPU memory buffer using the `dmabuf` file descriptor. If it fails (something incorrectly set on the Linux system), it fallbacks trying to map the GPU buffer with the legacy mode (`nvidia-peermem`). If it fails, an informative error is returned.

14.4.9.3.5.3 GPU BAR1 Size

Every time a GPU buffer is mapped to the NIC (e.g., buffers associated with send or receive queues), a portion of the GPU BAR1 mapping space is used. Therefore, it is important to check that the BAR1 mapping is large enough to hold all the bytes the DOCA GPUNetIO application is trying to map. To verify the BAR1 mapping space of a GPU you can use `nvidia-smi` :

BAR1 mapping

```
$ nvidia-smi -q

==============NVSMI LOG==============
.....
Attached GPUs                             : 1
GPU 00000000:CA:00.0
    Product Name                          : NVIDIA A100 80GB PCIe
    Product Architecture                  : Ampere
    Persistence Mode                      : Enabled
.....
    BAR1 Memory Usage
        Total                             : 131072 MiB
        Used                              : 1 MiB
        Free                              : 131071 MiB
```

By default, some GPUs (e.g. RTX models) may have a very small BAR1 size:

BAR1 mapping

```
$ nvidia-smi -q | grep -i bar -A 3
    BAR1 Memory Usage
    Total : 256 MiB
    Used : 6 MiB
    Free : 250 MiB
```

If the BAR1 size is not enough, DOCA GPUNetIO applications may exit with errors because DOCA mmap fails to map the GPU memory buffers to the NIC (e.g., `Failed to start mmap DOCA Driver call failure`). To overcome this issue, the GPU BAR1 must be increased from the BIOS. The system should have "Resizable BAR" option enabled. For further information, refer to this NVIDIA forum post.

14.4.9.4 Architecture

A GPU packet processing network application can be split into two fundamental phases:

- Setup on the CPU (devices configuration, memory allocation, launch of CUDA kernels, etc.)
- Main data path where GPU and NIC interact to exercise their functions

DOCA GPUNetIO provides different building blocks, some of them in combination with the DOCA Ethernet or DOCA RDMA library, to create a full pipeline running entirely on the GPU.

During the setup phase on the CPU, applications must:

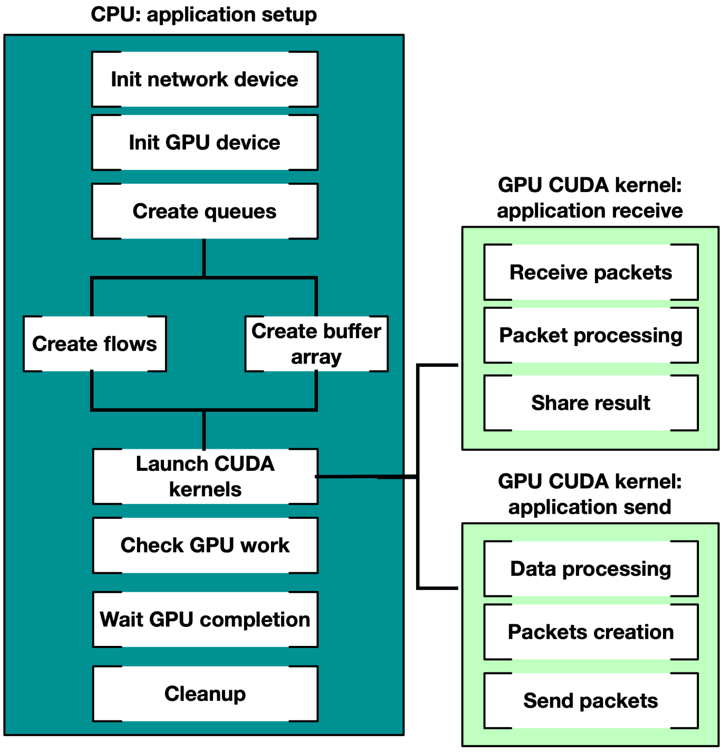
1. Prepare all the objects on the CPU.
2. Export a GPU handler for them.
3. Launch a CUDA kernel passing the object's GPU handler to work with the object during the data path.

For this reason, DOCA GPUNetIO is composed of two libraries:

- `libdoca_gpunetio` with functions invoked by CPU to prepare the GPU, allocate memory and objects
- `libdoca_gpunetio_device` with functions invoked by GPU within CUDA kernels during the data path

⚠ The pkgconfig file for the DOCA GPUNetIO shared library is `doca-gpunetio.pc`. However, there is no pkgconfig file for the DOCA GPUNetIO CUDA device's static library `/opt/mellanox/doca/lib/x86_64-linux-gnu/libdoca_gpunetio_device.a`, so it must be explicitly linked to the CUDA application if DOCA GPUNetIO CUDA device functions are required.

The following diagram presents the typical flow:



Refer to the NVIDIA DOCA GPU Packet Processing Application Guide for an example of using DOCA GPUNetIO to send and receive Ethernet packets.

14.4.9.5 API

This section details the specific structures and operations related to the main DOCA GPUNetIO API on CPU and GPU. GPUNetIO headers are:

- `doca_gpunetio.h` - CPU functions
- `doca_gpunetio_dev_buf.cuh` - GPU functions to manage a DOCA buffer array
- `doca_gpunetio_dev_eth_rxq.cuh` - GPU functions to manage a DOCA Ethernet receive queue
- `doca_gpunetio_dev_eth_txq.cuh` - GPU functions to manage a DOCA Ethernet send queue
- `doca_gpunetio_dev_sem.cuh` - GPU functions to manage a DOCA GPUNetIO semaphore
- `doca_gpunetio_dev_rdma.cuh` - GPU functions to manage a DOCA RDMA queue
- `doca_gpunetio_dev_dma.cuh` - GPU functions to manage a DOCA DMA queue

This section lists the main functions of DOCA GPUNetIO. To better understand their usage, refer to section "Building Blocks" which includes several code examples.

✓ To better understand structures, objects, and functions related to Ethernet send and receive, please refer to the DOCA Ethernet.

To better understand structures, objects, and functions related to RDMA operations, please refer to the DOCA RDMA.

To better understand structures, objects, and functions related to DMA operations, please refer to the DOCA DMA.

To better understand DOCA core objects like `doca_mmap` or `doca_buf_array`, please refer to the DOCA Core.

All DOCA Core and Ethernet object used with GPUNetIO have a GPU export function to obtain a GPU handler for that object. The following are a few examples:

- `doca_buf_array` is exported as `doca_gpu_buf_arr`:

  DOCA buf array

  ```
  struct doca_mmap *mmap;
  struct doca_buf_arr *buf_arr_cpu;
  struct doca_gpu_buf_arr *buf_arr_gpu;

  doca_mmap_create(&(mmap));
  /* Populate and start mmap */
  doca_buf_arr_create(mmap, &buf_arr_cpu);
  /* Populate and start buf arr attributes. Set datapath on GPU */
  /* Export the buf array CPU handler to a buf array GPU handler */
  doca_buf_arr_get_gpu_handle(buf_arr_cpu, &(buf_arr_gpu));
  /* To use the GPU handler, pass it as parameter of the CUDA kernel */
  cuda_kernel<<<...>>>(buf_arr_gpu, ...);
  ```

- `doca_eth_rxq` is exported as `doca_gpu_eth_rxq`:

  DOCA buf array

  ```
  struct doca_mmap *mmap;
  struct doca_eth_rxq *eth_rxq_cpu;
  struct doca_gpu_eth_rxq *eth_rxq_gpu;
  struct doca_dev *ddev;

  /* Create DOCA network device ddev */
  /* Create the DOCA Ethernet receive queue */
  doca_eth_rxq_create(ddev, MAX_NUM_PACKETS, MAX_PACKET_SIZE, &eth_rxq_cpu,);
  /* Populate and start Ethernet receive queue attributes. Set datapath on GPU */
  /* Export the Ethernet receive queue CPU handler to a Ethernet receive queue GPU handler */
  doca_eth_rxq_get_gpu_handle(eth_rxq_cpu, &(eth_rxq_gpu));
  /* To use the GPU handler, pass it as parameter of the CUDA kernel */
  cuda_kernel<<<...>>>(eth_rxq_gpu, ...);
  ```

14.4.9.5.1 CPU Functions

In this section there is the list of DOCA GPUNetIO functions that can be used on the CPU only.

14.4.9.5.1.1 doca_gpu_mem_type

This enum lists all the possible memory types that can be allocated with GPUNetIO.

```
enum doca_gpu_mem_type {
    DOCA_GPU_MEM_TYPE_GPU        = 0,
    DOCA_GPU_MEM_TYPE_GPU_CPU    = 1,
    DOCA_GPU_MEM_TYPE_CPU_GPU    = 2,
};
```

> With regards to the syntax, the text string after the `DOCA_GPU_MEM_TYPE_` prefix signifies `<where-memory-resides>_<who-has-access>`.

- `DOCA_GPU_MEM_TYPE_GPU` - memory resides on the GPU and is accessible from the GPU only
- `DOCA_GPU_MEM_TYPE_GPU_CPU` - memory resides on the GPU and is accessible also by the CPU
- `DOCA_GPU_MEM_TYPE_CPU_GPU` - memory resides on the CPU and is accessible also by the GPU

Typical usage of the `DOCA_GPU_MEM_TYPE_GPU_CPU` memory type is to send a notification from the CPU to the GPU (e.g., a CUDA kernel periodically checking to see if the exit condition set by the CPU is met).

14.4.9.5.1.2 doca_gpu_create

This is the first function a GPUNetIO application must invoke to create an handler on a GPU device. The function initializes a pointer to a structure in memory with type `struct doca_gpu *`.

```
doca_error_t doca_gpu_create(const char *gpu_bus_id, struct doca_gpu **gpu_dev);
```

- `gpu_bus_id` - `<PCIe-bus>:<device>.<function>` of the GPU device you want to use in your application
- `gpu_dev [out]` - GPUNetIO handler to that GPU device

To get the PCIe address, users can use the commands `lspci` or `nvidia-smi`.

14.4.9.5.1.3 doca_gpu_mem_alloc

This CPU function allocates different flavors of memory.

```
doca_error_t doca_gpu_mem_alloc(struct doca_gpu *gpu_dev, size_t size, size_t alignment, enum doca_gpu_mem_type
mtype, void **memptr_gpu, void **memptr_cpu)
```

- `gpu_dev` - GPUNetIO device handler
- `size` - Size, in bytes, of the memory area to allocate
- `alignment` - Memory address alignment to use. If 0, default one will be used
- `mtype` - Type of memory to allocate
- `memptr_gpu [out]` - GPU pointer to use to modify that memory from the GPU if memory is allocated on or is visible by the GPU
- `memptr_cpu [out]` - CPU pointer to use to modify that memory from the CPU if memory is allocated on or is visible by the CPU. Can be NULL if memory is GPU-only

Make sure to use the right pointer on the right device! If an application tries to access the memory using the `memptr_gpu` address from the CPU, a segmentation fault will result.

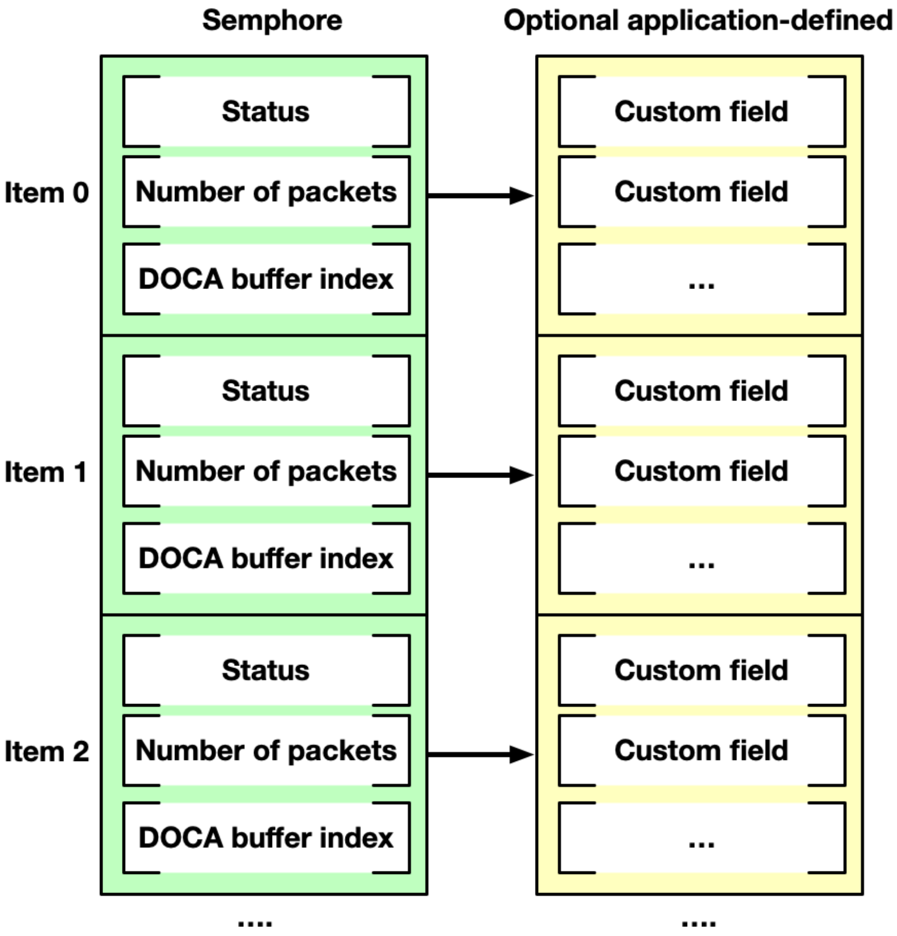
14.4.9.5.1.4 doca_gpu_semaphore_create

Creates a new instance of a DOCA GPUNetIO semaphore. A semaphore is composed by a list of items each having, by default, a status flag, number of packets, and the index of a `doca_gpu_buf` in a `doca_gpu_buf_arr`.

For example, a GPUNetIO semaphore can be used in applications where a CUDA kernel is responsible for receiving packets in a `doca_gpu_buf_arr` array associated with an Ethernet receive queue object, `doca_gpu_eth_rxq` (see section "doca_gpu_dev_eth_rxq_receive_*"), and dispatching packet info to a second CUDA kernel which processes them.

Another way to use a GPUNetIO semaphore is to exchange data across different entities like two CUDA kernels or a CUDA kernel and a CPU thread. The reason for this scenario may be that the CUDA kernel needs to provide the outcome of the packet processing to the CPU which would in turn compile a statistics report. Therefore, it is possible to associate a custom application-defined structure to each item in the semaphore. This way, the semaphore can be used as a message passing object.

Both situations are illustrated in the "Receive and Process" section.



Entities communicating through a semaphore must adopt a poll/update mechanism according to the following logic:

- Update:
    a. Populate the next item of the semaphore (packets' info and/or custom application-defined info).
    b. Set status flag to READY.
- Poll:
    a. Wait for the next item to have a status flag equal to `READY`.
    b. Read and process info.
    c. Set status flag to `DONE`.

```
doca_error_t doca_gpu_semaphore_create(struct doca_gpu *gpu_dev, struct doca_gpu_semaphore **semaphore)
```

- `gpu_dev` - GPUNetIO handler
- `semaphore [out]` - GPUNetIO semaphore handler associated to the GPU device

14.4.9.5.1.5 doca_gpu_semaphore_set_memory_type

This function defines the type of memory for the semaphore allocation.

```
doca_error_t doca_gpu_semaphore_set_memory_type(struct doca_gpu_semaphore *semaphore, enum doca_gpu_mem_type mtype)
```

- `semaphore` - GPUNetIO semaphore handler
- `mtype` - Type of memory to allocate the custom info structure
    - If the application must share packet info only across CUDA kernels, then `DOCA_GPU_MEM_GPU` is the suggested memory type.
    - If the application must share info from a CUDA kernel to a CPU (e.g., to report statistics or output of the pipeline computation), then `DOCA_GPU_MEM_CPU_GPU` is the suggested memory type

14.4.9.5.1.6 doca_gpu_semaphore_set_items_num

This function defines the number of items in a semaphore.

```
doca_error_t doca_gpu_semaphore_set_items_num(struct doca_gpu_semaphore *semaphore, uint32_t num_items)
```

- `semaphore` - GPUNetIO semaphore handler
- `num_items` - Number of items to allocate

14.4.9.5.1.7 doca_gpu_semaphore_set_custom_info

This function associates an application-specific structure to semaphore items as explained under "doca_gpu_semaphore_create".

```
doca_error_t doca_gpu_semaphore_set_custom_info(struct doca_gpu_semaphore *semaphore, uint32_t nbytes, enum
 doca_gpu_mem_type mtype)
```

- `semaphore` - GPUNetIO semaphore handler

- `nbytes` - Size of the custom info structure to associate
- `mtype` - Type of memory to allocate the custom info structure
    - If the application must share packet info only across CUDA kernels, then `DOCA_GPU_MEM_GPU` is the suggested memory type
    - If the application must share info from a CUDA kernel to a CPU (e.g., to report statistics or output of the pipeline computation), then `DOCA_GPU_MEM_CPU_GPU` is the suggested memory type

14.4.9.5.1.8 doca_gpu_semaphore_get_status

From the CPU, query the status of a semaphore item. If the semaphore is allocated with `DOCA_GPU_MEM_GPU`, this function results in a segmentation fault.

```
doca_error_t doca_gpu_semaphore_get_status(struct doca_gpu_semaphore *semaphore_cpu, uint32_t idx, enum
 doca_gpu_semaphore_status *status)
```

- `semaphore_cpu` - GPUNetIO semaphore CPU handler
- `idx` - Semaphore item index
- `status [out]` - Output semaphore status

14.4.9.5.1.9 doca_gpu_semaphore_get_custom_info_addr

From the CPU, retrieve the address of the custom info structure associated to a semaphore item. If the semaphore or the custom info is allocated with `DOCA_GPU_MEM_GPU` this function results in a segmentation fault.

```
doca_error_t doca_gpu_semaphore_get_custom_info_addr(struct doca_gpu_semaphore *semaphore_cpu, uint32_t idx, void
 **custom_info)
```

- `semaphore_cpu` - GPUNetIO semaphore CPU handler
- `idx` - Semaphore item index
- `custom_info [out]` - Output semaphore custom info address

14.4.9.5.2 DOCA PE

A DOCA Ethernet Txq context, exported for GPUNetIO usage, can be tracked via DOCA PE on the CPU side to check if there are errors when sending packets or to retrieve notification info after sending a packet with any of the `doca_gpu_dev_eth_txq_*_enqueue_*` functions on the GPU. An example can be found in the DOCA GPU packet processing application with ICMP traffic.

14.4.9.5.3 Strong Mode vs. Weak Mode

Some Ethernet and RDMA GPU functions present two modes of operation: Weak and strong.

- In weak mode, the application calculates the next available position in the queue. With the help of functions like `doca_gpu_eth_txq_get_info`, `doca_gpu_rdma_get_info`, or `doca_gpu_dev_rdma_recv_get_info` it is possible to know the next available position in the queue and the mask of the number of total entries in the queue (so the incremental descriptor index can be wrapped). In this mode, the developer must specify a queue descriptor number for where to enqueue the packet, ensuring that no descriptor in the queue is left empty. It's a bit more complex to manage but it should result in better performance and developer can emphasize GPU memory coalescing enqueuing sequential operations using sequential memory locations.

- In strong mode, the GPU function enqueues the Ethernet/RDMA operation in the next available position in the queue. It is simpler to manage as developer does not have to worry about operation's position, but it may introduce an extra latency to atomically guarantee the access of multiple threads to the same queue. Moreover, it does not guarantee that sequential operations refer to sequential memory locations.

  > ⚠ All strong mode functions work at the CUDA block level. That is, it is not possible to access the same Eth/RDMA queue at the same time from two different CUDA blocks.

In sections "Produce and Send" and "CUDA Kernel for RDMA Write", there are a few examples about how to use the weak mode API.

14.4.9.5.4 GPU Functions – Ethernet

This section provides a list of DOCA GPUNetIO functions that can be used for Ethernet network operations on the GPU only within a CUDA kernel.

14.4.9.5.4.1 doca_gpu_dev_eth_rxq_receive_*

To acquire packets in a CUDA kernel, DOCA GPUNetIO offers different flavors of the receive function for different scopes: per CUDA block, per CUDA warp, and per CUDA thread.

```
__device__ doca_error_t doca_gpu_dev_eth_rxq_receive_block(struct doca_gpu_eth_rxq *eth_rxq, uint32_t max_rx_pkts,
uint64_t timeout_ns, uint32_t *num_rx_pkts, uint64_t *doca_gpu_buf_idx)
__device__ doca_error_t doca_gpu_dev_eth_rxq_receive_warp(struct doca_gpu_eth_rxq *eth_rxq, uint32_t max_rx_pkts,
uint64_t timeout_ns, uint32_t *num_rx_pkts, uint64_t *doca_gpu_buf_idx)
__device__ doca_error_t doca_gpu_dev_eth_rxq_receive_thread(struct doca_gpu_eth_rxq *eth_rxq, uint32_t max_rx_pkts,
uint64_t timeout_ns, uint32_t *num_rx_pkts, uint64_t *doca_gpu_buf_idx)
```

- `eth_rxq` - Ethernet receive queue GPU handler
- `max_rx_pkts` - Maximum number of packets to receive. It ensures the number of packets returned by the function is lower or equal to this number.
- `timeout_ns` - Nanoseconds to wait for packets before returning
- `num_rx_pkts [out]` - Effective number of received packets. With CUDA block or warp scopes, this variable should be visible in memory by all the other threads (shared or global memory).
- `doca_gpu_buf_idx [out]` - DOCA buffer index of the first packet received in this function. With CUDA block or warp scopes, this variable should be visible in memory by all the other threads (shared or global memory).

> ⚠ If both `max_rx_pkts` and `timeout_ns` are 0, the function never returns.

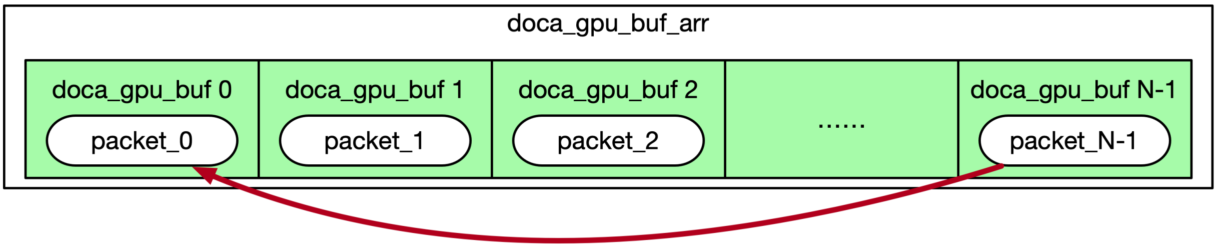
CUDA threads in the same scope (thread, warp, or block) must invoke the function on the same receive queue. The output parameters `num_rx_pkts` and `doca_gpu_buf_idx` must be visible by all threads in the scope (e.g., CUDA shared memory for warp and block).

Each packet received by this function goes to the `doca_gpu_buf_arr` internally created and associated with the Ethernet queues (see section "Building Blocks").

The function exits when `timeout_ns` is reached or when the maximum number of packets is received.

> For CUDA block scope, the block invoking the receive function must have at least 32 CUDA threads (i.e., one warp).

The output parameters indicate how many packets have been received (`num_rx_pkts`) and the index of the first received packet in the `doca_gpu_buf_arr` internally associated with the Ethernet receive queue. Packets are stored consecutively in the `doca_gpu_buf_arr` so if the function returns `num_rx_pkts=N` and `doca_gpu_buf_idx=X` , this means that all the `doca_gpu_buf` in the `doca_gpu_buf_arr` within the range `[X, .. ,X + (N-1)]` have been filled with packets.



The DOCA buffer array is treated in a circular fashion so that once the last DOCA buffer is filled by a packet, the queue circles back to the first DOCA buffer. There is no need for the application to lock or free `doca_gpu_buf_arr` buffers.

> It is the application's responsibility to consume packets before they are overwritten when circling back, properly dimensioning the DOCA buffer array size and scaling across multiple receive queues.

14.4.9.5.4.2 doca_gpu_send_flags

This enum lists all the possible flags for the txq functions. The usage of those flags makes sense if a DOCA PE has been attached to the DOCA Ethernet Txq context with GPU data path and a CPU thread, in a loop, keeps invoking `doca_pe_progress` .

> If no DOCA PE has been attached to the DOCA Ethernet Txq context, it's mandatory to use the `DOCA_GPU_SEND_FLAG_NONE` flag.

```
enum doca_gpu_mem_type {
    DOCA_GPU_SEND_FLAG_NONE         = 0,
    DOCA_GPU_SEND_FLAG_NOTIFY       = 1 << 0,
};
```

- `DOCA_GPU_SEND_FLAG_NONE` (default) - send is executed and no notification info is returned. If an error occurs, an event is generated. This error can be detected from the CPU side using DOCA PE.

- `DOCA_GPU_SEND_FLAG_NOTIFY` - once the send (or wait) is executed, return a notification with packet info. This notification can be detected from the CPU side using DOCA PE.

14.4.9.5.4.3 doca_gpu_dev_eth_txq_send_*

To send packets from a CUDA kernel, DOCA GPUNetIO offers a strong and weak modes for enqueuing a packet in the Ethernet TXQ. For both modes, the scope is the single CUDA thread each populating and enqueuing a different `doca_gpu_buf` from a `doca_gpu_buf_arr` in the send queue.

```
__device__ doca_error_t doca_gpu_dev_eth_txq_get_info(struct doca_gpu_eth_txq *eth_txq, uint32_t *curr_position,
uint32_t *mask_max_position)
```

- `eth_txq` - Ethernet send queue GPU handler
- `curr_position` - Next available position in the queue
- `mask_max_position` - Mask of the total number of positions in the queue

```
__device__ doca_error_t doca_gpu_dev_eth_txq_send_enqueue_strong(struct doca_gpu_eth_txq *eth_txq, const struct
 doca_gpu_buf *buf_ptr, const uint32_t nbytes, const uint32_t flags_bitmask)
```

- `eth_txq` - Ethernet send queue GPU handler
- `buf_ptr` - DOCA buffer from a DOCA GPU buffer array to be sent
- `nbytes` - Number of bytes to be sent in the packet
- `flags_bitmask` - One of the flags in the `doca_gpu_send_flags` enum

```
__device__ doca_error_t doca_gpu_dev_eth_txq_send_enqueue_weak(const struct doca_gpu_eth_txq *eth_txq, const struct
 doca_gpu_buf *buf_ptr, const uint32_t nbytes, const uint32_t ndescr, const uint32_t flags_bitmask)
```

- `eth_txq` - Ethernet send queue GPU handler
- `buf_ptr` - DOCA buffer from a DOCA GPU buffer array to be sent
- `nbytes` - Number of bytes to be sent in the packet
- `ndescr` - Position in the queue to place the packet. Range: 0 - `mask_max_position`.
- `flags_bitmask` - One of the flags in the `doca_gpu_send_flags` enum

14.4.9.5.4.4 doca_gpu_dev_eth_txq_wait_*

To enable Accurate Send Scheduling, the "wait on time" barrier (based on timestamp) must be set in the send queue before enqueuing more packets. Like `doca_gpu_dev_eth_txq_send_*`, `doca_gpu_dev_eth_txq_wait_*` also has a strong and weak mode.

```
__device__ doca_error_t doca_gpu_dev_eth_txq_wait_time_enqueue_strong(struct doca_gpu_eth_txq *eth_txq, const
uint64_t wait_on_time_value, const uint32_t flags_bitmask)
```

- `eth_txq` - Ethernet send queue GPU handler
- `wait_on_time_value` - Timestamp to specify when packets must be sent after this barrier
- `flags_bitmask` - One of the flags in the `doca_gpu_send_flags` enum

```
__device__ doca_error_t doca_gpu_dev_eth_txq_wait_time_enqueue_weak(struct doca_gpu_eth_txq *eth_txq, const
 uint64_t wait_on_time_value, const uint32_t ndescr, const uint32_t flags_bitmask)
```

- `eth_txq` - Ethernet send queue GPU handler

- `wait_on_time_value` - Timestamp to specify when packets must be sent after this barrier
- `ndescr` - Position in the queue to place the packet. Range: 0 - `mask_max_position`.
- `flags_bitmask` - One of the flags in the `doca_gpu_send_flags` enum

Please refer to section "GPUNetIO Samples" to understand how to enable and use Accurate Send Scheduling.

14.4.9.5.4.5 doca_gpu_dev_eth_txq_commit_*

After enqueuing all the packets to be sent and time barriers, a commit function must be invoked on the txq queue. The right commit function must be used according to the type of enqueue mode (i.e., strong or weak) used in `doca_gpu_dev_eth_txq_send_*` and `doca_gpu_dev_eth_txq_wait_*`.

```
__device__ doca_error_t doca_gpu_dev_eth_txq_commit_strong(struct doca_gpu_eth_txq *eth_txq)
```

- `eth_txq` - Ethernet send queue GPU handler

```
__device__ doca_error_t doca_gpu_dev_eth_txq_commit_weak(struct doca_gpu_eth_txq *eth_txq, const uint32_t
descr_num)
```

- `eth_txq` - Ethernet send queue GPU handler
- `descr_num` - Number of queue items enqueued thus far

Only one CUDA thread in the scope (CUDA block or CUDA warp) can invoke this function on the send queue after several enqueue operations. Typical flow is as follows:

1. All threads in the scope enqueue packets in the send queue.
2. Synchronization point.
3. Only one thread in the scope performs the send queue commit.

14.4.9.5.4.6 doca_gpu_dev_eth_txq_push

After committing, the items in the send queue must be actually pushed to the network card.

```
__device__ doca_error_t doca_gpu_dev_eth_txq_push(struct doca_gpu_eth_txq *eth_txq)
```

- `eth_txq` - Ethernet send queue GPU handler

Only one CUDA thread in the scope (CUDA block or CUDA warp) can invoke this function on the send queue after several enqueue or commit operations. Typical flow is as follows:

1. All threads in the scope enqueue packets in the send queue.
2. Synchronization point.
3. Only one thread in the scope does the send queue commit.
4. Only one thread in the scope does the send queue push.

Section "Produce and Send" provides an example where the scope is a block (e.g., each CUDA block operates on a different Ethernet send queue).

14.4.9.5.5 GPU Functions – RDMA

This section provides a list of DOCA GPUNetIO functions that can be used on the GPU only within a CUDA kernel to execute RDMA operations. These functions offer a strong and a weak mode.

```
__device__ doca_error_t __device__ doca_error_t doca_gpu_dev_rdma_get_info(struct doca_gpu_dev_rdma *rdma, uint32_t
*curr_position, uint32_t *mask_max_position)
```

- `rdma` - RDMA queue GPU handler
- `curr_position` - Next available position in the queue
- `mask_max_position` - Mask of the total number of positions in the queue

```
__device__ doca_error_t __device__ doca_error_t doca_gpu_dev_rdma_recv_get_info(struct doca_gpu_dev_rdma_r *rdma_r,
uint32_t *curr_position, uint32_t *mask_max_position)
```

- `rdma_r` - RDMA receive queue GPU handler
- `curr_position` - Next available position in the queue
- `mask_max_position` - Mask of the total number of positions in the queue

14.4.9.5.5.1 doca_gpu_dev_rdma_write_*

To RDMA write data onto a remote memory location from a CUDA kernel, DOCA GPUNetIO offers strong and weak modes for enqueuing operations on the RDMA queue. For both modes, the scope is the single CUDA thread.

```
__device__ doca_error_t doca_gpu_dev_rdma_write_strong(struct doca_gpu_dev_rdma *rdma,
                                                       struct doca_gpu_buf *remote_buf, uint64_t remote_offset,
                                                       struct doca_gpu_buf *local_buf, uint64_t local_offset,
                                                       size_t length, uint32_t imm,
                                                       const enum doca_gpu_dev_rdma_write_flags flags)
```

- `rdma` - RDMA queue GPU handler
- `remote_buf` - Remote DOCA buffer from a DOCA GPU buffer array to write data to
- `remote_offset` - Offset, in bytes, to write data to in the remote buffer
- `local_buf` - Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to write
- `local_offset` - Offset, in bytes, to fetch data from in the local buffer
- `length` - Number of bytes to write
- `imm` - Immediate value `uint32_t`
- `flags` - One of the flags in the `doca_gpu_dev_rdma_write_flags` enum

```
__device__ doca_error_t doca_gpu_dev_rdma_write_weak(struct doca_gpu_dev_rdma *rdma,
                                                     struct doca_gpu_buf *remote_buf, uint64_t remote_offset,
                                                     struct doca_gpu_buf *local_buf, uint64_t local_offset,
                                                     size_t length, uint32_t imm,
                                                     const enum doca_gpu_dev_rdma_write_flags flags,
                                                     uint32_t position);
```

- `rdma` - RDMA queue GPU handler
- `remote_buf` - Remote DOCA buffer from a DOCA GPU buffer array to write data to
- `remote_offset` - Offset, in bytes, to write data to in the remote buffer
- `local_buf` - Local DOCA buffer from a DOCA GPU buffer array where to fetch data to write

- `local_offset` - Offset, in bytes, to fetch data in the local buffer
- `length` - Number of bytes to write
- `imm` - Immediate value `uint32_t`
- `flags` - One of the flags in the `doca_gpu_dev_rdma_write_flags` enum
- `position` - Position in the queue to place the RDMA operation. Range: 0 - `mask_max_position`.

14.4.9.5.5.2 doca_gpu_dev_rdma_read_*

To RDMA read data onto a remote memory location from a CUDA kernel, DOCA GPUNetIO offers strong and weak modes to enqueue operations on the RDMA queue. For both modes, the scope is the single CUDA thread.

```
__device__ doca_error_t doca_gpu_dev_rdma_read_strong(struct doca_gpu_dev_rdma *rdma,
                                                      struct doca_gpu_buf *remote_buf, uint64_t remote_offset,
                                                      struct doca_gpu_buf *local_buf, uint64_t local_offset,
                                                      size_t length,
                                                      const uint32_t flags_bitmask)
```

- `rdma` - RDMA queue GPU handler
- `remote_buf` - Remote DOCA buffer from a DOCA GPU buffer array where to read data
- `remote_offset` - Offset in bytes to read data to in the remote buffer
- `local_buf` - Local DOCA buffer from a DOCA GPU buffer array where to store remote data
- `local_offset` - Offset in bytes to store data in the local buffer
- `length` - Number of bytes to be read
- `flags_bitmask` - Must be 0; reserved for future use

```
__device__ doca_error_t doca_gpu_dev_rdma_read_weak(struct doca_gpu_dev_rdma *rdma,
                                                    struct doca_gpu_buf *remote_buf, uint64_t remote_offset,
                                                    struct doca_gpu_buf *local_buf, uint64_t local_offset,
                                                    size_t length,
                                                    const uint32_t flags_bitmask,
                                                    uint32_t position);
```

- `rdma` - RDMA queue GPU handler
- `remote_buf` - Remote DOCA buffer from a DOCA GPU buffer array where to read data
- `remote_offset` - Offset in bytes to read data to in the remote buffer
- `local_buf` - Local DOCA buffer from a DOCA GPU buffer array where to store remote data
- `local_offset` - Offset in bytes to store data in the local buffer
- `length` - Number of bytes to be read
- `flags_bitmask` - Must be 0; reserved for future use
- `position` - Position in the queue to place the RDMA operation. Range: 0 - `mask_max_position`.

14.4.9.5.5.3 doca_gpu_dev_rdma_send_*

To RDMA send data from a CUDA kernel, DOCA GPUNetIO offers strong and weak modes for enqueuing operations on the RDMA queue. For both modes, the scope is the single CUDA thread.

```
__device__ doca_error_t doca_gpu_dev_rdma_send_strong(struct doca_gpu_dev_rdma *rdma,
                                                      struct doca_gpu_buf *local_buf, uint64_t local_offset,
                                                      size_t length, uint32_t imm,
                                                      const enum doca_gpu_dev_rdma_write_flags flags)
```

- `rdma` - RDMA queue GPU handler
- `local_buf` - Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to send
- `local_offset` - Offset in bytes to fetch data in the local buffer
- `length` - Number of bytes to send
- `imm` - Immediate value `uint32_t`
- `flags` - One of the flags in the `doca_gpu_dev_rdma_write_flags` enum

```
__device__ doca_error_t doca_gpu_dev_rdma_send_weak(struct doca_gpu_dev_rdma *rdma,
                                                    struct doca_gpu_buf *local_buf, uint64_t local_offset,
                                                    size_t length, uint32_t imm,
                                                    const enum doca_gpu_dev_rdma_write_flags flags,
                                                    uint32_t position);
```

- `rdma` - RDMA queue GPU handler
- `local_buf` - Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to send
- `local_offset` - Offset in bytes to fetch data in the local buffer
- `length` - Number of bytes to send
- `imm` - Immediate value `uint32_t`
- `flags` - One of the flags in the `doca_gpu_dev_rdma_write_flags` enum
- `position` - Position in the queue to place the RDMA operation. Range: 0 - `mask_max_position`.

14.4.9.5.5.4 doca_gpu_dev_rdma_commit_*

Once all RDMA write, send or read requests have been enqueue in the RDMA queue, a synchronization point must be reached to consolidate and execute those requests. Only 1 CUDA thread can invoke this function at a time.

```
__device__ doca_error_t doca_gpu_dev_rdma_commit_strong(struct doca_gpu_dev_rdma *rdma)
```

- `rdma` - RDMA queue GPU handler

```
__device__ doca_error_t doca_gpu_dev_rdma_commit_weak(struct doca_gpu_dev_rdma *rdma, uint32_t num_ops)
```

- `rdma` - RDMA queue GPU handler
- `num_ops` - Number of RDMA requests enqueued since the last commit

14.4.9.5.5.5 doca_gpu_dev_rdma_wait_all

After a commit, RDMA requests are executed by the network card as applications move forward doing other operations. If the application needs to verify all RDMA operations have been done by the network card, this "wait all" function can be used to wait for all previous posted operations. Only 1 CUDA thread can invoke this function at a time.

```
__device__ doca_error_t doca_gpu_dev_rdma_wait_all(struct doca_gpu_dev_rdma *rdma, uint32_t *num_ops)
```

- `rdma` - RDMA queue GPU handler
- `num_ops` - Output parameter. Function reports number of completed operations.

This function is optional.

14.4.9.5.5.6 doca_gpu_dev_rdma_recv_*

To receive data from an RDMA send, send with immediate, or write with immediate, the destination peer should post a receive operation. DOCA GPUNetIO RDMA receive operations must be done with a `doca_gpu_dev_rdma_r` handler. This handler can be obtained with the function `doca_gpu_dev_rdma_get_recv` .

All receive operations must use this object.

```
__device__ doca_error_t doca_gpu_dev_rdma_get_recv(struct doca_gpu_dev_rdma *rdma, struct doca_gpu_dev_rdma_r
**rdma_r)
```

- `rdma` - RDMA queue GPU handler
- `rdma_r` - RDMA receive queue GPU handler

Even for the receive side, in this case, DOCA GPUNetIO offers strong and weak modes for enqueuing operations on the RDMA queue. For both modes, the scope is the single CUDA thread.

```
__device__ doca_error_t doca_gpu_dev_rdma_recv_strong(struct doca_gpu_dev_rdma_r *rdma_r,
                                                      struct doca_gpu_buf *recv_buf,
                                                      size_t recv_length,
                                                      uint64_t recv_offset,
                                                      const uint32_t flags_bitmask)
```

- `rdma_r` - RDMA receive queue GPU handler
- `recv_buf` - Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to send
- `recv_length` - Number of bytes to send
- `recv_offset` - Offset in bytes to fetch data in the local buffer
- `flags_bitmask` - Must be 0; reserved for future use

```
__device__ doca_error_t doca_gpu_dev_rdma_recv_weak(struct doca_gpu_dev_rdma_r *rdma_r,
                                                    struct doca_gpu_buf *recv_buf,
                                                    size_t recv_length,
                                                    uint64_t recv_offset,
                                                    const uint32_t flags_bitmask,
                                                    uint32_t position);
```

- `rdma_r` - RDMA receive queue GPU handler
- `recv_buf` - Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to send
- `recv_length` - Number of bytes to send

- `recv_offset` - Offset in bytes to fetch data in the local buffer
- `flags_bitmask` - Must be 0; reserved for future use
- `position` - Position in the queue to place the RDMA operation. Range: 0 - `mask_max_position`.

14.4.9.5.5.7 doca_gpu_dev_rdma_recv_commit_*

After posting a number of RDMA receive, a commit function must be invoked to activate the receive in the queue. Only 1 CUDA thread can invoke this function at a time.

```
__device__ doca_error_t doca_gpu_dev_rdma_recv_commit_strong(struct doca_gpu_dev_rdma_r *rdma_r)
```

- `rdma_r` - RDMA receive queue GPU handler

```
__device__ doca_error_t doca_gpu_dev_rdma_recv_commit_weak(struct doca_gpu_dev_rdma_r *rdma_r, uint32_t num_ops)
```

- `rdma_r` - RDMA receive queue GPU handler
- `num_ops` - Number of RDMA receive requests enqueued since the last commit

14.4.9.5.5.8 doca_gpu_dev_rdma_recv_wait_all

This function waits for the completion of all previously posted RDMA receive operation. Only 1 CUDA thread can invoke this function at a time. It works in blocking or non-blocking mode.

```
 enum doca_gpu_dev_rdma_recv_wait_flags {
    DOCA_GPU_RDMA_RECV_WAIT_FLAG_NB = 0,  /**< Non-Blocking mode: the wait receive function
doca_gpu_dev_rdma_recv_wait
                                            *  checks if the receive operation happened (data has been received)
                                            *  and exit from the function. If nothing has been received,
                                            *  the function doesn't block the execution.
                                            */
    DOCA_GPU_RDMA_RECV_WAIT_FLAG_B  = 1, /**< Blocking mode: the wait receive function doca_gpu_dev_rdma_recv_wait
                                           *  blocks the execution waiting for the receive operations to be
executed.
                                           */
};
```

Function:

```
__device__ doca_error_t doca_gpu_dev_rdma_recv_wait_all(struct doca_gpu_dev_rdma_r *rdma_r, const enum
 doca_gpu_dev_rdma_recv_wait_flags flags, uint32_t *num_ops, uint32_t *imm_val)
```

- `rdma_r` - RDMA receive queue GPU handler
- `flags` - receive flags
- `num_ops` - Output parameter. Function reports number of completed operations.
- `imm_val` - Output parameter. Application-provided buffer where the function can store received immediate values, if any (or 0xFFFFFFFF if no immediate value is received). If `nullptr`, the function ignores this parameter.

14.4.9.5.6 GPU Functions – DMA

This section provides a list of DOCA GPUNetIO functions that can be used on the GPU only within a CUDA kernel to execute DMA operations.

14.4.9.5.6.1 doca_gpu_dev_dma_memcopy

This function allows a CUDA kernel to trigger a DMA memory copy operation through the DMA GPU engine. There is no strong/weak mode here, the DMA is assuming the strong behavior by default.

```
__device__ doca_error_t doca_gpu_dev_dma_memcpy(struct doca_gpu_dma *dma, struct doca_gpu_buf *src_buf, uint64_t
src_offset, struct doca_gpu_buf *dst_buf, uint64_t dst_offset, size_t length);
```

- `dma` - DMA queue GPU handler
- `src_buf` - memcpy source buffer
- `src_offset` - fetch data starting from this source buffer offset
- `dst_buf` - memcpy destination buffer
- `dst_offset` - copy data starting from this destination buffer offset
- `lenght` - number of bytes to copy

14.4.9.5.6.2 doca_gpu_dev_dma_commit

After posting several DMA memory copies, a commit function must be invoked to execute the operations enqueued in the DMA queue. Only 1 CUDA thread can invoke this function at a time.

```
__device__ doca_error_t doca_gpu_dev_dma_commit(struct doca_gpu_dma *dma);
```

- `dma` - DMA queue GPU handler

14.4.9.6 Building Blocks

This section explains general concepts behind the fundamental building blocks to use when creating a DOCA GPUNetIO application.

14.4.9.6.1 Initialize GPU and NIC

When DOCA GPUNetIO is used in combination with the NIC to send or receive Ethernet traffic, the following must be performed to properly set up the application and devices:

GPUNetIO setup

```
uint16_t dpdk_port_id;
struct doca_dev *ddev;
struct doca_gpu *gdev;
char *eal_param[3] = {"", "-a", "00:00.0"};

/* Initialize DPDK with empty device. DOCA device will hot-plug the network card later. */
rte_eal_init(3, eal_param);
/* Create DOCA device on a specific network card */
doca_dpdk_port_probe(&ddev);
get_dpdk_port_id_doca_dev(&ddev, &dpdk_port_id);
/* Create GPUNetIO handler on a specific GPU */
doca_gpu_create(gpu_pcie_address, &gdev);
```

The application would may have to enable different items depending on the task at hand.

14.4.9.6.2 Semaphore

If the DOCA application must dispatch some packets' info across CUDA kernels or from the CUDA kernel and some CPU thread, a semaphore must be created.

A semaphore is a list of items, allocated either on the GPU or CPU (depending on the use case) visible by both the GPU and CPU. This object can be used to discipline communication across items in the GPU pipeline between CUDA kernels or a CUDA kernel and a CPU thread.

By default, each semaphore item can hold info about its status (`FREE` , `READY` , `HOLD` , `DONE` , `ERROR`), the number of received packets, and an index of a `doca_gpu_buf` in a `doca_gpu_buf_arr` .

If the semaphore must be used to exchange data with the CPU, a preferred memory layout would be `DOCA_GPU_MEM_CPU_GPU` . Whereas, if the semaphore is only needed across CUDA kernels, `DOCA_GPU_MEM_GPU` is the best memory layout to use.

As an optional feature, if the application must pass more application-specific info through the semaphore items, it is possible to attach a custom structure to each item of the semaphore.

Semaphore

```
#define SEMAPHORE_ITEMS 1024

/* Application defined custom structure to pass info through semaphore items */
struct custom_info {
    int a;
    uint64_t b;
};

/* Semaphore to share info from the GPU to the CPU */
struct doca_gpu_semaphore *sem_to_cpu;
struct doca_gpu_semaphore_gpu *sem_to_cpu_gpu;

doca_gpu_semaphore_create(gdev, &sem_to_cpu);
doca_gpu_semaphore_set_memory_type(sem_to_cpu, DOCA_GPU_MEM_CPU_GPU);
doca_gpu_semaphore_set_items_num(sem_to_cpu, SEMAPHORE_ITEMS);
/* This is optional */
doca_gpu_semaphore_set_custom_info(sem_to_cpu, sizeof(struct custom_info), DOCA_GPU_MEM_CPU_GPU);
doca_gpu_semaphore_start(sem_to_cpu);
doca_gpu_semaphore_get_gpu_handle(sem_to_cpu, &sem_to_cpu_gpu);

/* Semaphore to share info across GPU CUDA kernels with no CPU involvment */
struct doca_gpu_semaphore *sem_to_gpu;
struct doca_gpu_semaphore_gpu *sem_to_gpu_gpu;

doca_gpu_semaphore_create(gdev, &sem_to_gpu);
doca_gpu_semaphore_set_memory_type(sem_to_gpu, DOCA_GPU_MEM_GPU);
doca_gpu_semaphore_set_items_num(sem_to_gpu, SEMAPHORE_ITEMS);
/* This is optional */
doca_gpu_semaphore_set_custom_info(sem_to_gpu, sizeof(struct custom_info), DOCA_GPU_MEM_GPU);
doca_gpu_semaphore_start(sem_to_gpu);
doca_gpu_semaphore_get_gpu_handle(sem_to_gpu, &sem_to_gpu_gpu);
```

14.4.9.6.3 Ethernet Queue with GPU Data Path

14.4.9.6.3.1 Receive Queue

If the DOCA application must receive Ethernet packets, receive queues must be created. The receive queue works in a circular way: At creation time, each receive queue is associated with a DOCA buffer array allocated on the GPU by the application. Each DOCA buffer of the buffer array has a maximum fixed size.

GPUNetIO receive

```
/* Start DPDK device */
rte_eth_dev_start(dpdk_port_id);
/* Initialise DOCA Flow */
struct doca_flow_port_cfg port_cfg;
port_cfg.port_id = port_id;
doca_flow_init(port_cfg);
doca_flow_port_start();

struct doca_dev *ddev;
struct doca_eth_rxq *eth_rxq_cpu;
struct doca_gpu_eth_rxq *eth_rxq_gpu;
struct doca_mmap *mmap;
void *gpu_buffer;

/* Create DOCA Ethernet receive queues */
doca_eth_rxq_create(ddev, MAX_PACKETS_NUM, MAX_PACKETS_SIZE, &eth_rxq_cpu);

/* Set Ethernet receive queue properties */
/* ... */

/* Create DOCA mmap in GPU memory to be used for the DOCA buffer array associated to this Ethernet queue */
doca_mmap_create(&mmap);
/* Set DOCA mmap properties */
doca_gpu_mem_alloc(gdev, buffer_size, alignment, DOCA_GPU_MEM_GPU, (void **)&gpu_buffer, NULL);
doca_mmap_start(mmap);
doca_eth_rxq_set_pkt_buffer(eth_rxq_cpu, mmap, 0, buffer_size);
/* This DOCA Ethernet Rxq object will be managed by the GPU */
doca_ctx_set_datapath_on_gpu();
/* Start the Ethernet queue object */
/* Export GPU handle for the receive queue */
doca_eth_rxq_get_gpu_handle(eth_rxq_cpu, &eth_rxq_gpu);
```

It is mandatory to associate DOCA Flow pipe(s) to the receive queues. Otherwise, the application cannot receive any packet.

14.4.9.6.3.2 Send Queue

If the DOCA application must send Ethernet packets, send queues must be created in combination with `doca_gpu_buf_arr` to prepare and send packets from GPU memory.

GPUNetIO receive

```
struct doca_dev *ddev;
struct doca_eth_txq *eth_txq_cpu;
struct doca_gpu_eth_txq *eth_txq_gpu;

/* Create DOCA Ethernet send queues */
doca_eth_txq_create(ddev, QUEUE_DEPTH, &eth_txq_cpu);
/* Set properties to send queues */

/* This DOCA Ethernet Rxq object will be managed by the GPU */
doca_ctx_set_datapath_on_gpu();
/* Start the Ethernet queue object */
/* Export GPU handle for the send queue */
doca_eth_txq_get_gpu_handle(eth_txq_cpu, &eth_txq_gpu);

/* Create DOCA mmap to define memory layout and type for the DOCA buf array */
struct doca_mmap *mmap;
doca_mmap_create(&mmap);
/* Set DOCA mmap properties */

/* Create DOCA buf arr and export it to GPU */
struct doca_buf_arr *buf_arr;
struct doca_gpu_buf_arr *buf_arr_gpu;
doca_buf_arr_create(mmap, &buf_arr);
/* Set DOCA buf array properties */
...
/* Export GPU handle for the buf arr */
doca_buf_arr_get_gpu_handle(buf_arr, &buf_arr_gpu);
```

14.4.9.6.3.3 Receive and Process

At this point, the application has created and initialized all the objects required by the GPU to exercise the data path to send or receive packets with GPUNetIO.

In this example, the application must receive packets from different queues with a receiver CUDA kernel and dispatch packet info to a second CUDA kernel responsible for packet processing.

The CPU launches the CUDA kernels and waits on the semaphore for output:

CPU code

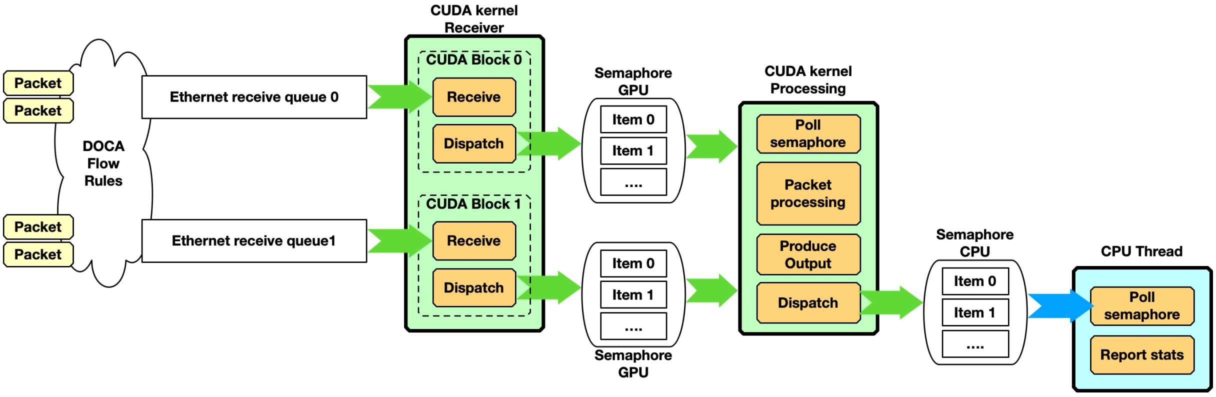
```
#define CUDA_THREADS 512
#define CUDA_BLOCKS 1
int semaphore_index = 0;
enum doca_gpu_semaphore_status status;
struct custom_info *gpu_info;

/* On the CPU */
cuda_kernel_receive_dispatch<<<CUDA_THREADS, CUDA_BLOCKS, ..., stream_0>>>(eth_rxq_gpu, sem_to_gpu_gpu)
cuda_kernel_process<<<CUDA_THREADS, CUDA_BLOCKS, ..., stream_1>>>(eth_rxq_gpu, sem_to_cpu_gpu, sem_to_gpu_gpu)

while(/* condition */) {
    doca_gpu_semaphore_get_status(sem_to_cpu, semaphore_index, &status);
    if (status == DOCA_GPU_SEMAPHORE_STATUS_READY) {
        doca_gpu_semaphore_get_custom_info_addr(sem_to_cpu, semaphore_index, (void **)&(gpu_info));
        report_info(gpu_info);
        doca_gpu_semaphore_set_status(sem_to_cpu, semaphore_index, DOCA_GPU_SEMAPHORE_STATUS_FREE);
        semaphore_index = (semaphore_index+1) % SEMAPHORE_ITEMS;
    }
}
```

On the GPU, the two CUDA kernels are running on different streams:

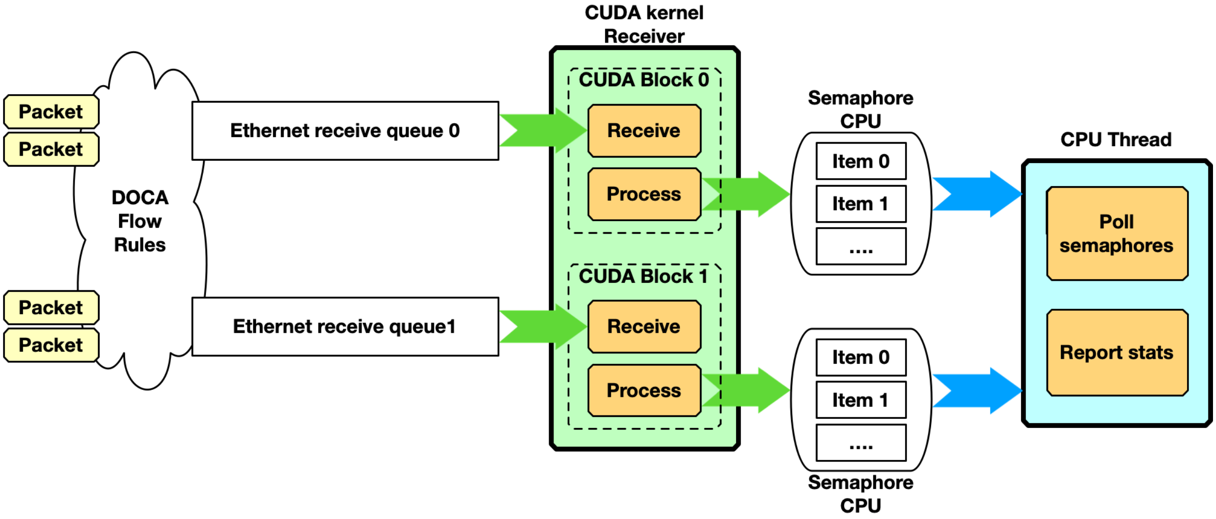
GPU code

```
cuda_kernel_receive_dispatch(eth_rxq_gpu, sem_to_gpu_gpu) {
    __shared__ uint32_t rx_pkt_num;
    __shared__ uint64_t rx_buf_idx;
    int semaphore_index = 0;

    while (/* exit condition */) {
        doca_gpu_dev_eth_rxq_receive_block(eth_rxq_gpu, MAX_NUM_RECEIVE_PACKETS, TIMEOUT_RECEIVE_NS, &rx_pkt_num,
&rx_buf_idx);
        if (threadIdx.x == 0 && rx_pkt_num > 0) {
            doca_gpu_dev_sem_set_packet_info(sem_to_gpu_gpu, semaphore_index, DOCA_GPU_SEMAPHORE_STATUS_READY,
rx_pkt_num, rx_buf_idx);
            semaphore_index = (semaphore_index+1) % SEMAPHORE_ITEMS;
        }
    }
}

cuda_kernel_process(eth_rxq_gpu, sem_to_cpu_gpu, sem_to_gpu_gpu) {
    __shared__ uint32_t rx_pkt_num;
    __shared__ uint64_t rx_buf_idx;
    int semaphore_index = 0;
    int thread_buf_idx = 0;
    struct doca_gpu_buf *buf_ptr;
    uintptr_t buf_addr;
    struct custom_info *gpu_info;

    while (/* exit condition */) {
        if (threadIdx.x == 0) {
            do {
                result = doca_gpu_dev_sem_get_packet_info_status(sem_to_gpu_gpu, semaphore_index,
DOCA_GPU_SEMAPHORE_STATUS_READY, &rx_pkt_num, &rx_buf_idx);
            } while(result != DOCA_ERROR_NOT_FOUND /* && other exit condition */);
        }
        __syncthreads();

        thread_buf_idx = threadIdx.x;
        while (thread_buf_idx < rx_pkt_num) {
            /* Get DOCA GPU buffer from the GPU buffer in the receive queue */
            doca_gpu_dev_eth_rxq_get_buf(eth_rxq_gpu, rx_buf_idx + thread_buf_idx, &buf_ptr);
            /* Get DOCA GPU buffer memory address */
            doca_gpu_dev_buf_get_addr(buf_ptr, &buf_addr);
            /*
             * Atomic here is has the entire CUDA block accesses the same semaphore to CPU.
             * Smarter implementation can be done at warp level, with multiple semaphores, etc.. to avoid this
atomic
             */
            int semaphore_index_tmp = atomicAdd_block(&semaphore_index, 1);
            semaphore_index_tmp = semaphore_index_tmp % SEMAPHORE_ITEMS;
            doca_gpu_dev_sem_get_custom_info_addr(sem_to_cpu_gpu, semaphore_index_tmp, (void **)&gpu_info);
            populate_custom_info(buf_addr, gpu_info);
            doca_gpu_dev_sem_set_status(sem_to_cpu_gpu, semaphore_index_tmp, DOCA_GPU_SEMAPHORE_STATUS_READY);
        }
        __syncthreads();

        if (threadIdx.x == 0) {
            doca_gpu_dev_sem_set_status(sem_to_gpu_gpu, semaphore_index, DOCA_GPU_SEMAPHORE_STATUS_READY);
        }
    }
}
```

This code can be represented with the following diagram when multiple queues and/or semaphores are used:



Please note that receiving and dispatching packets to another CUDA kernel is not required. A simpler scenario can have a single CUDA kernel receiving and processing packets:



The drawback of this approach is that the time between two receives depends on the time taken by the CUDA kernel to process received packets.

The type of pipeline that must be built heavily depends on the specific use case.

14.4.9.6.3.4 Produce and Send

In this example, the GPU produces some data, stores it into packets and then sends them over the network. The CPU launches the CUDA kernels and continues doing other work:

CPU code

```
#define CUDA_THREADS 512
#define CUDA_BLOCKS 1
int semaphore_index = 0;
enum doca_gpu_semaphore_status status;
struct custom_info *gpu_info;
```

```
/* On the CPU */
cuda_kernel_produce_send<<<CUDA_THREADS, CUDA_BLOCKS, ..., stream_0>>>(eth_txq_gpu, buf_arr_gpu)

/* do other stuff */
```

On the GPU, the CUDA kernel fills the packets with meaningful data and sends them. In the following example, the scope is CUDA block so each block uses a different DOCA Ethernet send queue:

GPU code

```
cuda_kernel_produce_send(eth_txq_gpu, buf_arr_gpu) {
    uint64_t doca_gpu_buf_idx = threadIdx.x;
    struct doca_gpu_buf *buf;
    uintptr_t buf_addr;
    uint32_t packet_len;
    uint32_t curr_position;
    uint32_t mask_max_position;
    uint32_t num_pkts_per_send = blockDim.x;

    /* Get last occupied position in the Tx queue */
    doca_gpu_dev_eth_txq_get_info(eth_txq_gpu, &curr_position, &mask_max_position);
    __syncthreads();

    while (/* exit condition */) {
        /* Each CUDA thread retrieves doca_gpu_buf from doca_gpu_buf_arr */
        doca_gpu_dev_buf_get_buf(buf_arr_gpu, doca_gpu_buf_idx, &buf);
        /* Get memory address of the packet in the doca_gpu_buf */
        doca_gpu_dev_buf_get_addr(buf, &buf_addr);

        /* Application produces data and crafts the packet in the doca_gpu_buf */
        populate_packet(buf_addr, &packet_len);

        /* Enqueue packet in the send queue with weak mode: each thread posts the packet in a different and
sequential position of the queue */
        doca_gpu_dev_eth_txq_send_enqueue_weak(eth_txq_gpu, buf, packet_len, ((curr_position + doca_gpu_buf_idx) &
mask_max_position), DOCA_GPU_SEND_FLAG_NONE);

        /* Synchronization point */
        __synchthreads();

        /* Only one CUDA thread in the block must commit and push the send queue */
        if (threadIdx.x == 0) {
            doca_gpu_dev_eth_txq_commit_weak(eth_txq_gpu, num_pkts_per_send);
            doca_gpu_dev_eth_txq_push(eth_txq_gpu);
        }
        /* Synchronization point */
        __synchthreads();

        /* Assume all threads in the block pushed a packet in the send queue */
        doca_gpu_buf_idx += blockDim.x;
    }
}
```

14.4.9.6.4 RDMA Queue with GPU Data Path

To execute RDMA operations from a GPU CUDA kernel, in the setup phase, the application must first create a DOCA RDMA queue, export the RDMA as context, and then set the datapath of the context on the GPU (as shown in the following code snippet).

The following is a pseudo-code to serve as a guide. Please refer to real function signatures in header files (`*.h`) and documentation for a complete overview of the functions.

GPU code

```
struct doca_dev *doca_device;       /* DOCA device */
struct doca_gpu *gpudev;        /* DOCA GPU device */
struct doca_rdma *rdma;         /* DOCA RDMA instance */
struct doca_gpu_dev_rdma *gpu_rdma; /* DOCA RDMA instance GPU handler */
struct doca_ctx *rdma_ctx;

// Initialize IBDev RDMA device
open_doca_device_with_ibdev_name(&doca_device)
// Initialize DPDK (hugepages not needed)
char *eal_param[4] = {"", "-a", "00:00.0", "--in-memory"};
rte_eal_init(4, eal_param);

// Initialize the GPU device
doca_gpu_create(&gpudev);
```

```
// Create the RDMA queue object with the DOCA device
doca_rdma_create(doca_device, &(rdma));
// Export the RDMA queue object context
rdma_ctx = doca_rdma_as_ctx(rdma)

// Set RDMA queue attributes

// Set GPU data path for the RDMA object
doca_ctx_set_datapath_on_gpu(ctx, gpudev)
doca_ctx_start(rdma_ctx);
```

At this point, the application has an RDMA queue usable from a GPU CUDA kernel. The next step would be to establish a connection using some OOB (out-of-band) mechanism (e.g., Linux sockets) to exchange RDMA queue info so each peer can connect to the other's queues.

To exchange data, users must create DOCA GPU buffer arrays to send or receive data. If the application also requires read or write, then the GPU memory associated with the buffer arrays must be exported and exchanged with the remote peers using the OOB mechanism.

GPU code

```
/* Create DOCA mmap to define memory layout and type for the DOCA buf array */
struct doca_mmap *mmap;
doca_mmap_create(&mmap);
/* Set DOCA mmap properties */
doca_mmap_start(mmap);
/* Export mmap info to share with remote peer */
doca_mmap_export_rdma(mmap, ...);

/* Exchange export info with remote peer */

/* Create DOCA buf arr and export it to GPU */
struct doca_buf_arr *buf_arr;
struct doca_gpu_buf_arr *buf_arr_gpu;
doca_buf_arr_create(mmap, &buf_arr);
/* Set DOCA buf array properties */
...
/* Export GPU handle for the buf arr */
doca_buf_arr_get_gpu_handle(buf_arr, &buf_arr_gpu);
```

Please refer to the "RDMA Client Server" sample as a basic layout to implement all the steps described in this section.

14.4.9.6.4.1 CUDA Kernel for RDMA Write

Assuming the RDMA queues and buffer arrays are correctly created and exchanged across peers, the application can launch a CUDA kernel to remotely write data. As typically applications use `strong` mode, the following code snippet shows how to use `weak` mode to post multiple writes from different CUDA threads in the same CUDA block.

GPU code

```
__global__ void rdma_write_bw(struct doca_gpu_dev_rdma *rdma_gpu, struct doca_gpu_buf_arr *local_buf_arr, struct
 doca_gpu_buf_arr *remote_buf_arr)
{
    struct doca_gpu_buf *remote_buf;
    struct doca_gpu_buf *local_buf;
    struct doca_gpu_dev_rdma *rdma_gpu;
    struct doca_gpu_buf_arr *server_local_buf_arr;
    struct doca_gpu_buf_arr *server_remote_buf_arr;
    uint32_t curr_position;
    uint32_t mask_max_position;
    uint32_t num_ops;

    doca_gpu_dev_buf_get_buf(server_local_buf_arr, threadIdx.x, &local_buf);
    doca_gpu_dev_buf_get_buf(server_remote_buf_arr, threadIdx.x, &remote_buf);

    /* Get RDMA queue current available position and mask of the max position number */
    doca_gpu_dev_rdma_get_info(rdma_gpu, &curr_position, &mask_max_position);

    doca_gpu_dev_rdma_write_weak(rdma_gpu,
                    /* Write into this remote buffer at offset 0 */
                    remote_buf, 0,
                    /* Fetch data from this local buffer at offset 0 */
```

```
                      local_buf, 0,
                      /* Number of bytes to write */
                      msg_size,
                      /* Don't use immediate */
                      0, DOCA_GPU_RDMA_WRITE_FLAG_NONE,
                      /* Position in the RDMA queue to post the write */
                      (curr_position + threadIdx.x) & mask_max_position);

        /* Wait all CUDA threads to post their RDMA Write */
        __syncthreads();

        if (threadIdx.x == 0) {
            /* Only 1 CUDA thread can push the write op just posted */
            doca_gpu_dev_rdma_commit_weak(rdma_gpu, blockDim.x);
            doca_gpu_dev_rdma_wait_all(rdma_gpu, &num_ops);
        }
        __syncthreads();
}
```

ⓘ The code in the "RDMA Client Server" sample shows how to use write and send with immediate flag set.

14.4.9.7 GPUNetIO Samples

This section contains two samples that show how to enable simple GPUNetIO features. Be sure to correctly set the following environment variables:

Build the sample

```
export PATH=${PATH}:/usr/local/cuda/bin
export CPATH="$(echo /usr/local/cuda/targets/{x86_64,sbsa}-linux/include | sed 's/ /:/'):${CPATH}"
export PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/usr/lib/pkgconfig:/opt/mellanox/grpc/lib/{x86_64,aarch64}-linux-gnu/
pkgconfig:/opt/mellanox/dpdk/lib/{x86_64,aarch64}-linux-gnu/pkgconfig:/opt/mellanox/doca/lib/{x86_64,aarch64}-
linux-gnu/pkgconfigexport LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/local/cuda/lib64:/opt/mellanox/gdrcopy/src:/opt/
mellanox/dpdk/lib/{x86_64,aarch64}-linux-gnu:/opt/mellanox/doca/lib/{x86_64,aarch64}-linux-gnu
```

ⓘ All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

⚠ **GPU architecture**

Please ensure the arch of your GPU is included in the `meson.build` file before building the samples (e.g., `sm_80` for Ampere, `sm_89` for L40, `sm_90` for H100, etc).

14.4.9.7.1 Ethernet Send Wait Time

The sample shows how to enable Accurate Send Scheduling (or wait-on-time) in the context of a GPUNetIO application. Accurate Send Scheduling is the ability of an NVIDIA NIC to send packets in the future according to application-provided timestamps.

⚠ This feature is supported on ConnectX-6 Dx and later.

ⓘ This NVIDIA blog post offers an example for how this feature has been used in 5G networks.

This DOCA GPUNetIO sample provides a simple application to send packets with Accurate Send Scheduling from the GPU.

14.4.9.7.1.1 Synchronizing Clocks

Before starting the sample, it is important to properly synchronize the CPU clock with the NIC clock. This way, timestamps provided by the system clock are synchronized with the time in the NIC.

For this purpose, at least the `phc2sys` service must be used. To install it on an Ubuntu system:

phc2sys

```
sudo apt install linuxptp
```

To start the `phc2sys` service properly, a config file must be created in `/lib/systemd/system/phc2sys.service`:

phc2sys

```
[Unit]
Description=Synchronize system clock or PTP hardware clock (PHC)
Documentation=man:phc2sys

[Service]
Restart=always
RestartSec=5s
Type=simple
ExecStart=/bin/sh -c "taskset -c 15 /usr/sbin/phc2sys -s /dev/ptp$(ethtool -T ens6f0 | grep PTP | awk '{print $4}')
-c CLOCK_REALTIME -n 24 -O 0 -R 256 -u 256"

[Install]
WantedBy=multi-user.target
```

Now `phc2sys` service can be started:

phc2sys

```
sudo systemctl stop systemd-timesyncd
sudo systemctl disable systemd-timesyncd
sudo systemctl daemon-reload
sudo systemctl start phc2sys.service
```

To check the status of `phc2sys`:

phc2sys

```
$ sudo systemctl status phc2sys.service

● phc2sys.service - Synchronize system clock or PTP hardware clock (PHC)
     Loaded: loaded (/lib/systemd/system/phc2sys.service; disabled; vendor preset: enabled)
     Active: active (running) since Mon 2023-04-03 10:59:13 UTC; 2 days ago
       Docs: man:phc2sys
   Main PID: 337824 (sh)
      Tasks: 2 (limit: 303788)
     Memory: 560.0K
        CPU: 52min 8.199s
     CGroup: /system.slice/phc2sys.service
               337824 /bin/sh -c "taskset -c 15 /usr/sbin/phc2sys -s /dev/ptp\$(ethtool -T enp23s0f1np1 | grep PTP
| awk '{print \$4}') -c CLOCK_REALTIME -n 24 -O 0 -R >
               337829 /usr/sbin/phc2sys -s /dev/ptp3 -c CLOCK_REALTIME -n 24 -O 0 -R 256 -u 256

Apr 05 16:35:52 doca-vr-045 phc2sys[337829]: [457395.040] CLOCK_REALTIME rms    8 max   18 freq +110532 +/-  27
delay   770 +/-   3
Apr 05 16:35:53 doca-vr-045 phc2sys[337829]: [457396.071] CLOCK_REALTIME rms    8 max   20 freq +110513 +/-  30
delay   769 +/-   3
Apr 05 16:35:54 doca-vr-045 phc2sys[337829]: [457397.102] CLOCK_REALTIME rms    8 max   18 freq +110527 +/-  30
delay   769 +/-   3
```

```
Apr 05 16:35:55 doca-vr-045 phc2sys[337829]: [457398.130] CLOCK_REALTIME rms    8 max   18 freq +110517 +/-  31
delay   769 +/-   3
Apr 05 16:35:56 doca-vr-045 phc2sys[337829]: [457399.159] CLOCK_REALTIME rms    8 max   19 freq +110523 +/-  32
delay   770 +/-   3
Apr 05 16:35:57 doca-vr-045 phc2sys[337829]: [457400.191] CLOCK_REALTIME rms    8 max   20 freq +110528 +/-  33
delay   770 +/-   3
Apr 05 16:35:58 doca-vr-045 phc2sys[337829]: [457401.221] CLOCK_REALTIME rms    8 max   19 freq +110512 +/-  38
delay   770 +/-   3
Apr 05 16:35:59 doca-vr-045 phc2sys[337829]: [457402.253] CLOCK_REALTIME rms    9 max   20 freq +110538 +/-  47
delay   770 +/-   4
Apr 05 16:36:00 doca-vr-045 phc2sys[337829]: [457403.281] CLOCK_REALTIME rms    8 max   21 freq +110517 +/-  38
delay   769 +/-   3
Apr 05 16:36:01 doca-vr-045 phc2sys[337829]: [457404.311] CLOCK_REALTIME rms    8 max   17 freq +110526 +/-  26
delay   769 +/-   3
...
```

At this point, the system and NIC clocks are synchronized so timestamps provided by the CPU are correctly interpreted by the NIC.

> The timestamps you get may not reflect the real time and day. To get that, you must properly set the `ptp4l` service with an external grand master on the system. Doing that is out of the scope of this sample.

14.4.9.7.1.2 Running the Sample

The sample is shipped with the source files that must be built:

phc2sys

```
# Ensure DOCA and DPDK are in the pkgconfig environment variable
cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_send_wait_time
meson build
ninja -C build
```

The sample sends 8 bursts of 32 raw Ethernet packets or 1kB to a dummy Ethernet address, `10:11:12:13:14:15`, in a timed way. Program the NIC to send every `t` nanoseconds (command line option `-t`).

The following example programs a system with GPU PCIe address `ca:00.0` and NIC PCIe address `17:00.0` to send 32 packets every 5 milliseconds:

Run

```
# Ensure DOCA and DPDK are in the LD_LIBRARY_PATH environment variable
$ sudo ./build/doca_gpunetio_send_wait_time -n 17:00.0 -g ca:00.0 -t 5000000[09:22:54:165778][1316878][DOCA][INF]
[gpunetio_send_wait_time_main.c:195][main] Starting the sample
[09:22:54:438260][1316878][DOCA][INF][gpunetio_send_wait_time_main.c:224][main] Sample configuration:
        GPU ca:00.0
        NIC 17:00.0
        Timeout 5000000ns
EAL: Detected CPU lcores: 128
...
EAL: Probe PCI driver: mlx5_pci (15b3:a2d6) device: 0000:17:00.0 (socket 0)
[09:22:54:819996][1316878][DOCA][INF][gpunetio_send_wait_time_sample.c:607][gpunetio_send_wait_time] Wait on time
 supported mode: DPDK
EAL: Probe PCI driver: gpu_cuda (10de:20b5) device: 0000:ca:00.0 (socket 1)
[09:22:54:830212][1316878][DOCA][INF][gpunetio_send_wait_time_sample.c:252][create_tx_buf] Mapping send queue
buffer (0x0x7f48e32a0000 size 262144B) with legacy nvidia-peermem mode
[09:22:54:832462][1316878][DOCA][INF][gpunetio_send_wait_time_sample.c:657][gpunetio_send_wait_time] Launching CUDA
kernel to send packets
[09:22:54:842945][1316878][DOCA][INF][gpunetio_send_wait_time_sample.c:664][gpunetio_send_wait_time] Waiting 10 sec
for 256 packets to be sent
[09:23:04:883309][1316878][DOCA][INF][gpunetio_send_wait_time_sample.c:684][gpunetio_send_wait_time] Sample
finished successfully
[09:23:04:883339][1316878][DOCA][INF][gpunetio_send_wait_time_main.c:239][main] Sample finished successfully
```

To verify that packets are actually sent at the right time, use a packet sniffer on the other side (e.g., `tcpdump`):

phc2sys

```
$ sudo tcpdump -i enp23s0f1np1 -A -s 64

17:12:23.480318 IP5 (invalid)
Sent from DOCA GPUNetIO...........................
....
17:12:23.480368 IP5 (invalid)
Sent from DOCA GPUNetIO...........................
# end of first burst of 32 packets, bump to +5ms
17:12:23.485321 IP5 (invalid)
Sent from DOCA GPUNetIO...........................
...
17:12:23.485369 IP5 (invalid)
Sent from DOCA GPUNetIO...........................
# end of second burst of 32 packets, bump to +5ms
17:12:23.490278 IP5 (invalid)
Sent from DOCA GPUNetIO...........................
...
```

The output should show a jump of approximately 5 milliseconds every 32 packets.

> ⚠ `tcpdump` may increase latency in sniffing packets and reporting the receive timestamp, so the difference between bursts of 32 packets reported may be less than expected, especially with small interval times like 500 microseconds (`-t 500000`).

14.4.9.7.2 Ethernet Simple Receive

This simple application shows the fundamental steps to build a DOCA GPUNetIO receiver application with one queue for UDP packets and one CUDA kernel receiving those packets from the GPU, printing packet info to the console.

> ❗ Invoking a `printf` from a CUDA kernel is not good practice for release software and should be used only to print debug information as it slows down the overall execution of the CUDA kernel.

To build and run the application:

Build the sample

```
# Ensure DOCA and DPDK are in the pkgconfig environment variable
cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_simple_receive
meson build
ninja -C build
```

To test the application, this guide assumes the usual setup with two machines: one with the DOCA receiver application and the second one acting as packet generator. As UDP packet generator, this example considers the `nping` application that can be easily installed easily on any Linux machine.

The command to send 10 UDP packets via `nping` on the packet generator machine is:

nping generator

```
$ nping --udp -c 10 -p 2090 192.168.1.1 --data-length 1024 --delay 500ms

Starting Nping 0.7.80 ( https://nmap.org/nping ) at 2023-11-20 11:05 UTC
SENT (0.0018s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (0.5018s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (1.0025s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (1.5025s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (2.0032s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (2.5033s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (3.0040s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (3.5040s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (4.0047s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (4.5048s) UDP packet with 1024 bytes to 192.168.1.1:2090

Max rtt: N/A | Min rtt: N/A | Avg rtt: N/A
UDP packets sent: 10 | Rcvd: 0 | Lost: 10 (100.00%)
Nping done: 1 IP address pinged in 5.50 seconds
```

Assuming the DOCA Simple Receive sample is waiting on the other machine at IP address `192.168.1.1`.

The DOCA Simple Receive sample is launched on a system with NIC at `17:00.1` PCIe address and GPU at `ca:00.0` PCIe address:

DOCA Simple Receive

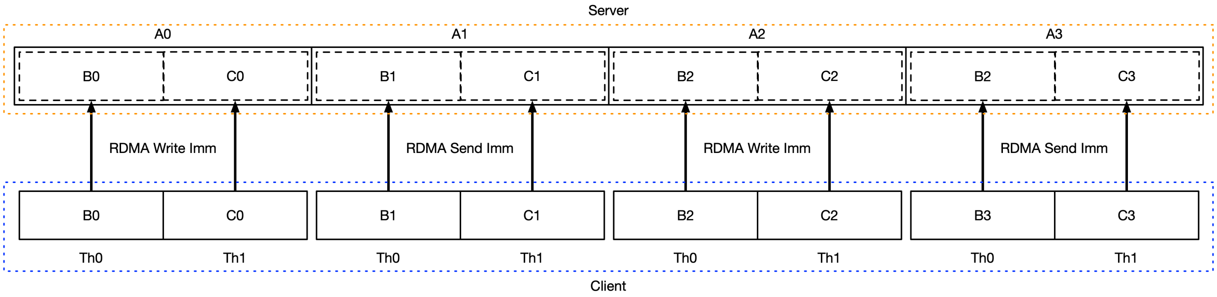
```
# Ensure DOCA and DPDK are in the LD_LIBRARY_PATH environment variable
$ sudo ./build/doca_gpunetio_simple_receive -n 17:00.1 -g ca:00.0
[11:00:30:397080][2328673][DOCA][INF][gpunetio_simple_receive_main.c:159][main] Starting the sample
[11:00:30:652622][2328673][DOCA][INF][gpunetio_simple_receive_main.c:189][main] Sample configuration:
    GPU ca:00.0
    NIC 17:00.1

EAL: Detected CPU lcores: 128
EAL: Detected NUMA nodes: 2
EAL: Detected shared linkage of DPDK
EAL: Multi-process socket /var/run/dpdk/rte/mp_socket
EAL: Selected IOVA mode 'PA'
EAL: VFIO support initialized
TELEMETRY: No legacy callbacks, legacy socket not created
EAL: Probe PCI driver: mlx5_pci (15b3:a2d6) device: 0000:17:00.1 (socket 0)
[11:00:31:036760][2328673][DOCA][WRN][engine_model.c:72][adapt_queue_depth] adapting queue depth to 128.
[11:00:31:928926][2328673][DOCA][WRN][engine_port.c:321][port_driver_process_properties] detected representor used
in VNF mode (driver port id 0)
EAL: Probe PCI driver: gpu_cuda (10de:20b5) device: 0000:ca:00.0 (socket 1)
[11:00:31:977261][2328673][DOCA][INF][gpunetio_simple_receive_sample.c:425][create_rxq] Creating Sample Eth Rxq

[11:00:31:977841][2328673][DOCA][INF][gpunetio_simple_receive_sample.c:466][create_rxq] Mapping receive queue
buffer (0x0x7f86cc000000 size 33554432B) with nvidia-peermem mode
[11:00:32:043182][2328673][DOCA][INF][gpunetio_simple_receive_sample.c:610][gpunetio_simple_receive] Launching CUDA
kernel to receive packets
[11:00:32:055193][2328673][DOCA][INF][gpunetio_simple_receive_sample.c:614][gpunetio_simple_receive] Waiting for
 termination
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9

# Type Ctrl+C to kill the sample

[11:01:44:265141][2328673][DOCA][INF][gpunetio_simple_receive_sample.c:45][signal_handler] Signal 2 received,
preparing to exit!
[11:01:44:265189][2328673][DOCA][INF][gpunetio_simple_receive_sample.c:620][gpunetio_simple_receive] Exiting from
sample
[11:01:44:265533][2328673][DOCA][INF][gpunetio_simple_receive_sample.c:362][destroy_rxq] Destroying Rxq
[11:01:44:307829][2328673][DOCA][INF][gpunetio_simple_receive_sample.c:631][gpunetio_simple_receive] Sample
finished successfully
[11:01:44:307861][2328673][DOCA][INF][gpunetio_simple_receive_main.c:204][main] Sample finished successfully
```

14.4.9.7.3 RDMA Client Server

This sample exhibits how to use the GPUNetIO RDMA API to receive and send/write with immediate using a single RDMA queue.

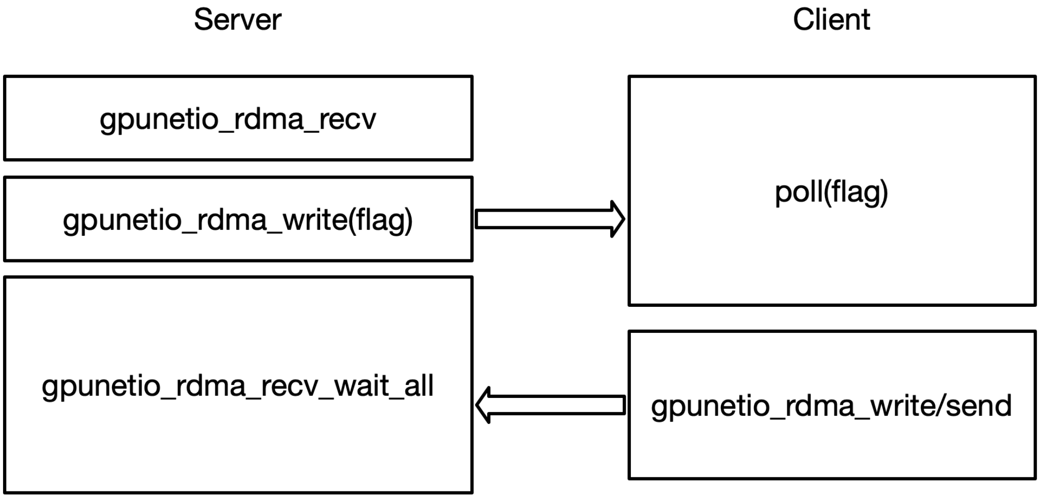
The server has a GPU buffer array A composed by `GPU_BUF_NUM doca_gpu_buf` elements, each 1kB in size. The client has two GPU buffer arrays, B and C, each composed by `GPU_BUF_NUM doca_gpu_buf` elements, each 512B in size.

The goal is for the client to fill a single server buffer of 1kB with two GPU buffers of 512B as illustrated in the following figure:



To show how to use RDMA write and send, even buffers are sent from the client with write immediate, while odd buffers are sent with send immediate. In both cases, the server must pre-post the RDMA receive operations.

For each buffer, the CUDA kernel code repeats the handshake:



Once all buffers are filled, the server double checks that all values are valid. The server output should be as follows:

DOCA RDMA Server side

```
# Ensure DOCA and DPDK are in the LD_LIBRARY_PATH environment variable
$ cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_rdma_client_server_write
$ ./build/doca_gpunetio_rdma_client_server_write -gpu 17:00.0 -d mlx5_0

[14:11:43:000930][1173110][DOCA][INF][gpunetio_rdma_client_server_write_main.c:250][main] Starting the sample
EAL: Detected CPU lcores: 64
EAL: Detected NUMA nodes: 2
EAL: Detected shared linkage of DPDK
EAL: Selected IOVA mode 'VA'
EAL: No free 2048 kB hugepages reported on node 0
EAL: No free 2048 kB hugepages reported on node 1
EAL: VFIO support initialized
TELEMETRY: No legacy callbacks, legacy socket not created
EAL: Probe PCI driver: gpu_cuda (10de:2331) device: 0000:17:00.0 (socket 0)
[14:11:43:686581][1173110][DOCA][ERR][rdma_common.c:64][oob_connection_server_setup] Socket created successfully
[14:11:43:686598][1173110][DOCA][INF][rdma_common.c:83][oob_connection_server_setup] Done with binding
[14:11:43:686610][1173110][DOCA][INF][rdma_common.c:91][oob_connection_server_setup] Listening for incoming
connections
[14:11:45:681523][1173110][DOCA][INF][rdma_common.c:105][oob_connection_server_setup] Client connected at IP:
192.168.2.28 and port: 46274
[14:11:45:681557][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:591][rdma_write_server] Send
connection details to remote peer size 216 str
[14:11:45:681613][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:604][rdma_write_server] Receive
remote connection details
[14:11:45:682110][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:622][rdma_write_server] Connect
DOCA RDMA to remote RDMA
```

```
[14:11:45:686065][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:111]
[create_memory_local_remote_server] Create local server mmap A context
[14:11:45:686976][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:143]
[create_memory_local_remote_server] Create local server mmap A context
[14:11:45:687907][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:151]
[create_memory_local_remote_server] Send exported mmap A to remote client
[14:11:45:687966][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:162]
[create_memory_local_remote_server] Receive client mmap F export
[14:11:45:771521][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:195]
[create_memory_local_remote_server] Create local DOCA buf array context A
[14:11:45:771660][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:207]
[create_memory_local_remote_server] Create local DOCA buf array context F
[14:11:45:771727][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:219]
[create_memory_local_remote_server] Create remote DOCA buf array context F
[14:11:45:771807][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:644][rdma_write_server] Before
launching CUDA kernel, buffer array A is:
[14:11:45:771822][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:646][rdma_write_server] Buffer 0
-> offset 0: 1111 | offset 128: 1111
[14:11:45:771837][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:646][rdma_write_server] Buffer 1
-> offset 0: 1111 | offset 128: 1111
[14:11:45:771851][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:646][rdma_write_server] Buffer 2
-> offset 0: 1111 | offset 128: 1111
[14:11:45:771864][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:646][rdma_write_server] Buffer 3
-> offset 0: 1111 | offset 128: 1111
RDMA Recv 2 ops completed with immediate values 0 and 1!
RDMA Recv 2 ops completed with immediate values 1 and 2!
RDMA Recv 2 ops completed with immediate values 2 and 3!
RDMA Recv 2 ops completed with immediate values 3 and 4!
[14:11:45:781561][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:671][rdma_write_server] After
launching CUDA kernel, buffer array A is:
[14:11:45:781574][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:673][rdma_write_server] Buffer 0
-> offset 0: 2222 | offset 128: 3333
[14:11:45:781583][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:673][rdma_write_server] Buffer 1
-> offset 0: 2222 | offset 128: 3333
[14:11:45:781593][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:673][rdma_write_server] Buffer 2
-> offset 0: 2222 | offset 128: 3333
[14:11:45:781602][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:673][rdma_write_server] Buffer 3
-> offset 0: 2222 | offset 128: 3333
[14:11:45:781640][1173110][DOCA][INF][gpunetio_rdma_client_server_write_main.c:294][main] Sample finished
successfully
```

On the other side, assuming the server is at IP address `192.168.2.28`, the client output should be as follows:

DOCA RDMA Client side

```
# Ensure DOCA and DPDK are in the LD_LIBRARY_PATH environment variable

$ cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_rdma_client_server_write
$ ./build/doca_gpunetio_rdma_client_server_write -gpu 17:00.0 -d mlx5_0 -c 192.168.2.28

[16:08:22:335744][160913][DOCA][INF][gpunetio_rdma_client_server_write_main.c:197][main] Starting the sample
EAL: Detected CPU lcores: 64
EAL: Detected NUMA nodes: 2
EAL: Detected shared linkage of DPDK
EAL: Selected IOVA mode 'PA'
EAL: No free 2048 kB hugepages reported on node 0
EAL: No free 2048 kB hugepages reported on node 1
EAL: VFIO support initialized
TELEMETRY: No legacy callbacks, legacy socket not created
EAL: Probe PCI driver: gpu_cuda (10de:2331) device: 0000:17:00.0 (socket 0)
[16:08:25:752916][160913][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:716][rdma_write_client] Function
create_rdma_resources completed correctly
[16:08:25:752932][160913][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:725][rdma_write_client] Got GPU
handle at 0x7f6596710000
[16:08:25:752944][160913][DOCA][INF][rdma_common.c:134][oob_connection_client_setup] Socket created successfully
[16:08:25:753316][160913][DOCA][INF][rdma_common.c:147][oob_connection_client_setup] Connected with server
successfully
......
Client waiting on flag 7f6596735000 for server to post RDMA Recvs
Thread 0 post rdma write imm 0
Thread 1 post rdma write imm 0
Client waiting on flag 7f6596735001 for server to post RDMA Recvs
Thread 0 post rdma send imm 1
Thread 1 post rdma send imm 1
Client waiting on flag 7f6596735002 for server to post RDMA Recvs
Thread 0 post rdma write imm 2
Thread 1 post rdma write imm 2
Client waiting on flag 7f6596735003 for server to post RDMA Recvs
Thread 0 post rdma send imm 3
Thread 1 post rdma send imm 3
[16:08:25:853454][160913][DOCA][INF][gpunetio_rdma_client_server_write_main.c:241][main] Sample finished
successfully
```

With RDMA, the network device must be specified by name (e.g., `mlx5_0`) instead of the PCIe address (as is the case for Ethernet).

Printing from a CUDA kernel is not recommended for performance. It may make sense for debugging purposes and for simple samples like this one.

14.4.9.7.4 GPU DMA Copy

This sample exhibits how to use the DOCA DMA and DOCA GPUNetIO libraries to DMA copy a memory buffer from the CPU to the GPU (with DOCA DMA CPU functions) and from the GPU to the CPU (with DOCA GPUNetIO DMA device functions) from a CUDA kernel. This sample requires a DPU as it uses the DMA engine on it.

DOCA RDMA Client side

```
$ cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_dma_memcpy

# Build the sample and then execute

$ ./build/doca_gpunetio_dma_memcpy -g 17:00.0 -n ca:00.0
[15:44:04:189462][862197][DOCA][INF][gpunetio_dma_memcpy_main.c:164][main] Starting the sample
EAL: Detected CPU lcores: 64
EAL: Detected NUMA nodes: 2
EAL: Detected shared linkage of DPDK
EAL: Selected IOVA mode 'VA'
EAL: No free 2048 kB hugepages reported on node 0
EAL: No free 2048 kB hugepages reported on node 1
EAL: VFIO support initialized
TELEMETRY: No legacy callbacks, legacy socket not created
EAL: Probe PCI driver: gpu_cuda (10de:2331) device: 0000:17:00.0 (socket 0)
[15:44:04:857251][862197][DOCA][INF][gpunetio_dma_memcpy_sample.c:211][init_sample_mem_objs] The CPU source buffer
value to be copied to GPU memory: This is a sample piece of text from CPU
[15:44:04:857359][862197][DOCA][WRN][doca_mmap.cpp:1743][doca_mmap_set_memrange] Mmap 0x55aec6206140: Memory range
isn't cache-line aligned - addr=0x55aec52ceb10. For best performance align address to 64B
[15:44:04:858839][862197][DOCA][INF][gpunetio_dma_memcpy_sample.c:158][init_sample_mem_objs] The GPU source buffer
value to be copied to CPU memory: This is a sample piece of text from GPU
[15:44:04:921702][862197][DOCA][INF][gpunetio_dma_memcpy_sample.c:570][submit_dma_memcpy_task] Success, DMA memcpy
job done successfully
CUDA KERNEL INFO: The GPU destination buffer value after the memcpy: This is a sample piece of text from CPU
CPU received message from GPU: This is a sample piece of text from GPU
[15:44:04:930087][862197][DOCA][INF][gpunetio_dma_memcpy_sample.c:364][gpu_dma_cleanup] Cleanup DMA ctx with GPU
data path
[15:44:04:932658][862197][DOCA][INF][gpunetio_dma_memcpy_sample.c:404][gpu_dma_cleanup] Cleanup DMA ctx with CPU
data path
[15:44:04:954156][862197][DOCA][INF][gpunetio_dma_memcpy_main.c:197][main] Sample finished successfully
```

14.4.10 DOCA App Shield

This guide provides instructions on using the DOCA App Shield API.

14.4.10.1 Introduction

DOCA App Shield API offers a solution for strong intrusion detection capabilities using the DPU services to collect and analyze data from the host's (or a VM on the host) memory in real time. This solution provides intrusion detection and forensics investigation in a way that is:

- Robust against attacks on a host machine
- Able to detect a wide range of attacks (including zero-day attacks)
- Least disruptive to the execution of host application (where current detection solutions hinder the performance of host applications)
- Transparent to the host, such that the host does not need to install anything (other than providing some files obtained from the `doca_apsh_config.py` tool)

App Shield uses a DMA device to access the host's memory and analyze it.

The App Shield API provides multiple functions that help with gathering data extracted from system's memory (e.g., processes list, modules list, connections). This data helps with detecting attacks on critical services or processes in a system (e.g., services that enforce integrity or privacy of the execution of different applications).

14.4.10.2 Prerequisites

1. Configure the NVIDIA® BlueField® networking platform's (DPU or SuperNIC) firmware.
    a. On BlueField, configure the PF base address register and NVMe emulation. Run:

    ```
    dpu> mlxconfig -d /dev/mst/mt41686_pciconf0 s PF_BAR2_SIZE=2 PF_BAR2_ENABLE=1
    ```

    If working with VFs, configure NVME emulation, SR-IOV, and number of VFs. Run:

    ```
    dpu> mlxconfig -d /dev/mst/mt41686_pciconf0 s NVME_EMULATION_ENABLE=1 SRIOV_EN=1 NUM_OF_VFS=<vf-
    number>
    ```

    b. Perform graceful shutdown and a cold boot from the host.

    > These configurations can be checked using the following command:
    >
    > ```
    > dpu> mlxconfig -d /dev/mst/mt41686_pciconf0 q | grep -E "NVME|BAR|SRIOV|NUM_OF_VFS"
    > ```

2. Download target system (host/VM) symbols.
    - For Ubuntu:

    ```
    host> sudo tee /etc/apt/sources.list.d/ddebs.list << EOF
    deb http://ddebs.ubuntu.com/ $(lsb_release -cs) main restricted universe multiverse
    deb http://ddebs.ubuntu.com/ $(lsb_release -cs)-updates main restricted universe multiverse
    deb http://ddebs.ubuntu.com/ $(lsb_release -cs)-proposed main restricted universe multiverse
    EOF
    host> sudo apt install ubuntu-dbgsym-keyring
    host> sudo apt-get update
    host> sudo apt-get install linux-image-$(uname -r)-dbgsym
    ```

    - For CentOS:

    ```
    host> yum install --enablerepo=base-debuginfo kernel-devel-$(uname -r) kernel-debuginfo-$(uname -r)
    kernel-debuginfo-common-$(uname -m)-$(uname -r)
    ```

    - No action is needed for Windows

3. Perform IOMMU passthrough. This stage is only necessary if IOMMU is not enabled by default (e.g., when the host is using an AMD CPU).

    > Skip this step if you are not sure whether it is needed. Return to it only if DMA fails with a message similar to the following in `dmesg`:
    >
    > ```
    > host> dmesg
    > [ 3839.822897] mlx5_core 0000:81:00.0: AMD-Vi: Event logged [IO_PAGE_FAULT domain=0x0047
    > address=0x2a0aff8 flags=0x0000]
    > ```

    a. Locate your OS's `grub` file (most likely `/boot/grub/grub.conf`, `/boot/grub2/grub.cfg`, or `/etc/default/grub`) and open it for editing. Run:

```
host> vim /etc/default/grub
```

b. Search for the line defining `GRUB_CMDLINE_LINUX_DEFAULT` and add the argument `iommu=pt`. For example:

```
GRUB_CMDLINE_LINUX_DEFAULT="iommu=pt <intel/amd>_iommu=on"
```

c. Run:

> Prior to performing a power cycle, make sure to do a graceful shutdown.

- For Ubuntu:

```
host> sudo update-grub
```

- For CentOS:

```
host> grub2-mkconfig -o /boot/grub2/grub.cfg
```

4. Prepare target:

a. Install DOCA on the target system.

b. Create the ZIP and JSON files. Run:

```
target-system> cd /opt/mellanox/doca/tools/
target-system> python3 doca_apsh_config.py --pid <pid-of-process-to-monitor> --os <windows/linux>
--path <path to dwarf2json executable  or pdbparse-to-json.py>
target-system> cp /opt/mellanox/doca/tools/*.* <shared-folder-with-baremetal>
dpu> scp <shared-folder-with-baremetal>/* <path-to-app-shield-binary>
```

If the target system does not have DOCA installed, the script can be copied from BlueField.
The required `dwaf2json` and `pdbparse-to-json.py` are not provided with DOCA.

> If the kernel and process `.exe` have not changed, there is no need to redo this step.

14.4.10.3 Dependencies

The library requires firmware version 24.32.1010 or higher.

14.4.10.4 API

For the library API reference, refer to the DOCA APSH API documentation in the NVIDIA DOCA Library APIs.

> The pkg-config (`*.pc` file) for the APSH library is `doca-apsh`.

The following subsections provide more details about the library API.

14.4.10.4.1 doca_apsh_dma_dev_set

To attach a DOCA DMA device to App Shield, calling this function is mandatory and must be done before calling `doca_apsh_start`.

```
doca_apsh_dma_dev_set(doca_apsh_ctx, doca_dev)
```

- `doca_apsh_ctx [in]` - App Shield opaque context struct
- `doca_dev [in]` - struct for DOCA Device with DMA capabilities

14.4.10.4.2 Capabilities Per System

For each initialized system, App Shield retrieves an array of the requested object according to the getter's name:

Getter Function Name	Functions Information	Functions Signature	Return Type
Get modules	Returns an array with information about the system modules (drivers) loaded into the kernel of the OS.	`doca_error_t doca_apsh_modules_get(struct doca_apsh_system *system, struct doca_apsh_module ***modules, int *modules_size);`	• Array of `struct doca_apsh_module` • `int` - size of the returned array • `doca_error` status
Get processes	Returns an array with information about each process running on the system.	`doca_error_t doca_apsh_processes_get(struct doca_apsh_system *system, struct doca_apsh_process ***processes, int *processes_size);`	• Array of `struct doca_apsh_process` • `int` - size of the returned array • `doca_error` status
Get library	For a specified process, this function returns an array with information about each library loaded into this process.	`doca_error_t doca_apsh_libs_get(struct doca_apsh_process *process, struct doca_apsh_lib ***libs, int *libs_size);`	• Array of `struct doca_apsh_lib` • `int` - size of the returned array • `doca_error` status

Getter Function Name	Functions Information	Functions Signature	Return Type
Get threads	For a specified process, this function returns an array with information about each thread running within this process.	`doca_error_t doca_apsh_threads_get(struct doca_apsh_process *process, struct doca_apsh_thread ***threads, int *threads_size);`	• Array of `struct doca_apsh_thread` • `int` - size of the returned array • `doca_error` status
Get virtual memory areas/ virtual address description	For a specified process, this function returns an array with information about each virtual memory area within this process.	`doca_error_t doca_apsh_vads_get(struct doca_apsh_process *process, struct doca_apsh_vad ***vads, int *vads_size);`	• Array of `struct doca_apsh_vma` • `int` - size of the returned array • `doca_error` status
Get privileges	For a specified process, this function returns an array with information about each possible privilege for this process, as described here. ⚠ Available on a Windows host only.	`doca_error_t doca_apsh_privileges_get(struct doca_apsh_process *process, struct doca_apsh_privilege ***privileges, int *privileges_size);`	• Array of `struct doca_apsh_privilege` • `int` - size of the returned array • `doca_error` status
Get environment variables	For a specified process, this function returns an array with information about each environment variable within this process. ⚠ Available on a Windows host only.	`doca_error_t doca_apsh_envars_get(struct doca_apsh_process *process, struct doca_apsh_envar ***envars, int *envars_size);`	• Array of `struct doca_apsh_envar` • `int` - size of the returned array • `doca_error` status

Getter Function Name	Functions Information	Functions Signature	Return Type
Get handles	For a specified process, this function returns an array with information about each handle this process holds. ⚠ Available on a Windows host only.	`doca_error_t doca_apsh_handles_get(struct doca_apsh_process *process, struct doca_apsh_handle ***handles, int *handles_size);`	• Array of `struct doca_apsh_handle` • `int` - size of the returned array • `doca_error` status
Get LDR modules	For a specified process, this function returns an array with information about each loaded module within this process. ⚠ Available on a Windows host only.	`doca_error_t doca_apsh_ldrmodules_get(struct doca_apsh_process *process, struct doca_apsh_ldrmodule ***ldrmodules, int *ldrmodules_size);`	• Array of `struct doca_apsh_ldrmodule` • `int` - size of the returned array • `doca_error` status
Process attestation	For a specified process, this function attests the memory pages of the process according to a precomputed golden hash file given as an input. ⚠ Single-threaded processes are supported at beta level.	`doca_error_t doca_apsh_attestation_get(struct doca_apsh_process *process, const char *exec_hash_map_path, struct doca_apsh_attestation ***attestation, int *attestation_size);`	• Array of `struct doca_apsh_attestation` • `int` - size of the returned array • `doca_error` status
Attestation refresh	Refreshes a single attestation handler of a process with a new snapshot.	`doca_error_t doca_apsh_attst_refresh(struct doca_apsh_attestation ***attestation, int *attestation_size);`	• Array of `struct doca_apsh_attestation` • `int` - size of the returned array • `doca_error` status

<table>
<tr><th>Getter Function Name</th><th>Functions Information</th><th>Functions Signature</th><th>Return Type</th></tr>
<tr>
<td>Get NetScan</td>
<td>This function scans the system's physical memory and returns an array with information about each socket that resides in the memory.
⚠ Only available on hosts with one of the following Windows 10 OS builds:
<table>
<tr><th>Arch</th><th>Build No.</th></tr>
<tr><td rowspan="6">x86</td><td>10240</td></tr>
<tr><td>10586</td></tr>
<tr><td>14393</td></tr>
<tr><td>15063</td></tr>
<tr><td>17134</td></tr>
<tr><td>19041</td></tr>
<tr><td rowspan="7">x64</td><td>15063</td></tr>
<tr><td>16299</td></tr>
<tr><td>17134</td></tr>
<tr><td>17763</td></tr>
<tr><td>18362</td></tr>
<tr><td>18363</td></tr>
<tr><td>19041</td></tr>
</table>

⚠ This feature is currently supported at beta level.</td>
<td><pre><code>doca_error_t doca_apsh_netscan_get(struct
doca_apsh_system *system, struct
doca_apsh_netscan ***connections, int
 *connections_size);</code></pre></td>
<td>• Array of <code>struct doca_apsh_netscan</code>
• <code>int</code> - size of the returned array
• <code>doca_error</code> status</td>
</tr>
<tr>
<td>Get process parameters</td>
<td>For a specified process, this function returns a struct object (not an array) with information about the process' parameters (ones not included in the "get processes" capability).
⚠ Available on a Windows host only.
⚠ This feature is currently supported at beta level.</td>
<td><pre><code>doca_error_t
doca_apsh_process_parameters_get(struct
doca_apsh_process *process, struct
doca_apsh_process_parameters
**process_parameters);</code></pre></td>
<td>• An object of <code>struct doca_apsh_process_paramters</code>
• <code>doca_error</code> status</td>
</tr>
</table>

Getter Function Name	Functions Information	Functions Signature	Return Type
Get security identifier (SID)	For a specified process, this function returns an array with information about each SID (security identifier) included in the process's security context. ⚠ Available on a Windows host only.	`doca_error_t doca_apsh_sids_get(struct doca_apsh_process *process, struct doca_apsh_sid ***sids, int *sids_size);`	- Array of `struct doca_apsh_sid` - `int` - size of the returned array - `doca_error` status
Perform Yara scan	For a specified process, this function returns an array with information about each Yara rule match found in the process's memory. ⚠ Available on a Windows host and Ubuntu 22.04 DPU.	`doca_error_t doca_apsh_yara_get(struct doca_apsh_process *process, enum doca_apsh_yara_rule *yara_rules_arr, uint32_t yara_rules_arr_size, uint64_t scan_type, struct doca_apsh_yara ***yara_matches, int *yara_matches_size);` ⚠ To get a better understanding of the arguments, refer to documentation in `doca_apsh.h`.	- Array of `struct doca_apsh_yara` - `int` - size of the returned array - `doca_error` status
Get containers	Returns an array with information about each container running on the system. ⚠ Available on a Linux host only. ⚠ Only available for containers on the following runtimes: - runc - containerd	`doca_error_t doca_apsh_containers_get(struct doca_apsh_system *system, struct doca_apsh_container ***containers, int *containers_size);`	- Array of `struct doca_apsh_container` - `int` - size of the returned array - `doca_error` status
Get container's processes	For a specified container, this function returns an array with information about each process running within this container. ⚠ Available on a Linux host only. ⚠ Only available for containers on the following runtimes: - runc - containerd	`doca_error_t doca_apsh_container_processes_get(struct doca_apsh_container *container, struct doca_apsh_process ***processes, int *processes_size);`	- Array of `struct doca_apsh_process` - `int` - size of the returned array - `doca_error` status

The following attribute getters return a specific attribute of an object, obtained from the array returned from the getter functions listed above, depending on the requested attribute:

```
doca_apsh_process_info_get(struct doca_apsh_proccess *process, enum doca_apsh_process_attr attr);
doca_apsh_module_info_get(struct doca_apsh_module *module, enum doca_apsh_module_attr attr);
doca_apsh_lib_info_get(struct doca_apsh_lib *lib, enum doca_apsh_lib_attr attr);
doca_apsh_thread_info_get(struct doca_apsh_thread *thread, enum doca_apsh_lib_attr attr);
doca_apsh_vad_info_get(struct doca_apsh_vad *vad, enum doca_apsh_vad_attr attr);
doca_apsh_privilege_info_get(struct doca_apsh_privilege *privilege, enum doca_apsh_privilege_attr attr);
doca_apsh_envar_info_get(struct doca_apsh_envar *envar, enum doca_apsh_envar_attr attr);
doca_apsh_handle_info_get(struct doca_apsh_handle *handle, enum doca_apsh_handle_attr attr);
doca_apsh_ldrmodule_info_get(struct doca_apsh_ldrmodule *ldrmodule, enum doca_apsh_ldrmodule_attr attr);
doca_apsh_attst_info_get(struct doca_apsh_attestation *attestation, enum doca_apsh_attestation_attr attr);
doca_apsh_netscan_info_get(struct doca_apsh_netscan *connection, enum doca_apsh_netscan_attr attr)
doca_apsh_process_parameters_info_get(struct doca_apsh_process_parameters *process_parameters, enum
 doca_apsh_process_parameters_attr attr);
doca_apsh_sid_info_get(struct doca_apsh_sid *sid, enum doca_apsh_sid_attr attr);
doca_apsh_yara_info_get(struct doca_apsh_yara *yara, enum doca_apsh_yara_attr attr);
doca_apsh_container_info_get(struct doca_apsh_container *container, enum doca_apsh_container_attr attr);
```

The return type of the attribute getter can be found in `doca_apsh_attr.h`.

Usage example:

```
const uint pid = doca_apsh_process_info_get(processes[i], DOCA_APSH_PROCESS_PID);
const char *proc_name = doca_apsh_process_info_get(processes[i], DOCA_APSH_PROCESS_COMM);
```

14.4.10.5 App Shield Initialization and Teardown

To use App Shield, users must initialize and configure two main structs. This section presents these structs and explains how to interact with them.

14.4.10.5.1 doca_apsh_ctx

`doca_apsh_ctx` is the basic struct used by App Shield which defines the DMA device used to perform the memory forensics techniques required to run App Shield.

> ⚠ The same `doca_apsh_ctx` struct may be used to run multiple App Shield instances over different systems (e.g., two different VMs on the host).

1. To acquire an instance of the `doca_apsh_ctx` struct, use the following function:

   ```
   struct doca_apsh_ctx *doca_apsh_create(void);
   ```

2. To configure the `doca_apsh_ctx` instance with DMA device to use:

   ```
   doca_error_t doca_apsh_dma_dev_set(struct doca_apsh_ctx *ctx, struct doca_dev *dma_dev);
   ```

3. To start the `doca_apsh_ctx` instance, call the following function:

   ```
   doca_error_t doca_apsh_start(struct doca_apsh_ctx *ctx);
   ```

4. To destroy the `doca_apsh_ctx` instance when it is no longer needed, call:

   ```
   void doca_apsh_destroy(struct doca_apsh_ctx *ctx);
   ```

14.4.10.5.2 doca_apsh_system

The `doca_apsh_system` struct is built on the `doca_apsh_ctx` instance. This struct is created per system running App Shield. `doca_apsh_system` defines multiple attributes used by App Shield to perform memory analysis over the specific system successfully.

1. To acquire an instance of the `doca_apsh_system` struct, use the following function:

   ```
   const uint pid = doca_apsh_process_info_get(processes[i], DOCA_APSH_PROCESS_PID);
   const char *proc_name = doca_apsh_process_info_get(processes[i], DOCA_APSH_PROCESS_COMM);
   ```

2. To configure different attributes for the system instance:
   - OS type - specifies the system's OS type.

     ```
     doca_error_t doca_apsh_sys_os_type_set(struct doca_apsh_system *ctx, enum doca_apsh_system_os
     os_type);
     ```

     > ⚠ Currently supported types: Windows or Linux.

   - System representor - specifies the representor of the device connected to the system for App Shield to run on (which can be a representor of VF/PF). For information on querying the DOCA device, refer to the DOCA Core.
     After acquiring the DOCA device, use the following function to configure it into the system instance:

     ```
     doca_error_t doca_apsh_sys_dev_set(struct doca_apsh_system *system, struct doca_dev_rep *dev);
     ```

   - System symbols map - includes information about the OS that App Shield is attempting to run on (e.g., Window 10 Build 18363) and the size and fields of the OS structures, which helps App Shield with the memory forensic techniques it uses to access and analyze these structures in the system's memory. This can be obtained by running the `doca_apsh_config.py` on the system machine.
     After obtaining it, run:

     ```
     doca_error_t doca_apsh_sys_os_symbol_map_set(struct doca_apsh_system *system, const char
     *system_os_symbol_map_path);
     ```

   - Memory regions - includes the physical addresses of the memory regions which are mapped for system memory RAM. This is needed to prevent App Shield from accessing other memory regions, such as memory mapped I/O regions. This can be obtained by running the `doca_apsh_config.py` tool on the system machine.
     After obtaining it, run:

     ```
     doca_error_t doca_apsh_sys_mem_region_set(struct doca_apsh_system *system, const char
     *system_mem_region_path);
     ```

   - KPGD file (optional and relevant only for Linux OS) - contains the KPGD physical address and the virtual address of `init_task`. This information is required since App Shield extracts data from the kernel struct in the physical memory. Thus, the kernel page directory table must translate the virtual addresses of these structs. This can be obtained by running the `doca_apsh_config.py` tool on the system machine with the

flag `find_kpgd=1`. Since setting this attribute is optional, App Shield can work without it, but providing it speeds up App Shield's initialization process.
After obtaining it, run:

```
doca_error_t doca_apsh_sys_kpgd_file_set(struct doca_apsh_system *system, const char
*system_kpgd_file_path);
```

3. To start the `doca_apsh_system`:

```
doca_error_t doca_apsh_system_start(struct doca_apsh_system *system);
```

4. To destroy the `doca_apsh_system` instance when it is no longer needed, call:

```
void doca_apsh_system_destroy(struct doca_apsh_system *system);
```

14.4.10.5.3 doca_apsh_config.py Tool

The `doca_apsh_config.py` tool is a python3 script which can be used to obtain all the attributes needed to run `doca_apsh_system` instance.

The following parameters are necessary to use the tool:

Parameter	Description
`pid` (optional)	The process ID of the process we want to run attestation capability on
`os` (mandatory)	The OS type of the machine (i.e., Linux or Windows)
`find_kpgd` (optional)	Relevant for Linux OS only, AS flag to enable/disable creating `kpgd_file.conf`. Default 0.
`files` (mandatory)	A list of files for the tool to create. File options: `hash`, `symbols`, `memregions`, `kpgd_file` (only relevant for Linux). ⚠ Make sure that the value set is appropriate for your setup.
`path` (mandatory)	• Linux - path to the `dwarf2json` executable. Default `./dwarf2json`. This file can be obtained by compiling the following project using Go. • Windows - path to `pdbparse-to-json.py`. Default `./pdbparse-to-json.py`. This file can be found here. ⚠ Make sure that the value set is appropriate for your setup.

The tool creates the following files:

- Symbol map - this file changes once the system kernel is updated or a kernel module is installed. The file does not change on system reboot.

- Memory regions – this file changes when adding or removing hardware or drivers that affect the system's memory map (e.g., when adding register addresses). The file does not change on system reboot.
- `hash.zip` – this file is required for attestation but is unnecessary for all other capabilities. The ZIP file contains the required data to attest to a single process. The file changes on library or executable update.
- `kpgd_file.conf` (relevant for Linux OS only) – helps with faster initialization of the library. The file changes on system reboot.

14.4.10.6 DOCA App Shield Samples

This section provides DOCA App Shield library sample implementations on top of BlueField.

> ⓘ All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

14.4.10.6.1 Sample Prerequisites

Follow the guidelines in section "Prerequisites" then copy the generated JSON files, `symbols.json` and `mem_regions.json`, to the `/tmp/` directory.

14.4.10.6.2 Running the Sample

1. Refer to the following documents:
    - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
    - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample:

    ```
    cd /opt/mellanox/doca/samples/doca_apsh/<sample_name>
    meson /tmp/build
    ninja -C /tmp/build
    ```

    > ⚠ The binary `doca_<sample_name>` will be created under `/tmp/build/`.

3. Sample (e.g., `apsh_libs_get`) usage:

    ```
    Usage: doca_apsh_libs_get [DOCA Flags] [Program Flags]

    DOCA Flags:
      -h, --help                        Print a help synopsis
      -v, --version                     Print program version information
      -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
    30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
      --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICA
    L, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
      -j, --json <path>                 Parse all command flags from an input json file

    Program Flags:
      -p, --pid                         Process ID of process to be analyzed
      -f, --vuid                        VUID of the System device
      -d, --dma                         DMA device name
      -s, --osty <windows|linux>        System OS type - windows/linux
    ```

4. For additional information per sample, use the `-h` option:

```
/tmp/build/doca_<sample_name> -h
```

14.4.10.6.3 Samples

14.4.10.6.3.1 Apsh Libs Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of loadable libraries of a specific process.

The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCI device via given vendor unique identifier (VUID).
5. Creating DOCA Apsh system handler.
6. Setting fields and starting Apsh system handler.
7. Getting the list of system process using Apsh API and searching for a specific process with the given PID.
8. Getting the list of process-loadable libraries using `doca_apsh_libs_get` Apsh API call.
9. Querying the libraries for 3 selected fields using `doca_apsh_lib_info_get` Apsh API call.
10. Printing libraries' attributes to the terminal.
11. Cleaning up.

References:

- `/opt/mellanox/doca/samples/doca_apsh/apsh_libs_get/apsh_libs_get_sample.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_libs_get/apsh_libs_get_main.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_libs_get/meson.build`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_common.c`; `/opt/mellanox/doca/samples/doca_apsh/apsh_common.h`

14.4.10.6.3.2 Apsh Modules Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of installed modules on a monitored system.
The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCI device via given VUID.
5. Creating DOCA Apsh system handler.
6. Setting fields and start Apsh system handler.
7. Getting the the list of system-installed modules using `doca_apsh_modules_get` Apsh API call.
8. Querying the names of modules using `doca_apsh_module_info_get` Apsh API call.

9. Printing the attributes of up to 5 modules attributes to the terminal.
10. Cleaning up.

References:

- `/opt/mellanox/doca/samples/doca_apsh/apsh_modules_get/apsh_modules_get_sample.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_modules_get/apsh_modules_get_main.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_modules_get/meson.build`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_common.c`; `/opt/mellanox/doca/samples/doca_apsh/apsh_common.h`

14.4.10.6.3.3 Apsh Pslist

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of running processes on a monitored system.
The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCI device via given VUID.
5. Creating DOCA Apsh system handler.
6. Setting fields and starting Apsh system handler.
7. Getting the list of processes running on the system using `doca_apsh_processes_get` Apsh API call.
8. Querying the processes for 4 chosen attributes using `doca_apsh_proc_info_get` Apsh API call.
9. Printing the attributes of up to 5 processes to the terminal.
10. Cleaning up.

References:

- `/opt/mellanox/doca/samples/doca_apsh/apsh_pslist/apsh_pslist_sample.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_pslist/apsh_pslist_main.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_pslist/meson.build`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_common.c`; `/opt/mellanox/doca/samples/doca_apsh/apsh_common.h`

14.4.10.6.3.4 Apsh Threads Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of threads of a specific process.
The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCI device via given VUID.
5. Creating DOCA Apsh system handler.

6. Setting fields and starting Apsh system handler.
7. Getting the list of system processes using Apsh API and searching for a specific process with the given PID.
8. Getting the list of process threads using `doca_apsh_threads_get` Apsh API call.
9. Querying the threads for up to 3 selected fields using `doca_apsh_thread_info_get` Apsh API call.
10. Printing thread attributes to the terminal.
11. Cleaning up.

References:

- `/opt/mellanox/doca/samples/doca_apsh/apsh_threads_get/apsh_threads_get_sample.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_threads_get/apsh_threads_get_main.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_threads_get/meson.build`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_common.c` ; `/opt/mellanox/doca/samples/doca_apsh/apsh_common.h`

14.4.10.6.3.5 Apsh Vads Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of virtual address descriptors (VADs) of a specific process.
The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and start the Apsh context.
4. Opening DOCA remote PCI device via given VUID.
5. Creating DOCA Apsh system handler.
6. Setting fields and starting Apsh system handler.
7. Getting the list of system processes using Apsh API and searching for a specific process with the given PID.
8. Getting the list of process VADs using `doca_apsh_vads_get` Apsh API call.
9. Querying the VADs for 3 selected fields using `doca_apsh_vad_info_get` Apsh API call.
10. Printing the attributes of up to 5 VADs to the terminal.
11. Cleaning up.

References:

- `/opt/mellanox/doca/samples/doca_apsh/apsh_vads_get/apsh_vads_get_sample.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_vads_get/apsh_vads_get_main.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_vads_get/meson.build`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_common.c` ; `/opt/mellanox/doca/samples/doca_apsh/apsh_common.h`

14.4.10.6.3.6 Apsh Envars Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of environment variables of a specific process.

⚠ This sample works only on target systems with Windows OS.

The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCIe device via given VUID.
5. Creating DOCA Apsh system handler.
6. Setting fields and starting Apsh system handler.
7. Getting the list of system processes using Apsh API and searching for a specific process with the given PID.
8. Getting the list of process envars using `doca_apsh_envars_get` Apsh API call.
9. Querying the envars for 2 selected fields using `doca_apsh_envar_info_get` Apsh API call.
10. Printing the envars attributes to the terminal.
11. Cleaning up.

References:

- `/opt/mellanox/doca/samples/doca_apsh/apsh_envars_get/apsh_envars_get_sample.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_envars_get/apsh_envars_get_main.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_envars_get/meson.build`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_common.c` ; `/opt/mellanox/doca/samples/doca_apsh/apsh_common.h`

14.4.10.6.3.7 Apsh Privileges Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of privileges of a specific process.

⚠ This sample works only on target systems with Windows OS.

The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCIe device via given VUID.
5. Creating DOCA Apsh system handler.
6. Setting fields and starting Apsh system handler.
7. Getting the list of system processes using Apsh API and searching for a specific process with the given PID.
8. Getting the list of process privileges using the `doca_apsh_privileges_get` Apsh API call.
9. Querying the privileges for 5 selected fields using the `doca_apsh_privilege_info_get` Apsh API call.
10. Printing the privileges attributes to the terminal.
11. Cleaning up.

References:

- `/opt/mellanox/doca/samples/doca_apsh/apsh_privileges_get/apsh_privileges_get_sample.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_privileges_get/apsh_privileges_get_main.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_privileges_get/meson.build`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_common.c`; `/opt/mellanox/doca/samples/doca_apsh/apsh_common.h`

14.4.10.6.3.8 Apsh Containers Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of running containers on a monitored system, as well as getting a list of processes for each container.

> ⚠ This sample works only on target systems with Linux OS.

The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCIe device using specific VUID.
5. Creating DOCA Apsh system handler.
6. Setting fields and starting Apsh system handler.
7. Getting the list of containers running on the system using `doca_apsh_containers_get` Apsh API call.
8. Querying the containers for container ID attribute using `doca_apsh_container_info_get` Apsh API call.
9. Getting list of processes for each container using `doca_apsh_container_processes_get` Apsh API call.
10. Printing the attributes of up to 5 processes to the terminal.
11. Cleaning up.

References:

- `/opt/mellanox/doca/samples/doca_apsh/apsh_containers_get/apsh_containers_get_sample.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_containers_get/apsh_containers_get_main.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_containers_get/meson.build`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_common.c`; `/opt/mellanox/doca/samples/doca_apsh/apsh_common.h`

14.4.11 DOCA Compress

This guide provides instructions on how to use the DOCA Compress API.

14.4.11.1 Introduction

DOCA Compress library provides an API to compress and decompress data using hardware acceleration, supporting both host and NVIDIA® BlueField® DPU memory regions.

The library provides an API for executing compress operations on DOCA buffers, where these buffers reside in either the DPU memory or host memory.

Using DOCA Compress, compress and decompress memory operations can be easily executed in an optimized, hardware-accelerated manner.

This document is intended for software developers wishing to accelerate their application's compress memory operations.

14.4.11.2 Prerequisites

The DOCA Compress library follows the architecture of a DOCA Core Context. It is recommended to read the following sections before proceeding:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

14.4.11.3 Changes From Previous Releases

14.4.11.3.1 Changes in 2.8

The following subsection(s) detail the `doca_compress` library updates in version 2.8.0.

14.4.11.3.1.1 Removed

- `doca_compress_cap_task_decompress_lz4_is_supported`
- `doca_compress_cap_task_decompress_lz4_get_max_buf_size`
- `doca_compress_cap_task_decompress_lz4_get_max_buf_list_len`
- `doca_compress_task_decompress_lz4_set_conf`
- `doca_compress_task_decompress_lz4_alloc_init`
- `doca_compress_task_decompress_lz4_as_task`
- `doca_compress_task_decompress_lz4_set_src`
- `doca_compress_task_decompress_lz4_get_src`
- `doca_compress_task_decompress_lz4_set_dst`
- `doca_compress_task_decompress_lz4_get_dst`
- `doca_compress_task_decompress_lz4_get_crc_cs`
- `doca_compress_task_decompress_lz4_get_adler_cs`

14.4.11.4 Environment

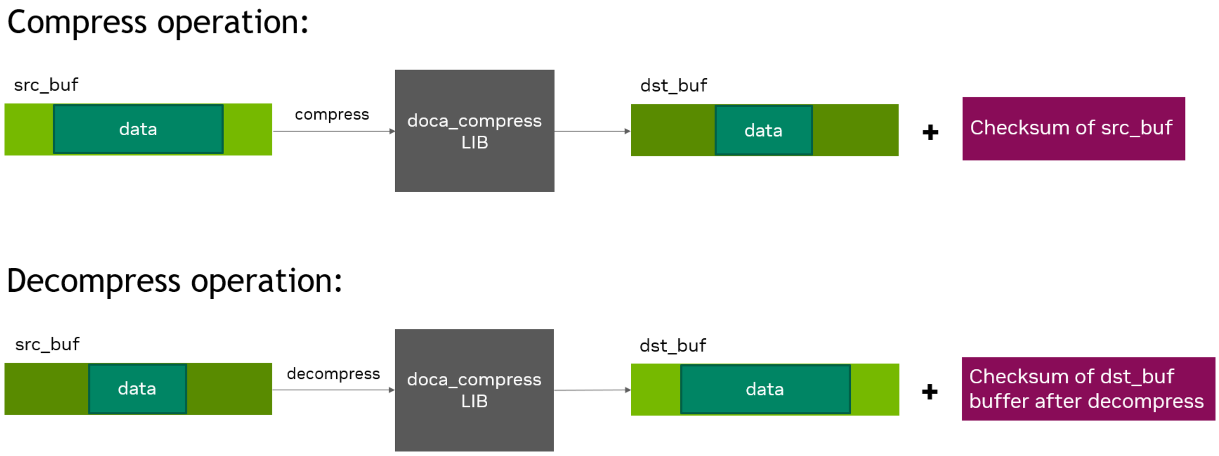
DOCA Compress-based applications can run either on the host machine or on the BlueField DPU target.

Compress can only be run with a DPU configured with DPU mode as described in NVIDIA BlueField Modes of Operation.

14.4.11.5 Architecture

DOCA Compress is a DOCA Context as defined by DOCA Core. See NVIDA DOCA Core Context for more information.

DOCA Compress leverages DOCA Core architecture to expose asynchronous tasks that are offloaded to hardware.



14.4.11.5.1 Supported Compress/Decompress Algorithms

For BlueField-2 devices, this library supports:

- Compress operation using the deflate algorithm
- Decompress operation using the deflate algorithm

For BlueField-3 devices, this library supports:

- Decompress operation using the deflate algorithm
- Decompress operation using the LZ4 algorithm

14.4.11.5.2 Supported Checksum Methods

Depending on the task type, the following checksum methods are produced and may be retrieved using the relevant getter functions:

- Adler - produced by the deflate compress and decompress tasks
- CRC - produced by all tasks
- xxHash - produced by the LZ4 decompress tasks

Refer to "Tasks" section for more information.

14.4.11.5.3 Objects

14.4.11.5.3.1 Device and Device Representor

The library requires a DOCA device to operate, the device is used to access memory and perform the actual copy. See DOCA Core Device Discovery for information.

For same BlueField DPU, it does not matter which device is used (PF/VF/SF), as all these devices utilize the same hardware component. If there are multiple DPUs, it is possible to create a Compress instance per DPU, providing each instance with a device from a different DPU.

To access memory that is not local (from the host to the DPU or vice versa), then the DPU side of the application must pick a device with an appropriate representor. See DOCA Core Device Representor Discovery.

The device must stay valid as long as the Compress instance is not destroyed.

14.4.11.5.3.2 Memory Buffers

All compress/decompress tasks require two DOCA buffers containing the destination and the source. Depending on the allocation pattern of the buffers, refer to the Inventory Types table.

Buffers must not be modified or read during the compress/decompress operation.

14.4.11.5.4 Source and Destination Location

DOCA Compress can process DOCA buffers that reside on the host, the DPU, or both.

14.4.11.5.4.1 Local Host

Source and destination buffers reside on the host and the compress library runs on the host.

14.4.11.5.4.2 Local DPU

Source and destination buffers reside on the DPU and the compress library runs on the DPU.

14.4.11.5.4.3 Remote

Source at Host, Destination at DPU

- The source resides on the host and is exported (DOCA mmap export) to the DPU
- The destination resides on the DPU
- The compress library runs on the DPU and compresses/decompresses the host source to the DPU destination

Source at DPU, Destination at Host

- The source resides on the DPU
- The destination resides on the host and is exported (DOCA mmap export) to the DPU
- Compress library runs on the DPU and compresses/decompresses the DPU source to the host destination

14.4.11.6 Configuration Phase

To start using the library, the user must go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context, to allow execution of tasks and retrieval of events.

14.4.11.6.1 Configurations

The context can be configured to match the use case of the application.

To find if a configuration is supported or what its min/max value is, refer to Device Support.

14.4.11.6.1.1 Mandatory Configurations

The following configurations must be set by the application before attempting to start the context:

- At least one task/event type must be configured. See configuration of Tasks.
- A device with appropriate support must be provided upon creation

14.4.11.6.2 Device Support

DOCA Compress requires a device to operate. To pick a device, see DOCA Core Device Discovery.

As device capabilities may change in the future (see DOCA Core Device Support), it is recommended to select your device using the following APIs:

14.4.11.6.2.1 Supported Tasks

- `doca_compress_cap_task_compress_deflate_is_supported`
- `doca_compress_cap_task_decompress_deflate_is_supported`
- `doca_compress_cap_task_decompress_lz4_stream_is_supported`
- `doca_compress_cap_task_decompress_lz4_block_is_supported`

14.4.11.6.2.2 Supported Buffer Size

- `doca_compress_cap_task_compress_deflate_get_max_buf_size`
- `doca_compress_cap_task_decompress_deflate_get_max_buf_size`
- `doca_compress_cap_task_decompress_lz4_stream_get_max_buf_size`
- `doca_compress_cap_task_decompress_lz4_block_get_max_buf_size`

14.4.11.6.3 Buffer Support

Tasks support buffers with the following features:

Buffer Type	Source Buffer	Destination Buffer
Linked List Buffer	Yes	No
Local mmap Buffer	Yes	Yes

Buffer Type	Source Buffer	Destination Buffer
mmap From PCI Export Buffer	Yes	Yes
mmap From RDMA Export Buffer	No	No

14.4.11.7 Execution Phase

This section describes execution on CPU or DPU using DOCA Core Progress Engine.

14.4.11.7.1 Tasks

14.4.11.7.1.1 Compress Deflate Task

This task facilitates compressing memory, with the deflate algorithm, using buffers as described in section "Buffer Support".

> ⚠ DOCA compress returns only the payload. To create a compressed file, (e.g., gzip), the developer must add a gzip header/trailer.

Task Configuration

Description	API to set the configuration	API to query support
Enable the task	`doca_compress_task_compress_deflate_set_conf`	`doca_compress_cap_task_compress_deflate_is_supported`
Number of tasks	`doca_compress_task_compress_deflate_set_conf`	`doca_compress_get_max_num_tasks` `(max total num tasks)`
Maximal buffer size	-	`doca_compress_cap_task_compress_deflate_get_max_buf_size`
Maximum buffer list size	-	`doca_compress_cap_task_compress_deflate_get_max_buf_list_len`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer pointing to the memory to be compressed	Only the data residing in the data segment is compressed
Destination buffer	Buffer pointing to where compressed memory will be stored	The data is compressed to the tail segment extending the data segment

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The source data is compressed to destination
- The destination buffer data segment is extended to include the compressed data
- Adler can be retrieved by calling `doca_compress_task_compress_deflate_get_adler_cs`
- CRC can be retrieved by calling `doca_compress_task_compress_deflate_get_crc_cs`

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the source and destination should not be read/written to
- Source and destination must not overlap
- Other limitations are described in DOCA Core Task

14.4.11.7.1.2 Decompress Deflate Task

This task facilitates decompressing memory, with the deflate algorithm, using buffers as described in section "Buffer Support".

> ⚠ DOCA decompress expects the payload alone. To decompress a file (e.g. gzip), the developer must strip the header/trailer.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_compress_task_decompress_deflate_set_conf`	`doca_compress_cap_task_decompress_deflate_is_supported`
Number of tasks	`doca_compress_task_decompress_deflate_set_conf`	`doca_compress_get_max_num_tasks (max-total-num-tasks)`
Maximal buffer size	-	`doca_compress_cap_task_decompress_deflate_get_max_buf_size`
Maximum buffer list size	-	`doca_compress_cap_task_decompress_deflate_get_max_buf_list_len`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
source buffer	Buffer pointing to the memory to be decompressed	Only the data residing in the data segment is decompressed
destination buffer	Buffer pointing to where decompressed memory will be stored	The data is decompressed to the tail segment extending the data segment

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The source data is decompressed to destination
- The destination buffer data segment is extended to include the decompressed data
- Adler can be retrieved by calling `doca_compress_task_decompress_deflate_get_adler_cs`
- CRC can be retrieved by calling `doca_compress_task_decompress_deflate_get_crc_cs`

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the source and destination should not be read/written to
- Source and destination must not overlap
- Other limitations are described in DOCA Core Task

14.4.11.7.1.3 Decompress LZ4 Tasks

These tasks facilitate decompressing memory with the LZ4 algorithm, using buffers as described in section "Buffer Support".

The main differences between the tasks is the input data format -

- The decompress LZ4 stream task expects a stream of one or more blocks, without the frame (i.e., the magic number, frame descriptor, and content checksum)
- The decompress LZ4 block task expects a single, compressed, data-only block (i.e., without block size or block checksum)

Decompress LZ4 Stream Task

This task facilitates decompressing memory with the LZ4 algorithm, using buffers as described in section "Buffer Support".

⚠ The decompress LZ4 stream task expects a stream of one or more blocks without the frame (i.e., the magic number, frame descriptor, and content checksum).

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_compress_task_decompress_lz4_stream_set_conf`	`doca_compress_cap_task_decompress_lz4_stream_is_supported`
Number of tasks	`doca_compress_task_decompress_lz4_stream_set_conf`	`doca_compress_get_max_num_tasks` `(max total num tasks)`
Maximal buffer size	-	`doca_compress_cap_task_decompress_lz4_stream_get_max_buf_size`
Maximum buffer list size	-	`doca_compress_cap_task_decompress_lz4_stream_get_max_buf_list_len`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Has block checksum Flag	A flag to indicate whether or not the blocks in the stream have a checksum	1 if the task should expect blocks in the stream to have a checksum; 0 otherwise
Are blocks independent flag	A flag to indicate whether or not each block depends on previous blocks in the stream	1 the the task should expect blocks to be independent; 0 otherwise (dependent blocks)
Source buffer	Buffer pointing to the memory to be decompressed	Only the data residing in the data segment is decompressed
Destination buffer	Buffer pointing to where decompressed memory will be stored	The data is decompressed to the tail segment extending the data segment

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully:

- The source data is decompressed to destination
- The destination buffer data segment is extended to include the decompressed data
- CRC can be retrieved by calling `doca_compress_task_decompress_lz4_stream_get_crc_cs`
- xxHash can be retrieved by calling `doca_compress_task_decompress_lz4_stream_get_xxh_cs`

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the source and destination should not be read/written to
- Source and destination must not overlap
- Other limitations are described in DOCA Core Task

Decompress LZ4 Block Task

This task facilitates decompressing memory with the LZ4 algorithm, using buffers as described in section "Buffer Support".

> ⚠ The decompress LZ4 block task expects a single, compressed, data-only block (i.e., without block size or block checksum).

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_compress_task_decompress_lz4_block_set_conf`	`doca_compress_cap_task_decompress_lz4_block_is_supported`
Number of tasks	`doca_compress_task_decompress_lz4_block_set_conf`	`doca_compress_get_max_num_tasks (max total num tasks)`
Maximal buffer size	-	`doca_compress_cap_task_decompress_lz4_block_get_max_buf_size`
Maximum buffer list size	-	`doca_compress_cap_task_decompress_lz4_block_get_max_buf_list_len`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer pointing to the memory to be decompressed	Only the data residing in the data segment will be decompressed
Destination buffer	Buffer pointing to where decompressed memory will be stored	The data is decompressed to the tail segment extending the data segment

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully:

- The source data is decompressed to destination
- The destination buffer data segment is extended to include the decompressed data
- CRC can be retrieved by calling `doca_compress_task_decompress_lz4_block_get_crc_cs`
- xxHash can be retrieved by calling `doca_compress_task_decompress_lz4_bloxk_get_xxh_cs`

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the source and destination should not be read/written to
- Source and destination must not overlap
- Other limitations are described in DOCA Core Task

14.4.11.7.2 Events

DOCA Compress exposes asynchronous events to notify about changes that happen unexpectedly according to DOCA Core architecture.

The only events DOCA Compress expose are common events (DOCA CTX state changed). See more info in DOCA Core Event.

14.4.11.8 State Machine

The DOCA Compress library follows the Context state machine described in DOCA Core Context State Machine.

This section describes how to move states and what is allowed in each state.

14.4.11.8.1 States

14.4.11.8.1.1 Idle

In this state, it is expected that application:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to Configurations
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop after making sure all tasks have been freed
Stopping	Call progress until all tasks are completed and freed

14.4.11.8.1.2 Starting

This state cannot be reached.

14.4.11.8.1.3 Running

In this state, it is expected that application:

- Allocates and submit tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocate previously configured task
- Submit a task
- Call stop

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

14.4.11.8.1.4 Stopping

In this state, it is expected that application:

- Calls progress to complete all inflight tasks (tasks will complete with failure)
- Frees any completed tasks

Allowed operations:

- Call progress

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and fatal error occurs
Running	Call stop without freeing all tasks

14.4.11.9 Alternative Datapath Options

DOCA Compress only supports datapath on CPU, see Execution Phase.

14.4.11.10 DOCA Compress Samples

The following samples illustrate how to use the DOCA Compress API to compress and decompress files.

> ⚠ DOCA Compress handles payload only unless the `zc` flag is used (available only for deflate samples). In that case, a zlib header and trailer are added in compression and it is considered as part of the input when decompressing.

> ⓘ All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

14.4.11.10.1 Running the Sample

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample:

   ```
   cd /opt/mellanox/doca/samples/doca_compress/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build
   ```

   > ⓘ The binary `doca_<sample_name>` is created under `/tmp/build/`.

3. Sample (e.g., `doca_compress_deflate`) usage:
   - Common arguments

     ```
     Usage: doca_<sample_name> [DOCA Flags] [Program Flags]

     DOCA Flags:
       -h, --help                        Print a help synopsis
       -v, --version                     Print program version information
       -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE,
     20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
       --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
     20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
       -j, --json <path>                 Parse all command flags from an input json file

     Program Flags:
       -p, --pci-addr                    DOCA device PCI device address
       -f, --file                        Input file to compress/decompress
       -o, --output                      Output file
       -c, --output-checksum             Output checksum
     ```

   - Sample-specific arguments

Sample	Argument	Description
Compress/Decompress Deflate	`-wf`, `-with-frame`	Write/read a file with a frame, compatible with default zlib settings

Sample	Argument	Description
Decompress LZ4 Stream	`-bc`, `--has-block-checksum`	Flag to indicate if blocks have a checksum
	`-bi`, `--are-blocks-independent`	Flag to indicate if blocks are independent
	`-wf`, `-with-frame`	Read a file compatible with an LZ4 frame

4. For additional information per sample, use the `-h` option:

```
/tmp/build/doca_<sample_name> -h
```

14.4.11.10.2 Samples

14.4.11.10.2.1 Compress/Decompress Deflate

This sample illustrates how to use DOCA Compress library to compress or decompress a file.

The sample logic includes:

1. Locating a DOCA device.
2. Initializing the required DOCA Core structures.
3. Populating DOCA memory map with two relevant buffers; one for the source data and one for the result.
4. Allocating elements in DOCA buffer inventory for each buffer.
5. Allocating and initializing a DOCA Compress deflate task or a DOCA Decompress deflate task.
6. Submitting the task.
7. Running the progress engine until the task is completed.
8. Writing the result into an output file, `out.txt`.
9. Destroying all DOCA Compress and DOCA Core structures.

References:

- `/opt/mellanox/doca/samples/doca_compress/compress_deflate/compress_deflate_sample.c`
- `/opt/mellanox/doca/samples/doca_compress/compress_deflate/compress_deflate_main.c`
- `/opt/mellanox/doca/samples/doca_compress/compress_deflate/meson.build`
- `/opt/mellanox/doca/samples/doca_compress/decompress_deflate/decompress_deflate_sample.c`
- `/opt/mellanox/doca/samples/doca_compress/decompress_deflate/decompress_deflate_main.c`
- `/opt/mellanox/doca/samples/doca_compress/decompress_deflate/meson.build`
- `/opt/mellanox/doca/samples/doca_compress/compress_common.h`
- `/opt/mellanox/doca/samples/doca_compress/compress_common.c`

14.4.11.10.2.2 Decompress LZ4 Stream

This sample illustrates how to use DOCA Compress library to decompress a file using the LZ4 stream decompress task.

The sample logic includes:

1. Locating a DOCA device.
2. Initializing the required DOCA Core structures.
3. Populating DOCA memory map with two relevant buffers; one for the source data and one for the result.
4. Allocating elements in DOCA buffer inventory for each buffer.
5. Allocating and initializing an DOCA Decompress LZ4 stream task.
6. Submitting the task.
7. Running the progress engine until the task is completed.
8. Writing the result into an output file, `out.txt`.
9. Destroying all DOCA Compress and DOCA Core structures.

References:

- `/opt/mellanox/doca/samples/doca_compress/decompress_lz4_stream/decompress_lz4_stream_sample.c`
- `/opt/mellanox/doca/samples/doca_compress/decompress_lz4_stream/decompress_lz4_stream_main.c`
- `/opt/mellanox/doca/samples/doca_compress/decompress_lz4_stream/meson.build`
- `/opt/mellanox/doca/samples/doca_compress/compress_common.h`
- `/opt/mellanox/doca/samples/doca_compress/compress_common.c`

14.4.11.10.3 Backward Compatibility

14.4.11.10.3.1 Decompress LZ4 Task

The decompress LZ4 task has been removed. To facilitates decompressing memory with the LZ4 algorithm, use the decompress LZ4 stream task or the decompress LZ4 block task instead.

14.4.12 DOCA SHA

This guide provides instructions on building and developing applications that calculate message digest using the SHA1, SHA2-256 or SHA2-512 algorithms.

14.4.12.1 Introduction

> ⚠ The DOCA SHA library is currently supported at alpha level.

The library provides an API for executing SHA operations on DOCA buffers, where the buffers reside in either local memory (i.e., within the same host) or host memory accessible by the NVIDIA® BlueField®-2 device (remote memory). Using DOCA SHA, complex cryptographic hash operations can be easily executed in an optimized, hardware-accelerated manner.

⚠ NVIDIA® BlueField®-3 does not support this library because it has no SHA acceleration engine.

This document is intended for software developers wishing to accelerate their applications' SHA calculations typically used in digital signature schemes or hash-based message authentication code calculations.

14.4.12.2 Prerequisites

This library follows the architecture of a DOCA Core context, it is recommended to read the following sections before:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

14.4.12.3 Environment

DOCA SHA-based applications can run either on the host machine or on the BlueField-2 DPU target.

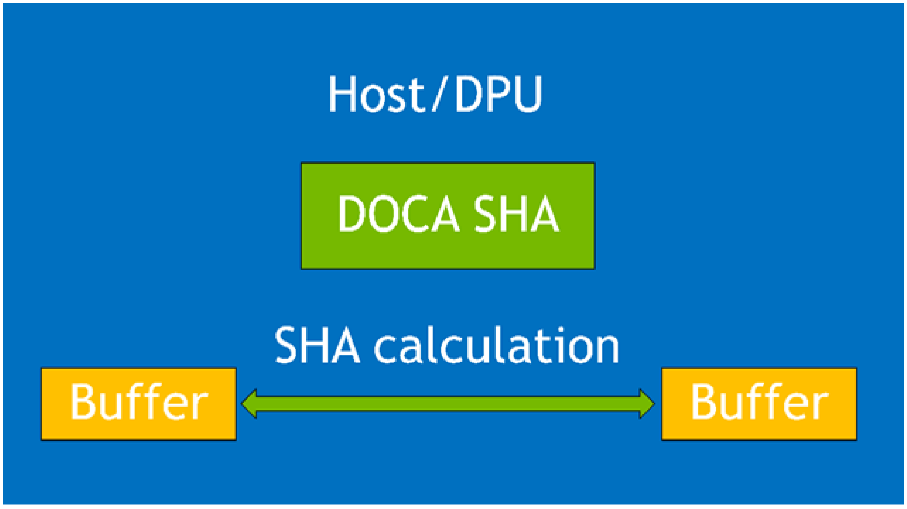
DOCA SHA calculations from the host to BlueField and vice versa can only be run when the DPU is configured in DPU mode.

14.4.12.4 Architecture

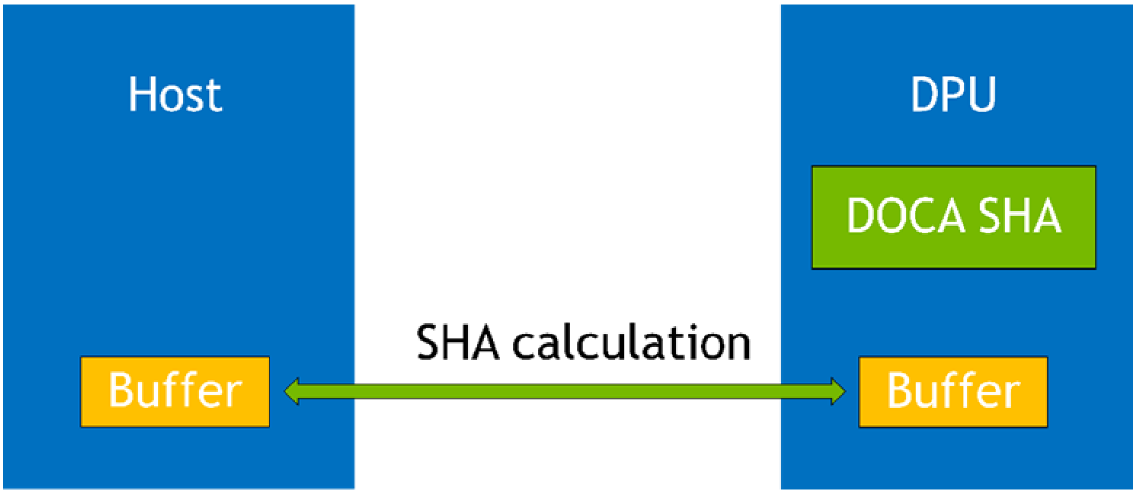
DOCA SHA is a DOCA Core Context. This library leverages the DOCA Core architecture to expose asynchronous tasks/events offloaded to hardware.

SHA can be used to calculate message digest as illustrated in the following diagrams:

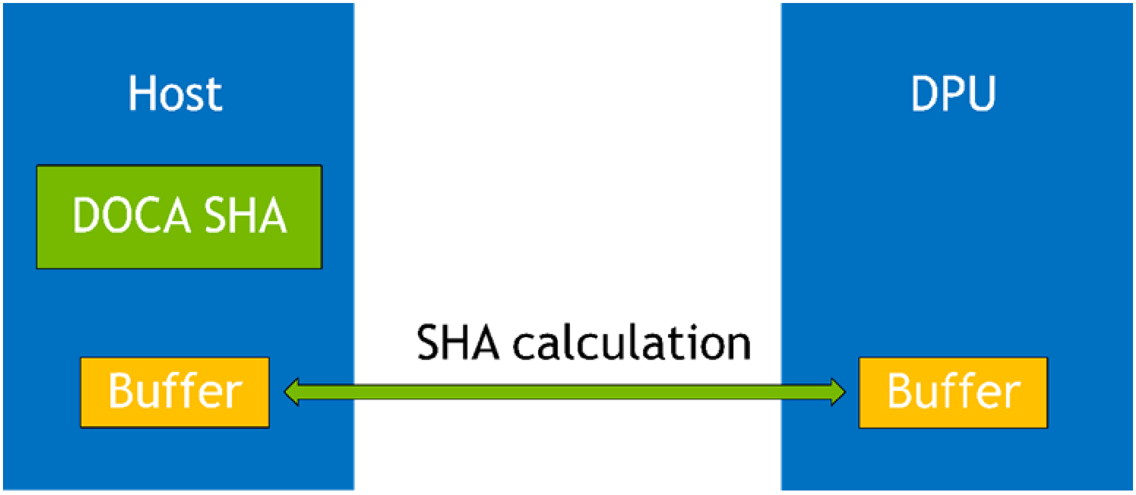
- SHA from local memory to local memory:



- Using the DPU to do SHA using the memory between the host and the DPU:



- Using the host to do SHA calculation using memory between the host and the DPU:



14.4.12.4.1 Objects

14.4.12.4.1.1 Device and Representor

The library requires a DOCA device to operate. The device is used to access memory and perform the actual SHA calculation. See DOCA Core Device Discovery.

For the same BlueField DPU, it does not matter which device is used (i.e., PF/VF/SF) as these devices utilize the same hardware component. If there are multiple DPUs, then it is possible to create a SHA instance per DPU, providing each instance with a device from a different DPU.

To access non-local memory (i.e., from the host to DPU or vice versa), the DPU side of the application must choose a device with an appropriate representor (see DOCA Core Device Representor Discovery). The device must stay valid for as long as the SHA instance is not destroyed.

14.4.12.4.1.2 Memory Buffers

The SHA task requires at least two DOCA buffers containing the destination and the source. Depending on the allocation pattern of the buffers, refer to the DOCA Core Inventory Types table.

Buffers must not be modified or read during the SHA operation. For information on what kind of memory is supported, refer to the table in section "Buffer Support".

14.4.12.5 Configuration Phase

To start using the library, users must go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context to allow the execution of tasks and retrieval of events.

14.4.12.5.1 Configurations

The context can be configured to match the application use case.

To find if a configuration is supported or its min/max value, refer to section "Device Support".

14.4.12.5.1.1 Mandatory Configurations

These configurations must be set by the application before attempting to start the context:

- At least one task/event type must be configured. See configuration of tasks and/or events in sections "Tasks" and "Events" respectively for information.
- A device with appropriate support must be provided upon creation

14.4.12.5.2 Device Support

DOCA SHA requires a device to operate. For information on choosing a device, see DOCA Core Device Discovery.

As device capabilities may change in the future (see DOCA Core Device Support) it is recommended to select your device using the following methods:

- `doca_sha_cap_task_hash_get_supported`
- `doca_sha_cap_task_partial_hash_get_supported`

Some devices can allow different capabilities such as:

- The maximum number of tasks
- The maximum source buffer size
- The minimum destination buffer size
- The maximum supported number of elements in DOCA linked-list buffer
- Check whether SHA1, SHA2-256 or SHA2-512 is supported

14.4.12.5.3 Buffer Support

Tasks support buffers with the following features:

Buffer Type	Source Buffer	Destination Buffer
Local mmap buffer	Yes	Yes
Mmap from PCIe export buffer	Yes	Yes
Mmap from RDMA export buffer	No	No

Buffer Type	Source Buffer	Destination Buffer
Linked list buffer	Yes	No

14.4.12.6 Execution Phase

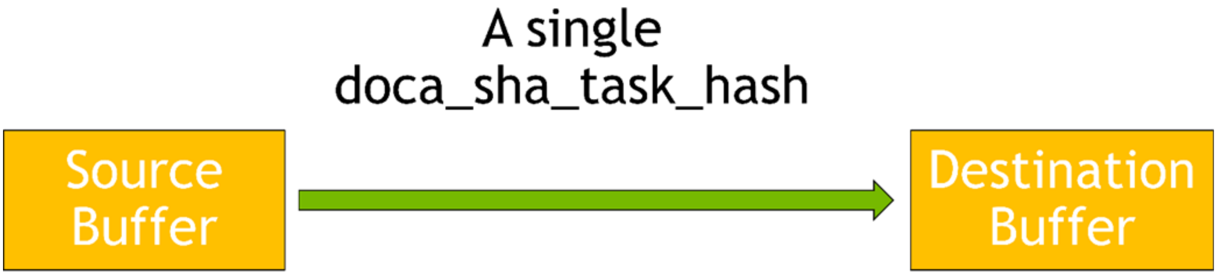
This section describes execution on the CPU using DOCA Core Progress Engine.

14.4.12.6.1 Tasks

DOCA SHA exposes asynchronous tasks that leverage DPU hardware according to DOCA Core architecture.

14.4.12.6.1.1 SHA Task

The SHA task `doca_sha_task_hash` allows one-shot SHA calculation using buffers as described in section "Buffer Support". One-shot means that the source buffer is used as a whole input, therefore, the SHA operation is completed after this task completion event arrives.



Task Configuration

Description	API to Set Configuration	API to Query Support
Enable the task	`doca_sha_task_hash_set_conf`	`doca_sha_cap_task_hash_get_supported`
Number of tasks	`doca_sha_task_hash_set_conf`	
Maximal source buffer size	-	`doca_sha_cap_get_max_src_buf_size`
Maximum source buffer list size	-	`doca_sha_cap_get_max_list_buf_num_elem`
Minimum destination buffer size	-	`doca_sha_cap_get_min_dst_buf_size`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer pointing to the memory to be used for SHA calculation	Only the data residing in the data segment is to be used
Destination buffer	Buffer pointing to the memory used for writing the SHA calculation result	The SHA result is appended to the tail segment
SHA algorithm type	SHA algorithm to be used in SHA calculation	Must be one of `DOCA_SHA_ALGORITHM_SHA1` , `DOCA_SHA_ALGORITHM_SHA256` , `DOCA_SHA_ALGORITHM_SHA512`

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The SHA calculation of data from the source buffer is successfully completed and the result is written to the destination buffer
- The destination buffer data segment is extended to include the SHA result data

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

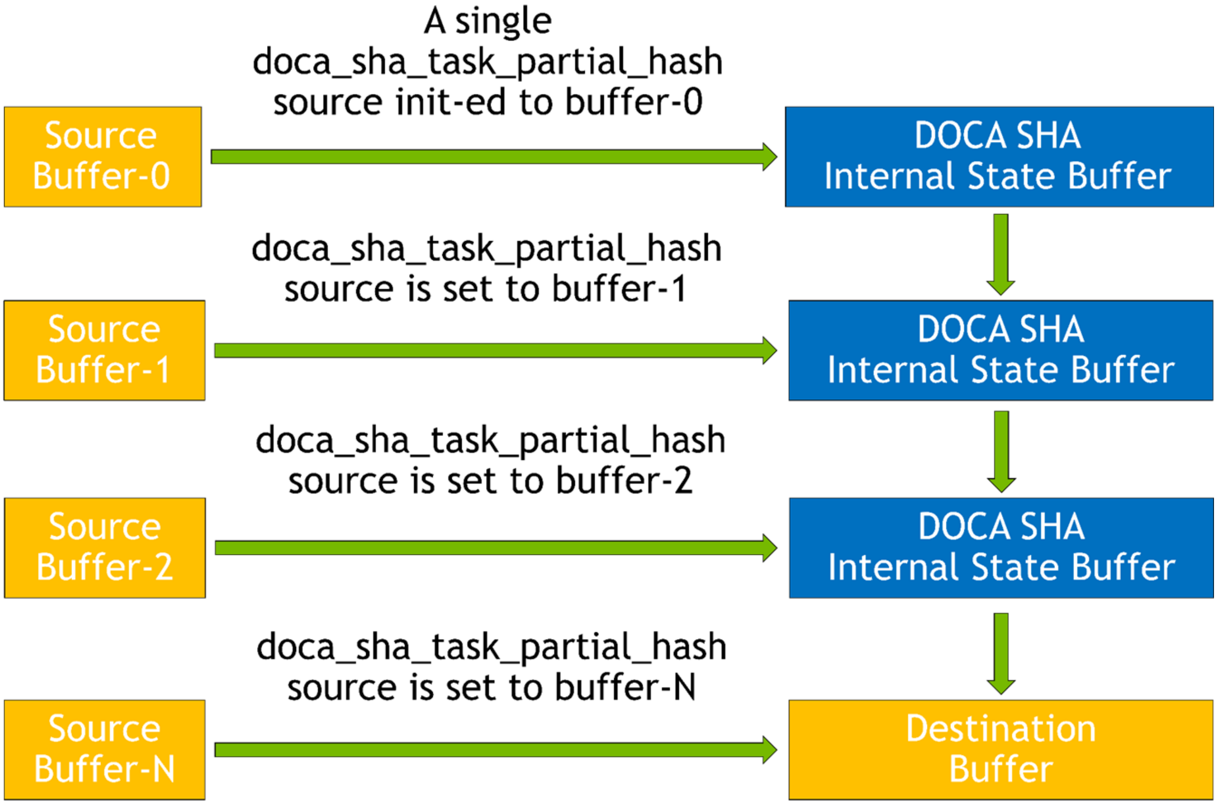
Task Limitations

- The operation is not atomic
- Once the task is submitted, the source and destination should not be read/written to
- Other limitations are described in DOCA Core Task

14.4.12.6.1.2 Partial-SHA Task

The partial-SHA task `doca_sha_task_partial_hash` allows stateful SHA calculation for a collection of messages. Using buffers as described in section "Buffer Support".

Stateful means that the input data is composed of many segments (may be spatial or timely non-consecutive), therefore, its SHA calculation requires more than one one-shot SHA operation to finish. During any stateful operation, other independent SHA tasks can also be executed.


A single
doca_sha_task_partial_hash
source init-ed to buffer-0
Source
Buffer-0
DOCA SHA
Internal State Buffer
doca_sha_task_partial_hash
source is set to buffer-1
Source
Buffer-1
DOCA SHA
Internal State Buffer
doca_sha_task_partial_hash
source is set to buffer-2
Source
Buffer-2
DOCA SHA
Internal State Buffer
doca_sha_task_partial_hash
source is set to buffer-N
Source
Buffer-N
Destination
Buffer

Task Configuration

Description	API to Set Configuration	API to Query Support
Enable the task	`doca_sha_task_partial_hash_set_conf`	`doca_sha_cap_task_partial_hash_get_supported`
Number of tasks	`doca_sha_task_partial_hash_set_conf`	
Maximal source buffer size	-	`doca_sha_cap_get_max_src_buf_size`
Maximum source buffer list size	-	`doca_sha_cap_get_max_list_buf_num_elem`
Minimum destination buffer size	-	`doca_sha_cap_get_min_dst_buf_size`
SHA block size		`doca_sha_cap_get_partial_hash_block_size`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer pointing to the memory to be used for SHA calculation	Only the data residing in the data segment is to be used. And the data length for the non-last data segment must be multiple of the SHA block size queried by `doca_sha_cap_get_partial_hash_block_size`
Destination buffer	Buffer pointing to the memory is used for writing the SHA calculation result	The SHA result is appended to the tail segment. During the whole calculation process, this buffer cannot be modified.
SHA algorithm type	SHA algorithm to be used in SHA calculation	Must be one of `DOCA_SHA_ALGORITHM_SHA1` , `DOCA_SHA_ALGORITHM_SHA256` , `DOCA_SHA_ALGORITHM_SHA512`
Whether the current source buffer is the last segment	Indicate whether the current source Buffer is the last segment data to be used for partial-SHA calculation	Use `doca_sha_task_partial_hash_set_is_final_buf` to set this property
Set source buffer	Use to set the subsequent source segment buffer after the initial `doca_sha_task_partial_hash` task is allocated	`doca_sha_task_partial_hash_set_src`

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The SHA calculation of data from the source buffer is successfully completed and the result is written to the destination buffer
- The destination buffer data segment is extended to include the SHA result data

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects is not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task is submitted, the source and destination should not be read/written to
- Other limitations are described in DOCA Core Task

14.4.12.6.2 Events

DOCA SHA exposes asynchronous events to notify about changes that happen unexpectedly according to the DOCA Core architecture.

The only events SHA exposes are common events as described in DOCA Core Event.

14.4.12.7 State Machine

The DOCA SHA library follows the context state machine as described in DOCA Core Context State Machine.

The following section describes moving states and what is allowed in each state.

14.4.12.7.1 Idle

In this state, it is expected that the application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to section "Configurations"
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop after making sure all tasks have been freed
Stopping	Call progress until all tasks are completed and freed

14.4.12.7.2 Starting

This state cannot be reached.

14.4.12.7.3 Running

In this state, it is expected that the application:

- Allocates and submits tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocating previously configured task
- Submitting a task
- Calling stop

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

14.4.12.7.4 Stopping

In this state, it is expected that the application:

- Calls progress to complete all inflight tasks (tasks complete with failure)
- Frees any completed tasks

Allowed operations:

- Calling progress

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and fatal error occurs
Running	Call stop without freeing all tasks

14.4.12.8 Alternative Datapath Options

DOCA SHA only supports datapath on the CPU. See section "Execution Phase".

14.4.12.9 DOCA SHA Samples

This section describes DOCA SHA samples based on the DOCA SHA library.

The samples in this section illustrate how to use the DOCA SHA API to do the following:

- Do SHA calculation of contents of a buffer, and write result to another buffer
- Chop the contents of a buffer into a collection of segments, and do partial-SHA calculation of this collection of segments, and write result to another

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

14.4.12.9.1 Running the Samples

1. Refer to the following documents:
    - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
    - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample:

```
cd /opt/mellanox/doca/samples/doca_sha/<sample_name>
meson/tmp/build
ninja -C/tmp/build
```

> The binary `doca_<sample_name>` is created under `/tmp/build/` .

3. Sample (e.g., `doca_sha_create`) usage:

```
Usage: doca_sha_create [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICA
L, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags from an input json file

Program Flags:
  -d, --data                        user data
```

4. For additional information per sample, use the `-h` option:

```
/tmp/build/doca_<sample_name>-h
```

14.4.12.9.2 Samples

14.4.12.9.2.1 SHA Create

This sample illustrates how to perform SHA calculation with DOCA SHA.

The sample logic includes:

1. Locating DOCA device.
2. Initializing required DOCA Core structures.
3. Setting the `task_pool` configuration for `doca_sha_task_hash` .
4. Populating DOCA memory map with two relevant buffers.
5. Allocating element in DOCA buffer inventory for each buffer.
6. Allocating and initializing a `doca_sha_task_hash` .
7. Submitting the task.
8. Retrieving task result once it is done.

Reference:

- `/opt/mellanox/doca/samples/doca_sha/sha_create/sha_create_sample.c`
- `/opt/mellanox/doca/samples/doca_sha/sha_create/sha_create_main.c`
- `/opt/mellanox/doca/samples/doca_sha/sha_create/meson.build`

14.4.12.9.2.2 SHA-Partial Create

This sample illustrates how to perform partial-SHA calculation for a collection of data segments with DOCA SHA.

The sample logic includes:

1. Locating DOCA device.

2. Initializing the required DOCA Core structures.
3. Setting the `task_pool` configuration for `doca_sha_task_partial_hash`.
4. Chopping the source data into a collection of data segments according to the selected SHA algorithm's block size
5. Populating DOCA memory map with needed buffers for all source data segments and destination buffer.
6. Allocating element in DOCA buffer inventory for the first source buffer and destination buffer.
7. Allocating and initializing a `doca_sha_task_partial_hash` with the first source buffer and the destination buffer.
8. Iteratively repeating the following sub-steps until all data segments are consumed:
    a. Submitting the `doca_sha_task_partial_hash`.
    b. Waiting for the submitted task to finish.
    c. Allocating a `doca_buf` for the next source segment and use `doca_sha_task_partial_hash_set_src` to set it as source buffer of the above allocated task.
    d. If it is the final segment, use `doca_sha_task_partial_hash_set_is_final_buf` to mark it in the allocate task.
9. Retrieving the result of the final iteration in the destination buffer as the full partial-SHA calculation result.
10. Destroying all SHA and DOCA Core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_sha/sha_partial_create/sha_partial_create_sample.c`
- `/opt/mellanox/doca/samples/doca_sha/sha_partial_create/sha_partial_create_main.c`
- `/opt/mellanox/doca/samples/doca_sha/sha_partial_create/meson.build`

14.4.13 DOCA Erasure Coding

This guide provides instructions on how to use the DOCA Erasure Coding API.

14.4.13.1 Introduction

> ⚠ This library is currently supported at alpha version.

The DOCA Erasure Coding (known also as forward error correction or FEC) library provides an API to encode and decode data using hardware acceleration, supporting both host and NVIDIA® BlueField®-3 (and higher) DPU memory regions.

DOCA Erasure Coding recovers lost data fragments by creating generic redundancy fragments (backup). Each redundancy block that the library creates can help recover any block in the original data should a total loss of fragment occur. This increases data redundancy and reduces data overhead.

The library provides an API for executing erasure coding (EC) operations on DOCA buffers residing in either the DPU or host memory.

This document is intended for software developers wishing to accelerate their application's EC memory operations.

14.4.13.1.1 Glossary

Familiarize yourself with the following terms to better understand the information in this document:

Term	Definition
Data	Original data, original blocks, blocks of original data to be protected/preserved
Coding matrix	Coefficients, the matrix used to generate the redundancy blocks and recovery
Redundancy blocks	Codes; encoded data; the extra blocks that help recover data loss
Encoding	The process of creating the redundancy blocks. Encoded data is referred to as the original blocks or redundancy blocks.
Decoding	The process of recovering the data. Decoded data is referred to as the original blocks alone.

14.4.13.2 Prerequisites

DOCA Erasure Coding library follows the architecture of a DOCA Core Context, it is recommended read the following sections before:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

14.4.13.3 Environment

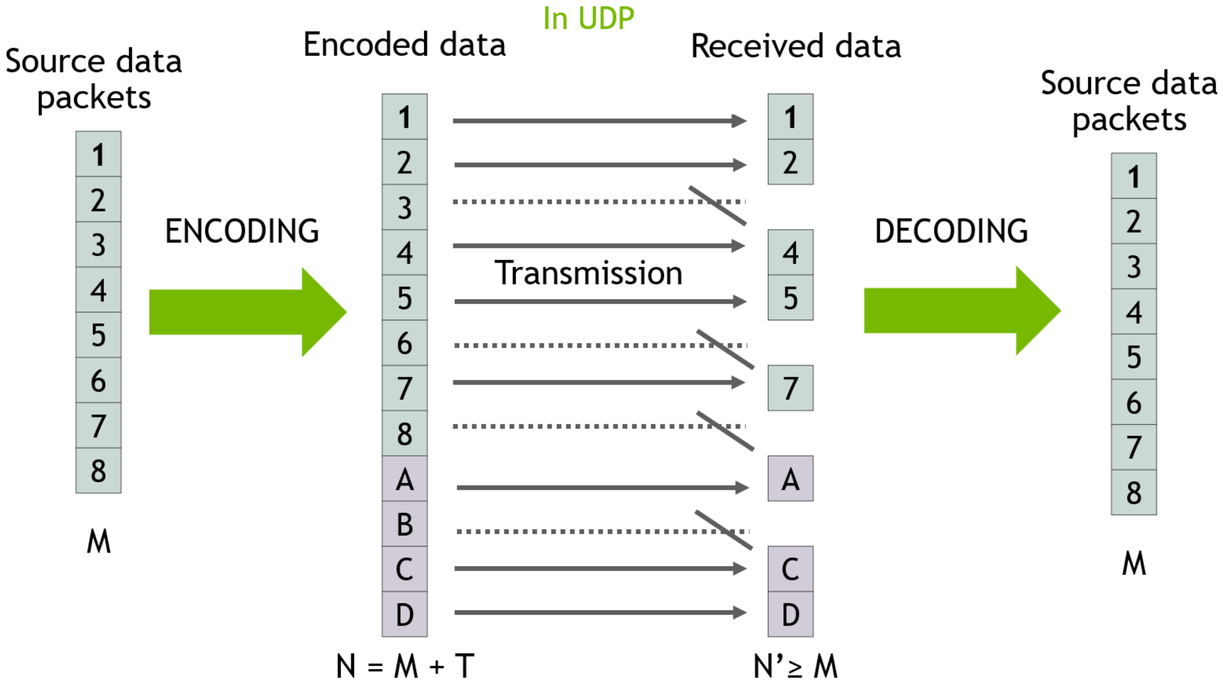
DOCA Erasure Coding-based applications can run either on the host machine or on the DPU target (NVIDIA® BlueField®-3 and above).

Erasure Coding can only be run with DPU configured in DPU mode as described in NVIDIA BlueField Modes of Operation.

14.4.13.4 Architecture

DOCA Erasure Coding is a DOCA Context as defined by DOCA Core. This library leverages the DOCA Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

The following diagram presents a high-level view of the EC transmission flow:

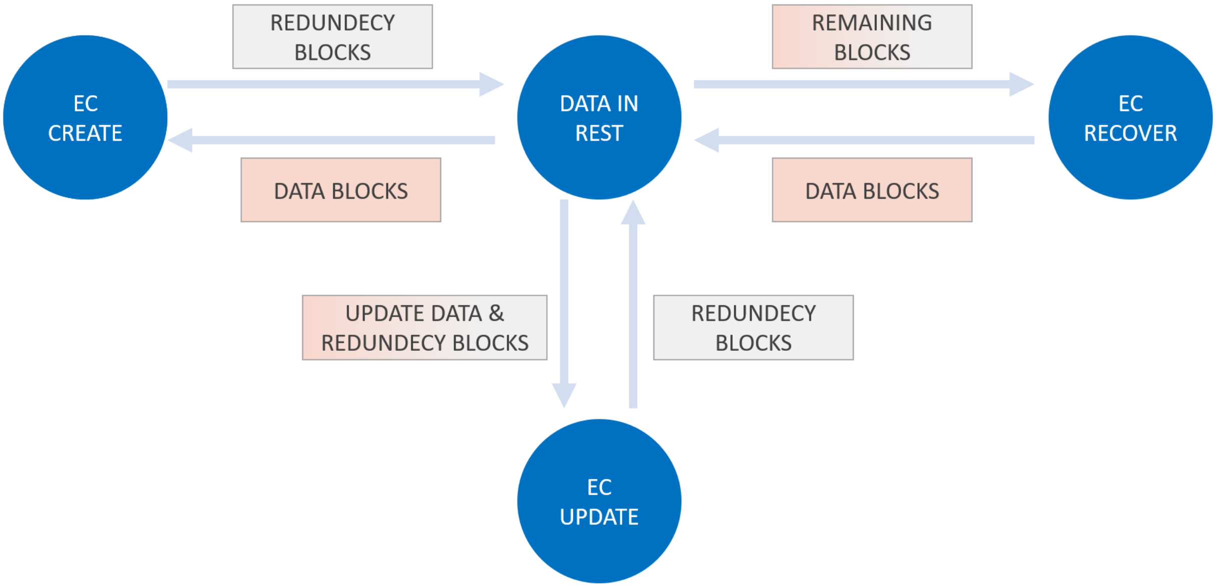


1. M packets are sent from the source (8 in this case).
2. Before the source send them, the source encode the data by adding to it T redundancy packets (4 in this case).
3. The packets are transmitted to the destination in UDP protocol. Some packets are lost and N' packets are received (in this case 4 packets are lost and 8 are received).
4. The destination decodes the data using all the packets available (both original data in green and redundancy data in red) and gets back the M original data packets.

14.4.13.4.1 Flows

Regular EC flow consists of the following elements:

1. Creating redundancy blocks from data (EC create).
2. Updating redundancy blocks from updated data (EC update).
3. Recovering data blocks from redundancy blocks (EC recover).



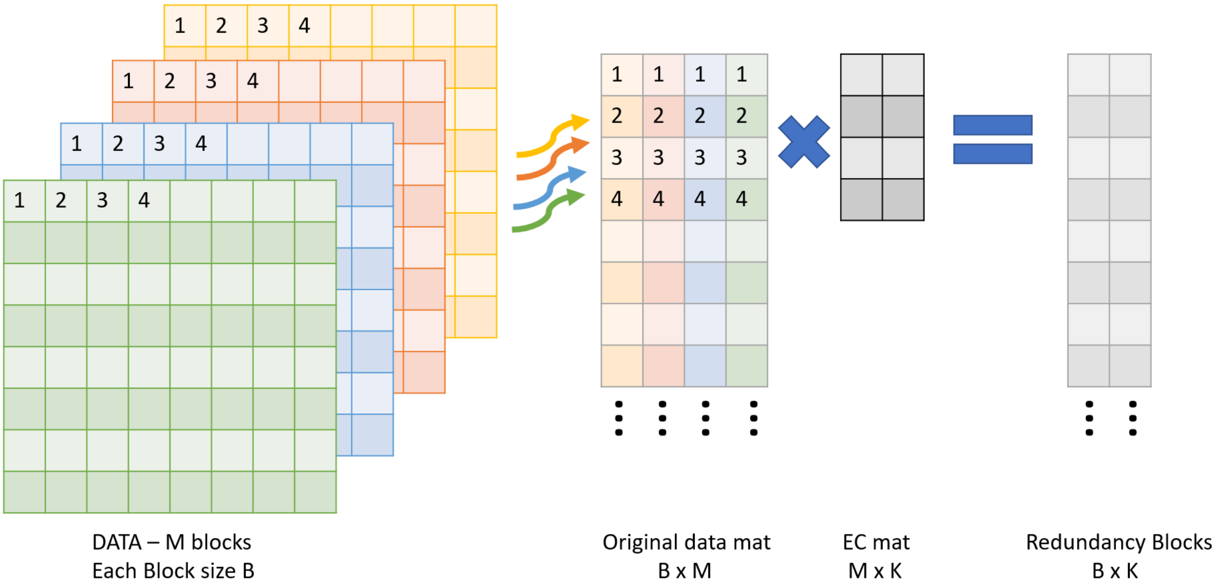
The following sections examine an M:K (where M is the original data and K is redundancy) EC.

14.4.13.4.2 Create Redundancy Blocks

The user must perform the following:

1. Input M data blocks via `doca_buf` (filled with data, each block size B).
2. Output K empty blocks via `doca_buf` (each block size B).
3. Use DOCA Erasure Coding to create a coding matrix of M by K via `doca_buf`.
4. Use DOCA Erasure Coding Create task to get the K output redundancy blocks.

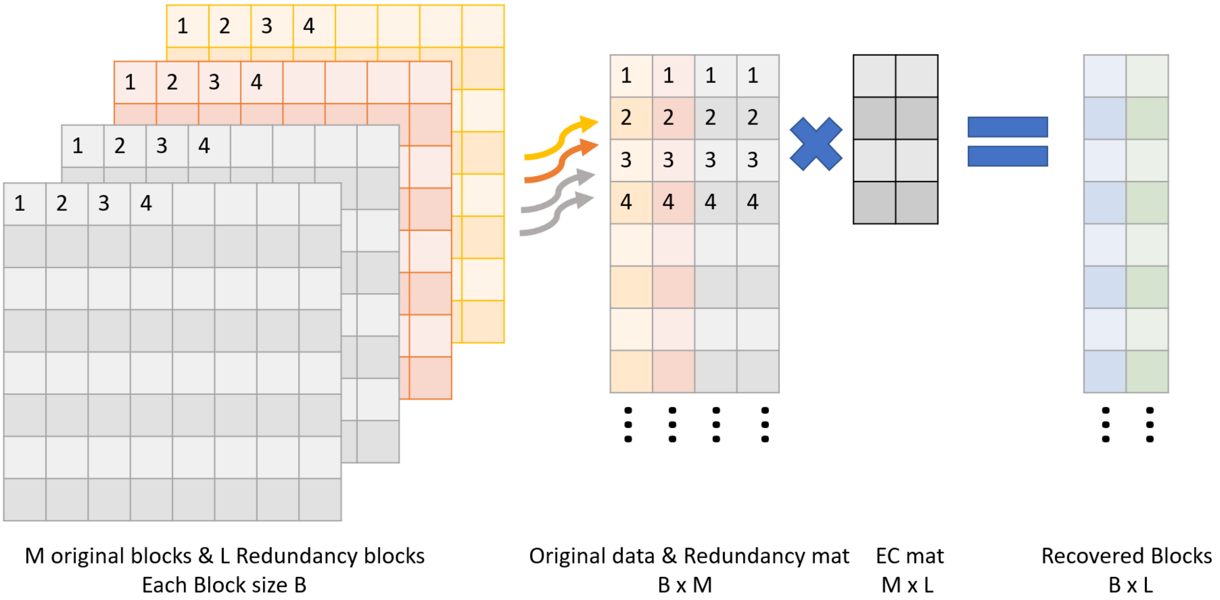
> ⚠ This step can be repeated in a stream use case, as the DPU would not be the recovery or update point.



14.4.13.4.3 Recover Block

The user must perform the following:

1. Input M-L original blocks via `doca_buf` (blocks that were not impaired).
2. Input L≤K (any) redundancy blocks via `doca_buf` (redundancy blocks originating from create/update tasks).
3. Input bitmask or array, indicating which blocks to recover.
4. Output L empty blocks via `doca_buf` (same size of data block).
5. Use DOCA Erasure Coding to create a recover coding matrix of M by L via `doca_buf` (unique per bitmask).
6. Use DOCA Erasure Coding Recover task to get the L output recovered data blocks.



14.4.13.4.4 Objects

14.4.13.4.4.1 Device and Device Representor

The DOCA Erasure Coding library requires a DOCA device to operate. The device is used to access memory and perform the encoding and decoding operations. See DOCA Core Device Discovery.

For same Bluefield card, it does not matter which device is used (PF/VF/SF), as all these devices utilize the same HW component. If there are multiple DPUs, then it is possible to create an EC instance per DPU, providing each instance with a device from a different DPU. To access memory that is not local (from the host to the DPU and vice versa), the DPU side of the application must pick a device with an appropriate representor. See DOCA Core Representor Device Discovery.

The device must stay valid until the EC instance is destroyed.

14.4.13.4.4.2 Memory Buffers

Executing any DOCA EC task requires two DOCA buffers, a source buffer and a destination buffer.

Depending on the allocation pattern of the buffers, refer to the Inventory Types table.

Buffers must not be modified or read during the execution of any task.

14.4.13.5 Configuration Phase

To start using the library, first, you need to go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context, to allow execution of tasks and retrieval of events.

14.4.13.5.1 Configurations

The context can be configured to match the application use case.

To find if a configuration is supported, or what the min/max value, please refer to Device Support.

14.4.13.5.1.1 Mandatory Configurations

These configurations are mandatory and must be set by the application before attempting to start the context:

- At least 1 task/event type needs to be configured. See configuration of Tasks.
- A device with appropriate support must be provided on creation.

14.4.13.5.2 Device Support

DOCA Erasure Coding needs a device to operate. For picking a device, see DOCA Core Device Discovery.

Erasure Coding can be used in BlueField-3 with some limitations (see architecture). Any device can be used PF/VF/SF.

As device capabilities may change in the future, it is recommended to choose your device using the following methods:

- `doca_ec_cap_task_galois_mul_is_supported`
- `doca_ec_cap_task_create_is_supported`
- `doca_ec_cap_task_update_is_supported`
- `doca_ec_cap_task_recover_is_supported`

Some devices can allow different capabilities as follows:

- The maximum buffer list length
- The maximum block size

⚠ Current BlueField-3 limitations:

- Data block count range: 1-128
- Redundancy block count: 1-32
- Block size: 64B-128MB

14.4.13.5.3 Buffer Support

Tasks support buffers with the following features:

Buffer Type	Source Buffer	Destination Buffer
Linked list buffer	Depends on the device; check the `max_buf_list_len` capability	No
Local mmap buffer	Yes	Yes
Mmap from PCIe export buffer	Yes	Yes
Mmap from RDMA export buffer	No	No

14.4.13.6 Execution Phase

This section describes execution on CPU or DPU using the DOCA Core Progress Engine.

14.4.13.6.1 Matrix Generate

All tasks require a coding matrix.

14.4.13.6.1.1 Matrix Type

DOCA EC provides 2 matrix types which are elaborated on in the following subsections.

Cauchy

Cauchy encoding matrix is constructed so that $a_{ij} = \frac{1}{(x_i + y_j)}$.

Where:

- $0 \leq i < \text{number of data blocks}$
- $0 \leq j < \text{number of redundancy blocks}$
- $x_i = i$
- $y_j = j + \text{number of data blocks}$

Vandermonde

Vandermonde encoding matrix is constructed so that $a_{ij} = (i + 1)^j$.

Where:

- $0 \leq i < \text{number of data blocks}$
- $0 \leq j < \text{number of redundancy blocks}$

Vandermonde matrix does not guarantee that every submatrix is invertible (i.e., the decode task may fail in some settings).

14.4.13.6.1.2 Matrix Functionality

Create

An encoding matrix is necessary for executing the create task, to create redundancy blocks.

The matrices used for updates and recovery are based on an encoding matrix.

The following subsections describe the available options for creating matrices.

Generic

Generic creation, with the `doca_ec_matrix_create()` function, is used for simple setup using one of matrix types provided by the library.

Input:

Name	Description
Type	One of matrix types provided by the library
Data block count	The number of original data blocks
Redundancy block count	The number of redundancy blocks

Custom

Custom creation, with the `doca_ec_matrix_create_from_raw()` function, is used if the desired type of matrix is not provided by the library.

Input:

Name	Description	Notes
Data	The data of a coding matrix	The size of the data should be `data_block_count` * `rdnc_block_count`
Data block count	The number of original data blocks	-
Redundancy block count	The number of redundancy blocks	-

Update

This matrix is necessary for executing the update task, to update the redundancy blocks after a change in the data blocks.

The matrix is created using the `doca_ec_matrix_create_update()` function.

Input:

Name	Description	Notes
Coding matrix	A coding matrix created by `doca_ec_matrix_create()` or `doca_ec_matrix_create_from_raw()`	-

Name	Description	Notes
Update indices	An array specifying the indices of the updated data blocks	- The indices must be in ascending order - The indices should match the order of the data blocks in the matrix creation function
Number of updates	The number of updated blocks. The length of the update indices array.	-

Recover

This matrix is necessary for executing the recover task, to recover original data blocks.

The matrix is created using the `doca_ec_matrix_create_recover()` function.

Input:

Name	Description	Notes
Coding matrix	A coding matrix created by `doca_ec_matrix_create()` or `doca_ec_matrix_create_from_raw()`	-
Missing indices	An array specifying the indices of the missing data blocks	- The indices must be in ascending order - The indices should match the order of the data blocks in the matrix creation function
Number of missing	The number of updated blocks. The length of the update indices array.	-

14.4.13.6.2 Tasks

14.4.13.6.2.1 Task Batching

DOCA Erasure Coding supports task batching mode, which is a task submit mode of work that allows aggregating multiple DOCA tasks of the same type and handling them as a single unit.

For more information on task batching, refer to DOCA Core Task.

DOCA Erasure Coding supports the flags `DOCA_TASK_SUBMIT_FLAG_FLUSH` and `DOCA_TASK_SUBMIT_FLAG_OPTIMIZE_REPORTS`.

14.4.13.6.2.2 Galois Mul Task

This task executes Galois multiplication between the original blocks and the coding matrix.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_ec_task_galois_mul_set_conf`	`doca_ec_cap_task_galois_mul_is_supported`
Maximum block size	-	`doca_ec_cap_get_max_block_size`
Maximum buffer list length	-	`doca_ec_cap_get_max_buf_list_len`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
coding matrix	A coding matrix as created by `doca_ec_matrix_create()` or `doca_ec_matrix_create_from_raw()`	-
source buffer	Source original data buffer, holding a sequence containing all original blocks (e.g., `block_1` , `block_2` , etc.); the order matters	• The data length of `src_buf` should be a multiplication of the block size • The data length should also be aligned to 64B and with a minimum size of 64B
destination buffer	A destination buffer for the multiplication outcome blocks. The sequence containing all multiplication outcome blocks (`dst_block_1` , `dst_block_2` , etc.) is written to it upon successful completion of the task.	• The data is written to the tail segment extending the data segment • The minimal available memory in `dst_buf` should be the number of redundancy blocks * the block size, aligned to 64B and, in any case, at least 64B.

Example for required buffer length

If a Galois multiplication task matrix is 10x4 (i.e., 10 original blocks, 4 multiplication outcome blocks), and the block size is 64KB:

- `src_buf` data length should be 10x64KB = 640KB
- The available memory for writing in `dst_buf` should be at least 4x64KB = 256KB

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The destination buffer holds a sequence containing all multiplication outcome blocks (e.g., `dst_block_1` , `dst_block_2` , etc.)
- The destination buffer data segment is extended to include the outcome blocks

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the source and destination buffer should not be read from/written to
- Source and destination buffers must not overlap
- Other limitations are described in DOCA Core Task

14.4.13.6.2.3 Create Task

This task creates redundancy blocks for the given original data blocks using a given coding matrix.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_ec_task_create_set_conf`	`doca_ec_cap_task_create_is_supported`
Maximum block size	-	`doca_ec_cap_get_max_block_size`
Maximum buffer list length	-	`doca_ec_cap_get_max_buf_list_len`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
coding matrix	A coding matrix created by `doca_ec_matrix_create()` or `doca_ec_matrix_create_from_raw()`	-
original data blocks	Source original data buffer, holding a sequence containing all original blocks (`block_1`, `block_2`, etc.); the order matters	• The data length of `original_data_blocks` should be a multiplication of the block size • The data length should also be aligned to 64B and with a minimum size of 64B
redundancy blocks	A destination buffer for the redundancy blocks. The sequence containing all redundancy blocks (`rdnc_block_1`, `rdnc_block_2`, etc.) is written to it upon successful completion of the task.	• The data will be written to the tail segment extending the data segment • The minimal available memory in `rdnc_blocks` should be the number of redundancy blocks * the block size, aligned to 64B and, in any case, at least 64B

Example for required buffer lengths

If a create task matrix is 10x4 (i.e., 10 original blocks, 4 redundancy blocks), and the block size is 64KB:

- `original_data_blocks` data length should be 10x64KB = 640KB
- The available memory for writing in `redundancy_blocks` should be at least 4x64KB = 256KB

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The destination buffer holds a sequence containing all redundancy blocks (`rdnc_block_1` , `rdnc_block_2` , etc.)
- The destination buffer data segment is extended to include the redundancy blocks

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task is submitted, the source and destination buffers should not be read from/written to
- Source and destination buffers must not overlap
- Other limitations are described in DOCA Core Task

14.4.13.6.2.4 Update Task

This task executes updates the redundancy blocks for the given original data blocks, using an update coding matrix.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_ec_task_update_set_conf`	`doca_ec_cap_task_update_is_supported`
Maximum block size	-	`doca_ec_cap_get_max_block_size`
Maximum buffer list length	-	`doca_ec_cap_get_max_buf_list_len`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
update matrix	An update coding matrix created by `doca_ec_matrix_create_update()` or `doca_ec_matrix_create_from_raw()`	-
original updated and RDNC blocks	A source buffer with data, holding a sequence containing the original data block and its updated data block, for each block that was updated, followed by the old redundancy blocks (`old_data_block_i`, `updated_data_block_i`, `old_data_block_j`, `updated_data_block_j`, ..., `rdnc_block_1`, `rdnc_block_2`, etc.)	• The data length of `original_updated_and_rdnc_blocks` should be a multiplication of the block size • The data length should also be aligned to 64B and with a minimum size of 64B
updated RDNC blocks	A destination buffer for the updated redundancy blocks. The sequence containing the updated redundancy blocks (`rdnc_block_1`, `rdnc_block_2`, etc.) is written to it upon successful completion of the task	• The data is written to the tail segment extending the data segment • The minimal available memory in `updated_rdnc_blocks` should be the number of redundancy blocks * the block size, aligned to 64B and, in any case, at least 64B

Example for required buffer lengths

using an update task matrix, in which 3 data block were updated and there are 4 redundancy blocks, and the block size is 64KB:

- `original_updated_and_rdnc_blocks` data length should be (3+3+4=10)x64KB = 640KB
- The available memory for writing in `updated_rdnc_blocks` should be at least 4x64KB = 256KB

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The destination buffer holds a sequence containing the updated redundancy blocks (`rdnc_block_1`, `rdnc_block_2`, etc.)
- The destination buffer data segment is extended to include the updated redundancy blocks

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects is not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the source and destination buffers should not be read from/written to
- Source and destination buffers must not overlap
- Other limitations described in DOCA Core Task

14.4.13.6.2.5 Recover Task

This task executes recovers data blocks for, using given available original data blocks and redundancy blocks and a given coding matrix.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_ec_task_recover_set_conf`	`doca_ec_cap_task_recover_is_supported`
Maximum block size	-	`doca_ec_cap_get_max_block_size`
Maximum buffer list length	-	`doca_ec_cap_get_max_buf_list_len`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
recover matrix	A coding matrix create by `doca_ec_matrix_create()` or `doca_ec_matrix_create_from_raw()`	-
available blocks	A source buffer with data, holding a sequence containing available data blocks and redundancy blocks (`data_block_a`, `data_block_b`, `data_block_c`, ..., `rdnc_block_x`, `rdnc_block_y`, etc.)	• The total number of blocks given should be equal to the number of original data blocks • The data length of `available_blocks` should be a multiplication of the block size • The data length should also be aligned to 64B and with a minimum size of 64B
recovered data blocks	A destination buffer for the recovered data blocks. The sequence containing the recovered data blocks (`data_block_i`, `data_block_j`, etc.) is written to it upon successful completion of the task	• The data is written to the tail segment extending the data segment • The minimal available memory in `recovered_data_blocks` should be the number of missing data blocks * the block size, aligned to 64B and, in any case, at least 64B.

Example for required buffer lengths

Using a recover task matrix, based on an original 10x4 coding matrix (i.e., 10 original blocks, 4 redundancy blocks), and a block size of 64KB:

- 10 available blocks should be given in total (e.g., 7 data blocks and 3 redundancy blocks)
- `available_blocks` data length should be 10x64KB = 640KB
- The available memory for writing in `recovered_data_blocks` should be at least 3x64KB = 192KB

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task is completed successfully the data is transformed to destination.

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task is submitted, the source and destination buffers should not be read from/written to
- Source and destination must not overlap
- The amount of blocks that can be recovered are limited to the number of redundancy blocks created
- Other limitations are described in DOCA Core Task

14.4.13.7 DOCA Erasure Coding Samples

This section provides DOCA Erasure Coding sample implementation on top of the BlueField-3 DPU (and higher).

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

14.4.13.7.1 Sample Prerequisites

N/A

14.4.13.7.2 Running the Sample

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample:

```
cd /opt/mellanox/doca/samples/doca_erasure_coding/<sample_name>
meson /tmp/build
ninja -C /tmp/build
```

> ⓘ The binary `doca_<sample_name>` is created under `/tmp/build/`.

3. Sample (e.g., `doca_erasure_coding_recover`) usage:

```
Usage: doca_erasure_coding_recover [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags from an input JSON file

Program Flags:
  -p, --pci-addr                    DOCA device PCI device address - default: 03:00.0
  -i, --input                       Input file/folder to ec - default: self
  -o, --output                      Output file/folder to ec - default: /tmp
  -b, --both                        Do both (encode & decode) - default: false
  -x, --matrix                      Matrix - {cauchy, vandermonde} - default: cauchy
  -t, --data                        Data block count - default: 2
  -r, --rdnc                        Redundancy block count - default: 2
  -d, --delete-index                Indices of data blocks to delete; comma-separated (i.e., 0,3,4) -
default: 0
```

> ⚠ Current BlueField-3 limitations:
> - Data block count range - 1-128
> - Redundancy block count - 1-32
> - Block size - 64B-128MB

4. For additional information per sample, use the `-h` option:

```
/tmp/build/doca_<sample_name> -h
```

14.4.13.7.3 Samples

14.4.13.7.3.1 Erasure Coding Recover

This sample illustrates how to use DOCA Erasure Coding (EC) library to encode and decode a file block (and entire file).

The sample logic includes 3 steps:

1. Encoding - create redundancy.
2. Deleting - simulating disaster.

3. Decoding - recovering data.

The encode logic includes:

1. Locating a DOCA device.
2. Initializing the required DOCA Core structures, such as the progress engine (PE), memory maps, and buffer inventory.
3. Reading source original data file and splitting it to a specified number of blocks, `<data block count>`, specified for the sample to the output directory.
4. Populating two DOCA memory maps with a memory range, one for the source data and one for the result.
5. Allocating buffers from DOCA buffer inventory for each memory range.
6. Creating an EC object.
7. Connecting the EC context to the PE.
8. Setting a state change callback function for the PE, with the following logic:
    - Printing a log with every state change
    - Indicating that the user may stop progress the PE once it is back in idle state
9. Setting the configuration to the EC create task, including setting callback functions as follows:
    - Successful completion callback:
        i. Writing the resulting redundancy blocks to the output directory (count is specified by `<redundancy block count>`).
        ii. Freeing the task.
        iii. Saving the result of the task and the callback. If there was an error in step a., the relevant error value is saved.
        iv. Stopping the context.
    - Failed completion callback:
        i. Saving the result of the task and the callback.
        ii. Freeing the task.
        iii. Stopping the context.
10. Creating EC encoding matrix by the matrix type specified to the sample.
11. Allocating and submitting an EC create task.
12. Progressing the PE until the context returns to idle state, either as a result of a successful run in which all tasks have been successfully completed, or as a result of a fatal error.
13. Destroying all EC and DOCA Core structures.

The delete logic includes:

1. Deleting the block files specified with `<indices of data blocks to delete>`.

The decode logic includes:

1. Locating a DOCA device.
2. Initializing the required DOCA Core structures, such as the PE, memory maps, and buffer inventory.
3. Reading the output directory (source remaining data) and determining the block size and which blocks are missing (needing recovery).
4. Populating two DOCA memory maps with a memory range, one for the source data and one for the result.
5. Allocating buffers from DOCA buffer inventory for each memory range.

6. Creating an EC object.
7. Connecting the EC context to the PE.
8. Setting a state change callback function for the PE, with the following logic:
    - Printing a log with every state change
    - Indicating that the user may stop progress the PE once it is back in idle state
9. Setting the configuration to the EC recover task, including setting callback functions as following:
    - Successful completion callback:
        i. Writing the resulting recovered blocks to the output directory.
        ii. Writing the recovered file to the output path.
        iii. Freeing the task.
        iv. Saving the result of the task and the callback. If there was an error in step a., the relevant error value is saved.
        v. Stopping the context.
    - Failed completion callback:
        i. Saving the result of the task and the callback.
        ii. Freeing the task.
        iii. Stopping the context.
10. Creating EC encoding matrix by the matrix type specified to the sample.
11. Creating EC decoding matrix, with `doca_ec_matrix_create_recover()` , using the encoding matrix.
12. Allocating and submitting an EC recover task.
13. Progressing the PE until the context returns to idle state, either as a result of a successful run in which all tasks have been successfully completed, or as a result of a fatal error.
14. Destroying all DOCA EC and DOCA Core structures.

References:

- `/opt/mellanox/doca/samples/doca_erasure_coding/doca_erasure_coding_recover/erasure_coding_recover_sample.c`
- `/opt/mellanox/doca/samples/doca_erasure_coding/doca_erasure_coding_recover/erasure_coding_recover_main.c`
- `/opt/mellanox/doca/samples/doca_erasure_coding/doca_erasure_coding_recover/meson.build`

14.4.14 DOCA AES-GCM

This guide provides instructions on building and developing applications that require data encryption and decryption using the AES-GCM algorithm.

14.4.14.1 Introduction

> The DOCA AES-GCM library is supported at alpha level.

The library provides an API for executing AES-GCM operations on DOCA buffers, where the buffers reside in either local memory (i.e., within the same host) or host memory accessible by the DPU (remote memory). Using DOCA AES-GCM, complex encrypt/decrypt operations can be easily executed in an optimized, hardware-accelerated manner.

This document is intended for software developers wishing to accelerate their application's encrypt/decrypt operations.

14.4.14.2 Prerequisites

This library follows the architecture of a DOCA Core context, it is recommended to read the following sections before:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

14.4.14.3 Environment

DOCA AES-GCM-based applications can run either on the host machine or on the NVIDIA® BlueField® DPU target.

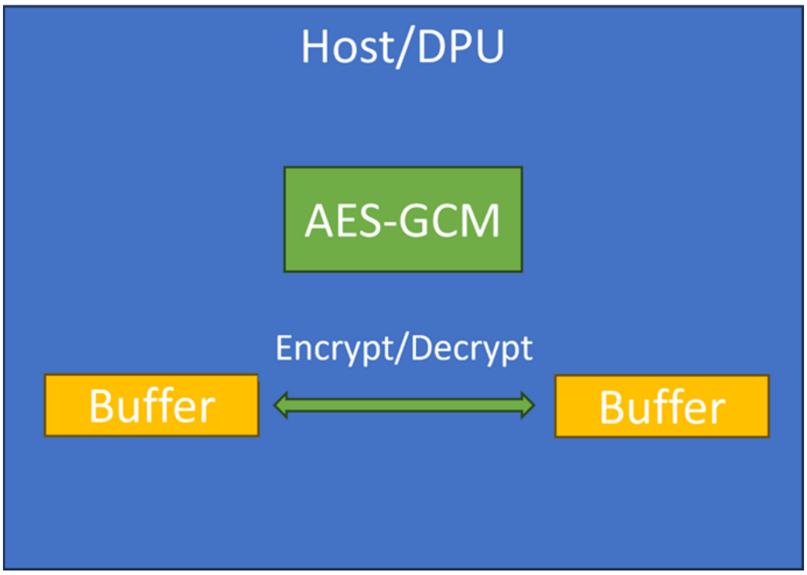
Encrypting/decrypting from the host to DPU and vice versa can only be run when the DPU is configured in DPU mode.

14.4.14.4 Architecture

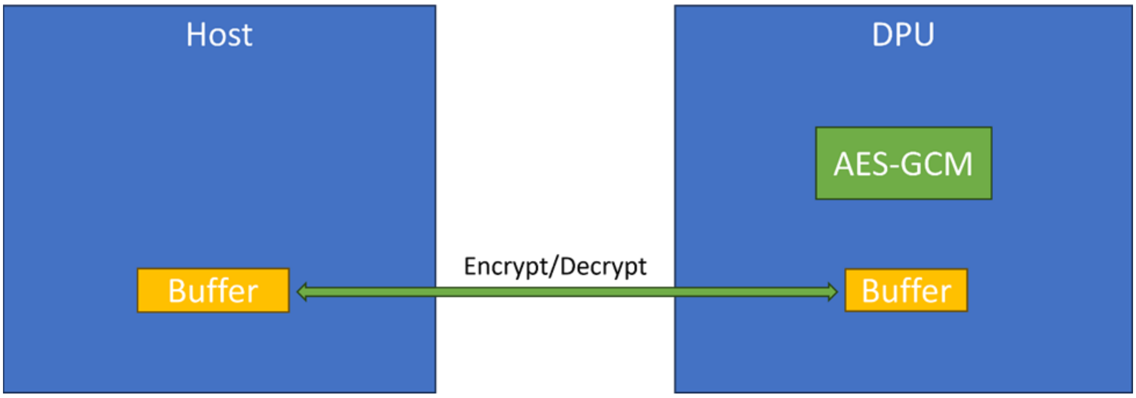
DOCA AES-GCM is a DOCA Core Context. This library leverages the DOCA Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

AES-GCM can be used to encrypt/decrypt data as illustrated in the following diagrams:

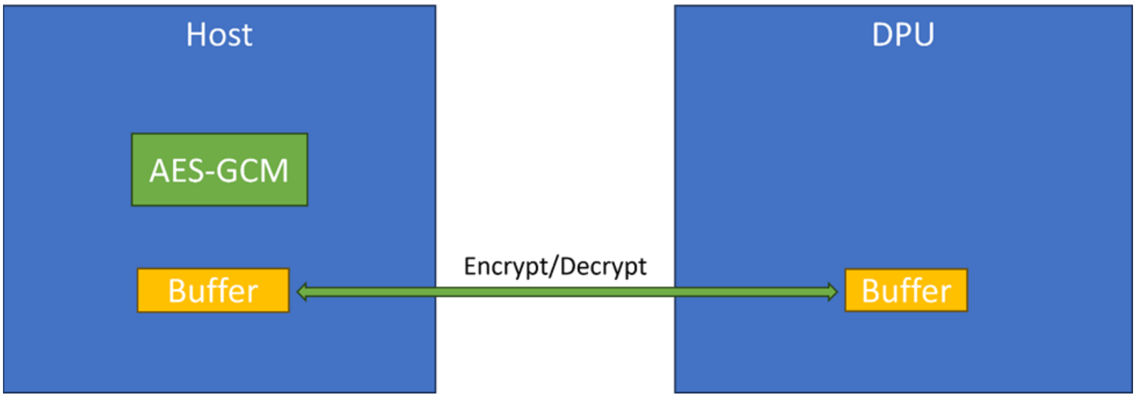
- Encrypt/decrypt from local memory to local memory:



- Using the DPU to copy memory between the host and the DPU:



- Using the host to copy memory between the host and the DPU:



14.4.14.4.1 Objects

14.4.14.4.1.1 Device and Representor

The library requires a DOCA device to operate. The device is used to access memory and perform the actual encrypt/decrypt. See DOCA Core Device Discovery.

For the same BlueField DPU, it does not matter which device is used (i.e., PF/VF/SF) as all these devices utilize the same hardware component. If there are multiple DPUs, then it is possible to create a AES-GCM instance per DPU, providing each instance with a device from a different DPU.

To access memory that is not local (i.e., from the host to DPU or vice versa), the DPU side of the application must pick a device with an appropriate representor (see DOCA Core Device Representor Discovery). The device must stay valid as long as AES-GCM instance is not destroyed.

14.4.14.4.1.2 Memory Buffers

The encrypt/decrypt task, requires two DOCA buffers containing the destination and the source.

Depending on the allocation pattern of the buffers, consider the DOCA Core Inventory Types table.

To find what kind of memory is supported, refer to the following table.

Buffers must not be modified or read during the encrypt/decrypt operation.

14.4.14.5 Configuration Phase

To start using the library users must go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context to allow execution of tasks and retrieval of events.

14.4.14.5.1 Configurations

The context can be configured to match the application use case.

To find if a configuration is supported or its min/max value, refer to Device Support.

14.4.14.5.1.1 Mandatory Configurations

These configurations must be set by the application before attempting to start the context:

- At least one task/event type must be configured. See configuration of Tasks and/or Events.
- A device with appropriate support must be provided upon creation

14.4.14.5.2 Device Support

DOCA AES-GCM requires a device to operate. For picking a device, see DOCA Core Device Discovery.

As device capabilities may change in the future (see DOCA Core Device Support) it is recommended to select your device using the following method:

- `doca_aes_gcm_cap_task_encrypt_is_supported`
- `doca_aes_gcm_cap_task_decrypt_is_supported`

Some devices can allow different capabilities as follows:

- The maximum number of tasks
- The maximum buffer size
- The maximum supported number of elements in DOCA linked-list buffer
- The maximum initialization vector length
- Check if authentication tag of size 96-bit is supported
- Check if authentication tag of size 128-bit is supported
- Check if a given AES-GCM key type is supported

14.4.14.5.3 Buffer Support

Tasks support buffers with the following features:

Buffer Type	Source Buffer	Destination Buffer
Local mmap buffer	Yes	Yes
Mmap from PCIe export buffer	Yes	Yes
Mmap from RDMA export buffer	No	No
Linked list buffer	Yes	No

14.4.14.6 Execution Phase

This section describes execution on the CPU using DOCA Core Progress Engine.

14.4.14.6.1 Tasks

DOCA AES-GCM exposes asynchronous tasks that leverage DPU hardware according to the DOCA Core architecture.

14.4.14.6.1.1 Encrypt Task

The encrypt task allows data encryption using buffers as described in Buffer Support.

Task Configuration

Description	API to Set Configuration	API to Query Support
Enable the task	`doca_aes_gcm_task_encrypt_set_conf`	`doca_aes_gcm_cap_task_encrypt_is_supported`
Number of tasks	`doca_aes_gcm_task_encrypt_set_conf`	`doca_aes_gcm_cap_get_max_num_tasks`
Maximal buffer size	-	`doca_aes_gcm_cap_task_encrypt_get_max_buf_size`
Maximum buffer list size	-	`doca_aes_gcm_cap_task_encrypt_get_max_list_buf_num_elem`
Maximum initialization vector length	-	`doca_aes_gcm_cap_task_encrypt_get_max_iv_length`
Enable authentication tag size	-	`doca_aes_gcm_cap_task_encrypt_is_tag_96_supported` `doca_aes_gcm_cap_task_encrypt_is_tag_128_supported`
Enable key type	-	`doca_aes_gcm_cap_task_encrypt_is_key_type_supported`

Task Input

Common input as described in DOCA Core Task

Name	Description	Notes
source buffer	Buffer pointing to the memory to be encrypted	Only the data residing in the data segment is encrypted
destination buffer	Buffer pointing to where memory is encrypted to	The encrypted data is appended to the tail segment

Name	Description	Notes
key	Key to encrypt the data	Created by the function `doca_aes_gcm_key_create` Users should use the same key to encrypt and decrypt the data
initialization vector (IV)	Initialization vector to be used by the AES-GCM algorithm	Users should use the same IV to encrypt and decrypt the data
initialization vector length	Initialization vector length that must be supplied for the AES-GCM algorithm	Represented in bytes, 0B-12B values are supported
authentication tag size	Authentication tag size to be supplied for the AES-GCM algorithm. The tag is automatically calculated and appended to the result buffer.	Represented in bytes, only 12B and 16B values are supported
additional authenticated data size	Additional authenticated data size to be supplied for the AES-GCM algorithm. This data, which should be present at the beginning of the source buffer, is will not encrypted but is authenticated.	Represented in bytes

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The data from the source buffer is encrypted and written to the destination buffer
- The destination buffer data segment is extended to include the encrypted data

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task is submitted, the source and destination should not be read/written to
- Other limitations are described in DOCA Core Task

14.4.14.6.1.2 Decrypt Task

The decrypt task allows data decryption. Using buffers as described in Buffer Support.

Task Configuration

Description	API to Set Configuration	API to Query Support
Enable the task	`doca_aes_gcm_task_decrypt_set_conf`	`doca_aes_gcm_cap_task_decrypt_is_supported`
Number of tasks	`doca_aes_gcm_task_decrypt_set_conf`	`doca_aes_gcm_cap_get_max_num_tasks`
Maximal buffer size	-	`doca_aes_gcm_cap_task_decrypt_get_max_buf_size`
Maximum buffer list size	-	`doca_aes_gcm_cap_task_decrypt_get_max_list_buf_num_elem`
Maximum initialization vector length	-	`doca_aes_gcm_cap_task_decrypt_get_max_iv_length`
Enable authentication tag size	-	`doca_aes_gcm_cap_task_decrypt_is_tag_96_supported` `doca_aes_gcm_cap_task_decrypt_is_tag_128_supported`
Enable key type	-	`doca_aes_gcm_cap_task_decrypt_is_key_type_supported`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer pointing to the memory to be decrypted	Only the data residing in the data segment is decrypted
Destination buffer	Buffer pointing to where memory is decrypted to	The decrypted data is appended to the tail segment extending the data segment
Key	Key to decrypt the data	Created by the function `doca_aes_gcm_key_create` The user should use the same key to encrypt and decrypt the data
Initialization vector (IV)	Initialization vector to be used by the AES-GCM algorithm	The user should use the same IV to encrypt and decrypt the data
Initialization vector length	Initialization vector length that must be supplied for the AES-GCM algorithm	Represented in bytes, 0B-12B values are supported
Authentication tag size	Authentication tag size to be supplied for the AES-GCM algorithm. The tag, present at the end of the source buffer, is verified and is not present in the destination buffer.	Represented in bytes, only 12B and 16B values are supported
Additional authenticated data size	Additional authenticated data size to be supplied for the AES-GCM algorithm. This data, present at the beginning of the source buffer, is not encrypted but is authenticated.	Represented in bytes

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The data from the source buffer is decrypted and written to the destination buffer
- The destination buffer data segment is extended to include the decrypted data

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects is not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task is submitted, the source and destination should not be read/written to
- Other limitations are described in DOCA Core Task

14.4.14.6.2 Events

DOCA AES-GCM exposes asynchronous events to notify about changes that happen unexpectedly according to the DOCA Core architecture.

The only events AES-GCM exposes are common events as described in DOCA Core Event.

14.4.14.7 State Machine

The DOCA AES-GCM library follows the Context state machine as described in DOCA Core Context State Machine.

The following section describes moving states and what is allowed in each state.

14.4.14.7.1 Idle

In this state, it is expected that the application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to Configurations
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context

Previous State	Transition Action
Running	Call stop after making sure all tasks have been freed
Stopping	Call progress until all tasks are completed and freed

14.4.14.7.2 Starting

This state cannot be reached.

14.4.14.7.3 Running

In this state, it is expected that the application:

- Allocates and submits tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocating previously configured task
- Submitting a task
- Calling stop

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

14.4.14.7.4 Stopping

In this state, it is expected that the application:

- Calls progress to complete all inflight tasks (tasks complete with failure)
- Frees any completed tasks

Allowed operations:

- Calling progress

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and fatal error occurs
Running	Call stop without freeing all tasks

14.4.14.8 Alternative Datapath Options

DOCA AES-GCM only supports datapath on the CPU. See Execution Phase.

14.4.14.9 DOCA AES-GCM Samples

This section describes DOCA AES-GCM samples based on the DOCA AES-GCM library.

The samples in this section illustrate how to use the DOCA AES-GCM API to do the following:

- Encrypt contents of a buffer to another buffer
- Decrypt contents of a buffer to another buffer

> All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

14.4.14.9.1 Running the Samples

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample:

   ```
   cd /opt/mellanox/doca/samples/doca_aes_gcm/<sample_name>
   meson/tmp/build
   ninja -C/tmp/build
   ```

   > The binary `doca_<sample_name>` is created under `/tmp/build/`.
3. Sample (e.g., `doca_aes_gcm_encrypt`) usage:

   ```
   Usage: doca_aes_gcm_encrypt [DOCA Flags] [Program Flags]

   DOCA Flags:
     -h, --help                        Print a help synopsis
     -v, --version                     Print program version information
     -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
   30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
     --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICA
   L, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
     -j, --json <path>                 Parse all command flags from an input json file

   Program Flags:
     -p, --pci-addr                    DOCA device PCI device address - default: 03:00.0
     -f, --file                        Input file to encrypt/decrypt
     -o, --output                      Output file - default: /tmp/out.txt
     -k, --key                         Raw key to encrypt/decrypt with, represented in hex format (32
    characters for 128-bit key, and 64 for 256-bit key) - default: 256-bit key, equals to zero
     -i, --iv                          Initialization vector, represented in hex format (0-24 characters for 0-9
   6-bit IV) - default: 96-bit IV, equals to zero
     -t, --tag size                    Authentication tag size. Tag size is in bytes and can be 12B or 16B -
   default: 12
     -a, --aad size                    Additional authenticated data size - default: 0
   ```
4. For additional information per sample, use the `-h` option:

   ```
   /tmp/build/doca_<sample_name>-h
   ```

14.4.14.9.2 Samples

14.4.14.9.2.1 AES-GCM Encrypt

This sample illustrates how to encrypt data with AES-GCM.

The sample logic includes:

1. Locating DOCA device.
2. Initializing required DOCA Core structures.
3. Setting the AES-GCM encrypt tasks configuration.
4. Populating DOCA memory map with two relevant buffers.
5. Allocating element in DOCA buffer inventory for each buffer.
6. Creating DOCA AES-GCM key.
7. Allocating and initializing AES-GCM encrypt task.
8. Submitting AES-GCM encrypt task.
9. Retrieving AES-GCM encrypt task once it is done.
10. Checking task result.
11. Destroying all AES-GCM and DOCA Core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_encrypt/aes_gcm_encrypt_sample.c`
- `/opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_encrypt/aes_gcm_encrypt_main.c`
- `/opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_encrypt/meson.build`

14.4.14.9.2.2 AES-GCM Decrypt

This sample illustrates how to decrypt data with AES-GCM.

The sample logic includes:

1. Locating DOCA device.
2. Initializing needed DOCA Core structures.
3. Setting the AES-GCM decrypt tasks configuration.
4. Populating DOCA memory map with two relevant buffers.
5. Allocating element in DOCA buffer inventory for each buffer.
6. Creating DOCA AES-GCM key.
7. Allocating and initializing AES-GCM decrypt task.
8. Submitting AES-GCM decrypt task.
9. Retrieving AES-GCM decrypt task once it is done.
10. Checking task result.
11. Destroying all AES-GCM and DOCA Core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_decrypt/aes_gcm_decrypt_sample.c`
- `/opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_decrypt/aes_gcm_decrypt_main.c`
- `/opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_decrypt/meson.build`

14.4.15 DOCA Rivermax

This guide provides instructions on building and developing applications that require media/data streaming.

14.4.15.1 Introduction

DOCA Rivermax (RMAX) is a DOCA API for NVIDIA® Rivermax®, an optimized networking SDK for media and data streaming applications. Rivermax leverages NVIDIA® BlueField® DPU hardware streaming acceleration technology which enables direct data transfers to and from the GPU, delivering best-in-class throughput and latency with minimal CPU utilization for streaming workloads.

This document is intended for software developers wishing to accelerate their networking operations.

14.4.15.2 Prerequisites

This library follows the architecture of DOCA Core Context. it is recommended read the following content before proceeding:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

14.4.15.3 Environment

DOCA Rivermax-based applications can run on the target DPU only.

DOCA Rivermax-based application must be run with root privileges.

- The Rivermax library must compile and run and Rivermax license to run applications. Refer to NVIDIA Rivermax SDK page to obtain that license.
- An IP address to the device being used must be set up.
- It is recommended to have at least 800 huge pages enabled to achieve maximum performance:

```
dpu> echo 1000000000 > /proc/sys/kernel/shmmax
dpu> echo 800 > /proc/sys/vm/nr_hugepages
```

14.4.15.4 Architecture

- DOCA Rivermax Input Stream is a DOCA Context as defined by DOCA Core
- DOCA Rivermax leverages DOCA Core architecture to expose asynchronous events that are offloaded to hardware
- DOCA Rivermax can be used to define input streams that allow packet acquisition on an IP port. Furthermore, the input stream can be split to TCP/UDP 5-tuples to allow separate handling of flows.

14.4.15.4.1 Objects

- `doca_rmax_flow` - is a flow object that represents an IP/port tuple
- `doca_rmax_in_stream` - is a `doca_ctx` that represents the input stream and can be thought of as a receive queue which scatters the received data into memory. Each stream can receive one or more flows.

14.4.15.5 Configuration Phase

To start using the library users must first go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context to allow execution of tasks and retrieval of events.

14.4.15.5.1 Configurations

The context can be configured to match the application use case.

To find if a configuration is supported or its min/max value, refer to section "Device Support".

14.4.15.5.1.1 Mandatory Configurations

These configurations must be set by the application before attempting to start the context:

- An event type must be configured. See configuration of Events.
- CPU affinity and then Rivermax library global initialization in this order. The following APIs can be used to achieve this `doca_rmax_set_cpu_affinity_mask()` and `doca_rmax_init()`
- The memory block that holds packet memory
- The number of stream elements
- Minimal packet segment size(s)
- Maximal packet segment size(s)

14.4.15.5.1.2 Optional Configurations

If the following configurations are not set, then a default value is used:

- The input stream type - defaults to generic
- The input stream packet's data scatter type - defaults to raw
- The input stream timestamp format - defaults to raw counter

14.4.15.5.2 Device Support

DOCA Rivermax Input Stream requires a device to operate. For picking a device see DOCA Core Device Discovery.

The device must be from within the DPU: Either a PF or SF.

It is recommended to choose your device using the following method:

- `doca_devinfo_get_ipv4_addr()`

Some devices can allow different capabilities as follows:

- PTP clock support.

14.4.15.5.3 Buffer Support

Memory block support buffers with the following features:

Buffer Type	Memory Block
Local mmap buffer	Yes
Mmap from PCIe export buffer	Yes
Mmap from RDMA export buffer	No
Linked list buffer	Yes (header split mode)

14.4.15.6 Execution Phase

This section describes execution on CPU using DOCA Core Progress Engine.

14.4.15.6.1 Events

DOCA Rivermax exposes asynchronous events to notify about changes that happen unexpectedly according to the DOCA Core architecture.

Common events are described in DOCA Core Event.

14.4.15.6.1.1 Rx Data

The Rx Data event is used by the stream to notify application that data has been received from the network.

Event Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	`doca_rmax_in_stream_event_rx_data_register`	-

Event Trigger Condition

The event is triggered anytime packet(s) arrive.

Event Output

Common output as described in DOCA Core Event.

In case of success, the following is provided:

- Number of packets received
- Time of arrival of the first packet
- Time of arrival of the last packet
- Sequence number of the first packet
- Array of memory blocks as configured by input stream

In case of error, the following is provided:

- An error code
- A human readable message

⚠ The parameters are valid only inside the event callback.

Event Handling

Once an event is triggered, the application may decide to process the received data.

14.4.15.6.2 Runtime Configurations

These configurations can be made after the context has been started:

- The minimal number of packets that the input stream must return in Rx event.
- The maximal number of packets that the input stream must return in Rx event.
- The receive timeout. The number of µsecs that library would do busy wait (polling) for reception of at least `min_packets` number of packets.

14.4.15.7 State Machine

The DOCA RMAX library follows the Context state machine as described in DOCA Core Context State Machine

The following section describes how to move to the state and what is allowed in each state.

14.4.15.7.1 Idle

In this state, it is expected that application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to Configurations
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop

14.4.15.7.2 Starting

This state is not expected to be reached.

14.4.15.7.3 Running

In this state, it is expected that application:

- Calls progress to receive events

Allowed operations:

- Calling stop
- Changing runtime configurations as described in Runtime Configurations

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

14.4.15.7.4 Stopping

This state is not expected to be reached.

14.4.15.8 DOCA Rivermax Samples

The samples illustrate how to use the DOCA Rivermax API to:

- List available devices, including their IP and supported capabilities
- Set CPU affinity for the internal Rivermax thread to achieve better performance
- Set the PTP clock device to be used internally in DOCA Rivermax
- Create a stream, create a flow and attach it to the created stream, and finally to start receiving data buffers (based on the attached flow)
- Create a stream in header-data split mode when packet headers and payload are split to different RX buffers

> All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

14.4.15.8.1 Running the Samples

1. Refer to the following documents:

    - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software
    - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples
2. To build a given sample:

   ```
   cd /opt/mellanox/doca/samples/doca_rmax/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build
   ```

   > The binary `doca_<sample_name>` is created under `/tmp/build/`.

3. Sample (e.g., `doca_rmax_create_stream`) usage:

   ```
   Usage: doca_rmax_create_stream [DOCA Flags] [Program Flags]

   DOCA Flags:
     -h, --help                              Print a help synopsis
     -v, --version                           Print program version information
     -l, --log-level                         Set the (numeric) log level for the program <10=DISABLE,
   20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
     -j, --json <path>                       Parse all command flags from an input json file

   Program Flags:
     -p, --pci_addr <PCI-ADDRESS>            PCI device address
   ```

   > When running DOCA Rivermax samples, the IPv4 address 192.168.105.2 must be configured to an available uplink prior to running it for the samples to run as expected:
   >
   > ```
   > $ifconfig p0 192.168.105.2
   > ```

4. For additional information per sample, use the `-h` option:

   ```
   /tmp/build/<sample_name> -h
   ```

14.4.15.8.2 Samples

14.4.15.8.2.1 List Devices

This sample illustrates how to list all available devices, dump their IPv4 addresses, and tell whether or not the PTP clock is supported.

The sample logic includes:

1. Initializing DOCA Rivermax library.
2. Iterating over the available devices.
3. Dumping their IPv4 addresses
4. Dumping whether a PTP clock is supported for each device.
5. Releasing DOCA Rivermax library.

References:

- `/opt/mellanox/doca/samples/doca_rmax/rmax_list_devices/rmax_list_devices_sample.c`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_list_devices/rmax_list_devices_main.c`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_list_devices/meson.build`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_common.h` ; `/opt/mellanox/doca/samples/doca_rmax/rmax_common.c`

14.4.15.8.2.2 Set CPU Affinity

This sample illustrates how to set the CPU affinity mask for Rivermax internal thread to achieve better performance. This parameter must be set before library initialization otherwise it will not be applied.

The sample logic includes:

1. Setting CPU affinity using the DOCA Rivermax API.
2. Initializing DOCA Rivermax library.
3. Releasing DOCA Rivermax library.

References:

- `/opt/mellanox/doca/samples/doca_rmax/rmax_set_affinity/rmax_set_affinity_sample.c`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_set_affinity/rmax_set_affinity_main.c`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_set_affinity/meson.build`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_common.h`; `/opt/mellanox/doca/samples/doca_rmax/rmax_common.c`

14.4.15.8.2.3 Set Clock

This sample illustrates how to set the PTP clock device to be used internally in DOCA Rivermax.

The sample logic includes:

1. Opening a DOCA device with a given PCIe address.
2. Initializing the DOCA Rivermax library.
3. Setting the device to use for obtaining PTP time.
4. Releasing the DOCA Rivermax library.

References:

- `/opt/mellanox/doca/samples/doca_rmax/rmax_set_clock/rmax_set_clock_sample.c`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_set_clock/rmax_set_clock_main.c`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_set_clock/meson.build`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_common.h`; `/opt/mellanox/doca/samples/doca_rmax/rmax_common.c`

14.4.15.8.2.4 Create Stream

This sample illustrates how to create a stream, create a flow and attach it to the created stream, and finally to start receiving data buffers (based on the attached flow).

The sample logic includes:

1. Opening a DOCA device with a given PCIe address.
2. Initializing the DOCA Rivermax library.
3. Creating an input stream.
4. Creating the context from the created stream.
5. Initializing DOCA Core related objects.
6. Setting the attributes of the created stream.
7. Creating a flow and attaching it to the created stream.
8. Starting to receive data buffers.
9. Clean up—detaches flow and destroys it, destroys created stream and DOCA Core related objects.

References:

- `/opt/mellanox/doca/samples/doca_rmax/rmax_create_stream/rmax_create_stream_sample.c`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_create_stream/rmax_create_stream_main.c`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_create_stream/meson.build`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_common.h`; `/opt/mellanox/doca/samples/doca_rmax/rmax_common.c`

14.4.15.8.2.5 Create Stream – Header-data Split Mode

This sample illustrates how to create a stream in header-data split mode when packet headers and payload are split to different RX buffers.

The sample logic includes:

1. Opening a DOCA device with a given PCIe address.
2. Initialize the DOCA Rivermax library.
3. Creating an input stream.
4. Creating a context from the created stream.
5. Initializing DOCA Core related objects.
6. Setting attributes of the created stream. Chaining buffers and setting header size to non-zero is essential to create a stream with header-data split mode.
7. Creating a flow and attaching it to the created stream.
8. Starting to receive data to split buffers.
9. Clean up—detaches flow and destroys it, destroys created stream and DOCA Core related objects.

References:

- `/opt/mellanox/doca/samples/doca_rmax/rmax_create_stream_hds/rmax_create_stream_hds_sample.c`

- `/opt/mellanox/doca/samples/doca_rmax/rmax_create_stream_hds/rmax_create_stream_hds_main.c`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_create_stream_hds/meson.build`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_common.h`; `/opt/mellanox/doca/samples/doca_rmax/rmax_common.c`

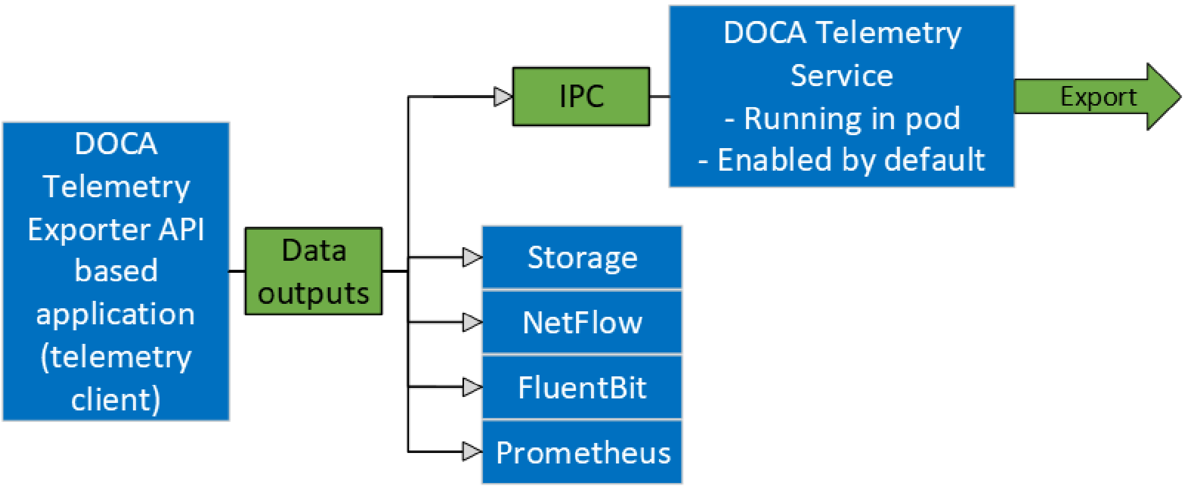
14.4.16 DOCA Telemetry Exporter

This guide provides an overview and configuration instructions for DOCA Telemetry Exporter API.

14.4.16.1 Introduction

DOCA Telemetry Exporter API offers a fast and convenient way to transfer user-defined data to DOCA Telemetry Service (DTS). In addition, the API provides several built-in outputs for user convenience, including saving data directly to storage, NetFlow, Fluent Bit forwarding, and Prometheus endpoint.

The following figure shows an overview of the telemetry exporter API. The telemetry exporter client side, based on the telemetry exporter API, collects user-defined telemetry and sends it to the DTS which runs as a container on BlueField. DTS does further data routing, including export with filtering. DTS can process several user-defined telemetry exporter clients and can collect pre-defined counters by itself. Additionally, telemetry exporter API has built-in data outputs that can be used from telemetry exporter client applications.



The following scenarios are available:

- Send data via IPC transport to DTS. For IPC, refer to Inter-process Communication.
- Write data as binary files to storage (for debugging data format).
- Export data directly from DOCA Telemetry Exporter API application using the following options:
  - Fluent Bit exports data through forwarding
  - NetFlow exports data from NetFlow API. Available from both API and DTS. See details in Data Outputs.
  - Prometheus creates Prometheus endpoint and keeps the most recent data to be scraped by Prometheus.

Users can either enable or disable any of the data outputs mentioned above. See Data Outputs to see how to enable each output.

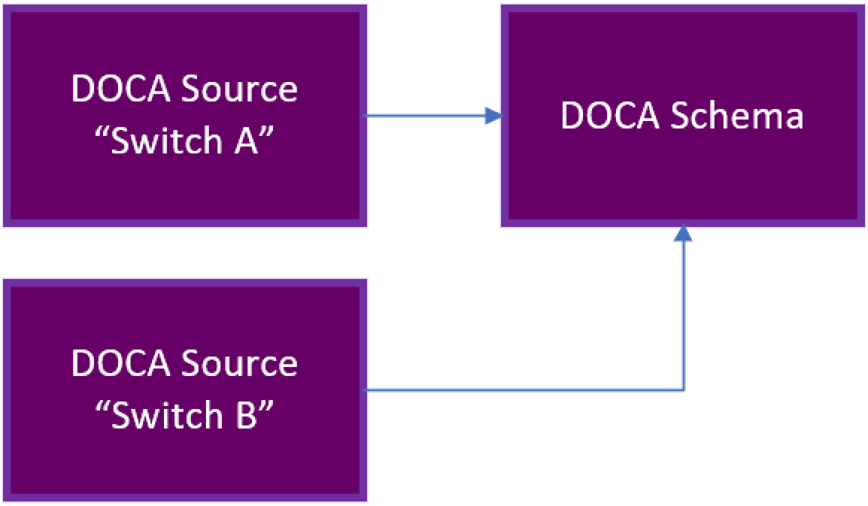
The library stores data in an internal buffer and flushes it to DTS/exporters in the following scenarios:

- Once the buffer is full. Buffer size is configurable with different attributes.
- When `doca_telemetry_exporter_source_flush(void *doca_source)` function is invoked.
- When the telemetry exporter client terminates. If the buffer has data, it is processed before the library's context cleanup.

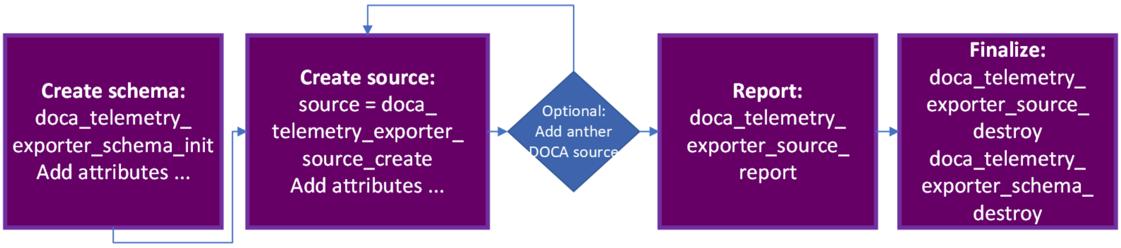
14.4.16.2 Architecture

DOCA Telemetry Exporter API is fundamentally built around four major parts:

- DOCA schema - defines a reusable structure (see doca_telemetry_exporter_type) of telemetry data which can be used by multiple sources



- Source - the unique identifier of the telemetry exporter source that periodically reports telemetry data.
- Report - exports the information to the DTS
- Finalize - releases all the resources


Create schema:
doca_telemetry_exporter_schema_init
Add attributes ...
Create source:
source = doca_telemetry_exporter_source_create
Add attributes ...
Optional:
Add anther
DOCA source
Report:
doca_telemetry_exporter_source_report
Finalize:
doca_telemetry_exporter_source_destroy
doca_telemetry_exporter_schema_destroy

14.4.16.2.1 DOCA Telemetry Exporter API Walkthrough

The NVIDIA DOCA Telemetry Exporter API's definitions can be found in the `doca_telemetry_exporter.h` file.

The following is a basic walkthrough of the needed steps for using the DOCA Telemetry Exporter API.

1. Create `doca_schema`.

   a. Initialize an empty schema with default attributes:

      ```
      struct doca_telemetry_exporter_schema *doca_schema;
      doca_telemetry_exporter_schema_init("example_doca_schema_name", &doca_schema);
      ```

   b. Set the following attributes if needed:
      - `doca_telemetry_exporter_schema_set_buffer_attr_*(…)`
      - `doca_telemetry_exporter_schema_set_file_write_*(…)`
      - `doca_telemetry_exporter_schema_set_ipc_*(…)`
   c. Add user event types:
      Event type (`struct doca_telemetry_exporter_type`) is the user-defined data structure that describes event fields. The user is allowed to add multiple fields to the event type. Each field has its own attributes that can be set (see example). Each event type is allocated an index (`doca_telemetry_exporter_type_index_t`) which can be used to refer to the event type in future API calls.

      ```
      struct doca_telemetry_exporter_type *doca_type;
      struct doca_telemetry_exporter_field *field1;

      doca_telemetry_exporter_type_create(&doca_type);
      doca_telemetry_exporter_field_create(&field1);

      doca_telemetry_exporter_field_set_name(field1, "sport");
      doca_telemetry_exporter_field_set_description(field1, "Source port")
      doca_telemetry_exporter_field_set_type_name(field1, DOCA_TELEMETRY_EXPORTER_FIELD_TYPE_UINT16);
      doca_telemetry_exporter_field_set_array_length(field1, 1);

      /* The user loses ownership on field1 after a successful invocation of the function */
      doca_telemetry_exporter_type_add_field(type, field1);

      /* Add more fields if needed */

      /* The user loses ownership on doca_type after a successful invocation of the function */
      doca_telemetry_exporter_schema_add_type(doca_schema, "example_event", doca_type, &type_index);
      ```

   d. Apply attributes and types to start using the schema:

      ```
      doca_telemetry_exporter_schema_start(doca_schema)
      ```

2. Create `doca_source`:
   a. Initialize:

      ```
      struct doca_telemetry_exporter_source *doca_source;
      doca_telemetry_exporter_source_create(doca_schema, &doca_source);
      ```

   b. Set source ID and tag:

      ```
      doca_telemetry_exporter_source_set_id(doca_source, "example id");
      doca_telemetry_exporter_source_set_tag(doca_source, "example tag");
      ```

   c. Apply attributes to start using the source:

      ```
      doca_telemetry_exporter_source_start(doca_source)
      ```

   You may optionally add more `doca_sources` if needed.
3. Collect the data per source and use:

   ```
   doca_telemetry_exporter_source_report(source, type_index, &my_app_test_ev1, num_events)
   ```

4. Finalize:
   a. For every source:

```
doca_telemetry_exporter_source_destroy(source)
```

b. Destroy:

```
doca_telemetry_exporter_schema_destroy(doca_schema)
```

Example implementation may be found in the `telemetry_export` DOCA sample (`telemetry_export_sample.c`).

14.4.16.2.2 DOCA Telemetry Exporter NetFlow API Walkthrough

The DOCA telemetry exporter API also supports NetFlow using DOCA Telemetry Exporter NetFlow API. This API is designed to allow customers to easily support the NetFlow protocol at the endpoint side. Once an endpoint produces NetFlow data using the API, the corresponding exporter can be used to send the data to a NetFlow collector.

The NVIDIA DOCA Telemetry Exporter Netflow API's definitions can be found in the `doca_telemetry_exporter_netflow.h` file.

The following are the steps to use the NetFlow API:

1. Initiate the API with an appropriate source ID:

   ```
   doca_telemetry_exporter_netflow_init(source_id)
   ```

2. Set the relevant attributes:
   - `doca_telemetry_exporter_netflow_set_buffer_*(…)`
   - `doca_telemetry_exporter_netflow_set_file_write_*(…)`
   - `doca_telemetry_exporter_netflow_set_ipc_*(…)`
   - `doca_telemetry_exporter_netflow_source_set_*()`
3. Start the API to use the configured attribute:

   ```
   doca_telemetry_exporter_netflow_start();
   ```

4. Form a desired NetFlow template and the corresponding NetFlow records.
5. Collect the NetFlow data.

   ```
   doca_telemetry_exporter_netflow_send(…)
   ```

6. (Optional) Flush the NetFlow data to send data immediately instead of waiting for the buffer to fill:

   ```
   doca_telemetry_exporter_netflow_flush()
   ```

7. Clean up the API:

   ```
   doca_telemetry_exporter_netflow_destroy()
   ```

Example implementation may be found in the `telemetry_export_netflow` DOCA sample (`telemetry_export_netflow_sample.c`).

14.4.16.3 API

Refer to NVIDIA DOCA Library APIs, for more detailed information on DOCA Telemetry Exporter API.

> The pkg-config (`*.pc` file) for the DOCA Telemetry Exporter library is `doca-telemetry-exporter` .

The following sections provide additional details about the library API.

Some attributes are optional as they are initialized with default values. Refer to the documentation of the setter functions of respective attributes for more information.

14.4.16.3.1 DOCA Telemetry Exporter Buffer Attributes

Buffer attributes are used to set the internal buffer size and data root used by all DOCA sources in the schema.

Configuring the attributes is optional as they are initialized with default values.

```
doca_telemetry_exporter_schema_set_buffer_size(doca_schema, 16 * 1024); /* 16KB - arbitrary value */
doca_telemetry_exporter_schema_set_buffer_data_root(doca_schema, "/opt/mellanox/doca/services/telemetry/data/");
```

- `buffer_size [in]` - the size of the internal buffer which accumulates the data before sending it to the outputs. Data is sent automatically once the internal buffer is full. Larger buffers mean fewer data transmissions and vice versa.
- `data_root [in]` - the path to where data is stored (if `file_write_enabled` is set to true). See section "DOCA Telemetry Exporter File Write Attributes".

14.4.16.3.2 DOCA Telemetry Exporter File Write Attributes

File write attributes are used to enable and configure data storage to the file system in binary format.

Configuring the attributes is optional as they are initialized with default values.

```
doca_telemetry_exporter_schema_set_file_write_enabled(doca_schema);
doca_telemetry_exporter_schema_set_file_write_max_size(doca_schema, 1 * 1024 * 1024); /* 1 MB */
doca_telemetry_exporter_schema_set_file_write_max_age(doca_schema, 60 * 60 * 1000000L); /* 1 Hour */
```

- `file_write_enable [in]` - use this function to enable storage. Storage/FileWrite is disabled by default.
- `file_write_max_size [in]` - maximum file size (in bytes) before a new file is created.
- `file_write_max_age [in]` - maximum file age (in microseconds) before a new file is created.

14.4.16.3.3 DOCA Telemetry Exporter IPC Attributes

IPC attributes are used to enable and configure IPC transport. IPC is disabled by default.

Configuring the attributes is optional as they are initialized with default values.

It is important to make sure that the IPC location matches the IPC location used by DTS, otherwise IPC communication will fail.

```
doca_telemetry_exporter_schema_set_ipc_enabled(doca_schema);
doca_telemetry_exporter_schema_set_ipc_sockets_dir(doca_schema, "/path/to/sockets/");
doca_telemetry_exporter_schema_set_ipc_reconnect_time(doca_schema, 100); /* 100 milliseconds */
doca_telemetry_exporter_schema_set_ipc_reconnect_tries(doca_schema, 3);
doca_telemetry_exporter_schema_set_ipc_socket_timeout(doca_schema, 3 * 1000) /* 3 seconds */
```

- `ipc_enabled [in]` - use this function to enable communication. IPC is disabled by default.
- `ipc_sockets_dir [in]` - a directory that contains UDS for IPC messages. Both the telemetry exporter program and DTS must use the same folder. DTS that runs on BlueField as a container has the default folder `/opt/mellanox/doca/services/telemetry/ipc_sockets`.
- `ipc_reconnect_time [in]` - maximum reconnection time in milliseconds after which the client is considered disconnected.
- `ipc_reconnect_tries [in]` - maximum reconnection attempts.
- `ipc_socket_timeout [in]` - timeout for the IPC socket.

14.4.16.3.4 DOCA Telemetry Exporter Source Attributes

Source attributes are used to create proper folder structure. All the data collected from the same host is written to the `source_id` folder under data root.

Sources attributes are mandatory and must be configured before invoking `doca_telemetry_exporter_source_start()`.

```
doca_telemetry_exporter_source_set_id(doca_source, "example_source");
doca_telemetry_exporter_source_set_tag(doca_source, "example_tag");
```

- `source_id [in]` - describes the data's origin. It is recommended to set it to the hostname. In later dataflow steps, data is aggregated from multiple hosts/DPUs and `source_id` helps navigate in it.
- `source_tag [in]` - a unique data identifier. It is recommended to set it to describe the data collected in the application. Several telemetry exporter apps can be deployed on a single node (host/DPU). In that case, each telemetry data would have a unique tag and all of them would share a single `source_id`.

14.4.16.3.5 DOCA Telemetry Exporter Netflow Collector Attributes

DOCA Telemetry Exporter NetFlow API attributes are optional and should only be used for debugging purposes. They represent the NetFlow collector's address while working locally, effectively enabling the local NetFlow exporter.

```
doca_telemetry_exporter_netflow_set_collector_addr("127.0.0.1");
doca_telemetry_exporter_netflow_set_collector_port(6343);
```

- `collector_addr [in]` - NetFlow collector's address (IP or name). Default value is `NULL`.

- `collector_port [in]` - NetFlow collector's port. Default value is `DOCA_NETFLOW_DEFAULT_PORT (2055)`.

14.4.16.3.6 doca_telemetry_exporter_source_report

The source report function is the heart of communication with the DTS. The report operation causes event data to be allocated to the internal buffer. Once the buffer is full, data is forwarded onward according to the set configuration.

```
doca_error_t doca_telemetry_exporter_source_report(struct doca_telemetry_exporter_source *doca_source,
                                                  doca_telemetry_exporter_type_index_t index,
                                                  void *data,
                                                  int count);
```

- `doca_source [in]` - a pointer to the `doca_telemetry_exporter_source` which reports the event
- `index [in]` - the event type index received when the schema was created
- `data [in]` - a pointer to the data buffer that needs to be sent
- `count [in]` - numbers of events to be written to the internal buffer

The function returns `DOCA_SUCCESS` if successful, or a `doca_error_t` if an error occurs. If a memory-related error occurs, try a larger buffer size that matches the event's size.

14.4.16.3.7 doca_telemetry_exporter_schema_add_type

This function allows adding a reusable telemetry data struct, also known as a schema. The schema allows sending a predefined data structure to the telemetry service. Note that it is mandatory to define a schema for proper functionality of the library. After adding the schemas, one needs to invoke the schema start function.

```
doca_error_t doca_telemetry_exporter_schema_add_type(struct doca_telemetry_exporter_schema *doca_schema,
                                                     const char *new_type_name,
                                                     struct doca_telemetry_exporter_type *type,
                                                     doca_telemetry_exporter_type_index_t *type_index);
```

- `doca_schema [in]` - a pointer to the schema to which the type is added
- `new_type_name [in]` - name of the new type
- `fields [in]` - user-defined fields to be used for the schema. Multiple fields can (and should) be added.
- `type_index [out]` - type index for the created type is written to this output variable

The function returns `DOCA_SUCCESS` if successful, or `doca_error_t` if an error occurs.

14.4.16.4 Telemetry Data Format

The internal data format consists of 2 parts: A schema containing metadata, and the actual binary data. When data is written to storage, the data schema is written in JSON format, and the data is written as binary files. In the case of IPC transport, both schema and binary data are sent to DTS. In the case of export, data is converted to the formats required by exporter.

Adding custom event types to the schema can be done using `doca_telemetry_exporter_schema_add_type` API call.

See available `DOCA_TELEMETRY_EXPORTER_FIELD_TYPE`s in `doca_telemetry_exporter.h`. See example of usage in `/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export/telemetry_export_sample.c`.

It is highly recommended to have the timestamp field as the first field since it is required by most databases. To get the current timestamp in the correct format use:

```
doca_error_t doca_telemetry_exporter_get_timestamp(doca_telemetry_exporter_timestamp_t *timestamp);
```

14.4.16.5 Data Outputs

This section describes available exporters:

- IPC
- NetFlow
- Fluent Bit
- Prometheus

Fluent Bit and Prometheus exporters are presented in both API and DTS. Even though DTS export is preferable, the API has the same possibilities for development flexibility.

14.4.16.5.1 Inter-process Communication

IPC transport automatically transfers the data from the telemetry-exporter-based program to DTS service.

It is implemented as a UNIX domain socket (UDS) sockets for short messages and shared memory for data. DTS and the telemetry-exporter-based program must share the same `ipc_sockets` directory.

When IPC transport is enabled, the data is sent from the DOCA-telemetry-exporter-based application to the DTS process via shared memory.

To enable IPC, use the `doca_telemetry_exporter_schema_set_ipc_enabled` API function.

IPC transport relies on system folders. For the host's usage, run the DOCA-telemetry-exporter-API-based application with `sudo` to be able to use IPC with system folders.

To check the IPC status for the current context, use:

```
doca_error_t doca_telemetry_exporter_check_ipc_status(struct doca_telemetry_exporter_source *doca_source,
                                                       doca_telemetry_exporter_ipc_status_t *status)
```

If IPC is enabled and for some reason connection is lost, it would try to automatically reconnect on every report's function call.

14.4.16.5.1.1 Using IPC with Non-container Application

When developing and testing a non-container DOCA Telemetry-Exporter-based program and its IPC interaction with DTS, some modifications are necessary in DTS's deployment for the program to interact with DTS over IPC:

- Shared memory mapping should be removed: `telemetry-ipc-shm`
- Host IPC should be enabled: `hostIPC`

File before the change:

```
spec:
  hostNetwork: true
  volumes:
  - name: telemetry-service-config
    hostPath:
      path: /opt/mellanox/doca/services/telemetry/config
      type: DirectoryOrCreate
  ...
  - name: telemetry-ipc-shm
    hostPath:
      path: /dev/shm/telemetry
      type: DirectoryOrCreate
  containers:
  ...
      volumeMounts:
      - name: telemetry-service-config
        mountPath: /config
  ...
      - name: telemetry-ipc-shm
        mountPath: /dev/shm
```

File after the change:

```
spec:
  hostNetwork: true
  hostIPC: true
  volumes:
  - name: telemetry-service-config
    hostPath:
      path: /opt/mellanox/doca/services/telemetry/config
      type: DirectoryOrCreate
  ...
  containers:
  ...
      volumeMounts:
      - name: telemetry-service-config
        mountPath: /config
```

These changes ensure that a DOCA-based program running outside of a container is able to communicate with DTS over IPC.

14.4.16.5.2 NetFlow

When the NetFlow exporter is enabled (NetFlow Collector Attributes are set), it sends the NetFlow data to the NetFlow collector specified by the attributes: Address and port. This exporter must be used when using DOCA Telemetry Exporter NetFlow API.

14.4.16.5.3 Fluent Bit

Fluent Bit export is based on `fluent_bit_configs` with `.exp` files for each destination. Every export file corresponds to one of Fluent Bit's destinations. All found and enabled `.exp` files are used as separate export destinations. Examples can be found after running DTS container under its configuration folder (`/opt/mellanox/doca/services/telemetry/config/fluent_bit_configs/`) .

All `.exp` files are documented in-place.

```
DPU# ls -l /opt/mellanox/doca/services/telemetry/config/fluent_bit_configs/
/opt/mellanox/doca/services/telemetry/config/fluent_bit_configs/:
total 56
-rw-r--r-- 1 root root  528 Oct 11 07:52 es.exp
-rw-r--r-- 1 root root  708 Oct 11 07:52 file.exp
-rw-r--r-- 1 root root 1135 Oct 11 07:52 forward.exp
-rw-r--r-- 1 root root  719 Oct 11 07:52 influx.exp
-rw-r--r-- 1 root root  571 Oct 11 07:52 stdout.exp
-rw-r--r-- 1 root root  578 Oct 11 07:52 stdout_raw.exp
-rw-r--r-- 1 root root 2137 Oct 11 07:52 ufm_enterprise.fset
```

Fluent Bit `.exp` files have 2-level data routing:

- `source_tags` in `.exp` files (documented in-place)
- Token-based filtering governed by `.fset` files (documented in `ufm_enterprise.fset`)

To run with Fluent Bit exporter, set `enable=1` in required `.exp` files and set the environment variables before running the application:

```
export FLUENT_BIT_EXPORT_ENABLE=1
export FLUENT_BIT_CONFIG_DIR=/path/to/fluent_bit_configs
export LD_LIBRARY_PATH=/opt/mellanox/collectx/lib
```

14.4.16.5.4 Prometheus

Prometheus exporter sets up endpoint (HTTP server) which keeps the most recent events data as text records.

The Prometheus server can scrape the data from the endpoint while the DOCA-Telemetry-Exporter-API-based application stays active.

Check the generic example of Prometheus records:

```
event_name_1{label_1="label_1_val", label_2="label_2_val", label_3="label_3_val", label_4="label_4_val"}
counter_value_1 timestamp_1
event_name_2{label_1="label_1_val", label_2="label_2_val", label_3="label_3_val", label_4="label_4_val"}
counter_value_2 timestamp_2
...
```

Labels are customizable metadata which can be set from data file. Events names could be filtered by token-based name-match according to `.fset` files.

Set the following environment variables before running.

```
# Set the endpoint host and port to enable export.
export PROMETHEUS_ENDPOINT=http://0.0.0.0:9101

# Set indexes as a comma-separated list to keep data for every index field. In
# this example most recent data will be kept for every record with unique
# `port_num`. If not set, only one data per source will be kept as the most
# recent.
export PROMETHEUS_INDEXES=Port_num

# Set path to a file with Prometheus custom labels. Use labels to store
# information about data source and indexes. If not set, the default labels
# will be used.
export CLX_METADATA_FILE=/path/to/labels.txt

# Set the folder which contains fset-files. If set, Prometheus will scrape
# only filtered data according to fieldsets.
export PROMETHEUS_CSET_DIR=/path/to/prometheus_cset
```

To scrape the data without the Prometheus server, use:

```
curl -s http://0.0.0.0:9101/metrics
```

Or:

```
curl -s http://0.0.0.0:9101/{fset_name}
```

14.4.16.6 DOCA Telemetry Exporter Samples

This section provides DOCA Telemetry Exporter sample implementations on top of the BlueField DPU.

The telemetry exporter samples in this document demonstrate an initial recommended configuration that covers two use cases:

- Standard DOCA Telemetry Exporter data
- DOCA Telemetry Exporter for NetFlow data

The telemetry exporter samples run on the BlueField. If write-to-file is enabled, telemetry data is stored to BlueField's storage. If inter-process communication (IPC) is enabled, data is sent to the DOCA Telemetry Service (DTS) running on the same BlueField.

For information on initializing and configuring DTS, refer to NVIDIA DOCA Telemetry Service Guide.

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

14.4.16.6.1 Running the Sample

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample:

   ```
   cd /opt/mellanox/doca/samples/doca_telemetry_exporter/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build
   ```

   The binary `doca_<sample_name>` will be created under `/tmp/build/`.
3. Sample (e.g., `telemetry_export`) usage:

   ```
   Usage: doca_telemetry_export [DOCA Flags]

   DOCA Flags:
     -h, --help                        Print a help synopsis
   ```

```
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICA
L, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags from an input json file
```

4. For additional information per sample, use the `-h` option:

```
/tmp/build/doca_<sample_name> -h
```

14.4.16.6.2 Samples

14.4.16.6.2.1 Telemetry Export

This sample illustrates how to use the telemetry exporter API. The sample uses a custom schema for telemetry exporter.

The sample logic includes:

1. Configuring schema attributes.
2. Initializing schema.
3. Creating telemetry exporter source.
4. Creating example events.
5. Reporting example events via DOCA Telemetry Exporter.
6. Destroying source and schema.

Reference:

- `/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export/telemetry_export_sample.c`
- `/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export/telemetry_export_main.c`
- `/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export/meson.build`

14.4.16.6.2.2 Telemetry Export NetFlow

This sample illustrates how to use the NetFlow functionality of the telemetry exporter API.

The sample logic includes:

1. Configuring NetFlow attributes.
2. Initializing NetFlow.
3. Creating telemetry exporter source.
4. Starting NetFlow.
5. Creating example events.
6. Reporting example events via DOCA Telemetry Exporter.
7. Destroying NetFlow.

Reference:

- `/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export_netflow/telemetry_export_netflow_sample.c`

- `/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export_netflow/telemetry_export_netflow_main.c`
- `/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export_netflowt/meson.build`

14.4.17 DOCA Telemetry Diagnostics

This guide provides instructions on building and developing applications which require collecting telemetry information provided by NVIDIA® BlueField and NVIDIA® ConnectX® families of networking platforms.

14.4.17.1 Introduction

The `doca_telemetry_diag` provides programable access to an on-device mechanism which allows sampling of diagnostic data (such as statistics and counters). The `doca_telemetry_diag` allows configuring such parameters as required data IDs or sampling period, and retrieving the generated information in several formats.

14.4.17.2 Architecture

Diagnostic data is stored in hardware as a cyclic buffer of samples. Each sample represents all the requested diagnostic data IDs and their corresponding sampling timestamps. The sampling period and the number of samples in the buffer can be configured.

The DOCA Telemetry Diagnostics library supports the following operational methods:

- Single sampling - the samples are stored and once the samples buffer is filled, sampling is terminated
- Repetitive sampling - when the sample buffer is filled, new samples override old samples
- On demand - the device does not collect samples. Upon query of the diagnostic data, the device fetches a single sample of the data.

Samples are retrieved by calling the `doca_telemetry_diag_query_counters` function. Multiple samples can be retrieved in a single call. The application defines the maximum number of samples it wishes to retrieve and supplies a buffer large enough to contain these samples (sample size can be received using a dedicated API). The library only retrieves new samples without duplications and returns fewer samples than requested if there are no more new samples.

14.4.17.2.1 Synchronized Start

Diagnostics data is sampled by the device every given sampling period. When sampling this way, each data entry in a sample may be recorded at a slightly different time.

Synchronized start mode enables diagnostics counters to begin all data measurements at the same time (i.e., during the same clock cycle). This way, the sample period is guaranteed to be identical for all samples.

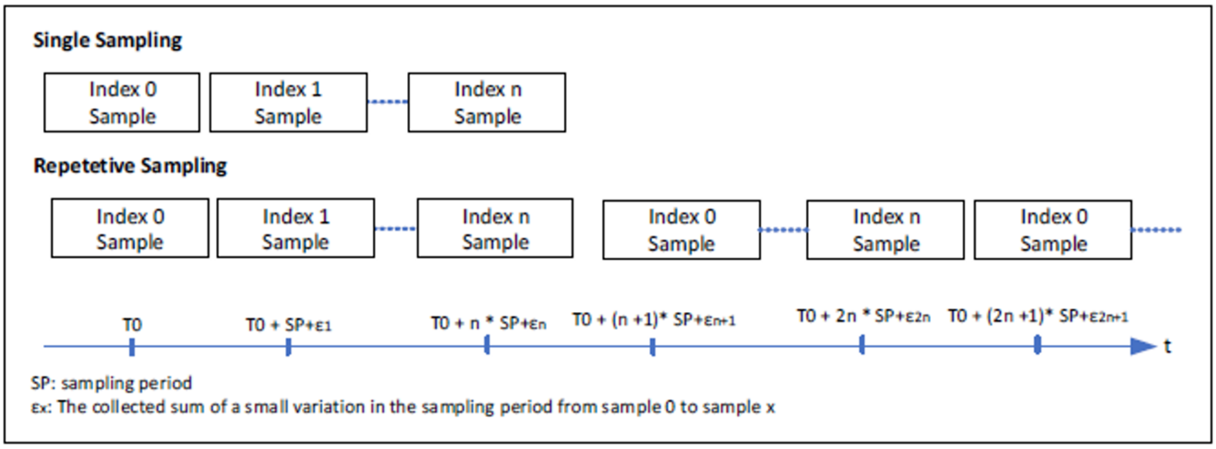
> ⚠ In synchronized start mode, counters are stopped during the collection time of each sample.

Not all data IDs can be sampled in synchronized start mode. Setting a data ID failure with the error code `DOCA_ERROR_BAD_CONFIG` indicates that the given data ID does not support synchronized start mode.

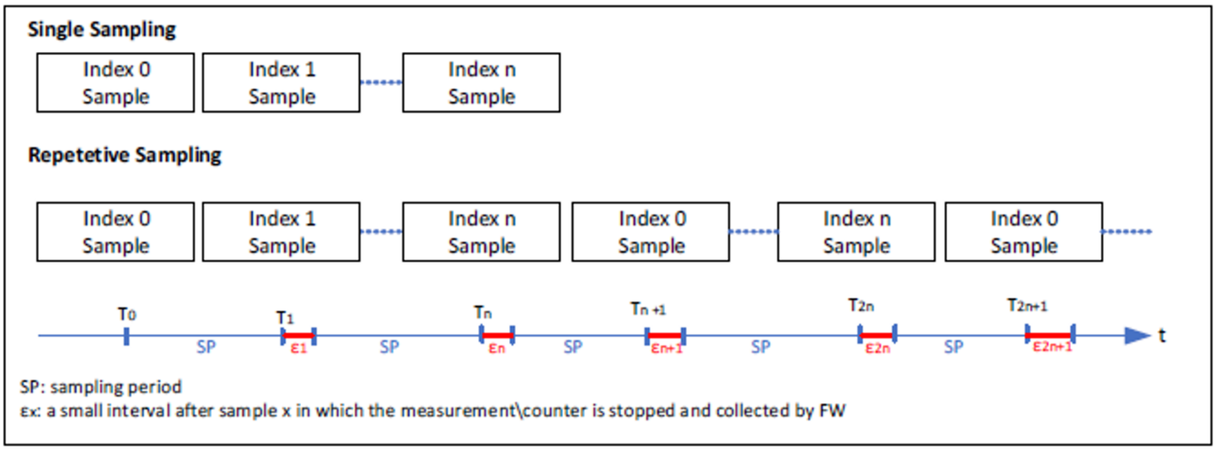
Synchronized start diagnostic counters can be cleared at the beginning of each sampling period.

The following diagrams illustrate how synchronized start affects the sampling timeline:

Without Synched Start:



With Synched Start:



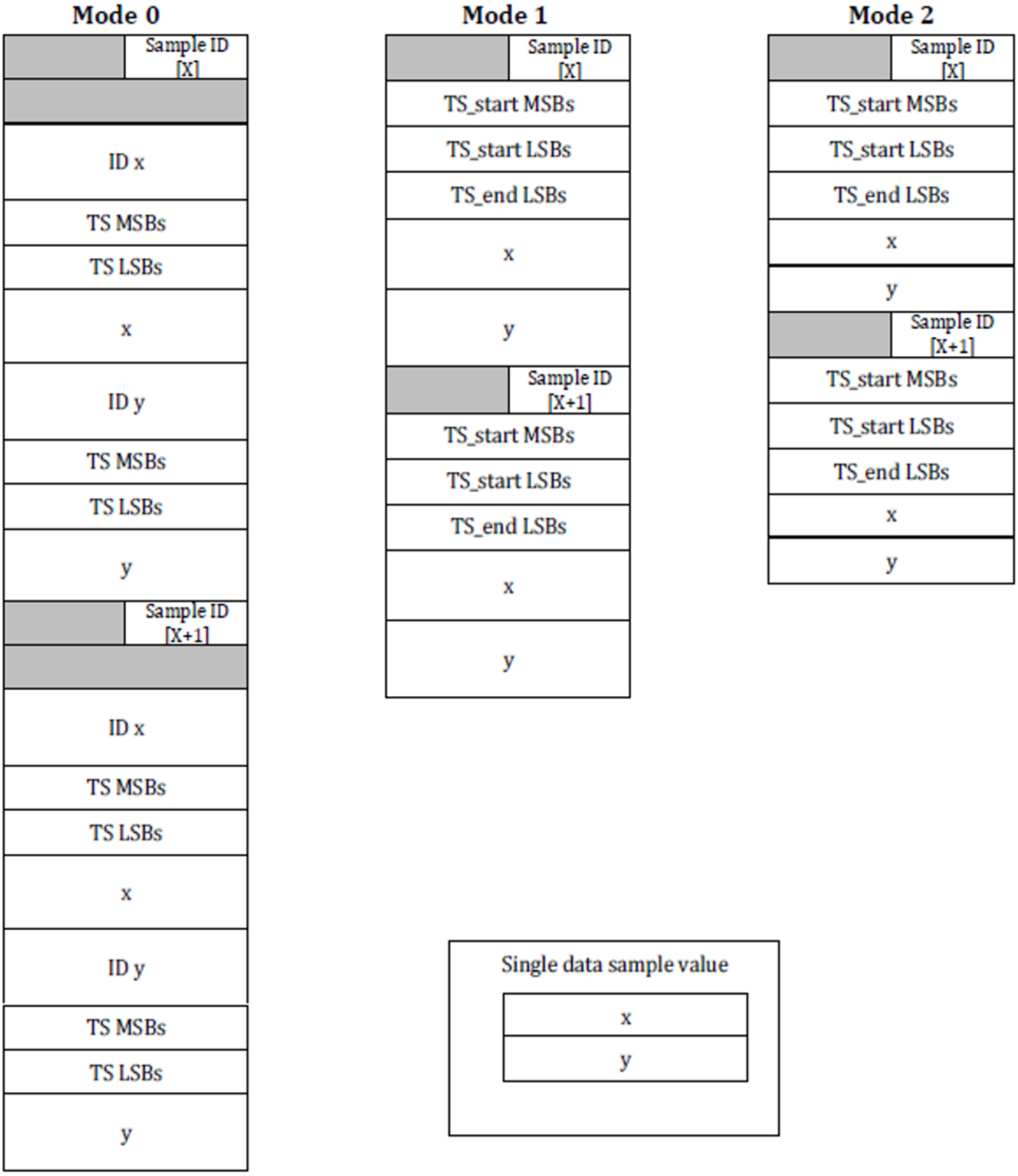
14.4.17.2.2 Output Formats

`doca_telemetry_diag` supports the following layout modes of the sampled data:

- Mode 0 - `data_id` is present in the output; data size is 64 bits; timestamp information per data
- Mode 1 - no `data_id` in the output; data size is 64 bits; timestamp information per sample (start and end)

- Mode 2 - no `data_id` in the output; data size is 32 bits; timestamp information per sample (start and end)

The sample layout of these modes is illustrated in the following diagrams:



14.4.17.2.3 Device and Ownership

`doca_telemetry_diag` requires a ConnectX/BlueField DOCA device to sample from. The device can be accessed using any of its physical functions (PFs). If multiple devices exist in a setup, a `doca_telemetry_diag` context should be created for each device.

`doca_telemetry_diag`, is designed to operate as a singleton per device. Upon creation, the `doca_telemetry_diag` context assumes control of the associated hardware resources to prevent conflicts and ensure accurate data sampling. In rare instances, ownership may be overridden (e.g., if a process crashed before releasing ownership). The `force_ownership` parameter may be used when creating the context from a second process.

> ⚠ Once ownership is enforced for one PF, it cannot be claimed by a different PF. It is recommended to always use PF0 to prevent potential conflicts.

14.4.17.2.4 State Machine

The `doca_telemetry_diag` context goes through the following states as it is being set up:

1. Idle - context is created. Ownership is taken. Capabilities can be queried. All configuration setters should be called except for configuring data IDs.
2. Configured - after calling `apply_configuration`. Internal initialization is called based on the applied configuration. Data IDs should be configured.
3. Ready - after setting the data IDs. Context is ready to start sampling.
4. Running - samples are generated and can be retrieved.

14.4.17.2.5 Data IDs

The on-device mechanism provides the following diagnostic data classes:

- Counter - monotonically increasing and counting different events in the device.
  - If `doca_telemetry_diag_set_data_clear` is set, the counters are cleared at the beginning of each sampling period (valid only if synchronized start mode is used and operational mode is set to single or repetitive sampling).
- Statistic - other collected diagnostic data about the performance of the device. Statistic diagnostic data is cleared on each sample.

Each diagnostic data is represented by a unique identifier, the data ID. Appendix "List of Supported Data IDs" lists the currently supported data IDs.

After applying the configuration, the list of data IDs to be sampled can be applied by calling `doca_telemetry_diag_apply_counters_list_by_id`. Not all combinations of data IDs can be configured. If any of `the_data_ids` fail to be configured, the operation fails, returning the index of the failed data ID and the reason of failure. The operation can be retried after omitting the faulty data ID.

14.4.17.3 Telemetry Diagnostics Sample

This section describes a telemetry diagnostics sample based on the `doca_telemetry_diag` library. The sample illustrates the utilization of DOCA telemetry diagnostics APIs to initialize and configure the `doca_telemetry_diag` context, as well as querying and parsing diagnostic counters.

Sample usage:

```
Usage: doca_telemetry_diag [DOCA Flags] [Program Flags]

DOCA Flags:
-h, --help                        Print a help synopsis
-v, --version                     Print program version information
-l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
--sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERRO
R, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
-j, --json <path>                 Parse all command flags from an input json file

Program Flags:
-p, --pci-addr                    DOCA device PCI device address
-o, --output                      Output CSV file - default: "/tmp/out.csv"
-rt, --sample-run-time            Total sample run time, in seconds
-sp, --sample-period              Sample period, in nanoseconds
-ns, --log-num-samples            Log max number of samples
-sr, --max-samples-per-read       Max num samples per read
-sm, --sync-mode                  Enable sync mode
```

14.4.17.4 Appendix - List of Supported Data IDs

The following table lists the data IDs currently supported by DOCA:

Name	Description	Data Class	Data ID
`port_rx_bytes`	The number of received bytes on the physical port [1]	Counter	0x10200001000000XX - XX - Port ID
`port_priority_rx_bytes`	The number of received bytes on the physical port and priority [1]	Counter	0x1020000200000YXX - XX - Port ID - Y - Priority
`port_rx_packets`	The number of received packets on the physical port [1]	Counter	0x10200003000000XX - XX - Port ID
`port_priority_rx_packets`	The number of received packets on the physical port and priority [1]	Counter	0x1020000400000YXX - XX - Port ID - Y - Priority
`port_rx_discard_buf_packets`	The number of received packets dropped due to lack of buffers on a physical port	Counter	0x10200005000000XX - XX - Port ID
`port_priority_rx_pauses_packets`	The number of link-layer pause packets received on a physical port and priority	Counter	0x1020000600000YXX - XX - Port ID - Y - Priority
`host_rx_transport_out_of_buffer_packets`	The number of dropped packets due to a lack of WQE for the associated QPs/RQs (excluding hairpin QPs/RQs)	Counter	0x10800002000000XX - XX - Host ID
`host_rx_transport_out_of_buffer_hairpin_packets`	The number of dropped packets due to a lack of WQE for the associated hairpin QPs/RQs	Counter	0x10800003000000XX - XX - Host ID
`port_rx_transport_ecn_packets`	The number of RoCEv2 packets received by the notification point which were marked for experiencing the congestion (i.e., ECN bits `11` on the ingress RoCE traffic), per port	Counter	0x10800004000000XX - XX - Local port
`port_rx_transport_cnp_handled_packets`	The number of CNP received packets handled by the Reaction Point, per port	Counter	0x10800005000000XX - XX - Local port

Name	Description	Data Class	Data ID
`port_tx_transport_cnp_sent_packets`	The number of CNP packets sent by the Notification Point, per port	Counter	0x11000001000000XX • XX - Local port
`tx_transport_done_due_to_cc_deschedule_events`	The number of QP descheduled due to congestion control rate limitation	Counter	0x1100000200000000
`port_tx_bytes`	The number of transmitted bytes on the physical port (excluding loopback traffic)	Counter	0x11400001000000XX • XX - Port ID
`port_priority_tx_bytes`	The number of transmitted bytes on the physical port and priority (excluding loopback traffic)	Counter	0x1140000200000YXX • XX - Port ID • Y - Priority
`port_tx_packets`	The number of transmitted packets on the physical port (excluding loopback traffic)	Counter	0x11400003000000XX • XX - Port ID
`port_priority_tx_packets`	The number of transmitted packets on the physical port and priority (excluding loopback traffic)	Counter	0x1140000400000YXX • XX - Port ID • Y - Priority
`port_priority_tx_pauses_packets`	The number of link-layer pause packets transmitted on a physical port and priority	Counter	0x1140000500000YXX XX - Port ID • Y - Priority
`pcie_link_inbound_bytes`	The number of bytes received from the PCIe toward the device, per PCIe link	Counter	0x1160000100ZZYYXX • XX - Node • YY - PCIe index • ZZ - Depth (0 - 63)
`pcie_link_outbound_bytes`	The number of bytes transmitted from the device toward the PCIe, per PCIe link	Counter	0x1160000200ZZYYXX • XX - Node • YY - PCIe index • ZZ - Depth (0 - 63)
`pcie_link_inbound_data_bytes`	The number of data bytes received from the PCIe (excluding headers) toward the device, per PCIe link	Counter	0x1160000200ZZYYXX • XX - Node • YY - PCIe index • ZZ - Depth (0 - 63)
`pcie_link_outbound_data_bytes`	The number of data bytes transmitted from the device toward the PCI (excluding headers), per PCIe link	Counter	0x1160000400ZZYYXX • XX - Node • YY - PCIe index • ZZ - Depth (0 - 63)
`pcie_link_write_stalled_time_no_posted_data_credits_ns`	The time period (in nanoseconds) in which the device had outbound posted write requests but stalled due to insufficient data credits per PCIe link	Counter	0x1160000500ZZYYXX • XX - Node • YY - PCIe index • ZZ - Depth (0 - 63)
`pcie_link_write_stalled_time_no_posted_header_credits_ns`	The time period (in nanoseconds) in which the device had outbound posted write requests but stalled due to insufficient header credits per PCIe link	Counter	0x1160000600ZZYYXX • XX - Node • YY - PCIe index • ZZ - Depth (0 - 63)

Name	Description	Data Class	Data ID
`pcie_link_read_stalled_time_no_non_posted_data_credits_ns`	The time period (in nanoseconds) in which the device had outbound non-posted read requests but stalled due to insufficient data credits per PCIe link	Counter	0x1160000700ZZYYXX - XX - Node - YY - PCIe index - ZZ - Depth (0 - 63)
`pcie_link_read_stalled_time_no_non_posted_header_credits_ns`	The time period (in nanoseconds) in which the device had outbound non-posted read requests but stalled due to insufficient header credits per PCIe link	Counter	0x1160000800ZZYYXX - XX - Node - YY - PCIe index - ZZ - Depth (0 - 63)
`pcie_link_read_stalled_time_no_completion_buffers_ns`	The time period (in nanoseconds) in which the device had outbound non-posted read requests but stalled due to no NIC completion buffers per PCIe link	Counter	0x1160000900ZZYYXX - XX - Node - YY - PCIe index - ZZ - Depth (0 - 63)
`pcie_link_tclass_read_stalled_time_ordering_ns`	The time period (in nanoseconds) in which the device had outbound non-posted read requests but stalled due to PCIe ordering semantics per PCIe link and PCIe tclass	Counter	0x1160000aZZZZYYXX - XX - Node - YY - PCIe index - ZZZZ - (tclass (0 - 7) << 6) | (Depth (0 - 63))
`pcie_link_latency_total_read_ns`	The total latency (in nanoseconds) for all PCIe read from the device per PCIe link Dividing this counter by `pcie_link_latency_total_read_packets` yields the average PCIe read latency of those reads.	Counter	0x1160000b00ZZYYXX - XX - Node - YY - PCIe index - ZZ - Depth (0 - 63)
`pcie_link_latency_total_read_packets`	The total number of packets used for the `pcie_link_latency_total_read_ns` calculation	Counter	0x1160000c00ZZYYXX - XX - Node - YY - PCIe index - ZZ - Depth (0 - 63)
`pcie_link_latency_max_read_ns`	The maximum latency (in nanoseconds) for a single PCIe read from the device per PCIe link	Statistic	0x1160000d00ZZYYXX - XX - Node - YY - PCIe index - ZZ - Depth (0 - 63)
`pcie_link_latency_min_read_ns`	The maximum latency (in nanoseconds) for a single PCIe read from the device per PCIe link	Statistic	0x1160000e00ZZYYXX - XX - Node - YY - PCIe index - ZZ - Depth (0 - 63)
`global_completion_engine_rx_cqes`	Number of responder (RX) CQEs	Counter	0x10c0000100000000
`function_completion_engine_rx_cqes`	Number of RX CQEs per function	Counter	0x10c000020000XXXX - XXXX - vhca_id
`global_completion_engine_tx_cqes`	Number of requestor (TX) CQEs	Counter	x10c0000400000000

Name	Description	Data Class	Data ID
`function_completion_engine_tx_cqes`	Number of TX CQEs per function	Counter	0x10c000050000XXXX • XXXX - vhca_id
`global_icmc_request`	Number of accesses to ICMC	Counter	0x1180000100000000
`global_icmc_hit`	Number of ICMC hits	Counter	0x1180000200000000
`global_icmc_miss`	Number of ICMC misses	Counter	0x1180000300000000

1. This counter includes loopback traffic and does not include packets discarded due to FCS, frame size, and similar errors. ↩ ↩ ↩ ↩

14.4.17.5 Known Limitations

Currently, the `doca_telemetry` library is supported at alpha level and is intended to allow developers to start testing applications using it.

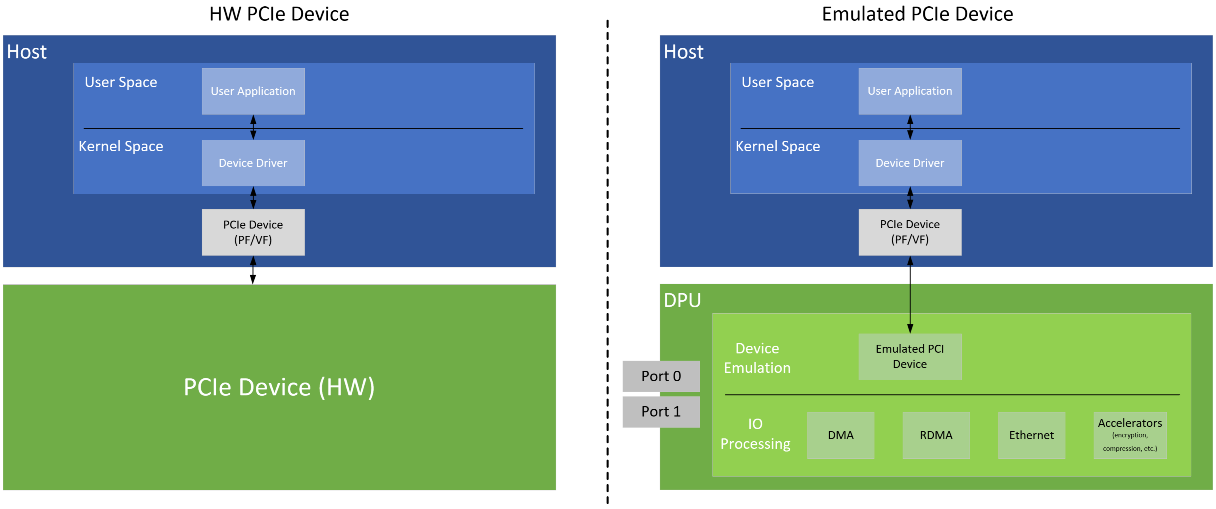
The following table lists the currently known limitations:

#	Item	Limitation
1	Output format	Only `DOCA_TELEMETRY_DIAG_OUTPUT_FORMAT_1` is supported.
2	Sample mode	Only `DOCA_TELEMETRY_DIAG_SAMPLE_MODE_REPETITIVE` is supported.

14.4.18 DOCA Device Emulation

14.4.18.1 Introduction

NVIDIA® BlueField® networking platforms (DPUs or SuperNICs) provide the ability to emulate a PCIe device. The DOCA Device Emulation subsystem provides a low-level software API for users to develop PCIe devices and their controllers. These APIs include discovery, configuration, hot plugging/unplugging, management, and IO path handling. In simpler terms, the libraries enable the user to implement a hardware PCIe function using software, such that the host is not aware that the PCIe function is emulated, and all interactions from the host are routed to software on the BlueField instead of actual hardware.



The diagram shows the potential for device emulation to replace a regular PCIe function of some PCIe device.

- On the left is a conventional setup where the host is connected to a PCIe device (e.g., NVMe SSD). On the host, user applications interact with the kernel driver of that device, using some software interface, and the driver communicates with the hardware/firmware of the device.
- On the right is a setup where the PCIe device is replaced with a BlueField with an application using DOCA Device Emulation. The application can use the DOCA DevEmu PCI library to control the device, and intercept any IOs written by the host to the PCIe device. Additionally, the application can use other DOCA libraries to perform IO processing (e.g., copying data from host memory using DMA, sending RDMA/Ethernet traffic) and other acceleration libraries for encryption, compression, etc.

14.4.18.2 Known Limitations

- This library is supported at alpha level; backward compatibility is not guaranteed
- VFs are not currently supported
- Some limitations apply when creating a generic emulated function, for more details refer to DOCA DevEmu PCI Generic Limitations.
- Consult your NVIDIA representative for limitations on the emulated device's behavior

14.4.18.3 DOCA DevEmu PCI

⚠ This library is supported at alpha level; backward compatibility is not guaranteed.

14.4.18.3.1 Introduction

DOCA DevEmu PCI is part of the DOCA Device Emulation subsystem. It provides low-level software APIs that allow management of an emulated PCIe device using the emulation capability of NVIDIA® BlueField® networking platforms.

It is a common layer for all PCIe emulation modules, such as DOCA DevEmu PCIe Generic Emulation, and DOCA DevEmu Virtio subsystem emulation.

14.4.18.3.2 Prerequisites

This library follows the architecture of a DOCA Core Context. It is recommended read the following sections beforehand:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

Generic device emulation is part of DOCA device emulation. It is recommended to read the following guides beforehand:

- DOCA Device Emulation

14.4.18.3.3 Environment

DOCA DevEmu PCI Emulation is supported only on the BlueField target. The BlueField must meet the following requirements

- DOCA version 2.7.0 or greater
- BlueField-3 firmware 32.41.1000 or higher

> Please refer to the DOCA Backward Compatibility Policy.

The library must be run with root privileges.

Perform the following:

1. Configure the BlueField to work in DPU mode as described in NVIDIA BlueField Modes of Operation.
2. Enable the PCIe switch emulation capability needed for hot plugging emulated PCIe devices. This can be done by running the following command on the host or BlueField:

   ```
   host/bf> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s PCI_SWITCH_EMULATION_ENABLE=1
   ```
3. Perform a BlueField system-level reset for the `mlxconfig` settings to take effect.

To support hot-plug feature, the host must have the following boot parameters:

- Intel CPU:

  ```
  intel_iommu=on iommu=pt pci=realloc
  ```
- AMD CPU:

  ```
  iommu=pt pci=realloc
  ```

This can be done using the following steps:

> This process may vary depending on the host OS. Users can find multiple guides online describing this process.

1. Add the boot parameters:

```
host> sudo nano /etc/default/grub
Find the variable
GRUB_CMDLINE_LINUX_DEFAULT="<existing-params>"
Add the params at the end
GRUB_CMDLINE_LINUX_DEFAULT="<existing-params> intel_iommu=on iommu=pt pci=realloc"
```

2. Update configuration.
   - For Ubuntu:

     ```
     host> update-grub
     ```

   - For RHEL:

     ```
     host> grub2-mkconfig -o /boot/grub2/grub.cfg
     ```

3. Perform warm boot.
4. Confirm that the parameters are in effect:

```
host> cat /proc/cmdline
<existing-params> intel_iommu=on iommu=pt pci=realloc
```

14.4.18.3.4 Architecture

The DOCA DevEmu PCI library provides 2 main software abstractions, the PCIe type, and the PCIe device. The PCIe type represents the configurations of the emulated device, while the PCIe device represents an instance of an emulated device. Furthermore, any PCIe device instance must be associated with a single PCIe type, while PCIe type can be associated with many PCIe devices.

14.4.18.3.4.1 Pre Defined PCI Type vs. Generic PCI Type

A PCIe type object can be acquired in 2 different ways:

- Acquire a pre-defined type, using emulation libraries of existing protocols such as DOCA DevEmu Virtio FS library
- Create from scratch using the DOCA DevEmu Generic library

In case of pre-defined type, the configurability of the type is limited.

14.4.18.3.4.2 PCIe Type Name

As part of the DOCA PCIe emulation, every type has a name assigned to it. This property is not part of the PCIe specification, but rather it is a mechanism in DOCA that uniquely identifies the PCIe type.

There cannot be 2 different PCIe types with the same name, even across different processes, unless the type in the second process is configured in identical manner to the first one. Furthermore, attempting to configure the second type with same name but with slight configuration difference will fail.

14.4.18.3.4.3 Create Emulated Device

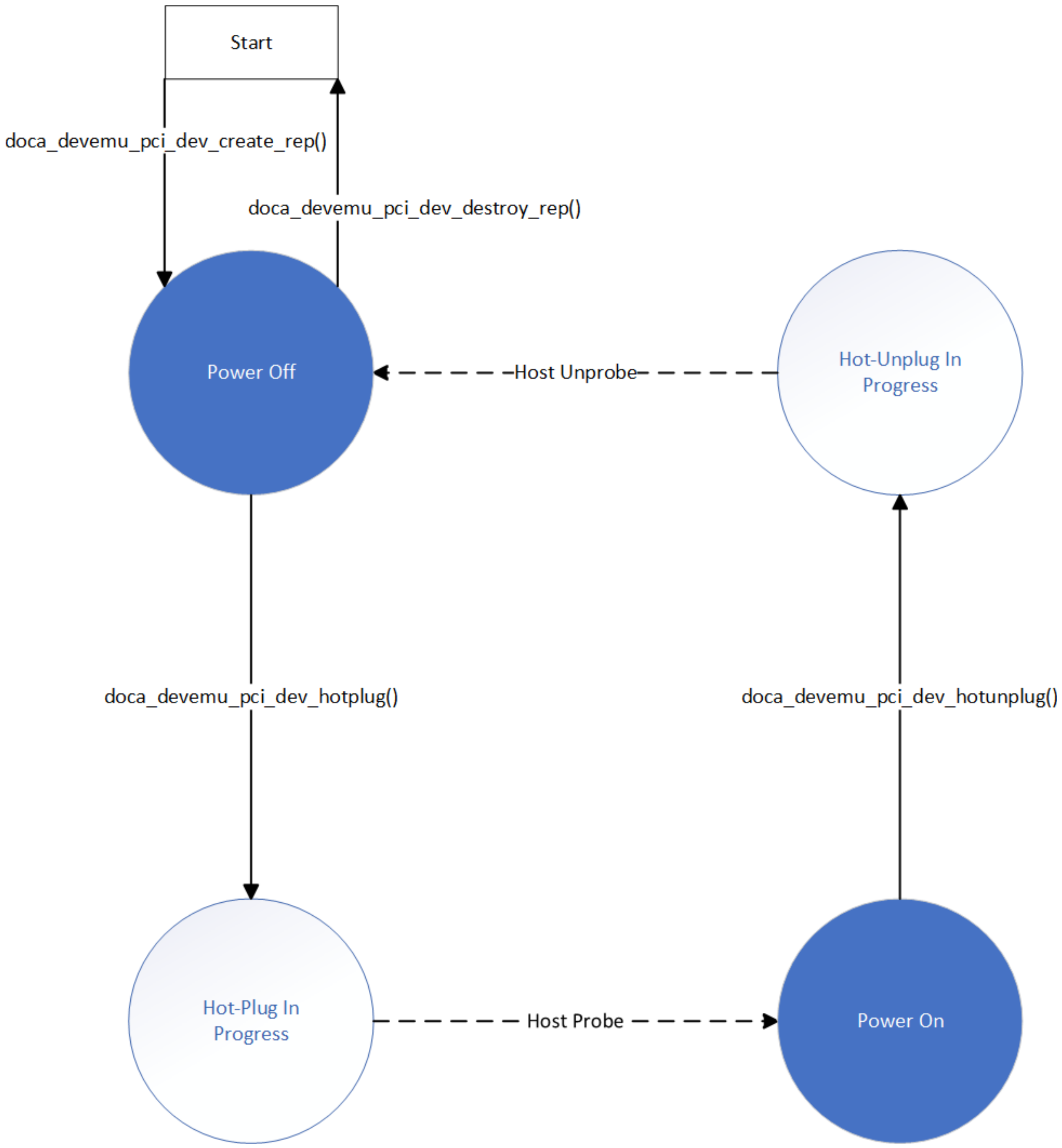
After configuring the desired DOCA Devemu PCIe type, it is possible to create an emulated device based on the configured type using `doca_devemu_pci_dev_create_rep`. This sequential process ensures that the DOCA DevEmu PCIe device is created with the specified parameters and configuration defined by the PCIe type object. Furthermore, it is possible to destroy the emulated device using `doca_devemu_pci_dev_destroy_rep`.

The created device representor starts in "power_off" state and is not visible to the host until hot-plug sequence is issued by the user, see Hot-plug Emulated Device. The device can then be destroyed only while in "power_off" state.

> ⓘ The created emulated device may outlive the application that created it, see Objects Lifecycle and Persistency.

14.4.18.3.4.4 Hot-plug Emulated Device

Hot-plugging refers to the process of emulating the physical attachment of a PCIe device to the host PCIe subsystem after the system has been powered on and initialized. Note that some operating systems require additional settings to enable the process of hot-plugging a PCIe device. For supported systems, this feature proves particularly advantageous for systems that need to remain operational at all times while expanding their hardware resources, such as additional storage and networking capabilities. DOCA DevEmu PCI provides software APIs that allow users to emulate this process in an asynchronous manner.


Start
doca_devemu_pci_dev_create_rep()
doca_devemu_pci_dev_destroy_rep()
Power Off
Host Unprobe
Hot-Unplug In Progress
doca_devemu_pci_dev_hotplug()
doca_devemu_pci_dev_hotunplug()
Hot-Plug In Progress
Host Probe
Power On

When creating a PCIe device object, if it starts in "power off" state, then the device is not yet visible to the host. It is possible then, from the BlueField, to hot-plug the device. This starts an async process of the device getting hot-plugged towards the host. Once the process completes, the emulated device transitions to "power on" and becomes visible to the host. Usually at this stage, the emulated device receives its BDF address. The hot-unplug process works in similar async manner.

Using DOCA API, the BlueField Arm can register to any changes to the hot-plug state of each emulated device using `doca_devemu_pci_dev_event_hotplug_state_change_register`.

14.4.18.3.4.5 Emulated Device Discovery

The emulated device is represented as a `doca_devinfo_rep`. It is possible to iterate through all the emulated devices as explained in DOCA Core Representor Discovery.

There are 2 ways of filtering the list of emulated devices:

- Get all emulated devices - use `DOCA_DEVINFO_REP_FILTER_EMULATED` as the filter argument in `doca_devinfo_rep_create_list`
- Get all emulated devices that belong to a certain type - `doca_devemu_pci_type_create_rep_list`

14.4.18.3.4.6 Objects Lifecycle and Persistency

This section creates distinction between firmware resources and software resources:

- Firmware resources persist until the next power cycle, and can be accessible from different processes on the BlueField Arm. Such resources are not cleared once the application exits.
- Software resources are representations of firmware resources, and are only relevant for the same thread

Using this terminology, it is possible to describe the objects as follows:

- The PCIe type object `doca_devemu_pci_type` represents a PCIe type firmware resource. The resource persists if any of the following apply:
  - There is at least 1 process holding reference to the PCIe type
  - There is at least 1 PCIe device firmware resource belonging to this type
- The emulated device representor, `doca_devinfo_rep`, represents an emulated PCIe function firmware resource:
  - `doca_devemu_pci_dev_create_rep` can be used to create such firmware resource
  - To destroy the firmware resource, `doca_devemu_pci_dev_destroy_rep` can be used
  - For static functions, the representor resource persists until configured otherwise in NVCONFIG
  - To find existing PCIe device firmware resources, use `doca_devemu_pci_type_create_rep_list`

14.4.18.3.4.7 Function Level Reset

The created emulated devices support PCIe function level reset (FLR).

Using DOCA API, the BlueField Arm can register to FLR event using `doca_devemu_pci_dev_event_flr_register`. Once the driver requests FLR, this event is triggered, calling the user provided callback.

Once FLR is detected, it is expected for the BlueField Arm to do the following:

- Destroy all resources related to the PCIe device. For information on such resources, refer to the guide of concrete PCIe type (generic/virtiofs).
- Stop the PCIe device
- Start the PCIe device again

14.4.18.3.5 Device Support

DOCA PCIe Device emulation requires a device to operate. For picking a device, see DOCA Core Device Discovery.

The device emulation library is only supported for BlueField-3.

As device capabilities may change in the future (see Capability Checking), it is recommended that users choose a device using the following method:

- `doca_devemu_pci_cap_type_is_hotplug_supported` - for create and hot-plug support
- `doca_devemu_pci_cap_type_is_mgmt_supported` - for device discovery only

14.4.18.3.6 PCIe Device

14.4.18.3.6.1 Configuration Phase

To start using the DOCA DevEmu PCI Device, users must first go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context to allow retrieval of events.

Configurations

The context can be configured to match the application use case.

To find if a configuration is supported or what its min/max value is, refer to Device Support.

Mandatory Configurations

All mandatory configurations are provided during the creation of the PCIe device.

These configurations are as follows:

- A DOCA DevEmu PCIe type object
- A DOCA Device Representor, representing an emulated function with the same type as the provided PCIe object type
- A DOCA Progress Engine object

Optional Configurations

These configurations are optional. If not set, then a default value is used:

- Registering to events as described in the "Events" section. By default, the user does not receive events.

14.4.18.3.6.2 Execution Phase

This section describes execution on CPU using DOCA Core Progress Engine.

Events

The DOCA DevEmu PCI device exposes asynchronous events to notify about sudden changes according to DOCA Core architecture.

Common events are described in DOCA Core Event.

Hotplug State Change

The hotplug state change event allows users to receive notifications whenever the hotplug state of the emulated device changes. See section "Hot-plug Emulated Device".

Event Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	`doca_devemu_pci_dev_event_hotplug_state_change_register`	`doca_devemu_pci_cap_type_is_hotplug_supported`

Event Trigger Condition

The event is triggered anytime an asynchronous transition happens as follows:

- `DOCA_DEVEMU_PCI_HP_STATE_PLUG_IN_PROGRESS` → `DOCA_DEVEMU_PCI_HP_STATE_POWER_ON`
- `DOCA_DEVEMU_PCI_HP_STATE_UNPLUG_IN_PROGRESS` → `DOCA_DEVEMU_PCI_HP_STATE_POWER_OFF`
- `DOCA_DEVEMU_PCI_HP_STATE_POWER_ON` → `DOCA_DEVEMU_PCI_HP_STATE_UNPLUG_IN_PROGRESS` (when initiated by the host)

Any transition initiated by user is not triggered (e.g., calling hotplug to transition from `POWER_OFF` to `PLUG_IN_PROGRESS`).

The following APIs can be used to initiate hotplug or hot-unplug transition processes:

- `doca_devemu_pci_dev_hotplug`
- `doca_devemu_pci_dev_hotunplug`

Event Output

Common output as described in DOCA Core Event.

Additionally, the internal cached hotplug state is updated and can be fetched using `doca_devemu_pci_dev_get_hotplug_state` .

Event Handling

Once the event is triggered, it means that the hotplug state has changed. The application is expected to do the following:

- Retrieve the new hotplug state using `doca_devemu_pci_dev_get_hotplug_state`

Function Level Reset

The FLR event allows users to receive notifications whenever the host initiates an FLR flow. See section "Function Level Reset".

Event Configuration

Description	API to Set the Configuration
Register to the event	`doca_devemu_pci_dev_event_flr_register`

Event Trigger Condition

The event is triggered anytime the host driver initiates an FLR flow. See section "Function Level Reset".

Event Output

Common output as described in DOCA Core Event.

Additionally, the internal cached FLR indicator is updated and can be fetched using `doca_devemu_pci_dev_is_flr`.

Event Handling

Once the event is triggered, it means that the host driver has initiated the FLR flow.

The user must handle the FLR flow by doing the following:

1. Flush all the outstanding requests back to the associated resource
2. Release all the PCIe device resources dynamically created after device start
3. Stop the PCIe device - `doca_ctx_stop`
4. Start the PCIe device again - `doca_ctx_start`
    - Call `doca_pe_progress` repeatedly until the PCIe device transitions to "running" state

For more information on starting the PCIe device again, refer to section "State Machine".

14.4.18.3.6.3 State Machine

The DOCA DevEmu PCI device object follows the context state machine as described in DOCA Core Context State Machine.

The following section describes how to transition to any state and what is allowed in each state.

Idle

In this state, it is expected that application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to section "Configurations"
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop after making sure all resources have been destroyed
Stopping	Call progress until all resources have been destroyed

Starting

In this state, it is expected that application:

- Calls progress to allow transition to next state

- Keeps context in this state until FLR flow is complete

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after receiving FLR event (i.e., while FLR is in progress)

Running

In this state, it is expected that application:

- Calls progress to receive events
- Creates/destroys PCIe device resources

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration
Starting	Call progress until FLR flow is completed

Stopping

In this state, it is expected that application:

- Destroys all emulated device resources as described in section "Function Level Reset".

Allowed operations:

- Destroying PCIe device resources

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call stop without freeing emulated device resources

14.4.18.3.7 DOCA DevEmu PCI Generic

⚠ This library is supported at alpha level; backward compatibility is not guaranteed.

This guide provides instructions on building and developing applications that require emulation of a generic PCIe device.

14.4.18.3.7.1 Introduction

DOCA DevEmu PCI Generic is part of the DOCA Device Emulation subsystem. It provides low-level software APIs that allow creation of a custom PCIe device using the emulation capability of NVIDIA® BlueField®.

For example, it enables emulating an NVMe device by creating a generic emulated device, configuring its capabilities and BAR to be compliant with the NVMe spec, and operating it from the DPU as necessary.

14.4.18.3.7.2 Prerequisites

This library follows the architecture of a DOCA Core Context. It is recommended read the following sections beforehand:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

Generic device emulation is part of DOCA PCIe device emulation. It is recommended to read the following guides beforehand:

- DOCA Device Emulation
- DOCA DevEmu PCI

14.4.18.3.7.3 Environment

DOCA DevEmu PCI Generic Emulation is supported only on the BlueField target. The BlueField must meet the following requirements:

- DOCA version 2.7.0 or greater
- BlueField-3 firmware 32.41.1000 or higher

ⓘ Please refer to the DOCA Backward Compatibility Policy.

Library must be run with root privileges.

Please refer to DOCA DevEmu PCI Environment, for further necessary configurations.

14.4.18.3.7.4 Architecture

DOCA DevEmu PCI Generic allows the creation of a generic PCI type. The PCI Type is part of the DOCA DevEmu PCI library. It is the component responsible for configuring the capabilities and bar layout of emulated devices.

The PCI Type can be considered as the template for creating emulated devices. Such that the user first configures a type, and then they can use it to create multiple emulated devices that have the same configuration.

For a more concrete example, consider that you would like to emulate an NVMe device, then you would create a type and configure its capabilities and BAR to be compliant with the NVMe spec, after that you can use the same type, to generate multiple NVMe emulated devices.

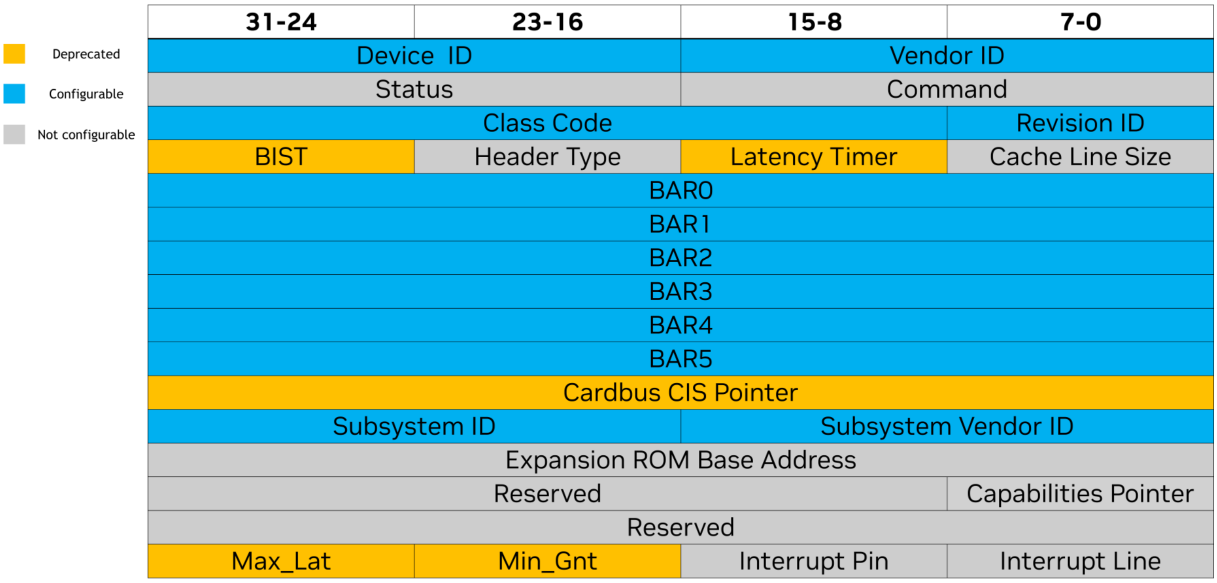
PCIe Configuration Space

The PCIe configuration space is 256 bytes long and has a header that is 64 bytes long. Each field can be referred to as a register (e.g., device ID).

Every PCIe device is required to implement the PCIe configuration space as defined in the PCIe specification.

The host can then read and/or write to registers in the PCIe configuration space. This allows the PCIe driver and the BIOS to interact with the device and perform the required setup.

It is possible to configure registers in the PCIe configuration space header as shown in the following diagram:



🟨 Deprecated

🟦 Configurable

⬜ Not configurable

31-24	23-16	15-8	7-0
Device ID		Vendor ID	
Status		Command	
Class Code			Revision ID
BIST	Header Type	Latency Timer	Cache Line Size
BAR0			
BAR1			
BAR2			
BAR3			
BAR4			
BAR5			
Cardbus CIS Pointer			
Subsystem ID		Subsystem Vendor ID	
Expansion ROM Base Address			
Reserved			Capabilities Pointer
Reserved			
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line

ⓘ 0x0 is the only supported header type (general device).

The following registers are read-only, and they are used to identify the device:

Register Name	Description	Example
Class Code	Defines the functionality of the device Can be further split into 3 values {class : subclass: prog IF}	0x020000 Class: 0x02 (Network Controller) Subclass: 0x00 (Ethernet Controller) Prog IF: 0x00 (N/A)
Revision ID	Unique identifier of the device revision Vendor allocates ID by itself	0x01 (Rev 01)
Vendor ID	Unique identifier of the chipset vendor Vendor allocates ID from the PCI-SIG	0x15b3 Nvidia
Device ID	Unique identifier of the chipset Vendor allocates ID by itself	0xa2dc BlueField-3 integrated ConnectX-7 network controller
Subsystem Vendor ID	Unique identifier of the card vendor Vendor allocates ID from the PCI-SIG	0x15b3 Nvidia
Subsystem ID	Unique identifier of the card Vendor allocates ID by itself	0x0051

BAR

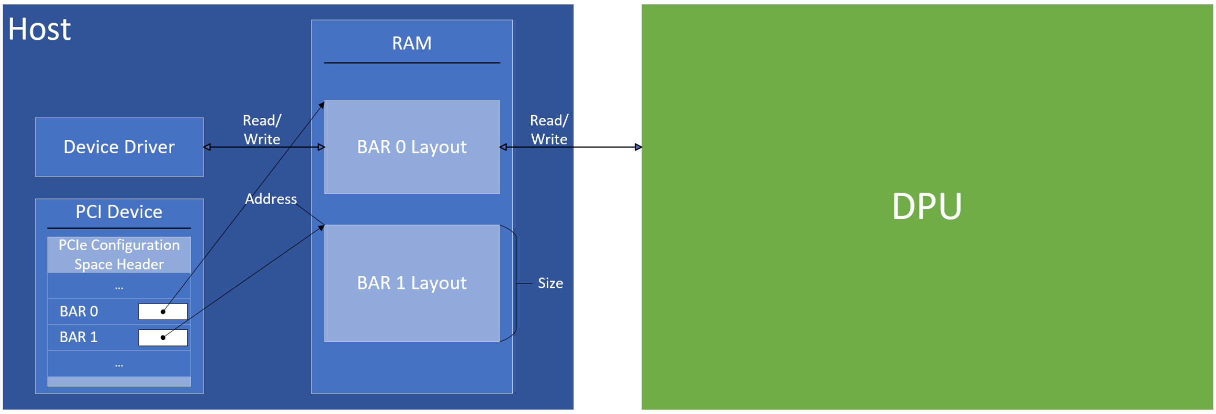
While the PCIe configuration space can be used to interact with the PCIe device, it is not enough to implement the functionality that is targeted by the device. Rather, it is only relevant for the PCIe layer.

To enable protocol-specific functionality, the device configures additional memory regions referred to as base address registers (BARs) that can be used by the host to interact with the device.

Different from the PCIe configuration space, BARs are defined by the device and interactions with them is device-specific. For example, the PCIe driver interacts with an NVMe device's PCIe configuration space according to the PCIe spec, while the NVMe driver interacts with the BAR regions according to the NVMe spec.

Any read/write requests on the BAR are typically routed to the hardware, but in case of an emulated device, the requests are routed to the software.

The DOCA DevEmu PCI type library provides APIs that allow software to pick the mechanism used for routing the requests to software, while taking into consideration common design patterns utilized in existing devices.



Each PCIe device can have up to 6 BARs with varying properties. During the PCIe bus enumeration process, the PCIe device must be able to advertise information about the layout of each BAR. Based on the advertised information, the BIOS/OS then allocates a memory region for each BAR and assigns the address to the relevant BAR in the PCIe configuration space header. The driver can then use the assigned memory address to perform reads/writes to the BAR.

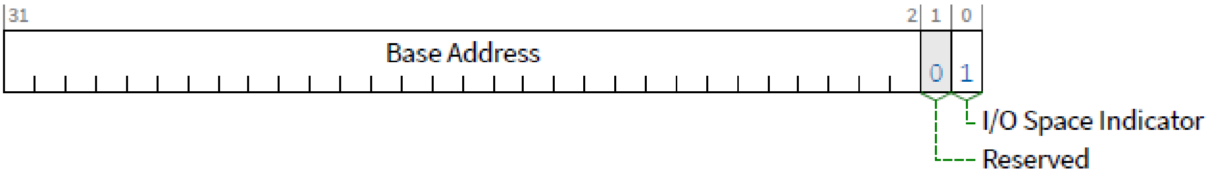
BAR Layout

The PCIe device must be able to provide information with regards to each BAR's layout.

The layout can be split into 2 types, each with their own properties as detailed in the following subsections.

I/O Mapped

According to the PCIe specification, the following represents the I/O mapped BAR:



Additionally, the BAR register is responsible for advertising the requested size during enumeration.

> The size must be a power of 2.

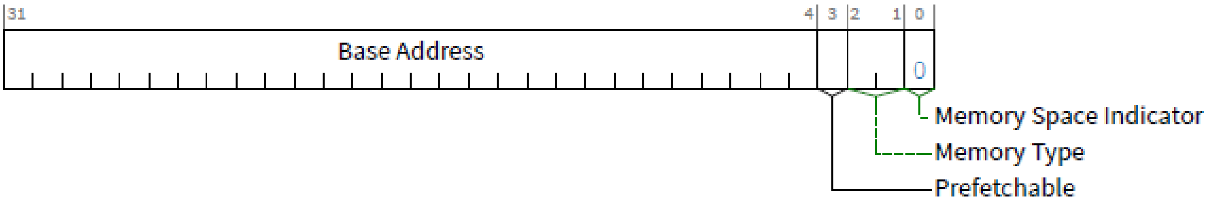
Users can use the following API to set a BAR as I/O mapped:

```
doca_devemu_pci_type_set_io_bar_conf(struct doca_devemu_pci_type *pci_type, uint8_t id, uint8_t log_sz)
```

- `id` - the BAR ID
- `log_sz` - the log of the BAR size

Memory Mapped

According to the PCIe specification, the following represents the memory mapped BAR:



Additionally, the BAR register is responsible for advertising the requested size during enumeration.

> ⓘ The size must be a power of 2.

The memory mapped BAR allows a 64-bit address to be assigned. To achieve this, users must specify the bar Memory Type as 64-bit, and then set the next BAR's (BAR ID + 1) size to be 0.

Setting the pre-fetchable bit indicates that reads to the BAR have no side-effects.

Users can use the following API to set a BAR as memory mapped:

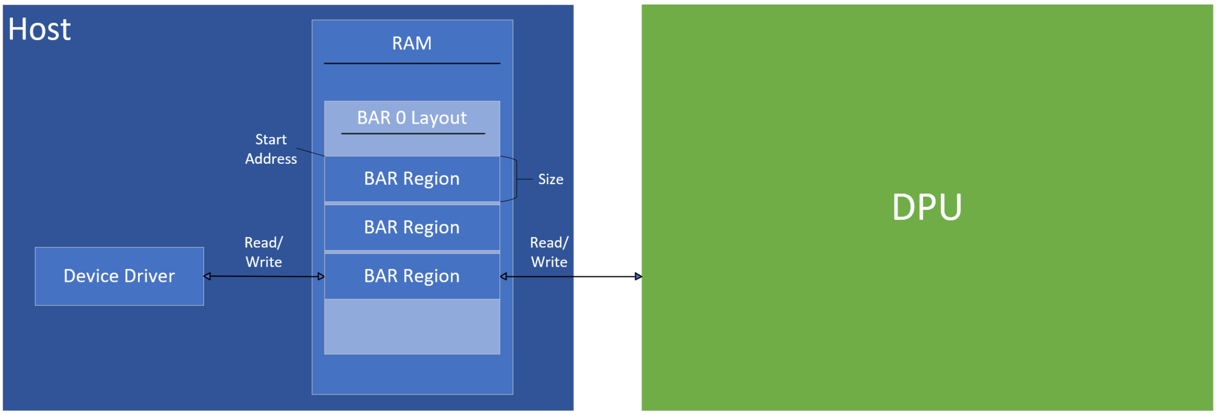
```
doca_devemu_pci_type_set_memory_bar_conf(struct doca_devemu_pci_type *pci_type, uint8_t id, uint8_t log_sz, enum
 doca_devemu_pci_bar_mem_type memory_type, uint8_t prefetchable)
```

- `id` - the BAR ID
- `log_sz` - the log of the BAR size. If set to 0, then the size is considered as 0 (instead of 1).
- `memory_type` - specifies the memory type of the BAR. If set to 64-bit, then the next BAR must have `log_sz` set to 0.
- `prefetchable` - indicates whether the BAR memory is pre-fetchable or not (a value of 1 or 0 respectively)

BAR Regions

BAR regions refer to memory regions that make up a BAR layout. This is not something that is part of the PCIe specification, rather it is a DOCA concept that allows the user to customize behavior of the BAR when interacted with by the host.

The BAR region defines the behavior when the host performs a read/write to an address within the BAR, such that every address falls in some memory region as defined by the user.



Common Configuration

All BAR regions have these configurations in common:

- `id` - the BAR ID that the region is part of
- `start_addr` - the start address of the region within the BAR layout relative to the BAR. 0 indicates the start of the BAR layout.
- `size` - the size of the BAR region

Currently, there are 4 BAR region types, defining different behavior:

- Stateful
- DB by offset
- DB by data
- MSIX table
- MSIX PBA

Generic Control Path (Stateful BAR Region)

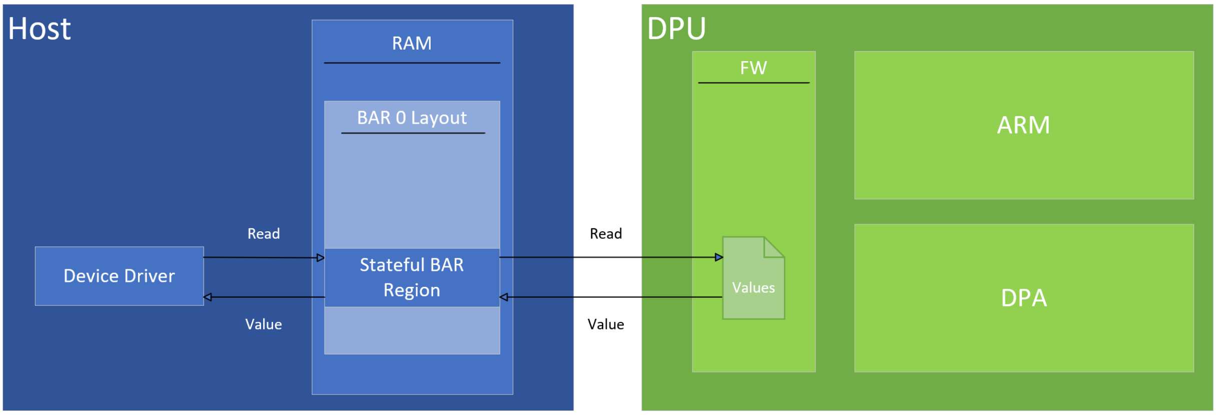
Stateful region can be used as a shared memory, such that the contents are maintained in firmware. A read from the driver returns the latest value, while a write updates the value and triggers an event to software running on the DPU.

This can be useful for communication between the driver and the device, during the control path (e.g., exposing capabilities, initialization).

> ⓘ Some limitations apply, please see Limitations section

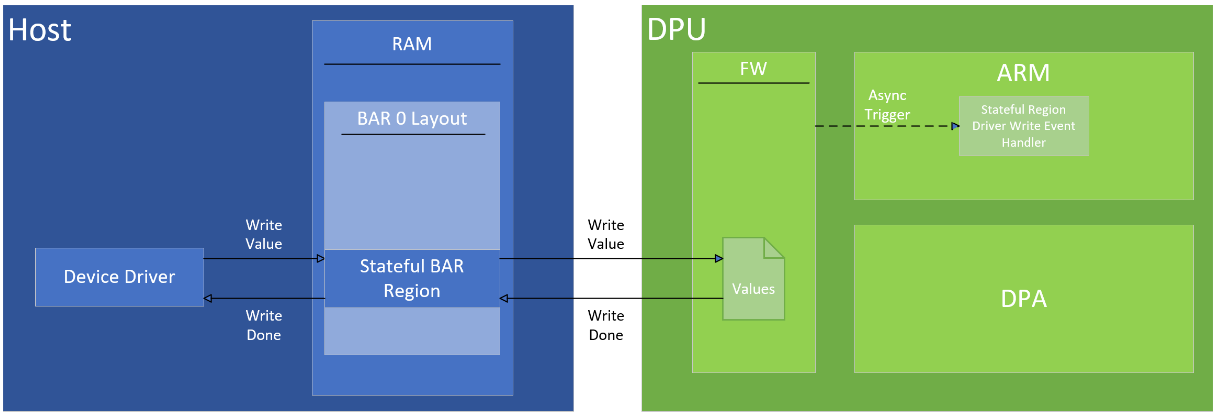
Driver Read

A read from the driver returns the latest value written to the region, whether written by the host or by the driver itself.



Driver Write

A write from the driver updates the value at the written address and notifies software running on the Arm that a write has occurred. The notification on the Arm arrives as an asynchronous event (see `doca_devemu_pci_dev_event_bar_stateful_region_driver_write`).



The event that arrives to Arm software is asynchronous such that it may arrive after the driver has completed the write.

DPU Read

The DPU can read the values of the stateful region using `doca_devemu_pci_dev_query_bar_stateful_region_values`. This returns the latest snapshot of the stateful region values. It can be particularly useful to find what was written by the driver after the "stateful region driver write event" occurs.

DPU Write

The DPU can write the values of the stateful region using `doca_devemu_pci_dev_modify_bar_stateful_region_values`. This updates the values such that subsequent reads from the driver or the DPU returns these values.

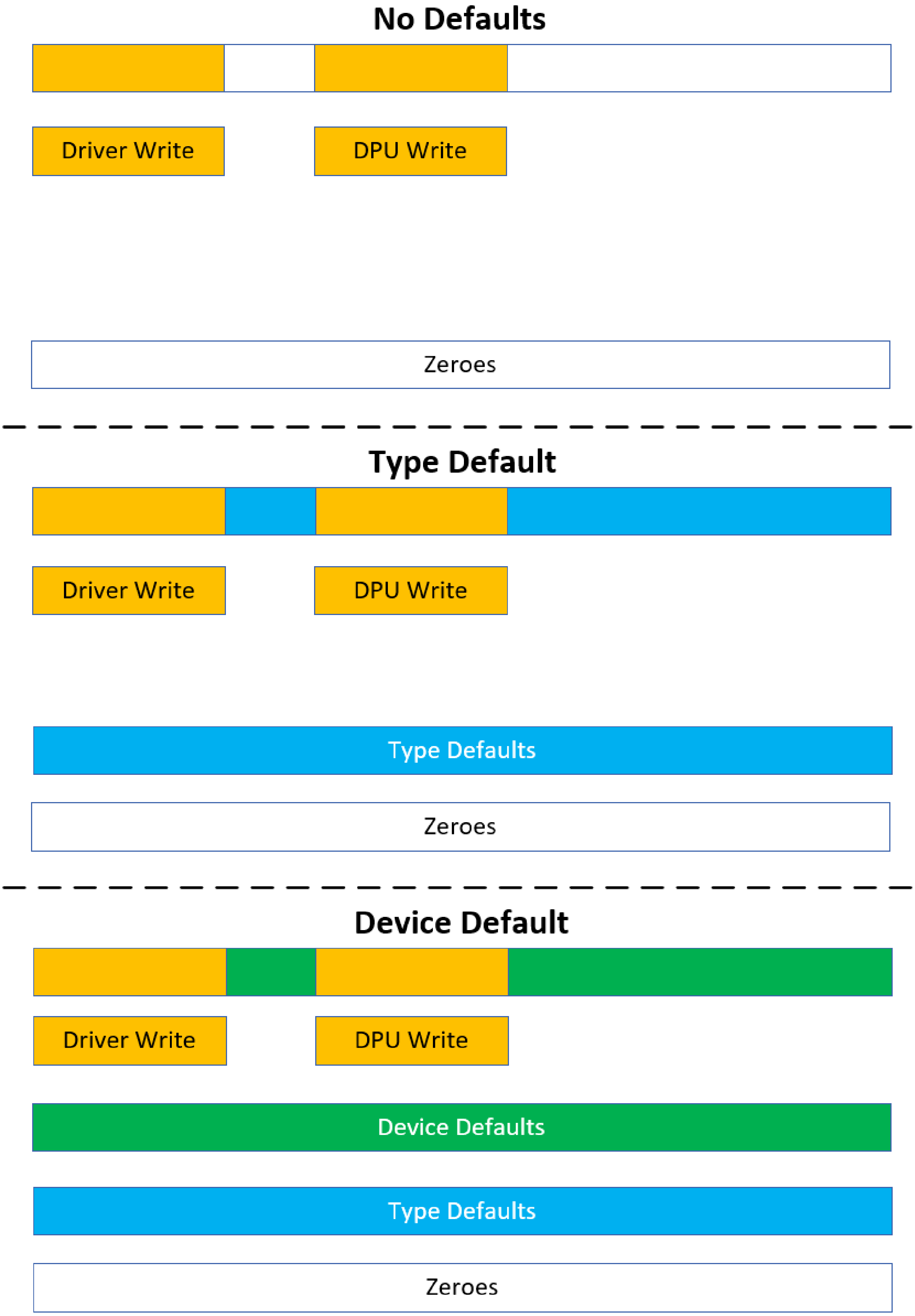
Default Values

The DPU is able to set default values to the stateful region. Default values come in 2 layers:

- Type default values – these values are set for all devices that have the same type. This can be set only if no device currently exists.
- Device default values – these values are set for a specific device and take affect on the next FLR cycle or the next hotplug of the device

A read of the stateful region follows the following hierarchy:

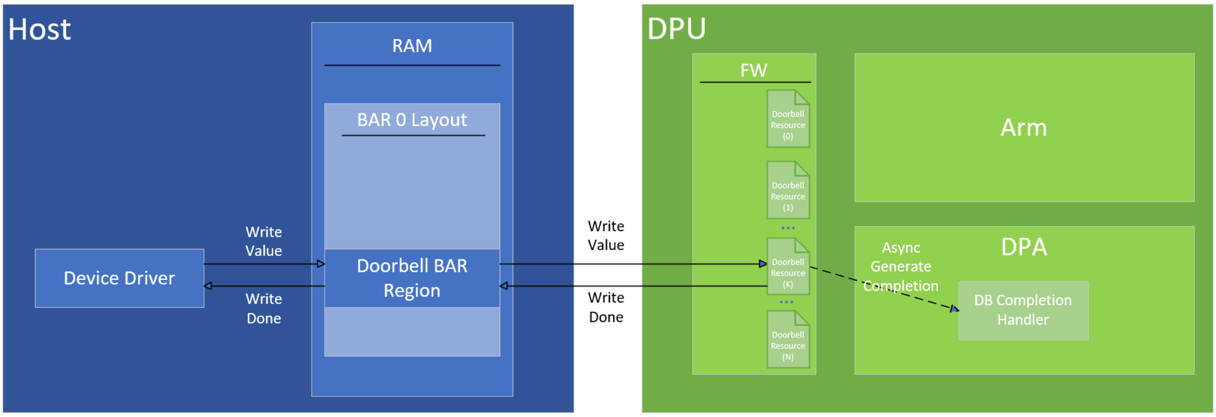
1. Return the latest value as written by the host or driver (whichever was done last).
2. Return the device default values.
3. Return the type default values.
4. Return 0.


No Defaults
Driver Write
DPU Write
Zeroes
Type Default
Driver Write
DPU Write
Type Defaults
Zeroes
Device Default
Driver Write
DPU Write
Device Defaults
Type Defaults
Zeroes

Generic Data Path (DB BAR Region)

Doorbell (DB) regions can be used to implement a consumer-producer queue between the driver and the DPU, such that a write from the driver would trigger an event on the DPU through DPA, allowing it to fetch the written value. This can be useful for communication between the driver and the device, during the data path allowing IO processing.

While DBs are not part of the PCIe specification, it is a widely used mechanism by vendors (e.g., RDMA QP, NVMe SQ, virtio VQ, etc).



The same DB region can be used to manage multiple DBs, such that each DB can be used to implement a queue.

The DPU software can utilize DB resources individually:

- Each DB resource has a unique zero-based index referred to as DB ID
- DB resource can be managed (create/destroy/modify/query) individually
- Each DB resource has a separate notification mechanism. That is, the notification on DPU is triggered for each DB separately.

Driver Write

The DB usually consists of a numeric value (e.g., `uint32_t`) representing the consumer/producer index of the queue.

When the driver writes to the DB region, the related DB resource gets updated with the written value, and a notification is sent to the DPU.

When driver writes to the DB BAR region it must adhere to the following:

- The size of the write must match the size of the DB value (e.g., `uint32_t`)
- The offset within the region must be aligned to the DB stride size or the DB size

The flow would look something as the following:

- Driver performs a write of the DB value at some offset within the DB BAR region
- DPU calculates the DB ID that the write is intended for. Depending on the region type:
    - DB by offset - DPU calculates the DB ID based on the write offset relative to the DB BAR region
    - DB by data - DPU parses the written DB value and extracts the DB ID from it
- DPU updates the DB resource with the matching DB ID to the value written by the driver
- DPU sends a notification to the DPA application, informing it that the value of DB with DB ID has been updated by the driver

Driver Read

The driver should not attempt to read from the DB region. Doing so results in anomalous behavior.

BlueField Write

The BlueField can update the value of each DB resource individually using `doca_devemu_pci_db_modify_value`. This produces similar side effects as though the driver updated the value using a write to the DB region.

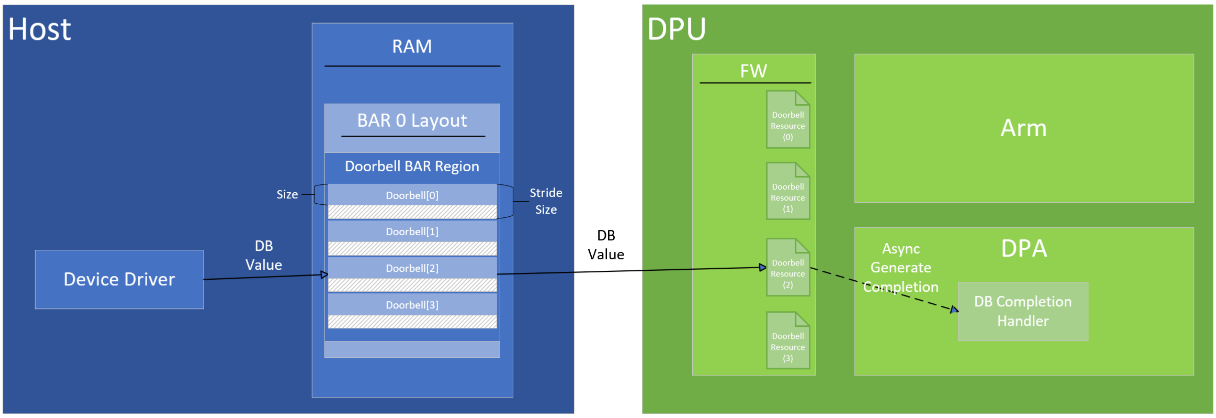
BlueField Read

The BlueField can read the value of each DB resource individually using one of the following methods:

- Read the value from the BlueField Arm using `doca_devemu_pci_db_query_value`
- Read the value from the DPA using `doca_dpa_dev_devemu_pci_db_get_value`

The first option is a time consuming operation and is only recommended for the control path. In the data path, it is recommended to use the second option only.

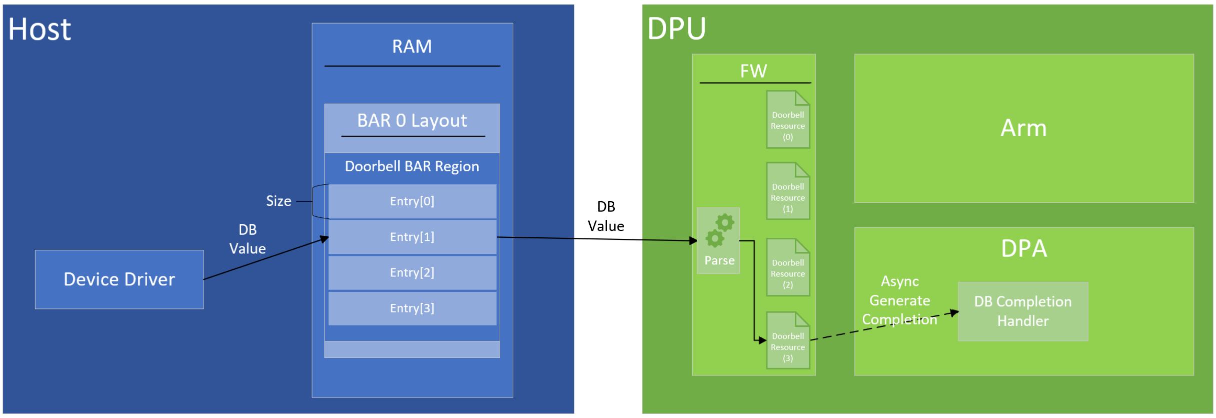
DB by Offset



The API `doca_devemu_pci_type_set_bar_db_region_by_offset_conf` can be used to set up DB by offset region. When the driver writes a DB value using this region, the DPU receives a notification for the relevant DB resource, based on the write offset, such that the DB ID is calculated as follows: `db_id` = `write_offset` / `db_stride_size`.

> The area that is part of the stride but not part of the doorbell, should not be used for any read/write operation, doing so will result in undefined anomalous.

DB by Data



The API `doca_devemu_pci_type_set_bar_db_region_by_data_conf` can be used to set up DB by data region. When the driver writes a DB value using this region, the DPU receives a notification for the relevant DB resource based on the written DB value, such that there is no relation between the write offset and the DB triggered. This DB region assumes that the DB ID is embedded within the DB value written by the driver. When setting up this region, the user must specify where the Most Significant Byte (MSB) and Least Significant Byte (LSB) of the DB ID are embedded in the DB value.

The DPU follows these steps to extract the DB ID from the DB value:

- Driver writes the DB value
- BlueField extracts the bytes between MSB and LSB
- DPU compares MSB index with LSB index
  - If MSB index greater than LSB index: The extracted value is interpreted as Little Endian
  - If LSB index greater than MSB index: The extracted value is interpreted as Big Endian

Example:

DB size is 4 bytes, LSB is 1, and MSB is 3.

- Driver writes value `0xCCDDEEFF` to DB region at index 0 in Little Endian
  - The value is written to memory as follows: `[0]=FF [1]=EE [2]=DD [3]=CC`
- The relevant bytes, are the following: `[1]=EE [2]=DD [3]=CC`
- Since MSB (3) is greater than LSB (1), the value is interpreted as Little Endian: `db_id = 0xCCDDEE`

MSI-X Capability (MSI-X BAR Region)

Message signaled interrupts extended (MSI-X) is commonly used by PCIe devices to send interrupts over the PCIe bus to the host driver. DOCA APIs allow users to expose the MSI-X capability as per the PCIe specification, and to later use it to send interrupts to the host driver.

To configure it, users must provide the following:

- The number of MSI-X vectors which can be done using `doca_devemu_pci_type_set_num_msix`
- Define an MSI-X table
- Define an MSI-X PBA

MSI-X Table BAR Region

As per the PCIe specification, to expose the MSI-X capability, the device must designate a memory region within its BAR as an MSI-X table region. In DOCA, this can be done using `doca_devemu_pci_type_set_bar_msix_table_region_conf`.

MSI-X PBA BAR Region

As per the PCIe specification, to expose the MSI-X capability, the device must designate a memory region within its BAR as an MSI-X pending bit array (PBA) region. In DOCA, this can be done using `doca_devemu_pci_type_set_bar_msix_pba_region_conf`.

Raising MSI-X From DPU

It is possible to raise an MSI-X for each vector individually. This can be done only using the DPA API `doca_dpa_dev_devemu_pci_msix_raise`.

DMA Memory

Some operations require accessing memory which is set up by the host driver. DOCA's device emulation APIs allow users to access such I/O memory using the DOCA mmap (see DOCA Core Memory Subsystem).

After starting the PCIe device, it is possible to acquire an mmap that references the host memory using `doca_devemu_pci_mmap_create`. After creating this mmap, it is possible to configure it by providing:

- Access permissions
- Host memory range
- DOCA devices that can access the memory

The mmap can then be used to create buffers that reference memory on the host. The buffers' addresses would not be locally accessible (i.e., CPU cannot dereference the address), instead the addresses would be I/O addresses as defined by the host driver.

The buffers created from the mmap can then be used with other DOCA libraries and accept a `doca_buf` as an input. This includes:

- DOCA DMA
- DOCA RDMA
- DOCA Ethernet
- DOCA AES-GCM

Function Level Reset

FLR can be handled as described in DOCA DevEmu PCI FLR. Additionally, users must ensure that the following resources are destroyed before stopping the PCIe device:

- Doorbells created using `doca_devemu_pci_db_create_on_dpa`
- MSI-X vectors created using `doca_devemu_pci_msix_create_on_dpa`
- Memory maps created using `doca_devemu_pci_mmap_create`

Limitations

Based on explanation in "Driver Write", user can assume that DOCA DevEmu PCI Generic supports creating emulated PCI devices with the limitation that when a driver writes to a register, the value is immediately available for subsequent reads from the same register. However, this immediate availability does not ensure that any required internal actions triggered by the write have been completed. It is recommended to rely on specific different register values to confirm completion of the write action. For instance, when implementing a write-to-clear operation, e.g. writing 1 to register A to clear register B, it is advisable to poll register B until it indicates the desired state. This approach ensures that the write action has been successfully executed. If a device specification requires certain actions to be completed before exposing written values for subsequent reads, such a device cannot be emulated using the DOCA DevEmu PCI generic framework.

14.4.18.3.7.5 Device Support

DOCA PCI Device emulation requires a device to operate. For information on picking a device, see DOCA DevEmu PCI Device Support.

Some devices can allow different capabilities as follows:

- The maximum number of emulated devices
- The maximum number of different PCIe types
- The maximum number of BARs
- The maximum BAR size
- The maximum number of doorbells
- The maximum number of MSI-X vectors
- For each BAR region type there are capabilities for:
  - Whether the region is supported
  - The maximum number of regions with this type
  - The start address alignment of the region
  - The size alignment of the region
  - The min/max size of the region

As the list of capabilities can be long, it is recommended to use the NVIDIA DOCA Capabilities Print Tool to get an overview of all the available capabilities.

Run the tool as root user as follows:

```
$ sudo /opt/mellanox/doca/tools/doca_caps -p <pci-address> -b devemu_pci
Example output: PCI: 0000:03:00.0
    devemu_pci
        max_hotplug_devices                           15
        max_pci_types                                 2
        type_log_min_bar_size                         12
        type_log_max_bar_size                         30
        type_max_num_msix                             11
        type_max_num_db                               64
        type_log_min_db_size                          1
        type_log_max_db_size                          2
        type_log_min_db_stride_size                   2
        type_log_max_db_stride_size                   12
        type_max_bars                                 2
        bar_max_bar_regions                           12
        type_max_bar_regions                          12
        bar_db_region_identify_by_offset              supported
        bar_db_region_identify_by_data                supported
        bar_db_region_block_size                      4096
        bar_db_region_max_num_region_blocks           16
        type_max_bar_db_regions                       2
        bar_max_bar_db_regions                        2
        bar_db_region_start_addr_alignment            4096
        bar_stateful_region_block_size                64
        bar_stateful_region_max_num_region_blocks     4
        type_max_bar_stateful_regions                 1
        bar_max_bar_stateful_regions                  1
        bar_stateful_region_start_addr_alignment      64
        bar_msix_table_region_block_size              4096
        bar_msix_table_region_max_num_region_blocks   1
        type_max_bar_msix_table_regions               1
        bar_max_bar_msix_table_regions                1
        bar_msix_table_region_start_addr_alignment    4096
        bar_msix_pba_region_block_size                4096
```

```
          bar_msix_pba_region_max_num_region_blocks     1
          type_max_bar_msix_pba_regions                 1
          bar_max_bar_msix_pba_regions                  1
          bar_msix_pba_region_start_addr_alignment      4096
          bar_is_32_bit_supported                       unsupported
          bar_is_1_mb_supported                         unsupported
          bar_is_64_bit_supported                       supported
          pci_type_hotplug                              supported
          pci_type_mgmt                                 supported
```

14.4.18.3.7.6 PCI Type

Configurations

This section describes the configurations of the DOCA DevEmu PCI Type object, that can be provided before start.

To find if a configuration is supported or what its min/max value is, refer to Device Support.

Mandatory Configurations

The following are mandatory configurations and must be provided before starting the PCI type:

- A DOCA device that is an emulation manager or hotplug manager. See Device Support.

Optional Configurations

The following configurations are optional:

- The PCIe device ID
- The PCIe vendor ID
- The PCIe subsystem ID
- The PCIe subsystem vendor ID
- The PCIe revision ID
- The PCIe class code
- The number of MSI-X vectors for MSI-X capability
- One or more memory mapped BARs
- One or more I/O mapped BARs
- One or more DB region
- An MSI-X table and PBA regions
- One or more stateful regions

If these configurations are not set then a default value is used.

14.4.18.3.7.7 PCI Device

Configuration Phase

This section describes additional configuration options, on top of the ones already described in DOCA DevEmu PCI Device Configuration Phase.

Configurations

The context can be configured to match the application's use case.

To find if a configuration is supported or what its min/max value is, refer to Device Support.

Optional Configurations

The following configurations are optional:

- Setting the stateful regions' default values - If not set, then the type default values are used. See stateful region default values for more.

Execution Phase

This section describes additional events, on top of the ones already described in DOCA DevEmu PCI Device Events.

Events

DOCA DevEmu PCI Device exposes asynchronous events to notify about changes that happen suddenly according to the DOCA Core architecture.

Common events are described in DOCA Core Event.

BAR Stateful Region Driver Write

The stateful region driver write event allows you to receive notifications whenever the host driver writes to the stateful BAR region. See section "Driver Write" for more information.

Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	`doca_devemu_pci_dev_event_bar_stateful_region_driver_write_register`	`doca_devemu_pci_cap_type_get_max_bar_stateful_regions`

If there are multiple stateful regions for the same device, then registration is done separately for each region. The details provided on registration (i.e., `bar_id` and start address) must match a region previously configured for PCIe type.

Trigger Condition

The event is triggered anytime the host driver writes to the stateful region. See section "Driver Write" for more information.

Output

Common output as described in DOCA Core Event.

Additionally, the event callback receives an event object of type `struct doca_devemu_pci_dev_event_bar_stateful_region_driver_write` which can be used to retrieve:

- The DOCA DevEmu PCI Device representing the emulated device that triggered the event - `doca_devemu_pci_dev_event_bar_stateful_region_driver_write_get_pci_dev`
- The ID of the BAR containing the stateful region - `doca_devemu_pci_dev_event_bar_stateful_region_driver_write_get_bar_id`
- The start address of the stateful region - `doca_devemu_pci_dev_event_bar_stateful_region_driver_write_get_bar_region_start_addr`

Event Handling

Once the event is triggered, it means that the host driver has written to someplace in the region.

The user must perform either of the following:

- Query the new values of the stateful region - `doca_devemu_pci_dev_query_bar_stateful_region_values`
- Modify the values of the stateful region - `doca_devemu_pci_dev_modify_bar_stateful_region_values`

It is possible also to do both. However, it is important that the memory areas that the host wrote to are either queried or overwritten with a modify operation.

> ⚠ Failure to do so results in a recurring event. For example, if the host wrote to the first half of the region, but BlueField Arm only queries the second half of the region after receiving the event. Then the library retriggers the event, assuming that the user did not handle the event.

14.4.18.3.7.8 PCI Device DB

After the PCIe device has been created, it can be used to create DB objects, each DB object represents a DB resources identified by a DB ID. See Generic Data Path (DB BAR Region).

When creating the DB, the DB ID must be provided, this can hold different meaning for DB by offset and DB by data. The DB object can then be used to get a notification to the DPA once a driver write occurs, and to fetch the latest value using the DPA.

Configuration

The flow for creating and configuring a DB should be as follows:

1. Create the DB object:

   ```
   arm> doca_devemu_pci_db_create_on_dpa
   ```

2. (Optional) Query the DB value:

   ```
   arm> doca_devemu_pci_db_query_value
   ```

3. (Optional) Modify the DB value:

   ```
   arm> doca_devemu_pci_db_modify_value
   ```

4. Get the DB DPA handle for referencing the DB from the DPA:

```
arm> doca_devemu_pci_db_get_dpa_handle
```

5. Bind the DB to the DB completion context using the handle from the previous step:

```
dpa> doca_dpa_dev_devemu_pci_db_completion_bind_db
```

> It is important to perform this step before the next one. Otherwise, the DB completion context will start receiving completions for an unbound DB.

6. Start the DB to start receiving completions on DPA:

```
arm> doca_devemu_pci_db_start
```

> Once DB is started, a completion is immediately generated on the DPA.

Similarly the flow for destroying a DB would look as follows:

1. Stop the DB to stop receiving completions:

```
arm> doca_devemu_pci_db_stop
```

> This step ensures that no additional completions will arrive for this DB

2. Acknowledge all completions related to this DB:

```
dpa> doca_dpa_dev_devemu_pci_db_completion_ack
```

> This step ensures that existing completions have been processed.

3. Unbind the DB from the DB completion context:

```
dpa> doca_dpa_dev_devemu_pci_db_completion_unbind_db
```

> Make sure to not perform this step more than once.

4. Destroy the DB object:

```
arm> doca_devemu_pci_db_destroy
```

Fetching DBs on DPA

To fetch DBs on DPA, a DB completion context can be used. The DB completion context serves the following purposes:

- Notifying a DPA thread that a DB value has been updated (wakes up thread)
- Providing information about which DB has been updated

The following flow shows how to use the same DB completion context to get notified whenever any of the DBs are updated, and to find which DBs were actually updated, and finally to get the DBs' values:

1. Get DB completion element:

   ```
   doca_dpa_dev_devemu_pci_get_db_completion
   ```
2. Get DB from completion:

   ```
   doca_dpa_dev_devemu_pci_db_completion_element_get_db_properties
   ```
3. Store the DB (e.g., in an array).
4. Repeat steps 1-3 until there are no more completions.
5. Acknowledge the number of received completions:

   ```
   doca_dpa_dev_devemu_pci_db_completion_ack
   ```
6. Request notification on DPA for the next completion:

   ```
   doca_dpa_dev_devemu_pci_db_completion_request_notification
   ```
7. Go over the DBs stored in step 3 and for each DB:
   1. Request a notification for the next time the host driver writes to this DB:

      ```
      doca_dpa_dev_devemu_pci_db_request_notification
      ```
   2. Get the most recent value of the DB:

      ```
      doca_dpa_dev_devemu_pci_db_get_value
      ```

Query/Modify DB from Arm

It is possible to query the DB value of a particular DB using `doca_devemu_pci_db_query_value` on the Arm. Similarly, it is possible to modify the DB value using `doca_devemu_pci_db_modify_value`. When modifying the DB value, the side effects of such modification is the same as if the host driver updated the DB value.

> Querying and modifying operations from the Arm are time consuming and should be used in the control path only. Fetching DBs on DPA is the recommended approach for retrieval of DB values in the data path.

14.4.18.3.7.9 PCIe Device MSI-X Vector

After the PCIe device has been created, it can be used to create MSI-X objects. Each MSI-X object represents an MSI-X vector identified by the vector index.

The MSI-X object can be used to send a notification to the host driver from the DPA.

Configuration

The MSI-X object can be created using `doca_devemu_pci_msix_create_on_dpa`. An MSI-X vector index must be provided during creation, this is a value in the range [0, `num_msix`), such that `num_msix` is the value previously set using `doca_devemu_pci_type_set_num_msix`.

Once the MSI-X object is created, `doca_devemu_pci_msix_get_dpa_handle` can be used to get a DPA handle for use within the DPA.

Raising MSI-X

The MSI-X object can be used on the DPA to raise an MSI-X vector using `doca_dpa_dev_devemu_pci_msix_raise`.

14.4.18.3.7.10 DOCA DevEmu Generic Samples

This section describes DOCA DevEmu Generic samples.

The samples illustrate how to use the DOCA DevEmu Generic API to do the following:

- List details about emulated devices with same generic type
- Create and hot-plug/hot-unplug an emulated device with a generic type
- Handle Host driver write using stateful region
- Handle Host driver write using DB region
- Raise MSI-X to the Host driver
- Perform DMA operation to copy memory buffer between the Host driver and the DPU Arm

Structure

All the samples utilize the same generic PCI type. The configurations of the type reside in `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h`

The structure for some samples is as follows:

- `/opt/mellanox/doca/samples/doca_devemu/<sample directory>`
    a. `dpu`
        i. `host`
        ii. `device`
    b. `host`

Samples following this structure will have two binaries: `dpu` (1) and `host` (2), the former should be run on the BlueField and represents the controller of the emulated device, while the latter should be run on the host and represents the host driver.

For simplicity, the host (2) side is based on the VFIO driver, allowing development of a driver in user-space.

Within the `dpu` (a) directory, there is a `host` (a) and `device` (b) directories. `host` in this case refers to the BlueField Arm processor, while `device` refers to the DPA processor. Both directories are compiled into a single binary.

Running the Samples

1. Refer to the following documents:

- NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
- NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample:

```
cd /opt/mellanox/doca/samples/doca_devemu/<sample_name>[/dpu or /host]
meson /tmp/build
ninja -C /tmp/build
```

> The binary `doca_<sample_name>[_dpu or _host]` is created under `/tmp/build/`.

3. Sample (e.g., `doca_devemu_pci_device_db`) usage:
   a. BlueField side (`doca_devemu_pci_device_db_dpu`):

```
Usage: doca_devemu_pci_device_db_dpu [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags from an input json file

Program Flags:
  -p, --pci-addr                    The DOCA device PCI address. Format: XXXX:XX:XX.X or XX:XX.X
  -u, --vuid                        DOCA Devemu emulated device VUID. Sample will use this device
to handle Doorbells from Host
  -r, --region-index                The index of the DB region as defined in
 devemu_pci_type_config.h. Integer
  -i, --db-id                       The DB ID of the DB. Sample will listen on DBs related to this
DB ID. Integer
```

   b. Host side (`doca_devemu_pci_device_db_host`):

```
Usage: doca_devemu_pci_device_db_host [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags from an input json file

Program Flags:
  -p, --pci-addr                    PCI address of the emulated device. Format: XXXX:XX:XX.X
  -g, --vfio-group                  VFIO group ID of the device. Integer
  -r, --region-index                The index of the DB region as defined in
 devemu_pci_type_config.h. Integer
  -d, --db-index                    The index of the Doorbell to write to. The sample will write at
byte offset (db-index * db-stride)
  -w, --db-value                    A 4B value to write to the Doorbell. Will be written in Big
Endian
```

4. For additional information per sample, use the `-h` option:

```
/tmp/build/<sample_name> -h
```

Additional sample setup:

- The BlueField samples require the emulated device to be already hot-plugged:
  - Such samples expect the VUID of the hot-plugged device (`-u, --vuid`)
  - The list sample can be used to find if any hot-plugged devices exist and what their VUID is
  - The hot-plug sample can be used to hot plug a device if no such device already exists

- The host samples require the emulated device to be already hot-plugged and that the device is bound to the VFIO driver:
    - The samples expect 2 parameters `-p` (`--pci-addr`) and `-g` (`--vfio-group`) of the emulated device as seen by the host
    - The PCI Device List sample can be used from the BlueField to find the PCIe address of the emulated device on the host
    - Once the PCIe address is found, the host can use the script `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bind.py` to bind the VFIO driver

      ```
      $ sudo python3 /opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bind.py <pcie-address-of-emulated-dev>
      ```

        - The script is a python3 script which expects the PCIe address of the emulated device as a positional argument (e.g., `0000:3e:00.0`)
        - The script outputs the VFIO group ID
        - The script must be used only once after the device is hot-plugged towards the host for the first time
- The hot-unplug sample requires that the device be unbound from the VFIO driver:
    - Use the script located at `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bind.py` from the host to unbind the VFIO driver as follows:

      ```
      $ sudo python3 /opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bind.py <pcie-address-of-emulated-dev> --unbind
      ```

        - This python3 script expects the PCIe address of the emulated device as a positional argument (e.g., `0000:3e:00.0`) along with the `--unbind` argument

Samples

PCI Device List

This sample illustrates how to list all emulated devices that have the generic type configured in `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h` .

The sample logic includes:

1. Initializing the generic PCIe type based on `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h` .
2. Creating a list of all emulated devices belonging to this type.
3. Iterating over the emulated devices.
4. Dumping their VUID.
5. Dumping their PCIe address as seen by the host.
6. Releasing the resources.

References:

- `/opt/mellanox/doca/samples/doca_devemu/`
    - `devemu_pci_device_list/`
        - `devemu_pci_device_list_sample.c`
        - `devemu_pci_device_list_main.c`
        - `meson.build`

    - `devemu_pci_common.h` ; `devemu_pci_common.c`
    - `devemu_pci_type_config.h`

PCI Device Hot-Plug

This sample illustrates how to create and hot-plug/hot-unplug an emulated device that has the generic type configured in `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h` .

The sample logic includes:

1. Initializing the generic PCIe type based on `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h` .
2. Acquiring the emulated device representor:
    - If the user did not provide VUID as input, then creating and using a new emulated device.
    - If the user provided VUID as an input, then searching for an existing emulated device with a matching VUID and using it.
3. Creating a PCIe device context to manage the emulated device and connecting it to a progress engine (PE).
4. Registering to the PCIe device's hot-plug state change event.
5. Initializing hot-plug/hot-unplug of the device:
    a. If the user did not provide VUID as input, then initializing hot-plug flow of the device.
    b. If the user provided VUID as input, then initializing hot-unplug flow of the device.
6. Using the PE to poll for hot-plug state change event.
7. Waiting until hot-plug state transitions to expected state (power on or power off).
8. Cleaning up resources.
    - If hot-unplug was requested, then the emulated device is destroyed as well.
    - Otherwise, the emulated device persists.

References:

- `/opt/mellanox/doca/samples/doca_devemu/`
    - `devemu_pci_device_hotplug/`
        - `devemu_pci_device_hotplug_sample.c`
        - `devemu_pci_device_hotplug_main.c`
        - `meson.build`
    - `devemu_pci_common.h` ; `devemu_pci_common.c`
    - `devemu_pci_type_config.h`

PCI Device Stateful Region

This sample illustrates how the host driver can write to a stateful region, and how the BlueField Arm can handle the write operation.

This sample consists of a host sample and BlueField sample. It is necessary to follow the additional sample setup detailed previously.

The BlueField sample logic includes:

1. Initializing the generic PCIe type based on `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h`.
2. Acquiring the emulated device representor that matches the provided VUID.
3. Creating a PCIe device context to manage the emulated device, and connecting it to a progress engine (PE).
4. For each stateful region configured in `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h`, registering to the PCIe device's stateful region write event.
5. Using the PE to poll for driver write to any of the stateful regions.
   - Every time the host driver writes to the stateful region, the handler is invoked and performs the following:
     1. Queries the values of the stateful region that the host wrote to.
     2. Logs the values of the stateful region.
   - The sample polls indefinitely until the user presses [Ctrl+c] to close the sample.
6. Cleaning up resources.

The host sample logic includes:

1. Initializing the VFIO device with a matching PCIe address and VFIO group.
2. Mapping the stateful memory region from the BAR to the process address space.
3. Writing the values provided as input to the beginning of the stateful region.

References:

- `/opt/mellanox/doca/samples/doca_devemu/`
  - `devemu_pci_device_stateful_region/dpu/`
    - `devemu_pci_device_stateful_region_dpu_sample.c`
    - `devemu_pci_device_stateful_region_dpu_main.c`
    - `meson.build`
  - `devemu_pci_device_stateful_region/host/`
    - `devemu_pci_device_stateful_region_host_sample.c`
    - `devemu_pci_device_stateful_region_host_main.c`
    - `meson.build`
  - `devemu_pci_common.h`; `devemu_pci_common.c`
  - `devemu_pci_host_common.h`; `devemu_pci_host_common.c`
  - `devemu_pci_type_config.h`

PCI Device DB

This sample illustrates how the host driver can ring the doorbell and how the BlueField can retrieve the doorbell value. The sample also demonstrates how to handle FLR.

This sample consists of a host sample and BlueField sample. It is necessary to follow the additional sample setup detailed previously.

The BlueField sample logic includes:

- Host (BlueField Arm) logic:
  a. Initializing the generic PCIe type based on `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h`.
  b. Initializing DPA resources:

        i. Creating DPA instance, and associating it with the DPA application.
        ii. Creating DPA thread and associating it with the DPA DB handler.
        iii. Creating DB completion context and associating it with the DPA thread.
    c. Acquiring the emulated device representor that matches the provided VUID.
    d. Creating a PCIe device context to manage the emulated device, and connecting it to progress engine (PE).
    e. Registering to the context state changes event.
    f. Registering to the PCIe device FLR event.
    g. Using the PE to poll for any of the following:
        i. Every time the PCIe device context state transitions to running, the handler performs the following:
            1. Creates a DB object.
            2. Makes RPC to DPA, to initialize the DB object.
        ii. Every time the PCIe device context state transitions to stopping, the handler performs the following:
            1. Makes RPC to DPA, to un-initialize the DB object.
            2. Destroys the DB object.
        iii. Every time the host driver initializes or destroys the VFIO device, an FLR event is triggered. The FLR handler performs the following:
            1. Destroys DB object.
            2. Stops the PCIe device context.
            3. Starts the PCIe device context again.
        iv. The sample polls indefinitely until the user presses [Ctrl+c] to close the sample.

        > ⚠ During this time, the DPA may start receiving DBs from the host.

    h. Cleaning up resources.

- Device (BlueField DPA) logic:
    a. Initializing application RPC:
        i. Setting the global context to point to the DB completion context DPA handle.
        ii. Binding DB to the doorbell completion context.
    b. Un-initializing application RPC:
        i. Unbinding DB from the doorbell completion context.
    c. DB handler:
        i. Getting DB completion element from completion context.
        ii. Getting DB handle from the DB completion element.
        iii. Acknowledging the DB completion element.
        iv. Requesting notification from DB completion context.
        v. Requesting notification from DB.
        vi. Getting DB value from DB.

The host sample logic includes:

1. Initializing the VFIO device with its matching PCIe address and VFIO group.
2. Mapping the DB memory region from the BAR to the process address space.
3. Writing the value provided as input to the DB region at the given offset.

References:

- `/opt/mellanox/doca/samples/doca_devemu/`
  - `devemu_pci_device_db/dpu/`
    - `host/`
      - `devemu_pci_device_db_dpu_sample.c`
    - `device/`
      - `devemu_pci_device_db_dpu_kernels_dev.c`
    - `devemu_pci_device_db_dpu_main.c`
    - `meson.build`
  - `devemu_pci_device_db/host/`
    - `devemu_pci_device_db_host_sample.c`
    - `devemu_pci_device_db_host_main.c`
    - `meson.build`
  - `devemu_pci_common.h`; `devemu_pci_common.c`
  - `devemu_pci_host_common.h`; `devemu_pci_host_common.c`
  - `devemu_pci_type_config.h`

PCI Device MSI-X

This sample illustrates how BlueField can raise an MSI-X vector, sending a signal towards the host, and shows how the host can retrieve this signal.

This sample consists of a host sample and a BlueField sample. It is necessary to follow the additional sample setup detailed previously.

The BlueField sample logic includes:

- Host (BlueField Arm) logic:
  a. Initializing the generic PCIe type based on `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h`.
  b. Initializing DPA resources:
     i. Creating a DPA instance and associating it with the DPA application.
     ii. Creating a DPA thread and associating it with the DPA DB handler.
  c. Acquiring the emulated device representor that matches the provided VUID.
  d. Creating a PCIe device context to manage the emulated device and connecting it to a progress engine (PE).
  e. Creating an MSI-X vector and acquiring its DPA handle.
  f. Sending an RPC to the DPA to raise the MSI-X vector.
  g. Cleaning up resources.
- Device (BlueField DPA) logic:
  a. Raising the MSI-X RPC by using the MSI-X vector handle.

The host sample logic includes:

1. Initializing the VFIO device with the matching PCIe address and VFIO group.
2. Mapping each MSI-X vector to a different FD.
3. Reading events from the FDs in a loop.
   a. Once the DPU raises MSI-X, the FD matching the MSI-X vector returns an event which is then printed to the screen.

b. The sample polls the FDs indefinitely until the user presses [Ctrl+c] to close the sample.

References:

- `/opt/mellanox/doca/samples/doca_devemu/`
    - `devemu_pci_device_msix/dpu/`
        - `host/`
            - `devemu_pci_device_msix_dpu_sample.c`
        - `device/`
            - `devemu_pci_device_msix_dpu_kernels_dev.c`
        - `devemu_pci_device_msix_dpu_main.c`
        - `meson.build`
    - `devemu_pci_device_msix/host/`
        - `devemu_pci_device_msix_host_sample.c`
        - `devemu_pci_device_msix_host_main.c`
        - `meson.build`
    - `devemu_pci_common.h` ; `devemu_pci_common.c`
    - `devemu_pci_host_common.h` ; `devemu_pci_host_common.c`
    - `devemu_pci_type_config.h`

PCI Device DMA

This sample illustrates how the host driver can set up memory for DMA, then the DPU can use that memory to copy a string from the BlueField to the host and from the host to the BlueField.

This sample consists of a host sample and a BlueField sample. It is necessary to follow the additional sample setup detailed previously.

The BlueField sample logic includes:

1. Initializing the generic PCIe type based on `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h` .
2. Acquiring the emulated device representor that matches the provided VUID.
3. Creating a PCIe device context to manage the emulated device and connecting it to a progress engine (PE).
4. Creating a DMA context to use for copying memory across the host and BlueField.
5. Setting up an mmap representing the host driver memory buffer.
6. Setting up an mmap representing a local memory buffer.
7. Use the DMA context to copy memory from host to BlueField.
8. Use the DMA context to copy memory from BlueField to host.
9. Cleaning up resources.

The host sample logic includes:

1. Initializing the VFIO device with the matching PCIe address and VFIO group.
2. Allocating memory buffer.
3. Mapping the memory buffer to I/O memory. The BlueField can now access the memory using the I/O address through DMA.
4. Copying the string provided by user to the memory buffer.

5. Waiting for the BlueField to write to the memory buffer.
6. Un-mapping the memory buffer.
7. Cleaning up resources.

References:

- `/opt/mellanox/doca/samples/doca_devemu/`
    - `devemu_pci_device_dma/dpu/`
        - `devemu_pci_device_dma_dpu_sample.c`
        - `devemu_pci_device_dma_dpu_main.c`
        - `meson.build`
    - `devemu_pci_device_dma/host/`
        - `devemu_pci_device_dma_host_sample.c`
        - `devemu_pci_device_dma_host_main.c`
        - `meson.build`

14.4.18.4 DOCA DevEmu Virtio

⚠ This library is supported at alpha level; backward compatibility is not guaranteed.

14.4.18.4.1 Introduction

DOCA DevEmu Virtio, which is part of the DOCA Device Emulation subsystem, introduces low-level software APIs that provide building blocks for developing and manipulating virtio devices using the device emulation capability of NVIDIA® BlueField®. This subsystem incorporates a core library that handles a common logic for various types of virtio devices, such as virtio-FS. One of its key responsibilities is managing the standard "device reset" procedure outlined in the virtio specification. This core library serves as a foundation for implementing shared functionalities across different virtio device types, ensuring consistency and efficiency in device operations and behaviors.

DOCA provides support for emulating virtio devices over the PCIe bus. The PCIe transport is commonly used for virtio devices. Configuration, discovery, and features related to PCIe (such as MSI-X and PCIe device hot plug/unplug) are managed through the DOCA DevEmu PCI APIs. This modular design enables each layer within the DOCA Device Emulation subsystem to manage its own business logic and facilitates seamless integration with the other layers, ensuring independent functionality and operation throughout the system.

This subsystem also includes device-specific libraries for various virtio device types (e.g., a library for a virtio-FS device).

From the host's perspective, there is no difference between para-virtual, DOCA-emulated, and actual hardware devices. The host uses the same virtio device drivers to operate the device under all circumstances.

14.4.18.4.2 Prerequisites

Virtio device emulation is part of the DOCA Device Emulation subsystem. It is, therefore, recommended to read the following guides beforehand:

- DOCA Device Emulation
- DOCA DevEmu PCI

14.4.18.4.3 Environment

DOCA DevEmu Virtio is supported on the BlueField target only.

The BlueField must meet the following requirements

- DOCA version 2.7.0 or greater
- BlueField-3 firmware 32.41.1000 or higher

> Please refer to the DOCA Backward Compatibility Policy.

Library must be run with root privileges.

14.4.18.4.4 Architecture

The DOCA DevEmu Virtio core library provides the following software abstractions:

- Virtio type - extends the PCIe type, represents common/default virtio configurations of emulated virtio devices
- Virtio device - extends the PCIe device, represents an instance of an emulated virtio device
- Virtio IO context - represents a progress context which is responsible for processing virtio descriptors and their associated virtio queues

DOCA DevEmu Virtio library does not provide APIs to configure the entire BAR layout of the virtio device as this configuration is done internally. However, the library offers APIs to configure some of the registers within the common configuration structure (see Virtio Device).

14.4.18.4.4.1 Virtio Common Configuration

According to the virtio specification, the common PCIe configuration structure layout is as follows:

virtio_pci_common_cfg

```
struct virtio_pci_common_cfg {
    /* About the whole device. */
    le32 device_feature_select; /* read-write */
    le32 device_feature; /* read-only for driver */
    le32 driver_feature_select; /* read-write */
    le32 driver_feature; /* read-write */
    le16 config_msix_vector; /* read-write */
    le16 num_queues; /* read-only for driver */
    u8 device_status; /* read-write */
    u8 config_generation; /* read-only for driver */

    /* About a specific virtqueue. */
    le16 queue_select; /* read-write */
    le16 queue_size; /* read-write */
    le16 queue_msix_vector; /* read-write */
    le16 queue_enable; /* read-write */
    le16 queue_notify_off; /* read-only for driver */
    le64 queue_desc; /* read-write */
    le64 queue_driver; /* read-write */
    le64 queue_device; /* read-write */
    le16 queue_notify_data; /* read-only for driver */
    le16 queue_reset; /* read-write */
};
```

The DOCA DevEmu Virtio core library provides the ability to configure some of the listed registers using the appropriate setters.

14.4.18.4.4.2 Virtio Type

The virtio type extends the PCIe type and describes the common/default configuration of emulated virtio devices, including the common virtio configuration space registers (such as `num_queues`, `queue_size`, and others).

Virtio type is currently read-only (i.e., only getter APIs are available to retrieve information). The following methods can be used for this purpose:

- `doca_devemu_virtio_type_get_num_queues` - for getting the default initial value of the `num_queues` register for the associated virtio devices
- `doca_devemu_virtio_type_get_queue_size` - for getting the default initial value of the `queue_size` register for the associated virtio devices
- `doca_devemu_virtio_type_get_device_features_63_0` - for getting the default initial values of the `device_feature` bits (0-63) for the associated virtio devices
- `doca_devemu_virtio_type_get_config_generation` - for getting the default initial value of the `config_generation` register for the associated virtio devices

The default virtio type is extended by a virtio device's specific type (e.g., virtio-FS type) and cannot be created on demand.

14.4.18.4.4.3 Virtio Device

The virtio device extends the PCIe device. Before using the DOCA DevEmu Virtio device, it is recommended to read the guidelines of DOCA DevEmu PCI device and DOCA Core context configuration phase.

The virtio device is extended by a virtio-specific device (e.g., virtio FS device) and cannot be created on demand.

Virtio Device Configurations

The virtio device context can be configured to match the application use case and optimize the utilization of system resources.

Mandatory Configurations

The mandatory configurations are as follows:

- `doca_devemu_virtio_dev_set_num_required_running_virtio_io_ctxs` - to set the number of required running virtio IO contexts to be bound to the virtio device context. The virtio device context does not move to `running` state (according to the DOCA Core context state machine) before having this amount of running virtio IO contexts bound to it.
- `doca_devemu_virtio_dev_event_reset_register` - to register to the virtio device reset event. This configuration is mandatory

Optional Configurations

The optional configurations are as follows:

- `doca_devemu_virtio_dev_set_device_features_63_0` - to set the values of the `device_feature` bits (0-63). If not set, the default value is taken from the virtio type configuration.
- `doca_devemu_virtio_dev_set_num_queues` - to set the value of the `num_queues` register. If not set, the default value is taken from the virtio type configuration.
- `doca_devemu_virtio_dev_set_queue_size` - to set the value of the `queue_size` register for all virtio queues. If not set, the default value is taken from the virtio type configuration.

Events

DOCA DevEmu Virtio device exposes asynchronous events to notify about sudden changes, according to DOCA Core architecture.

> ⓘ Common events are described in DOCA DevEmu PCI Device events and in DOCA Core Event.

Reset Event

The reset event allows users to receive notifications whenever the device reset flow is initialized by the device driver. Upon receiving this event, it is guaranteed that no further requests are routed to the user via any associated virtio IO context until the reset flow is completed.

To complete the reset flow the user must:

1. Flush all outstanding requests back to the virtio IO context associated with the request.
2. Perform one of the following:
   - Call `doca_devemu_virtio_dev_reset_complete`.
   - Follow FLR flow:
     1. `doca_ctx_stop` - stop the virtio device with its associated virtio IO contexts and wait until the device and its associated virtio IO contexts transition to `idle` state
     2. `doca_ctx_start` - start the virtio device with its associated virtio IO contexts and wait until the device and its associated virtio IO contexts transition to `running` state

Now, the device and its associated virtio IO contexts should be fully operational again, the device is allowed to route new requests via any associated virtio IO context.

14.4.18.4.4.4 Virtio IO

The virtio IO context extends the DOCA Core context. Before using the DOCA DevEmu Virtio IO, it is recommended to read the guidelines of DOCA Core context configuration phase.

This context is associated with a single DOCA virtio device and is bound to the virtio device context upon start. The virtio IO context is a thread-unsafe object and is progressed by a single DOCA Core progress engine. Usually, users configure a single virtio IO context per BlueField core used by the application service.

The virtio IO context is responsible to route new incoming virtio requests towards the application and to complete handled requests back to the device driver. It can only route requests while in `running` state and when its associated virtio device is also in `running` state.

14.4.18.4.5 DOCA DevEmu Virtio-FS

> ⚠ This library is supported at alpha level; backward compatibility is not guaranteed.

14.4.18.4.5.1 Introduction

The DOCA DevEmu Virtio-FS library is part of the DOCA DevEmu Virtio subsystem. It provides low-level software APIs that provide building blocks for developing and manipulating virtio filesystem devices using the device emulation capability of NVIDIA® BlueField® DPUs.

DOCA supports emulating virtio-FS devices over the PCIe bus. The PCIe transport is the common transport used for virtio devices. Configuration, discovery, and features related to PCIe (e.g., MSI-X and PCIe device hot plug/unplug) are managed through the DOCA DevEmu PCI APIs. Configuring common virtio registers and handling generic virtio logic (e.g., virtio device reset flow) is handled by the DOCA Virtio common library. This modular design enables each layer within the DOCA Device Emulation subsystem to manage its own business logic. It facilitates seamless integration with the other layers, ensuring independent functionality and operation throughout the system.

The DOCA Devemu Virtio-FS library efficiently handles virtio descriptors, carrying FUSE requests, sent by the device driver, and translating them into abstract virtio-FS requests which are then routed to the user. This translation process ensures that the underlying device-specific acceleration details are abstracted away, allowing applications to interact with abstracted virtio-FS requests.

Users of this library are responsible for developing a virtio-FS controller, which manages the underlying DOCA Devemu Virtio-FS device alongside an external backend file system which is outside DOCA's scope. The controller application is designed to receive DOCA Virtio-FS requests and process them according to virtio-FS and FUSE specifications, translating FUSE-based commands into the appropriate backend filesystem protocol.

14.4.18.4.5.2 Prerequisites

Virtio-FS device emulation is part of DOCA DevEmu Virtio subsystem. It is, therefore, recommended to read the following guides before proceeding:

- DOCA Device Emulation
- DOCA DevEmu PCI
- DOCA DevEmu Virtio

14.4.18.4.5.3 Environment

DOCA DevEmu Virtio-FS is supported on the BlueField target only. The BlueField must meet the following requirements:

- DOCA version 2.7.0 or greater
- BlueField-3 firmware 32.41.1000 or higher

> ⓘ Please refer to the DOCA Backward Compatibility Policy.

> ⚠ Library must be run with root privileges.

Perform the following:

1. Configure BlueField to work in DPU mode as described in NVIDIA BlueField Modes of Operation.
2. Enable emulation by running the following on the host or DPU:

   ```
   host/dpu> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s VIRTIO_FS_EMULATION_ENABLE=1
   ```

3. Configure the number of static virtio-FS physical functions and the number of MSIX for each physical function to expose. This can be done by running the following command on the DPU:

   ```
   host/dpu> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s VIRTIO_FS_EMULATION_NUM_PF=2
   VIRTIO_FS_EMULATION_NUM_MSIX=18
   ```

4. Perform a BlueField system reboot for the `mlxconfig` settings to take effect.

> ⚠ DOCA does not support hot plugging virtio-FS PF devices into the host PCIe subsystem or SR-IOV for virtio-FS devices.

14.4.18.4.5.4 Architecture

The DOCA DevEmu Virtio-FS library provides the following main software abstractions:

- The virtio-FS type - extends the virtio type; represents common/default virtio-FS configurations of emulated virtio-FS devices
- The virtio-FS device - extends the virtio device; represents an instance of an emulated virtio-FS device
- The virtio-FS IO context - extends the virtio IO context; represents a progress context responsible for processing virtio descriptors, carrying FUSE requests, and their associated virtio queues (e.g., hiprio, request, admin, and notification queues).
- The virtio-FS request

Virtio-FS Feature Bits

According to the virtio specification, a virtio-FS device may report support for `VIRTIO_FS_F_NOTIFICATION` which indicates the ability to handle FUSE notify messages sent via the notification queue.

> ⚠ Currently, DOCA does not support reporting the `VIRTIO_FS_F_NOTIFICATION` feature to the driver.

Virtio-FS Configuration Layout

According to the virtio specification, the virtio-FS configuration structure layout is as follows:

virtio_fs_config

```
struct virtio_fs_config {
    char tag[36];
    le32 num_request_queues;
    le32 notify_buf_size;
};
```

The `tag` and `num_request_queues` fields are always available. The `notify_buf_size` field is only available when `VIRTIO_FS_F_NOTIFICATION` is set.

Currently, there is no support for reporting the `VIRTIO_FS_F_NOTIFICATION` feature to the driver. Therefore, `notify_buf_size` field is not available.

Virtio-FS Type

The virtio-FS type extends the virtio type and describes the common/default configuration of emulated virtio-FS devices, including some of the virtio-FS configuration space registers (e.g., `num_request_queues`).

Currently, the virtio-FS type is read-only (i.e., only getter APIs are available to retrieve information). The following method can be used for this purpose:

- `doca_devemu_vfs_type_get_num_request_queues` - to get the default initial value of the `num_request_queues` register for the associated virtio-FS devices

DOCA supports the default virtio-FS type. To retrieve the default virtio-FS type, users use the following method:

- `doca_devemu_vfs_is_default_vfs_type_supported` - check if the default DOCA Virtio-FS type is supported by the device. If supported:
    - `doca_dev_open` - open supported DOCA device
    - `doca_devemu_vfs_find_default_vfs_type_by_dev` - get the default DOCA Virtio-FS type associated with the device

Virtio-FS Device

The virtio-FS device extends the virtio device. Before using the DOCA DevEmu Virtio-FS device, it is recommended to read the guidelines of DOCA DevEmu Virtio device, DOCA DevEmu PCI device, and DOCA Core context configuration phase.

This section describes how to create, configure, and operate the virtio-FS device.

Virtio-FS Device Configurations

The virtio-FS emulated device might be in several different visibility levels from the host point of view:

- Visible/non-visible to the PCIe subsystem - If the device is visible to the PCIe subsystem, the user is not able to configure PCIe-related parameters (e.g., number of MSI-X vector, `subsystem_id`).

- Visible/non-visible to the virtio subsystem - If the device is visible to the virtio subsystem, the user is not be able to configure virtio-related parameters (e.g., number of queues, `queue_size`).

The flow for creating and configuring a virtio-FS device is as follows:

1. `doca_devemu_vfs_dev_create` - Create a new DOCA DevEmu Virtio-FS device instance.
2. `doca_devemu_vfs_dev_set_tag` - Set a unique tag for the device according to the virtio specification.
3. `doca_devemu_vfs_dev_set_num_request_queues` - Set the number of request queues for the device.
4. `doca_devemu_vfs_dev_set_vfs_req_user_data_size` - Set the user data size of the virtio-FS request. If set, a buffer with this size is allocated for each DOCA DevEmu Virtio-FS on behalf of the user.
5. Configure virtio-related parameters as described in DOCA Virtio configurations.

   > ⚠ `doca_devemu_virtio_dev_set_num_queues` should be equal to the number of request queues +1 (for the `hiprio` queue) since DOCA does not currently support the virtio-FS notification queue.

6. Configure PCIe-related parameters as described in DOCA DevEmu PCI configurations.
7. `doca_ctx_start` - Start the virtio-FS device context to finalize the configuration phase.
   - The virtio-FS device object follows the DOCA context state machine as described in DOCA Core context state machine
   - The virtio-FS device context moves to `running` state after the initial number of virtio IO contexts is bound to it and turns to `running` state, as described at DOCA DevEmu Virtio configurations

At this point, the DOCA Devemu Virtio-FS context is fully operational.

Mandatory Configurations

The following are mandatory configurations:

- `doca_devemu_vfs_dev_set_tag` - set a unique tag for the device

Optional Configurations

The optional configurations are as follows:

- `doca_devemu_vfs_dev_set_num_request_queues` - set the number of request queues for the device. If not set, the default value is taken from the virtio-FS type configuration.
- `doca_devemu_vfs_dev_set_vfs_req_user_data_size` - set the user data size of the virtio-FS request. If not set, user data size defaults to 0.

Virtio-FS Device Events

DOCA DevEmu Virtio-FS device exposes asynchronous events to notify about changes that happen out of the blue, according to the DOCA Core architecture.

Common events are described in DOCA DevEmu Virtio device events, DOCA DevEmu PCI device events and in DOCA Core event.

Virtio-FS IO

The virtio-FS IO context extends the Virtio IO Context. To start using the DOCA DevEmu Virtio-FS IO it is recommended to read the guidelines of DOCA DevEmu Virtio IO and DOCA Core context configuration phase.

This section describes how to create, configure and operate the virtio-FS IO context.

Virtio-FS IO Configurations

The flow for creating and configuring a virtio-FS IO context should be as follows:

1. `doca_devemu_vfs_io_create` - Create a new DOCA DevEmu Virtio-FS IO instance.
2. `doca_devemu_vfs_io_event_vfs_req_notice_register` - Register event handler for incoming virtio-FS requests.
3. `doca_ctx_start` - Start the virtio-FS IO context to finalize the configuration phase. The virtio-FS IO object follows the DOCA Core context state machine. The virtio-FS device context moves to `running` state after the initial number of virtio-FS IO contexts is bound to it and moves to `running` state (as described at DOCA DevEmu Virtio configurations).

Mandatory Configurations

The following are mandatory configurations:

- `doca_devemu_vfs_io_event_vfs_req_notice_register` - Register event handler for incoming virtio-FS requests is mandatory

Virtio-FS Request

The virtio-FS request object serves as an abstraction for handling requests arriving on virtio-FS queues, including high-priority, request, or notification queues. These requests are initially generated by the device driver through created virtio queues and then routed to the user via a registered event handler, which is set up using `doca_devemu_vfs_io_event_vfs_req_notice_register`, on the associated virtio IO context. This event handler, issued by the DOCA Virtio FS library, ensures that users can receive and process virtio-FS requests effectively within their application. Once the event handler is called, the ownership of the virtio-FS request and the associated request user data move to the user. The request ownership moves back to the associated virtio IO context once it is completed by the user by calling `doca_devemu_vfs_req_complete`.

The following APIs operate a virtio-FS request:

1. `doca_devemu_vfs_req_get_datain` - Get a DOCA buffer representing the data-in of the virtio-FS request. This DOCA buffer represents the host memory for the device-readable part of the request according to the virtio specification.
2. `doca_devemu_vfs_req_get_dataout` - Get a DOCA buffer representing the data-out of the virtio-FS request. This DOCA buffer represents the host memory for the device-writable part of the request according to the virtio specification.

3. `doca_devemu_vfs_req_complete` - Complete the virtio-FS request. The associated virtio-FS IO context completes the request toward the device driver according to the virtio-FS specification.

14.4.18.4.5.5 Discovery

Emulated virtio-FS PCIe functions are represented by a `doca_devinfo_rep`. To find the suitable `doca_devinfo_rep` that is used as the input parameter for `doca_devemu_vfs_dev_create`, users should first discover the existing device representors using the below:

1. `doca_devinfo_create_list` - Get a list of all DOCA devices.
2. `doca_devemu_vfs_is_default_vfs_type_supported` - Check whether the device can manage device associated to virtio-FS type.
3. If supported:
   a. `doca_dev_open` - Get an instance of the DOCA device that can be used as virtio-FS emulation manager.
   b. `doca_devemu_vfs_find_default_vfs_type_by_dev` - Get the default virtio-FS device type.
   c. `doca_devemu_vfs_type_as_pci_type` - Cast virtio-FS type to PCIe type.
   d. `doca_devemu_pci_type_rep_list_create` - Create a list of all available representor devices for the virtio-FS type.
4. At this point, the user can choose the preferred representor device, open it using `doca_dev_rep_open`, and proceed with the flow described in section "Virtio-FS Device Configurations".

14.4.18.4.5.6 Initialization

This section describes the initialization flow of a DOCA DevEmu Virtio-FS device and one or more DOCA DevEmu Virtio-FS IO contexts (4 in this example). In this procedure, the user sets up and prepares the environment before starting to receive control path events (from the virtio-FS device context) and IO requests (from the virtio-FS IO contexts). During initialization, the user should configure various essential components to ensure correct behavior.

The user should perform the following:

1. Choose 4 Arm cores to run the application threads on.
2. Create 4 DOCA Core progress engine (PE) objects (`pe1` , `pe2` , `pe3` , `pe4`).
3. Find the suitable representor device according to the Discovery flow or any other method.
4. Create, configure, and start a new virtio-FS device according to the virtio-FS device configuration flow. Assume `pe1` is associated with the virtio-FS device and `doca_devemu_virtio_dev_set_num_required_running_virtio_io_ctxs` is set to 4.
5. Create, configure, and start 4 new virtio-FS IO contexts according to the virtio-FS IO configuration flow. Assume `pe1` , `pe2` , `pe3` , and `pe4` are associated with each of the 4 virtio-FS IO contexts respectively.
6. At this point, the 4 virtio-FS IO contexts transition to `running` state, followed by the virtio-FS device context transitioning to `running` state.

⚠ During the initialization flow, it is guaranteed that no virtio/PCIe control path or IO path events are generated until the virtio-FS device has transitioned to `running` state.

14.4.18.4.5.7 Teardown

This section describes the teardown flow of DOCA DevEmu Virtio-FS device and one or more DOCA DevEmu Virtio-FS IO contexts (4 in this example). In this procedure, the user cleans all the resources allocated in the initialization flow and all the outstanding events and requests.

The user should perform the following:

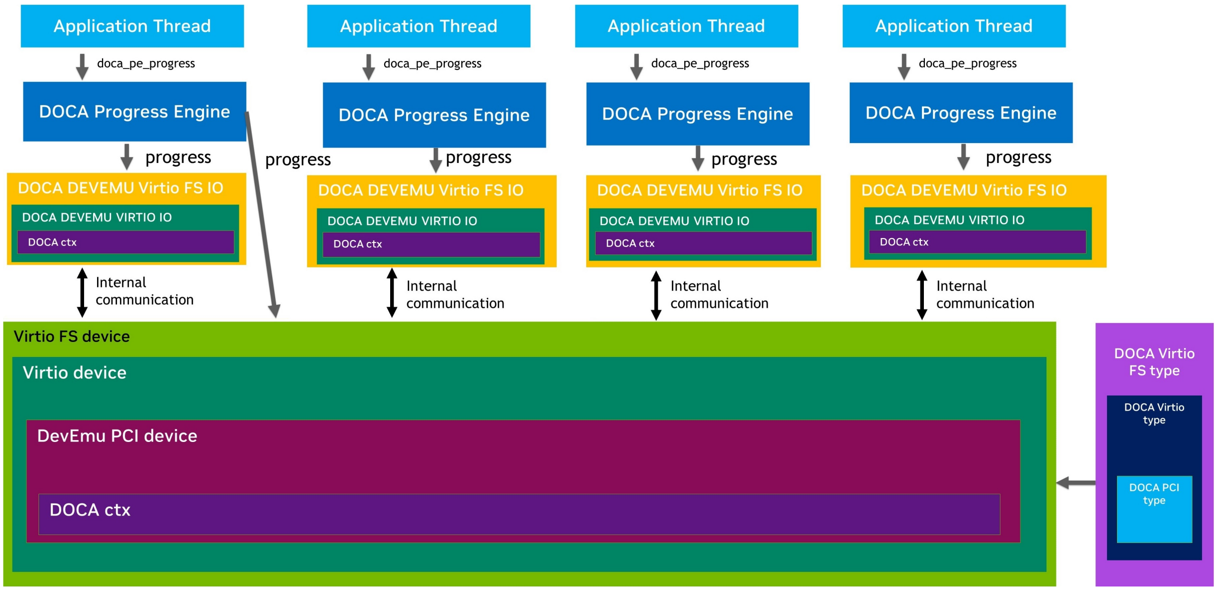
1. Start the teardown flow by calling `doca_ctx_stop`. This causes the DOCA Virtio-FS device context to transition to `stopping` state. It is guaranteed that no virtio/PCIe control path events is generated during this state.
2. Call `doca_ctx_stop` for any DOCA Virtio-FS IO context. This causes the DOCA Virtio-FS IO context to transition to `stopping` state. It is guaranteed that no IO path events are generated during this state.
3. Flush all outstanding virtio-FS requests to the associated virtio-FS IO contexts by calling `doca_devemu_vfs_req_complete`. Upon completing all the requests associated with a virtio-FS IO context, the DOCA Virtio-FS IO context transitions to `idle` state.
4. At this point, it is safe to destroy the virtio-FS IO context by calling `doca_devemu_vfs_io_destroy`. Destroying a virtio-FS IO context not in `idle` state will fail.
5. Once all 4 virtio-FS IO contexts associated with the virtio-FS device transition to `idle` state, the DOCA Virtio-FS device context transitions to `idle` state as well.
6. At this point, it is safe to destroy the virtio-FS device context by calling `doca_devemu_vfs_dev_destroy`. Destroying a virtio-FS device context not in `idle` state will fail.

14.4.18.4.5.8 Execution Phase

This section describes execution on BlueField Arm cores using several DOCA Core PE objects (one per core):

1. Choose 4 Arm cores to run the application threads on.
2. Create 4 DOCA Core PE objects. The application threads should periodically call `doca_pe_progress` to advance all DOCA contexts associated with the PE.
3. Create, configure, and start the DOCA Virtio-FS device.
4. Create, configure, and start 4 DOCA Virtio-FS IO contexts.

The progress of DOCA Virtio-FS objects is illustrated by the following diagram:



Control Path

The DOCA Virtio-FS device context extends the DOCA Virtio device context (which extends the DOCA PCIe device context). This means that the DOCA Virtio-FS device control path is comprised by all the object it extends (i.e., DOCA Context, DOCA DevEmu PCI device, and DOCA DevEmu Virtio device).

The following events can be triggered by a virtio-FS device context:

- DOCA context state change events as described in DOCA Core context state machine and in DOCA DevEmu PCI state machine
- DOCA DevEmu PCI FLR flow
- DOCA DevEmu Virtio reset flow

The DOCA Virtio-FS IO context extends the DOCA Virtio IO context (which extends the DOCA core context). This means that the DOCA Virtio-FS IO context control path is comprised by all the object it extends (i.e., DOCA Context and DOCA DevEmu Virtio IO).

The following events can be triggered by a Virtio-FS IO context:

- DOCA context state change events as described in DOCA Core context state machine

In addition to the control path events, the DOCA DevEmu Virtio-FS IO context also produces IO path events as described in IO path.

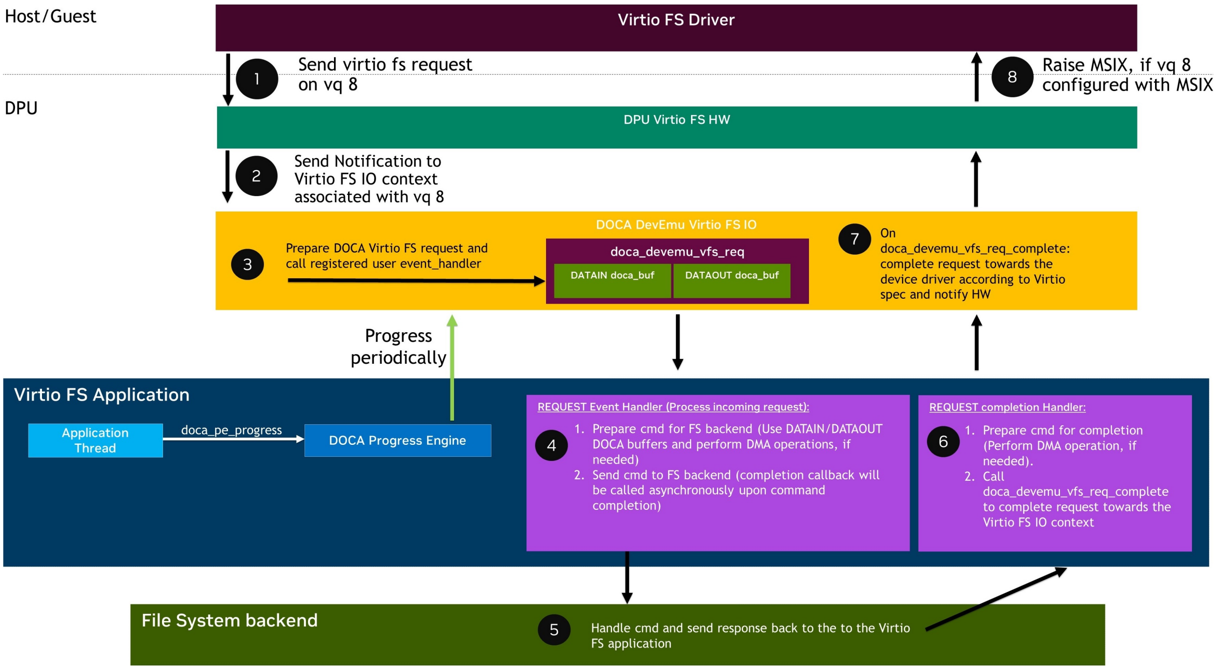
IO Path

This section describes the flow for a single virtio-FS request sent by the device driver until its completion.

It is assumed that the user properly configured an event handler for an incoming virtio-FS request as explained in section "Virtio-FS IO Configurations".

It is also assumed that the user is familiar with the virtio-FS specification and has the ability to perform DMA operations to/from the host using DOCA DMA or any other suitable method.

The DOCA virtio-FS flow is illustrated in the following diagram:


Host/Guest
DPU
Virtio FS Driver
1
Send virtio fs request on vq 8
8
Raise MSIX, if vq 8 configured with MSIX
DPU Virtio FS HW
2
Send Notification to Virtio FS IO context associated with vq 8
DOCA DevEmu Virtio FS IO
3
Prepare DOCA Virtio FS request and call registered user event_handler
doca_devemu_vfs_req
DATAIN doca_buf
DATAOUT doca_buf
7
On doca_devemu_vfs_req_complete: complete request towards the device driver according to Virtio spec and notify HW
Progress periodically
Virtio FS Application
Application Thread
doca_pe_progress
DOCA Progress Engine
REQUEST Event Handler (Process incoming request):
4
1. Prepare cmd for FS backend (Use DATAIN/DATAOUT DOCA buffers and perform DMA operations, if needed)
2. Send cmd to FS backend (completion callback will be called asynchronously upon command completion)
REQUEST completion Handler:
6
1. Prepare cmd for completion (Perform DMA operation, if needed).
2. Call doca_devemu_vfs_req_complete to complete request towards the Virtio FS IO context
File System backend
5
Handle cmd and send response back to the to the Virtio FS application

14.5 DOCA Utils

This section includes modules that may be used by application developers to speed up their development process.

This section contains the following pages:

- DOCA Arg Parser

14.5.1 DOCA Arg Parser

This guide provides an overview and configuration instructions for DOCA Arg Parser API.

14.5.1.1 Introduction

The Arg Parser module makes it simple to create a user command-line interface to supply program arguments. The module supports both regular command-line arguments and flags from a JSON file.

It also creates help and usage messages for all possible flags when the user provides invalid inputs to the program.
General notes about DOCA Arg Parser:

- Arg Parser checks a variety of errors including invalid arguments and invalid types, and it prints the error along with program usage and exits when it encounters an error
- The module uses long flags as JSON keys
- The options `-j` and `--json` are reserved for Arg Parser JSON and cannot be used

14.5.1.2 API

For the library API reference, refer to ARGP API documentation in NVIDIA DOCA Library APIs.

> ⚠ The pkg-config (`*.pc` file) for the Arg Parser library is `doca-argp` .

The following sections provide additional details about the library API.

14.5.1.2.1 doca_argp_param

The data structure contains the program flag details needed to process DOCA ARGP. These details are used to generate usage information for the flag, identify if the user passed the flag in the command line and notify the program about the flag's value.

```
struct doca_argp_param;
```

14.5.1.2.2 doca_argp_param_create

Creates a DOCA ARGP parameter instance. The user is required to update the param attributes by calling the respective setter functions and registering the param by calling `doca_argp_register_param()` .

```
doca_error_t doca_argp_param_create(struct doca_argp_param **param);
```

- `param [out]` - DOCA ARGP param structure with unset attributes

14.5.1.2.3 doca_argp_register_param

Calling this function registers the program flags in the Arg Parser database. The registration includes flag details. Those details are used to parse the input arguments and generate usage print.

> ⚠ The user must register all program flags before calling `doca_argp_start()` .

```
doca_error_t doca_argp_register_param(struct doca_argp_param *input_param);
```

- `input_param [in]` - program flag details

14.5.1.2.4 doca_argp_set_dpdk_program

Marks the programs as a DPDK program. Once ARGP is finished with the parsing, DPDK (EAL) flags are forwarded to the program by calling the given callback function.

```
void doca_argp_set_dpdk_program(dpdk_callback callback);
```

- `callback [in]` - callback function to handle DPDK (EAL) flags.

14.5.1.2.5 doca_argp_start

Calling this function starts the classification of command-line mode or JSON mode and is responsible for parsing DPDK flags if needed. If the program is triggered with a JSON file, the DPDK flags are parsed from the file and constructed in the correct format. DPDK flags are forwarded back to the program by calling the registered callback.

```
doca_error_t doca_argp_start(int argc, char **argv);
```

- `argc [in]` - number of input arguments
- `argv [in]` - program command-line arguments

14.5.1.3 DPDK Flags

The following table lists the supported DPDK flags:

Short Flag	Long Flag/ JSON Key	Flag Description	JSON Content	JSON Content Description

`a`	`devices`	Add a PCIe device to the list of devices to probe	`"devices": [`
			`  {`
			`    "device": "regex",`
			`    "id": "03:00.0"`
			`  },`
			`  {`
			`    "device": "sf",`
			`    "id": "4",`
			`    "sft": true`
			`  },`
			`  {`
			`    "device": "sf",`
			`    "id": "5",`
			`    "hws": true`
			`  },`
			`  {`
			`    "device": "vf",`
			`    "id": "b1:00.3"`
			`  },`
			`  {`
			`    "device": "pf",`
			`    "id": "03:00.0",`
			`    "sft": true`
			`  },`
			`  {`
			`    "device": "gpu",`
			`    "id": "06:00.0"`
			`  }`
			`]`
				Passing configuration for 6 devices:
				1. RegEx device with PCIe address `03:00.0`.
				2. SF device with number 4, SFT enabled.
				3. SF device with number 5, HW steering enabled.
				4. VF device with PCIe `b1:00.3`.
				5. PF device with PCIe address `03:00.0`, SFT enabled.
				6. GPU device with PCIe address `06:00.0`.

`c`	`core-mask`	Hexadecimal bitmask of cores to run on	`"core-mask": "0xff"`	Set core mask with value `0xff`
`l`	`core-list`	List of cores to run on	`"core-list": "0-4"`	Limit program to use five cores (core-0 to core-4)

⚠ Additional DPDK flags may be added in the "flags" JSON field.

14.5.1.4 DOCA General Flags

The following table lists the supported DOCA general flags:

Short Flag	Long Flag/ JSON Key	Flag Description	JSON Content	JSON Content Description
`h`	`help`	Print a help synopsis	N/A	Supported only on CLI
`v`	`version`	Print program version information	N/A	Supported only on CLI
`l`	`log-level`	Sets the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70	`"log-level": 60`	Set the log level to DEBUG mode
N/A	`sdk-log-level`	Sets the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70	`"sdk-log-level": 40`	Set the SDK log level to WARNING mode
`j`	`json`	Parse all command flags from an input json file	N/A	Supported only on CLI

14.5.1.5 DOCA Program Flags

The flags for each program can be found in the document dedicated to that program, including instructions on how to run it, whether by providing a JSON file or by using the command-line interface.

14.5.1.6 JSON File Example

An application JSON file can be found under `/opt/mellanox/applications/[APP name]/bin/[APP name]_params.json`.

```
{
  "doca_dpdk_flags": {
    // -a - Add a device to the allow list.
    "devices": [
      {
        "device": "sf",
        "id": "4",
        "sft": true
      },
      {
        "device": "sf",
        "id": "5",
        "sft": true
      }
    ],
    // -c - Hexadecimal bitmask of cores to run on
    "core-mask": "0xff",
    // Additional DPDK (EAL) flags (if needed)
    "flags": ""
  },
  "doca_general_flags": {
    // -l - Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO,
60=DEBUG, 70=TRACE>
    "log-level": 60,
    // --sdk-log-level - Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
    "sdk-log-level": 40,
  },
  // flags below are for DMA Copy application.
  "doca_program_flags":{
    // -f - Full path for file to be copied/saved
    "file": "/tmp/dma_copy_test.txt",
    // -p - commm channel doca device pci address
    "pci-addr": "03:00.0",
    // -r - comm channel doca device representor pci address
    "rep-pci": "b1:00.0"
  }
}
```

14.6 DOCA Drivers

This section describes underlying drivers included in DOCA and includes the following pages:

- DOCA UCX
- MLX Drivers (MLNX_OFED)

14.6.1 DOCA UCX

This guide provides instructions for developing applications on top of the UCX library.

14.6.1.1 Introduction

Unified Communication X (UCX) is an optimized point-to-point communication framework.

UCX exposes a set of abstract communication primitives that utilize the best available hardware resources and offloads, such as active messages, tagged send/receive, remote memory read/write, atomic operations, and various synchronization routines. The supported hardware types include RDMA (InfiniBand and RoCE), TCP, GPUs, and shared memory.

UCX facilitates rapid development by providing a high-level API, masking the low-level details, while maintaining high-performance and scalability.

UCX implements best practices for transfer of messages of all sizes, based on the accumulated experience gained from applications running on the world's largest datacenters and supercomputers.

14.6.1.2 Prerequisites

UCX runtime libraries are installed as part of the DOCA installation.

UCX is used the same way from the host and the DPU side.

Any active network device available on the system might be used by UCX, including network devices that might be unreachable to the remote peer.

If one of the destinations is not reachable via a certain network device (e.g., a BlueField cannot reach another BlueField via `tmfifo_net0`), UCX communication may fail.

To resolve this, use the UCX environment variable `UCX_NET_DEVICES` to specify which devices UCX can use. For example:

```
export UCX_NET_DEVICES=enp3s0f0s0,enp3s0f1s0
```

Or:

```
env UCX_NET_DEVICES=enp3s0f0s0,enp3s0f1s0 <UCX-program>
```

Using the command `show_gids` on the BlueField one can obtain the mlx device name and the port of an SF. Then that can be used to limit the UCX network interfaces and allow IB. For example:

```
dpu> show_gids
DEV     PORT    INDEX   GID                                      IPv4                    VER  DEV
---     ----    -----   ---                                      ------------            ---  ---
mlx5_2  1       0       fe80:0000:0000:0000:0052:72ff:fe63:1651                          v2   enp3s0f0s0
mlx5_3  1       0       fe80:0000:0000:0000:0032:6bff:fe13:f13a                          v2   enp3s0f1s0
dpu> env UCX_NET_DEVICES=mlx5_2:1,mlx5_3:1 <UCX-program>
```

When RDMACM is not available, it is also required to list the Ethernet devices in `UCX_NET_DEVICES` configuration, so they could be used for TCP-based connection establishment. For example:

```
dpu> env UCX_NET_DEVICES=enp3s0f0s0,enp3s0f1s0,mlx5_2:1,mlx5_3:1 <UCX-program>
```

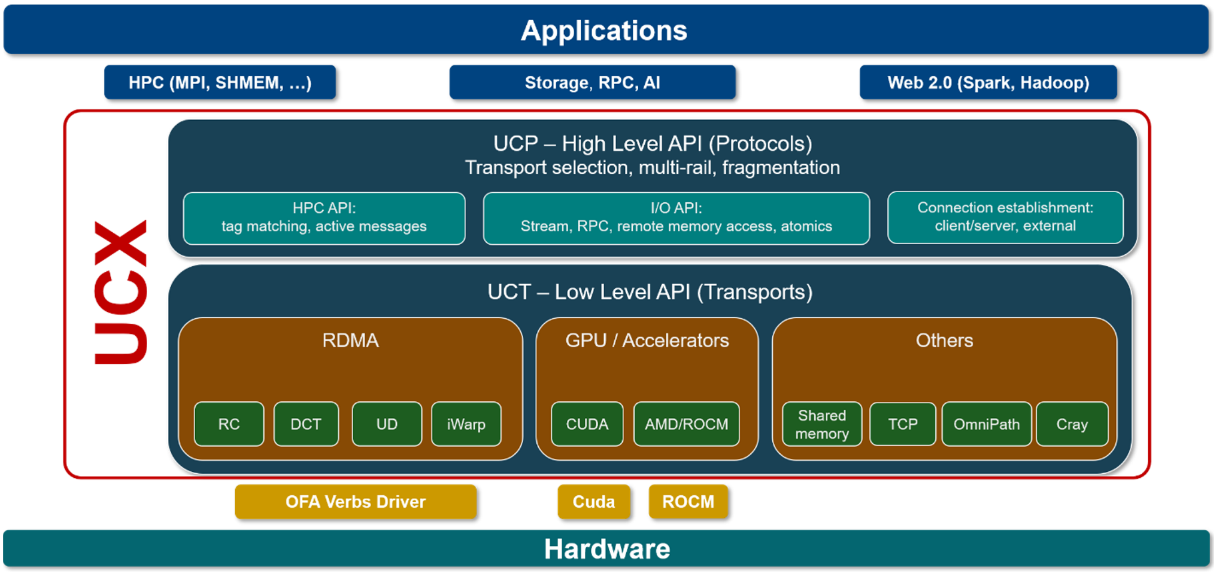
14.6.1.3 Architecture

The following image describes the software layers of UCX middleware.

On the upper layer, various applications that utilize high-speed communications are built on top of the UCX high-level API (UCP).

UCP layer implements the business logic to utilize, combine, and manipulate different transports to achieve the best possible performance for different use cases. This logic decides which transports must be used for each message, which types of basic hardware communication primitives to use, how to fragment messages, etc.

UCT, the transport API, is a hardware abstraction layer that brings different types of communication devices to a common denominator. There are multiple communication primitives defined by UCT API, but each transport service may implement only some of them—preferably the ones that are natively supported by the underlying hardware. UCT users (e.g., UCP) are expected to handle the missing communication primitives defined by UCT API but not implemented by a transport service.



14.6.1.3.1 UCP Objects

This section describes the high-level communication objects that are used by most applications written on top of UCX.

14.6.1.3.1.1 UCP Context (ucp_context_h)

The context is the top-level object and it defines the scope of all other UCX objects. It is possible to create multiple contexts in the same process to have a complete separation of hardware and memory resources.

14.6.1.3.1.2 UCP Worker (ucp_worker_h)

The worker represents a communication state and its associated network resources. It is responsible for sending and processing incoming messages and handling all network-related events. All point-to-point connections are created in the scope of a particular worker.

A worker object can be defined to support usage from multiple threads. However, due to lock contention, the performance is better when a given worker is used most of the time from one thread.

The worker progresses communications either by active polling, waiting for asynchronous events, or a combination of both.

14.6.1.3.1.3 UCP Endpoint (ucp_ep_h)

The endpoint represents a connection from a local worker to a remote worker. That remote worker may be created in any place that is reachable by one of the communication networks supported by UCT layer. That could be, for example, on a different host in the fabric, the same host, on the DPU, or even in the same process.

14.6.1.3.1.4 UCP Listener (ucp_listener_h)

The listener binds to a network port number on the underlying operating system, and dispatches incoming connection requests. The incoming connection request can be used to create a matching endpoint on the server (passive) side or rejected and released.

14.6.1.3.1.5 UCP Request (ucp_request_h)

The request object is created by one of the non-blocking communications primitives in a case where the operation could not be completed immediately in-place. The application is expected to check the request for completion, either by testing it directly, or by associating a custom callback with the request.

14.6.1.4 API

This section describes the main UCX APIs for high-speed communications. For the full reference, refer to UCX API specification.

UCX exposes two kinds of API: the high-level UCP API and the low-level UCT (transport) API. For most applications, it is recommended to use only the UCP API, since it relieves much of the burden of handling each transport's capabilities, limitations, and performance traits.

Many of the APIs accept a structure pointer with a `field_mask` as an argument. This method is used to provide backward ABI/API compatibility: If new function arguments are introduced, they are added as new fields in the struct, so the function signature does not change. In addition, `field_mask` specifies which struct fields are valid from the caller's (user application) perspective. UCX only accesses the fields enabled by this bitmask and uses default values for the remaining struct fields.

Some APIs require passing user-defined callbacks as a method to get notifications about specific events. Unless otherwise specified, such callbacks are called from the context of the `ucp_worker_progress()` call (detailed below), and are expected to complete quickly or defer some of their tasks to another thread (to avoid timeouts and starvation of processing from other network events).

> ⚠ The pkg-config (`*.pc` file) for the UCX library is named `ucx` .

The following sections provide additional details about the library API.

14.6.1.4.1 ucs_status_t

An `enum` type that holds all UCX error codes.

14.6.1.4.2 ucp_init

```
ucs_status_t ucp_init(const ucp_params_t *params, const ucp_config_t *config, ucp_context_h *context_p)
```

- `params [in]` - points to a structure with optional parameters. All fields are optional except `features`, which must be set.

- `config [in]` - optional, can be NULL for default behavior. Configuration can be obtained by calling `ucp_config_read()`.

  > ⚠ The supported configuration options can change between UCX versions. The full list can be obtained by running the `ucx_info` CLI tool:
  >
  > ```
  > ucx_info -c -f
  > ```

- `context_p [out]` - a pointer to a location in memory for the created UCP context

The function returns an error code as defined by `ucs_status_t`.

This function creates a new UCP top-level context and returns it by value in the `context_p` argument.

14.6.1.4.3 ucp_cleanup

```
void ucp_cleanup(ucp_context_h context_p)
```

- `context_p [in]` - a UCP context instance

This function destroys a previously created context. Prior to calling this function, any other resources created on this context (e.g., workers or endpoints) must be destroyed.

14.6.1.4.4 ucp_worker_create

```
ucs_status_t ucp_worker_create(ucp_context_h context, const ucp_worker_params_t *params, ucp_worker_h *worker_p)
```

- `context [in]` - an existing UCP context
- `params [in]` - points to a structure with configuration parameters. All fields are optional. Commonly, only the field `thread_mode` is used. Possible `thread_mode` values are as follows:
  - `UCS_THREAD_MODE_SINGLE` - only one specific thread (typically, the one that created the worker) is used to access the worker and its associated endpoints.
  - `UCS_THREAD_MODE_SERIALIZED` - multiple threads can access the worker and its associated endpoints, but only one at a time. This implies an exclusion mechanism (e.g., locking) implemented in the application. Sometimes, more expensive bus flushing instructions are needed with serialized mode, compared to single thread mode.
  - `UCS_THREAD_MODE_MULTI` - multiple threads can access the worker at any given time. UCX takes care of the locking internally. As of version 1.12, it is implemented as a global lock on the worker.
- `worker_p [out]` - a pointer to a location in memory for the created worker

The function returns an error code as defined by `ucs_status_t`.

This function creates a new UCP worker on a previously created context and returns it by value in the `worker_p` argument.

> ⚠ When `ucp_worker_create()` succeeds, the caller is still expected to check the actual thread mode the worker was created with by calling `ucp_worker_query()` API, and take the necessary actions (for example, report an error or fallback) if the returned thread mode is not as expected to be.

14.6.1.4.5 ucp_worker_destroy

```
void ucp_worker_destroy(ucp_worker_h worker)
```

- `context_p [in]` - an UCP worker instance

This function destroys a previously created worker. Prior to calling this function, all associated endpoints and listeners must be destroyed.

Destroying the worker may cause communication errors on any remote peer that has an open endpoint to this worker. These errors are handled according to that endpoint's error handling configuration (detailed in section `ucp_ep_create`).

14.6.1.4.6 ucp_listener_create

```
ucs_status_t ucp_listener_create(ucp_worker_h worker, const ucp_listener_params_t *params, ucp_listener_h
*listener_p)
```

- `worker [in]` - an existing UCP worker
- `params [in]` - points to a structure with configuration parameters. The fields `sockaddr` and `conn_handler` are mandatory, but the rest of the fields are optional.
  - `sockaddr` - specifies IPv4/IPv6 address to listen for connections. The semantics are similar to the built-in bind() function. `INADDR_ANY/INADDR6_ANY` can be used to listen on all network interfaces. If the port number is set to 0, a random unused port is selected. The actual port number can be obtained by calling the `ucp_listener_query()` API.
  - `conn_handler` - a callback for handling incoming connection requests along with an associated user-defined argument. The callback type is defined as:

    ```
    void (*ucp_listener_conn_callback_t) (ucp_conn_request_h conn_request, void *arg)
    ```

    Whenever a remote endpoint is created through this listener, this callback is called on the listener side with a new `conn_request` object representing the incoming connection, and the user-defined argument `arg` that is passed to `ucp_listener_create()`.
    The callback is expected to process this connection request by either creating an endpoint for it (pass `conn_request` as a parameter to `ucp_ep_create`, including on a different worker), or rejecting and destroying it (call `ucp_listener_reject`). This does not have to happen immediately. The callback may put the connection request on an internal application queue and process it later.
- `listener_p [out]` - a pointer to a location in memory for the created listener

The function returns an error code as defined by `ucs_status_t`.

This function creates a new listener object to accept incoming connections on a specific network port, and returns it by value in the `listener_p` argument.

14.6.1.4.7 ucp_listener_destroy

```
void ucp_listener_destroy(ucp_listener_h listener_p)
```

- `listener_p [in]` - a listener instance

This function destroys a previously created listener. Prior to calling this function, any connection requests that were reported by `conn_handler` are expected to be processed. Pending connection requests that have not been reported to the application yet, or new connection requests that arrive after this function is called, are rejected.

14.6.1.4.8 ucp_ep_create

```
ucs_status_t ucp_ep_create(ucp_worker_h worker, const ucp_ep_params_t *params, ucp_ep_h *ep_p)
```

- `worker [in]` - an existing UCP worker
- `params [in]` - Points to a structure with configuration parameters. A creation mode field must be set. Other fields are optional. Commonly used fields are described in the following subsections.
- `ep_p [in]` - a pointer to a location in memory for the created endpoint

The function returns an error code as defined by `ucs_status_t`.

This function creates a new connection to a remote peer and returns it by value in the `ep_p` parameter. The new endpoint can be used for communication immediately after it is created, though some operations may be queued internally and sent after the underlying connection is established.

14.6.1.4.8.1 Create Modes (ucp_ep_params_t)

There are three ways the endpoint can be created:

- Client connects to a remote listener
  In this case, the `sockaddr` field specifies the remote IPv4/IPv6 address and port number. The `flags` field must be enabled and must include the `UCP_EP_PARAMS_FLAGS_CLIENT_SERVER` flag. Optionally, from UCX version 1.13 on, the `local_sockaddr` field may be used to specify a local source device address to bind to.
- Server creates an endpoint due to an incoming connection request
  In this case, the `conn_request` field must be set to this connection request. Such endpoint can optionally be created on a different worker, not the same one this connection request was accepted on.
- Create an endpoint to a specific worker address
  In this case, the field `address` must be set to point to a remote worker's address. That address (and its length) must be obtained on the remote side by calling `ucp_worker_query()` and sent using an application-defined method (e.g., TCP socket, or other existing

communication mechanism). The internal structure of the address is opaque and may change in different versions.

14.6.1.4.8.2 User-Defined Error Handling (ucp_ep_params_t)

By default, unexpected errors on the connection (e.g., network disconnection or aborted remote process) generate a fatal failure. To enable graceful error handing, several parameters must be set during endpoint creation:

- The `err_mode` field must be set to `UCP_ERR_HANDLING_MODE_PEER`. This guarantees that send requests are always completed (successfully or error). Otherwise, network errors are considered fatal and abort the application without giving it a chance to perform cleanup or fallback flows.
- `The err_handler.cb` field must be set to a user-defined callback which is called if a connection error occurs. The error handler is defined as follows:

  ```
  void (*ucp_err_handler_cb_t)(void *arg, ucp_ep_h ep, ucs_status_t status)
  ```

  The callback parameters are the user-defined argument (passed in `user_data`), the endpoint handle on which the error happened, and the error code.
  After this callback, no more communications should be done on the endpoint. The application is expected to close the endpoint.
- The `user_data` field must be set to a user-defined argument passed to the `err_handler` c allback

14.6.1.4.9 ucs_status_ptr_t

```
typedef void* ucs_status_ptr_t;
```

This function is commonly used as a return value for non-blocking operations.

The return value of `ucs_status_ptr_t` combines a status code and a request pointer which may be one of the following:

- A NULL pointer indicating that the operation has completed successfully in-place. The user-provided callback, if there is one, is not called.
- An error status, that can be detected by the `UCS_PTR_IS_ERR(status)` macro and extracted by `UCS_PTR_STATUS(status)`.
- Otherwise, the status is a request pointer which can also be detected by the `UCS_PTR_IS_PTR(status)` macro. This means that the communication operation has started (or was queued) but not yet completed. The completion is reported by calling the user-provided callback (in `ucp_request_param_t`) or through an explicit check on the request status by calling `ucp_request_check_status()`.

14.6.1.4.10 ucp_ep_close_nbx

```
ucs_status_ptr_t ucp_ep_close_nbx(ucp_ep_h ep, const ucp_request_param_t *param)
```

- `ep [in]` - an existing UCP endpoint

- `param [in]` - points to a structure that defines how the closing operation is performed. The `flags` field of the `param` structure specifies which method to use to close the endpoint:
    - `UCP_EP_CLOSE_MODE_FORCE` - close the endpoint immediately without attempting to flush outstanding operation. Some requests already completed on the transport level may complete successfully, others may be completed with an error status. In the latter case, it is not known whether they have reached the destination process or completed there.
      Closing an endpoint this way is equivalent to calling `close()` on a TCP socket and can generate a connection error on the remote side. Therefore, to use this mode, both the local and remote endpoints must be created with the `err_mode` parameter set to `UCP_ERR_HANDLING_MODE_PEER`.
    - `UCP_EP_CLOSE_MODE_FLUSH` - synchronize with the remote peer and flush outstanding operations. Some operations may be canceled and complete with the status `UCS_ERR_CANCELED`. However, it is guaranteed that they did not complete on the remote peer as well.

The function returns a status pointer to check the operation's status. `NULL` means success.

This function starts the process of closing a previously created endpoint. The function is non-blocking, and the returned value is a status pointer used to indicate when the endpoint is fully destroyed. For more information, refer to section Communications.

14.6.1.4.11 ucp_request_param_t

```
struct ucp_request_param_t {
    uint32_t op_attr_mask;
    uint32_t flags;
    union ucp_request_param_t cb;
    void *user_data;
    ucp_datatype_t datatype;
    /* Some other fields that are rarely used */
    …
}
```

- `op_attr_mask [in]` - mask of enabled fields and several control flags.
- `flags [in]` - operation-specific flags. Each API method defines its own set of flags for this field.
- `cb [in]` - callback for when the operation is completed.
- `user_data [in]` - user-defined argument passed to the completion callback.
- `datatype [in]` - may be used to specify a custom data layout for the data buffer (not `user_data`) that is provided to the communication API. If this parameter is not set, the data buffer is treated as a contiguous byte buffer.

The fields of `ucp_request_param_t` specify several common attributes and flags that are used to control how the communications request is allocated and completed. This is aimed to optimize different use-cases.

14.6.1.4.12 ucp_worker_progress

```
unsigned ucp_worker_progress(ucp_worker_h worker)
```

- `worker [in]` - an existing UCP worker

The function returns a non-zero value if any communication has been progressed. Otherwise, it returns zero.

This function progresses outstanding communications on the worker. This includes polling hardware and shared memory queues, calling callbacks, pushing pending operations to the network devices, advancing the state of complex protocols, progressing connection establishment process, and more.

Though some transports, such as RDMA, offload do much of the heavy lifting, the initiation and completion of communication operations still must be performed explicitly by the process. UCX does not spawn additional progress threads. Instead, it is expected that the upper-layer application spawns its own progress thread, as needed, to call `ucp_worker_progress()` .

> ⚠ This function cannot be used from inside a callback.

14.6.1.4.13 ucp_am_send_nbx

```
ucs_status_ptr_t ucp_am_send_nbx(ucp_ep_h ep, unsigned id, const void *header,
                                 size_t header_length, const void *buffer,
                                 size_t count, const ucp_request_param_t *param)
```

- `ep [in]` - connection to send the active message on. Previously returned from `ucp_ep_create()` .
- `id [in]` - active message identifier. This is an arbitrary 16-bit integer value defined by the application and used to select the active message callback to call on the receiver side. This allows handling different types of messages by different callback functions.
- `header [in]` - pointer to a user-defined header for an active message
- `header_length [in]` - length of the header to send. Usually, the header is small and, in any case, it should be no larger than the `max_am_header` worker attribute, as returned from `ucp_worker_query()` . The header size could vary depending on the available transports and is usually expected to be at least 256 bytes.
- `buffer [in]` - pointer to the active message payload
- `count [in]` - number of elements in the payload buffer. By default, each element is a single byte, so this is the byte-length of the buffer. Other data layouts, such as IO vector (IOV) list, could be specified by `param->datatype` .
- `param [in]` - additional parameters controlling request completion semantics. The relevant field is only `flags` and it can be set to a combination of the following flags:
  - `UCP_AM_SEND_FLAG_REPLY` - force passing `reply_ep` to the callback on the receiver side. This can increase the internal header size and add some overhead.
  - `UCP_AM_SEND_FLAG_EAGER` - force using eager protocol (details below).
  - `UCP_AM_SEND_FLAG_RNDV` - force using rendezvous protocol (details below).

The active message can be sent either by the eager or rendezvous protocol. Eager protocol means the data buffer is available on the receiver immediately during the callback, while the rendezvous protocol requires fetching the data using an additional call to `ucp_am_recv_data_nbx()`, allowing it to be placed directly to an application-selected buffer. By default, smaller messages are sent via eager protocol, and larger messages use rendezvous protocol. This can be overridden using `UCP_AM_SEND_FLAG_EAGER` or `UCP_AM_SEND_FLAG_RNDV`.

> ⚠ `UCP_AM_SEND_FLAG_EAGER` and `UCP_AM_SEND_FLAG_RNDV` are mutually exclusive.

The function returns a status pointer to check the operation's status. `NULL` means success.

This function initiates sending of an active message from the initiator side. As a result, a designated callback (registered by `ucp_worker_set_am_recv_handler`) is called on the receiver side to handle this message. The function is non-blocking, so if the send operation is not completed immediately, a request handle is retuned.

14.6.1.4.14 ucp_worker_set_am_recv_handler

```
ucs_status_t ucp_worker_set_am_recv_handler(ucp_worker_h worker, const ucp_am_handler_param_t *param)
```

- `worker [in]` - an existing UCP worker.
- `param [in]` - set callback configurations. See more below.

The function returns a non-zero value if any communication has been progressed. Otherwise, it returns zero.

This function registers a callback for processing active messages on the given worker.

The following are the mandatory fields to set in `param`:

- `id` - active message identifier to bind with the registered callback. Callback is invoked when receiving incoming messages with the same ID.
- `arg` - a user-defined argument to pass to the active message callback.
- `cb` - a user-defined callback to invoke when an active message arrives. The callback is defined as:

```
ucs_status_t (*ucp_am_recv_callback_t)(void *arg, const void *header,
              size_t header_length, void *data,
              size_t length,
              const ucp_am_recv_param_t *param)
```

The following are the parameters passed from UCX to the callback:

- `arg` - the same user-defined argument passed to `ucp_worker_set_am_recv_handler`.
- `header` - points to the active message header as defined by the sender side while sending the active message. The header should be consumed by the callback since it is not valid after the callback returns.
- `header_length` - valid size of the buffer pointer by `header`.

- `data` - pointer to the data or an opaque handle that can be used to fetch the data according to the `UCP_AM_RECV_ATTR_FLAG_RNDV` flag in the field `param->recv_attr`. When flag is on, this is an opaque handle.
- `length` - length of the active message data (even if the data argument is an opaque handle and not the actual data).
- `param` - pointer to additional parameters of the incoming message. The relevant fields are:
  - `recv_attr` - flags providing more information about the incoming message.
  - `reply_ep` - if `UCP_AM_RECV_ATTR_FIELD_REPLY_EP` is set in `recv_attr`, then this field holds a handle to an endpoint that can be used to send replies to the active message sender.

The callback is expected to return `UCS_OK` if the message data has been consumed or if `UCP_AM_RECV_ATTR_FLAG_RNDV` is set in `recv_attr`. Otherwise, the if `UCP_AM_RECV_ATTR_FLAG_DATA` is set in `recv_attr`, the callback is allowed to keep the data for later processing (by adding it to an internal application queue, for example). In this case, the callback should return `UCS_INPROGRESS` as indication that the data should persist.

When a message arrives with `UCP_AM_RECV_ATTR_FLAG_RNDV` flag, the function `ucp_am_recv_data_nbx` must be used to fetch the data from the sender.

14.6.1.4.15 ucp_am_recv_data_nbx

```
ucs_status_ptr_t ucp_am_recv_data_nbx(ucp_worker_h worker, void *data_desc,
                                      void *buffer, size_t count,
                                      const ucp_request_param_t *param)
```

- `worker [in]` - UCP worker object to use for initiating the receive operation.

  > ⚠ The connection handle (endpoint) is not needed.

- `data_desc [in]` - handle for the data to receive. Obtained from the `data` argument for the active message callback.
- `buffer [in]` - receive buffer for the incoming data.
- `count [in]` - number of elements in the payload buffer. By default, each element is a single byte, so this is the byte-length of the buffer. Other data layouts, such as the IOV list, may be specified by `param->datatype`.
- `param [in]` - additional parameters that control request allocation and completion reporting. No specific flags are needed for this function.

The function returns a status pointer to check the operation's status. `NULL` means success.

This function is used for rendezvous active messages. The function initiates the process of fetching data from the sender side into an application-defined receive buffer. It is expected to be used when an active message callback is called with the `UCP_AM_RECV_ATTR_FLAG_RNDV` flag set in `params->recv_attr` field.

14.6.1.5 UCX Best Practices

14.6.1.5.1 Initialization

An application using UCX will usually create one global context (`ucp_context_h`) then create one or more workers (`ucp_worker_h`). Each worker consumes some memory for send/receive buffers, so it is not recommended to create too many workers. The rule of thumb is that the number of workers should be roughly tied to the number of CPU cores/threads.

The mapping of workers to threads is defined by the application's use case, for example:

- A single-threaded application does not need more than one worker
- A simple implementation of a multi-threaded application can create one or more workers in multi-threaded mode. These workers can be used by any thread.
- A multi-threaded application with a strong affinity between the thread and CPU core can create a dedicated worker per thread. These workers can be created in a single-threaded mode.
- Applications with many threads can implement a pool of workers and use one randomly or assign some to threads temporarily.

> ⚠ If there are multiple workers, each of them needs to create its own set of endpoints, since every endpoint connects a specific pair of workers.

To initiate communications, the application should create endpoints (`ucp_ep_h`) connected to the remote peers. There are two main methods to create an endpoint: Either by connecting directly to a remote worker's address, or by creating a listener object (`ucp_listener_h`) and connecting to remote IP address and port. These methods are described in more detail in the `ucp_ep_create()` s ection.

14.6.1.5.2 Communications

After initializing the UCP context, worker, and endpoints, the application can start using the endpoint for communications. Usually, endpoints are associated with application-level object that represents a connection.

Most communication operations follow a similar pattern: A non-blocking function (with `_nbx` suffix) receives a pointer to the `ucp_request_param_t` structure and returns `ucs_status_ptr_t` . Using a struct pointer allows extending the operations and while maintaining backward compatibility.

There are several types of communication methods supported by UCP intended for different kinds of applications. The recommended method for most applications is active messages which mean that the initiator can send arbitrary data to the responder, and the responder invokes a callback that can access this data.

14.6.2 MLX Drivers (MLNX_OFED)

Unable to render include or excerpt-include. Could not retrieve page.

The chapter contains the following sections:

- InfiniBand Network
- Storage Protocols
- Virtualization
- Resiliency
- Docker Containers
- HPC-X
- Fast Driver Unload

14.6.2.1 InfiniBand Network

The chapter contains the following sections:

- InfiniBand Interface
- NVIDIA SM
- QoS - Quality of Service
- IP over InfiniBand (IPoIB)
- Advanced Transport
- Optimized Memory Access
- NVIDIA PeerDirect
- CPU Overhead Distribution
- Out-of-Order (OOO) Data Placement
- IB Router
- MAD Congestion Control

14.6.2.1.1 InfiniBand Interface

14.6.2.1.1.1 Port Type Management

For information on port type management of ConnectX-4 and above adapter cards, please refer to Port Type Management/VPI Cards Configuration section.

14.6.2.1.1.2 RDMA Counters

- RDMA counters are available only through sysfs located under:
    - `# /sys/class/infiniband/<device>/ports/*/hw_counters/`
    - `# /sys/class/infiniband/<device>/ports/*/counters`

For mlx5 port and RDMA counters, refer to the Understanding mlx5 Linux Counters Community post.

14.6.2.1.2 NVIDIA SM

NVIDIA SM is an InfiniBand compliant Subnet Manager (SM). It is provided as a fixed flow executable called "opensm", accompanied by a testing application called osmtest. NVIDIA SM implements an InfiniBand compliant SM according to the InfiniBand Architecture Specification chapters: Management Model, Subnet Management, and Subnet Administration.

14.6.2.1.2.1 OpenSM Application

OpenSM is an InfiniBand compliant Subnet Manager and Subnet Administrator that runs on top of the NVIDIA OFED stack. OpenSM performs the InfiniBand specification's required tasks for initializing InfiniBand hardware. One SM must be running for each InfiniBand subnet.
OpenSM defaults were designed to meet the common case usage on clusters with up to a few hundred nodes. Thus, in this default mode, OpenSM will scan the IB fabric, initialize it, and sweep occasionally for changes.
OpenSM attaches to a specific IB port on the local machine and configures only the fabric connected to it. (If the local machine has other IB ports, OpenSM will ignore the fabrics connected to those other ports). If no port is specified, opensm will select the first "best" available port. opensm can also present the available ports and prompt for a port number to attach to.
By default, the OpenSM run is logged to/var/log/opensm.log. All errors reported in this log file should be treated as indicators of IB fabric health issues. (Note that when a fatal and non-recoverable error occurs, OpenSM will exit). opensm.log should include the message "SUBNET UP" if OpenSM was able to set up the subnet correctly.

Syntax

```
opensm [OPTIONS]
```

For the complete list of OpenSM options, please run:

```
opensm --help / -h / -?
```

Environment Variables

The following environment variables control OpenSM behavior:

- OSM_TMP_DIR - controls the directory in which the temporary files generated by OpenSM are created. These files are: opensm-subnet.lst, opensm.fdbs, and opensm.mcfdbs. By default, this directory is /var/log.
- OSM_CACHE_DIR - opensm stores certain data to the disk such that subsequent runs are consistent. The default directory used is /var/cache/opensm. The following file is included in it:
  `guid2lid` - stores the LID range assigned to each GUID

Signaling

When OpenSM receives a HUP signal, it starts a new heavy sweep as if a trap has been received or a topology change has been found.
Also, SIGUSR1 can be used to trigger a reopen of /var/log/opensm.log for logrotate purposes.

Running OpenSM as Daemon

OpenSM can also run as daemon. To run OpenSM in this mode, enter:

```
host1# service opensmd start
```

14.6.2.1.2.2 osmtest

osmtest is a test program for validating the InfiniBand Subnet Manager and Subnet Administrator. osmtest provides a test suite for opensm. It can create an inventory file of all available nodes, ports, and PathRecords, including all their fields. It can also verify the existing inventory with all the object fields and matches it to a pre-saved one.
osmtest has the following test flows:

- Multicast Compliancy test
- Event Forwarding test
- Service Record registration test
- RMPP stress test
- Small SA Queries stress test

For further information, please refer to the tool's man page.

14.6.2.1.2.3 Partitions

OpenSM enables the configuration of partitions (PKeys) in an InfiniBand fabric. By default, OpenSM searches for the partitions configuration file under the name /etc/opensm/partitions.conf. To change this filename, you can use opensm with the '--Pconfig' or '-P' flags.
The default partition is created by OpenSM unconditionally, even when a partition configuration file does not exist or cannot be accessed.
The default partition has a P_Key value of 0x7fff. The port out of which runs OpenSM is assigned full membership in the default partition. All other end-ports are assigned partial membership.

> - Adding a new partition to the partition.conf file, does not require SM restart, but signalling SM process via a HUP signal (e.g pkill -HUP opensm).
> - The default partition cannot be removed.

> Adjustments to the Port GUIDs, including additions, removals, or membership alterations (denoted as "<PortGUID>=[full|limited|both]" in the "Partition Definition") can be applied with a HUP signal to the Subnet Manager process (e.g pkill -HUP opensm).

> Performing changes in the ipoib_bc_flags (ipoib/sl/scope/rate/mtu) and mgroup flags of an existing partition requires a restart of the Subnet Manager to take effect.

File Format

> Line content followed after '#' character is comment and ignored by parser.

General File Format

```
<Partition Definition>:\[<newline>\]<Partition Properties>
```

- <Partition Definition>:

```
[PartitionName][=PKey][,indx0][,ipoib_bc_flags][,defmember=full|limited]
```

where:

PartitionName	String, will be used with logging. When omitted empty string will be used.
PKey	P_Key value for this partition. Only low 15 bits will be used. When omitted will be auto-generated.
indx0	Indicates that this pkey should be inserted in block 0 index 0.
ipoib_bc_flags	Used to indicate/specify IPoIB capability of this partition.
defmember=full|limited|both	Specifies default membership for port GUID list. Default is limited.

ipoib_bc_flags are:

ipoib	Indicates that this partition may be used for IPoIB, as a result the IPoIB broadcast group will be created with the flags given, if any.
rate=<val>	Specifies rate for this IPoIB MC group (default is 3 (10GBps))
mtu=<val>	Specifies MTU for this IPoIB MC group (default is 4 (2048))
sl=<val>	Specifies SL for this IPoIB MC group (default is 0)
scope=<val>	Specifies scope for this IPoIB MC group (default is 2 (link local))

- <Partition Properties>:

```
\[<Port list>|<MCast Group>\]* | <Port list>
```

    - <Port List>:

```
<Port Specifier>[,<Port Specifier>]
```

        - <Port Specifier>:

```
<PortGUID>[=[full|limited|both]]
```

where

PortGUID	GUID of partition member EndPort. Hexadecimal numbers should start from 0x, decimal numbers are accepted too.
full, limited	Indicates full and/or limited membership for this both port. When omitted (or unrecognized) limited membership is assumed. Both indicate full and limited membership for this port.

- <MCast Group>:

```
mgid=gid[,mgroup_flag]*<newline>
```

where:

mgid=gid	gid specified is verified to be a Multicast address IP groups are verified to match the rate and mtu of the broadcast group. The P_Key bits of the mgid for IP groups are verified to either match the P_Key specified in by "Partition Definition" or if they are 0x0000 the P_Key will be copied into those bits.	
mgroup_flag	rate=<val>	Specifies rate for this MC group (default is 3 (10GBps))
	mtu=<val>	Specifies MTU for this MC group (default is 4 (2048))
	sl=<val>	Specifies SL for this MC group (default is 0)
	scope=<val>	Specifies scope for this MC group (default is 2 (link local)). Multiple scope settings are permitted for a partition. NOTE: This overwrites the scope nibble of the specified mgid. Furthermore specifying multiple scope settings will result in multiple MC groups being created.
	qkey=<val>	Specifies the Q_Key for this MC group (default: 0x0b1b for IP groups, 0 for other groups)
	tclass=<val>	Specifies tclass for this MC group (default is 0)
	FlowLabel=<val>	Specifies FlowLabel for this MC group (default is 0)

Note that values for rate, MTU, and scope should be specified as defined in the IBTA specification (for example, mtu=4 for 2048). To use 4K MTU, edit that entry to "mtu=5" (5 indicates 4K MTU to that specific partition).

PortGUIDs list:

```
PortGUID    GUID of partition member EndPort. Hexadecimal numbers should start from 0x, decimal numbers are
accepted too.
full or limited indicates full or limited membership for this port. When omitted (or unrecognized) limited
membership is assumed.
```

There are some useful keywords for PortGUID definition:

- 'ALL_CAS' means all Channel Adapter end ports in this subnet
- 'ALL_VCAS' means all virtual end ports in the subnet
- 'ALL_SWITCHES' means all Switch end ports in this subnet
- 'ALL_ROUTERS' means all Router end ports in this subnet
- 'SELF' means subnet manager's port. An empty list means that there are no ports in this partition

Notes:

- White space is permitted between delimiters ('=', ',',':',';').
- PartitionName does not need to be unique, PKey does need to be unique. If PKey is repeated then those partition configurations will be merged and the first PartitionName will be used (see the next note).
- It is possible to split partition configuration in more than one definition, but then PKey should be explicitly specified (otherwise different PKey values will be generated for those definitions).

Examples:

```
Default=0x7fff : ALL, SELF=full ;
Default=0x7fff : ALL, ALL_SWITCHES=full, SELF=full ;

NewPartition , ipoib : 0x123456=full, 0x3456789034=limi, 0x2134af2306 ;

YetAnotherOne = 0x300 : SELF=full ;
YetAnotherOne = 0x300 : ALL=limited ;

ShareIO = 0x80 , defmember=full : 0x123451, 0x123452;
# 0x123453, 0x123454 will be limited
ShareIO = 0x80 : 0x123453, 0x123454, 0x123455=full;
# 0x123456, 0x123457 will be limited
ShareIO = 0x80 : defmember=limited : 0x123456, 0x123457, 0x123458=full;
ShareIO = 0x80 , defmember=full : 0x123459, 0x12345a;
ShareIO = 0x80 , defmember=full : 0x12345b, 0x12345c=limited, 0x12345d;

# multicast groups added to default
Default=0x7fff,ipoib:
mgid=ff12:401b::0707,sl=1 # random IPv4 group
mgid=ff12:601b::16 # MLDv2-capable routers
mgid=ff12:401b::16 # IGMP
mgid=ff12:601b::2 # All routers
mgid=ff12::1,sl=1,Q_Key=0xDEADBEEF,rate=3,mtu=2 # random group
ALL=full;
```

The following rule is equivalent to how OpenSM used to run prior to the partition manager:

```
Default=0x7fff,ipoib:ALL=full;
```

14.6.2.1.2.4 Effect of Topology Changes

If a link is added or removed, OpenSM may not recalculate the routes that do not have to change. A route has to change if the port is no longer UP or no longer the MinHop. When routing changes are performed, the same algorithm for balancing the routes is invoked.
In the case of using the file-based routing, any topology changes are currently ignored. The 'file'

routing engine just loads the LFTs from the file specified, with no reaction to real topology. Obviously, this will not be able to recheck LIDs (by GUID) for disconnected nodes, and LFTs for non-existent switches will be skipped. Multicast is not affected by 'file' routing engine (this uses min hop tables).

14.6.2.1.2.5 Routing Algorithms

OpenSM offers the following routing engines:

1. Min Hop Algorithm
   Based on the minimum hops to each node where the path length is optimized.
2. UPDN Algorithm
   Based on the minimum hops to each node, but it is constrained to ranking rules. This algorithm should be chosen if the subnet is not a pure Fat Tree, and a deadlock may occur due to a loop in the subnet.
3. Fat-tree Routing Algorithm
   This algorithm optimizes routing for a congestion-free "shift" communication pattern. It should be chosen if a subnet is a symmetrical Fat Tree of various types, not just a K-ary-N-Tree: non-constant K, not fully staffed, and for any CBB ratio. Similar to UPDN, Fat Tree routing is constrained to ranking rules.
4. DOR Routing Algorithm
   Based on the Min Hop algorithm, but avoids port equalization except for redundant links between the same two switches. This provides deadlock free routes for hypercubes when the fabric is cabled as a hypercube and for meshes when cabled as a mesh.
5. Torus-2QoS Routing Algorithm
   Based on the DOR Unicast routing algorithm specialized for 2D/3D torus topologies. Torus-2QoS provides deadlock-free routing while supporting two quality of service (QoS) levels. Additionally, it can route around multiple failed fabric links or a single failed fabric switch without introducing deadlocks, and without changing path SL values granted before the failure.
6. Routing Chains
   Allows routing configuration of different parts of a single InfiniBand subnet by different routing engines. In the current release, minhop/updn/ftree/dor/torus-2QoS/pqft can be combined.

⚠ Please note that LASH Routing Algorithm is not supported.

MINHOP/UPDN/DOR routing algorithms are comprised of two stages:

1. MinHop matrix calculation. How many hops are required to get from each port to each LID. The algorithm to fill these tables is different if you run standard (min hop) or Up/Down. For standard routing, a "relaxation" algorithm is used to propagate min hop from every destination LID through neighbor switches. For Up/Down routing, a BFS from every target is used. The BFS tracks link direction (up or down) and avoid steps that will perform up after a down step was used.
2. Once MinHop matrices exist, each switch is visited and for each target LID a decision is made as to what port should be used to get to that LID. This step is common to standard and Up/

Down routing. Each port has a counter counting the number of target LIDs going through it. When there are multiple alternative ports with same MinHop to a LID, the one with less previously assigned ports is selected.
If LMC > 0, more checks are added. Within each group of LIDs assigned to same target port:

a. Use only ports which have same MinHop
b. First prefer the ones that go to different systemImageGuid (then the previous LID of the same LMC group)
c. If none, prefer those which go through another NodeGuid
d. Fall back to the number of paths method (if all go to same node).

Min Hop Algorithm

The Min Hop algorithm is invoked by default if no routing algorithm is specified. It can also be invoked by specifying '-R minhop'.
The Min Hop algorithm is divided into two stages: computation of min-hop tables on every switch and LFT output port assignment. Link subscription is also equalized with the ability to override based on port GUID. The latter is supplied by:

```
-i <equalize-ignore-guids-file>
-ignore-guids <equalize-ignore-guids-file>
```

This option provides the means to define a set of ports (by GUIDs) that will be ignored by the link load equalization algorithm.
LMC awareness routes based on a (remote) system or on a switch basis.

UPDN Algorithm

The UPDN algorithm is designed to prevent deadlocks from occurring in loops of the subnet. A loop-deadlock is a situation in which it is no longer possible to send data between any two hosts connected through the loop. As such, the UPDN routing algorithm should be sent if the subnet is not a pure Fat Tree, and one of its loops may experience a deadlock (due, for example, to high pressure).
The UPDN algorithm is based on the following main stages:

1. Auto-detect root nodes - based on the CA hop length from any switch in the subnet, a statistical histogram is built for each switch (hop num vs the number of occurrences). If the histogram reflects a specific column (higher than others) for a certain node, then it is marked as a root node. Since the algorithm is statistical, it may not find any root nodes. The list of the root nodes found by this auto-detect stage is used by the ranking process stage.

   > ⚠ The user can override the node list manually.

   > ⚠ If this stage cannot find any root nodes, and the user did not specify a GUID list file, OpenSM defaults back to the Min Hop routing algorithm.

2. Ranking process - All root switch nodes (found in stage 1) are assigned a rank of 0. Using the BFS algorithm, the rest of the switch nodes in the subnet are ranked incrementally. This ranking aids in the process of enforcing rules that ensure loop-free paths.
3. Min Hop Table setting - after ranking is done, a BFS algorithm is run from each (CA or switch) node in the subnet. During the BFS process, the FDB table of each switch node traversed by

BFS is updated, in reference to the starting node, based on the ranking rules and GUID values.

At the end of the process, the updated FDB tables ensure loop-free paths through the subnet.

UPDN Algorithm Usage

Activation through OpenSM:

- Use '-R updn' option (instead of old '-u') to activate the UPDN algorithm.
- Use '-a <root_guid_file>' for adding an UPDN GUID file that contains the root nodes for ranking. If the `-a' option is not used, OpenSM uses its auto-detect root nodes algorithm.

Notes on the GUID list file:

- A valid GUID file specifies one GUID in each line. Lines with an invalid format will be discarded
- The user should specify the root switch GUIDs

Fat-tree Routing Algorithm

The fat-tree algorithm optimizes routing for "shift" communication pattern. It should be chosen if a subnet is a symmetrical or almost symmetrical fat-tree of various types. It supports not just K- ary-N-Trees, by handling for non-constant K, cases where not all leafs (CAs) are present, any Constant Bisectional Ratio (CBB)ratio. As in UPDN, fat-tree also prevents credit-loop-dead- locks.
If the root GUID file is not provided ('-a' or '-root_guid_file' options), the topology has to be pure fat-tree that complies with the following rules:

- Tree rank should be between two and eight (inclusively)
- Switches of the same rank should have the same number of UP-going port groups, unless they are root switches, in which case the shouldn't have UP-going ports at all.
  Note: Ports that are connected to the same remote switch are referenced as ‘port group’.
- Switches of the same rank should have the same number of DOWN-going port groups, unless they are leaf switches.
- Switches of the same rank should have the same number of ports in each UP-going port group.
- Switches of the same rank should have the same number of ports in each DOWN-going port group.
- All the CAs have to be at the same tree level (rank).

If the root GUID file is provided, the topology does not have to be pure fat-tree, and it should only comply with the following rules:

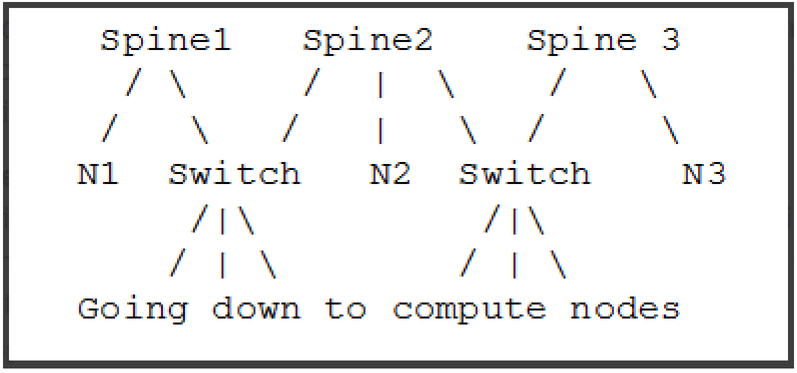
- Tree rank should be between two and eight (inclusively)
- All the Compute Nodes have to be at the same tree level (rank). Note that non-compute node CAs are allowed here to be at different tree ranks.
  Note: List of compute nodes (CNs) can be specified using ‘-u’ or ‘--cn_guid_file’ OpenSM options.

Topologies that do not comply cause a fallback to min-hop routing. Note that this can also occur on link failures which cause the topology to no longer be a "pure" fat-tree.
Note that although fat-tree algorithm supports trees with non-integer CBB ratio, the routing will not be as balanced as in case of integer CBB ratio. In addition to this, although the algorithm allows leaf

switches to have any number of CAs, the closer the tree is to be fully populated, the more effective the "shift" communication pattern will be. In general, even if the root list is provided, the closer the topology to a pure and symmetrical fat-tree, the more optimal the routing will be.
The algorithm also dumps the compute node ordering file (opensm-ftree-ca-order.dump) in the same directory where the OpenSM log resides. This ordering file provides the CN order that may be used to create efficient communication pattern, that will match the routing tables.

Routing between non-CN Nodes

The use of the io_guid_file option allows non-CN nodes to be located on different levels in the fat tree. In such case, it is not guaranteed that the Fat Tree algorithm will route between two non-CN nodes. In the scheme below, N1, N2 , and N3 are non-CN nodes. Although all the CN have routes to and from them, there will not necessarily be a route between N1,N2 and N3. Such routes would require to use at least one of the switches the wrong way around.



```
   Spine1    Spine2     Spine 3
    / \      /  |  \     /    \
   /   \    /   |   \   /      \
  N1   Switch   N2  Switch     N3
        /|\            /|\
       / | \          / | \
  Going down to compute nodes
```

To solve this problem, a list of non-CN nodes can be specified by \'-G\' or \'--io_guid_file\' option. These nodes will be allowed to use switches the wrong way around a specific number of times (specified by \'-H\' or \'--max_reverse_hops\'. With the proper max_reverse_hops and io_guid_file values, you can ensure full connectivity in the Fat Tree. In the scheme above, with a max_reverse_hop of 1, routes will be instantiated between N1<->N2 and N2<->N3. With a max_reverse_hops value of 2, N1,N2 and N3 will all have routes between them.

Using max_reverse_hops creates routes that use the switch in a counter-stream way. This option should never be used to connect nodes with high bandwidth traffic between them! It should only be used to allow connectivity for HA purposes or similar. Also having routes the other way around can cause credit loops.

Activation through OpenSM

Use '-R ftree' option to activate the fat-tree algorithm.

LMC > 0 is not supported by fat-tree routing. If this is specified, the default routing algorithm is invoked instead.

DOR Routing Algorithm

The Dimension Order Routing algorithm is based on the Min Hop algorithm and so uses shortest paths. Instead of spreading traffic out across different paths with the same shortest distance, it chooses among the available shortest paths based on an ordering of dimensions. Each port must be consistently cabled to represent a hypercube dimension or a mesh dimension. Paths are grown from a destination back to a source using the lowest dimension (port) of available paths at each step. This provides the ordering necessary to avoid deadlock. When there are multiple links between any

two switches, they still represent only one dimension and traffic is balanced across them unless port equalization is turned off. In the case of hypercubes, the same port must be used throughout the fabric to represent the hypercube dimension and match on both ends of the cable. In the case of meshes, the dimension should consistently use the same pair of ports, one port on one end of the cable, and the other port on the other end, continuing along the mesh dimension.
Use '-R dor' option to activate the DOR algorithm.

Torus-2QoS Routing Algorithm

Torus-2QoS is a routing algorithm designed for large-scale 2D/3D torus fabrics. The torus-2QoS routing engine can provide the following functionality on a 2D/3D torus:

- Free of credit loops routing
- Two levels of QoS, assuming switches support 8 data VLs
- Ability to route around a single failed switch, and/or multiple failed links, without:
    - introducing credit loops
    - changing path SL values
- Very short run times, with good scaling properties as fabric size increases

Unicast Routing

Torus-2 QoS is a DOR-based algorithm that avoids deadlocks that would otherwise occur in a torus using the concept of a dateline for each torus dimension. It encodes into a path SL which datelines the path crosses as follows:

```
sl = 0;
for (d = 0; d < torus_dimensions; d++)
/* path_crosses_dateline(d) returns 0 or 1 */
sl |= path_crosses_dateline(d) << d;
```

For a 3D torus, that leaves one SL bit free, which torus-2 QoS uses to implement two QoS levels. Torus-2 QoS also makes use of the output port dependence of switch SL2VL maps to encode into one VL bit the information encoded in three SL bits. It computes in which torus coordinate direc- tion each inter-switch link "points", and writes SL2VL maps for such ports as follows:

```
for (sl = 0; sl < 16; sl ++)
/* cdir(port) reports which torus coordinate direction a switch port
* "points" in, and returns 0, 1, or 2 */
sl2vl(iport,oport,sl) = 0x1 & (sl >> cdir(oport));
```

Thus, on a pristine 3D torus, i.e., in the absence of failed fabric switches, torus-2 QoS consumes 8 SL values (SL bits 0-2) and 2 VL values (VL bit 0) per QoS level to provide deadlock-free routing on a 3D torus. Torus-2 QoS routes around link failure by "taking the long way around" any 1D ring interrupted by a link failure. For example, consider the 2D 6x5 torus below, where switches are denoted by [+a-zA-Z]:

```
       |    |    |    |    |    |
  4  --+----+----+----+----+----+--
       |    |    |    |    |    |
  3  --+----+----+----D----+----+--
       |    |    |    |    |    |
  2  --+----+----I----r----+----+--
       |    |    |    |    |    |
  1  --m----S----n----T----o----p--
       |    |    |    |    |    |
y=0  --+----+----+----+----+----+--
       |    |    |    |    |    |

     x=0    1    2    3    4    5
```

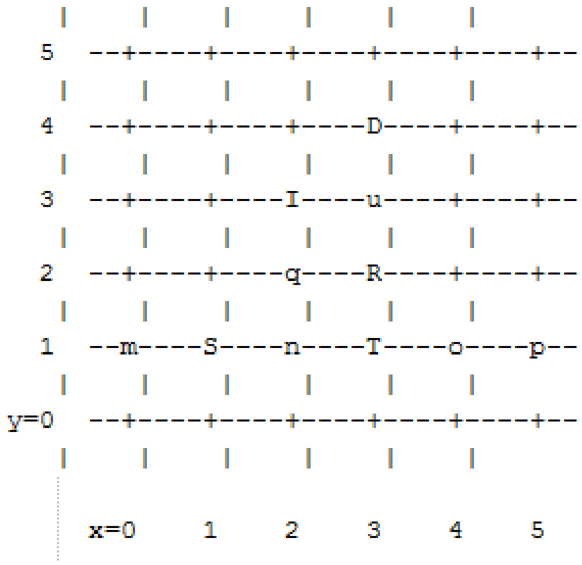
For a pristine fabric the path from S to D would be S-n-T-r-D. In the event that either link S-n or n-T has failed, torus-2QoS would use the path S-m-p-o-T-r-D.

Note that it can do this without changing the path SL value; once the 1D ring m-S-n-T-o-p-m has been broken by failure, path segments using it cannot contribute to deadlock, and the x-direction dateline (between, say, x=5 and x=0) can be ignored for path segments on that ring. One result of this is that torus-2QoS can route around many simultaneous link failures, as long as no 1D ring is broken into disjoint segments. For example, if links n-T and T-o have both failed, that ring has been broken into two disjoint segments, T and o-p-m-S-n. Torus-2QoS checks for such issues, reports if they are found, and refuses to route such fabrics.

Note that in the case where there are multiple parallel links between a pair of switches, torus-2QoS will allocate routes across such links in a round-robin fashion, based on ports at the path destination switch that are active and not used for inter-switch links. Should a link that is one of several such parallel links fail, routes are redistributed across the remaining links. When the last of such a set of parallel links fails, traffic is rerouted as described above.

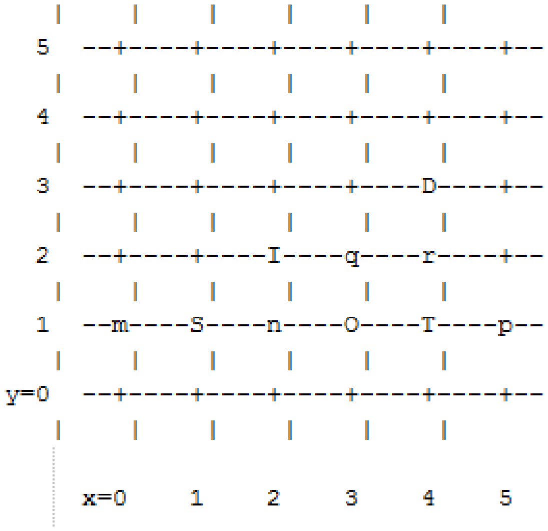
Handling a failed switch under DOR requires introducing into a path at least one turn that would be otherwise "illegal", i.e. not allowed by DOR rules. Torus-2QoS will introduce such a turn as close as possible to the failed switch in order to route around it. n the above example, suppose switch T has failed, and consider the path from S to D. Torus-2QoS will produce the path S-n-I-r-D, rather than the S-n-T-r-D path for a pristine torus, by introducing an early turn at n. Normal DOR rules will cause traffic arriving at switch I to be forwarded to switch r; for traffic arriving from I due to the "early" turn at n, this will generate an "illegal" turn at I.

Torus-2QoS will also use the input port dependence of SL2VL maps to set VL bit 1 (which would be otherwise unused) for y-x, z-x, and z-y turns, i.e., those turns that are illegal under DOR. This causes the first hop after any such turn to use a separate set of VL values, and prevents deadlock in the presence of a single failed switch. For any given path, only the hops after a turn that is illegal under DOR can contribute to a credit loop that leads to deadlock. So in the example above with failed switch T, the location of the illegal turn at I in the path from S to D requires that any credit loop caused by that turn must encircle the failed switch at T. Thus the second and later hops after the illegal turn at I (i.e., hop r-D) cannot contribute to a credit loop because they cannot be used to construct a loop encircling T. The hop I-r uses a separate VL, so it cannot contribute to a credit loop encircling T. Extending this argument shows that in addition to being capable of routing around a single switch failure without introducing deadlock, torus-2QoS can also route around multiple failed switches on the condition they are adjacent in the last dimension routed by DOR. For example, consider the following case on a 6x6 2D torus:



```
      |    |    |    |    |    |
 5   --+----+----+----+----+----+--
      |    |    |    |    |    |
 4   --+----+----+----D----+----+--
      |    |    |    |    |    |
 3   --+----+----I----u----+----+--
      |    |    |    |    |    |
 2   --+----+----q----R----+----+--
      |    |    |    |    |    |
 1   --m----S----n----T----o----p--
      |    |    |    |    |    |
y=0  --+----+----+----+----+----+--
      |    |    |    |    |    |

     x=0   1    2    3    4    5
```

Suppose switches T and R have failed, and consider the path from S to D. Torus-2QoS will generate the path S-n-q-I-u-D, with an illegal turn at switch I, and with hop I-u using a VL with bit 1 set. As a further example, consider a case that torus-2QoS cannot route without deadlock: two failed switches adjacent in a dimension that is not the last dimension routed by DOR; here the failed switches are O and T:



```
      |    |    |    |    |    |
 5   --+----+----+----+----+----+--
      |    |    |    |    |    |
 4   --+----+----+----+----+----+--
      |    |    |    |    |    |
 3   --+----+----+----+----D----+--
      |    |    |    |    |    |
 2   --+----+----I----q----r----+--
      |    |    |    |    |    |
 1   --m----S----n----O----T----p--
      |    |    |    |    |    |
y=0  --+----+----+----+----+----+--
      |    |    |    |    |    |

     x=0   1    2    3    4    5
```

In a pristine fabric, torus-2QoS would generate the path from S to D as S-n-O-T-r-D. With failed switches O and T, torus-2QoS will generate the path S-n-I-q-r-D, with an illegal turn at switch I, and with hop I-q using a VL with bit 1 set. In contrast to the earlier examples, the second hop after the illegal turn, q-r, can be used to construct a credit loop encircling the failed switches.

Multicast Routing

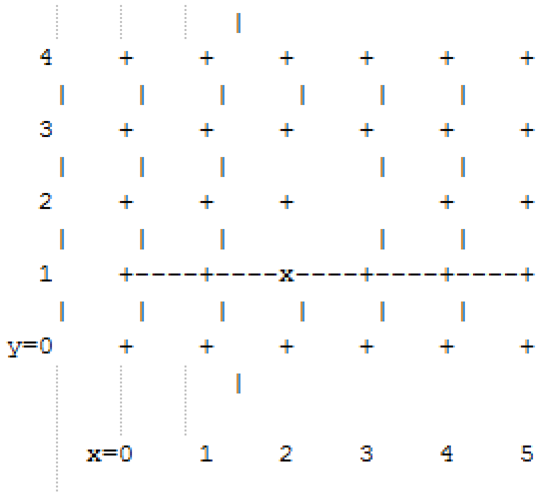
Since torus-2QoS uses all four available SL bits, and the three data VL bits that are typically available in current switches, there is no way to use SL/VL values to separate multicast traffic from unicast traffic. Thus, torus-2QoS must generate multicast routing such that credit loops cannot arise from a combination of multicast and unicast path segments. It turns out that it is possible to construct spanning trees for multicast routing that have that property. For the 2D 6x5 torus example above, here is the full-fabric spanning tree that torus-2QoS will construct, where "x" is the root switch and each "+" is a non-root switch:

```
  4     +     +     +     +     +     +
        |     |     |     |     |     |
  3     +     +     +     +     +     +
        |     |     |     |     |     |
  2     +-----+-----+-----x-----+-----+
        |     |     |     |     |     |
  1     +     +     +     +     +     +
        |     |     |     |     |     |
y=0     +     +     +     +     +     +

      x=0     1     2     3     4     5
```

For multicast traffic routed from root to tip, every turn in the above spanning tree is a legal DOR turn. For traffic routed from tip to root, and some traffic routed through the root, turns are not legal DOR turns. However, to construct a credit loop, the union of multicast routing on this spanning tree with DOR unicast routing can only provide 3 of the 4 turns needed for the loop. In addition, if none of the above spanning tree branches crosses a dateline used for unicast credit loop avoidance on a torus, and if multicast traffic is confined to SL 0 or SL 8 (recall that torus-2QoS uses SL bit 3 to differentiate QoS level), then multicast traffic also cannot contribute to the "ring" credit loops that are otherwise possible in a torus. Torus-2QoS uses these ideas to create a master spanning tree. Every multicast group spanning tree will be constructed as a subset of the master tree, with the same root as the master tree. Such multicast group spanning trees will in general not be optimal for groups which are a subset of the full fabric. However, this compromise must be made to enable support for two QoS levels on a torus while preventing credit loops. In the presence of link or switch failures that result in a fabric for which torus-2QoS can generate credit-loop-free unicast routes, it is also possible to generate a master spanning tree for multicast that retains the required properties. For example, consider that same 2D 6x5 torus, with the link from (2,2) to (3,2) failed. Torus-2QoS will generate the following master spanning tree:

```
  4     +     +     +     +     +     +
        |     |     |     |     |     |
  3     +     +     +     +     +     +
        |     |     |     |     |     |
  2   --+-----+-----+     x-----+-----+--
        |     |     |     |     |     |
  1     +     +     +     +     +     +
        |     |     |     |     |     |
y=0     +     +     +     +     +     +

      x=0     1     2     3     4     5
```

Two things are notable about this master spanning tree. First, assuming the x dateline was between x=5 and x=0, this spanning tree has a branch that crosses the dateline. However, just as for unicast, crossing a dateline on a 1D ring (here, the ring for y=2) that is broken by a failure cannot contribute to a torus credit loop. Second, this spanning tree is no longer optimal even for multicast groups that encompass the entire fabric. That, unfortunately, is a compromise that must be made to retain the other desirable properties of torus-2QoS routing. In the event that a single switch fails, torus-2QoS will generate a master spanning tree that has no "extra" turns by appropriately selecting a root switch. In the 2D 6x5 torus example, assume now that the switch at (3,2) (i.e., the root for a pristine fabric), fails. Torus-2QoS will generate the following master spanning tree for that case:



```
              |
  4    +    +    +    +    +    +
   |    |    |    |    |    |
  3    +    +    +    +    +    +
   |    |    |         |    |
  2    +    +    +         +    +
   |    |    |         |    |
  1    +----+----x----+----+----+
   |    |    |    |    |    |
y=0    +    +    +    +    +    +
              |

     x=0    1    2    3    4    5
```

Assuming the dateline was between y=4 and y=0, this spanning tree has a branch that crosses a dateline. However, this cannot contribute to credit loops as it occurs on a 1D ring (the ring for x=3) that is broken by failure, as in the above example.

Torus Topology Discovery

The algorithm used by torus-2QoS to construct the torus topology from the undirected graph representing the fabric requires that the radix of each dimension be configured via torus-2QoS.conf. It also requires that the torus topology be "seeded"; for a 3D torus this requires configuring four switches that define the three coordinate directions of the torus. Given this starting information, the algorithm is to examine the cube formed by the eight switch locations bounded by the corners (x,y,z) and (x+1,y+1,z+1). Based on switches already placed into the torus topology at some of these locations, the algorithm examines 4-loops of inter-switch links to find the one that is consistent with a face of the cube of switch locations and adds its switches to the discovered topology in the correct locations.

Because the algorithm is based on examining the topology of 4-loops of links, a torus with one or more radix-4 dimensions requires extra initial seed configuration. See torus-2QoS.conf(5) for details. Torus-2QoS will detect and report when it has an insufficient configuration for a torus with radix-4 dimensions.

In the event the torus is significantly degraded, i.e., there are many missing switches or links, it may happen that torus-2QoS is unable to place into the torus some switches and/or links that were discovered in the fabric, and will generate a warning in that case. A similar condition occurs if torus-2QoS is misconfigured, i.e., the radix of a torus dimension as configured does not match the radix of that torus dimension as wired, and many switches/links in the fabric will not be placed into the torus.

Quality Of Service Configuration

OpenSM will not program switches and channel adapters with SL2VL maps or VL arbitration configuration unless it is invoked with -Q. Since torus-2QoS depends on such functionality for correct operation, always invoke OpenSM with -Q when torus-2QoS is in the list of routing engines. Any quality of service configuration method supported by OpenSM will work with torus-2QoS, subject to the following limitations and considerations. For all routing engines supported by OpenSM except torus-2QoS, there is a one-to-one correspondence between QoS level and SL. Torus-2QoS can only

support two quality of service levels, so only the high-order bit of any SL value used for unicast QoS configuration will be honored by torus-2QoS. For multicast QoS configuration, only SL values 0 and 8 should be used with torus-2QoS.
Since SL to VL map configuration must be under the complete control of torus-2QoS, any configuration via qos_sl2vl, qos_swe_sl2vl, etc., must and will be ignored, and a warning will be generated. Torus-2QoS uses VL values 0-3 to implement one of its supported QoS levels, and VL values 4-7 to implement the other. Hard-to-diagnose application issues may arise if traffic is not delivered fairly across each of these two VL ranges. Torus-2QoS will detect and warn if VL arbitration is configured unfairly across VLs in the range 0-3, and also in the range 4-7. Note that the default OpenSM VL arbitration configuration does not meet this constraint, so all torus-2QoS users should configure VL arbitration via qos_vlarb_high, qos_vlarb_low, etc.

Operational Considerations
Any routing algorithm for a torus IB fabric must employ path SL values to avoid credit loops. As a result, all applications run over such fabrics must perform a path record query to obtain the correct path SL for connection setup. Applications that use rdma_cm for connection setup will automatically meet this requirement.
If a change in fabric topology causes changes in path SL values required to route without credit loops, in general, all applications would need to repath to avoid message deadlock. Since torus-2QoS has the ability to reroute after a single switch failure without changing path SL values, repathing by running applications is not required when the fabric is routed with torus-2QoS. Torus-2QoS can provide unchanging path SL values in the presence of subnet manager failover provided that all OpenSM instances have the same idea of dateline location. See torus- 2QoS.conf(5) for details. Torus-2QoS will detect configurations of failed switches and links that prevent routing that is free of credit loops and will log warnings and refuse to route. If "no_fall- back" was configured in the list of OpenSM routing engines, then no other routing engine will attempt to route the fabric. In that case, all paths that do not transit the failed components will continue to work, and the subset of paths that are still operational will continue to remain free of credit loops. OpenSM will continue to attempt to route the fabric after every sweep interval and after any change (such as a link up) in the fabric topology. When the fabric components are repaired, full functionality will be restored. In the event OpenSM was configured to allow some other engine to route the fabric if torus-2QoS fails, then credit loops and message deadlock are likely if torus-2QoS had previously routed the fabric successfully. Even if the other engine is capable of routing a torus without credit loops, applications that built connections with path SL values granted under torus-2QoS will likely experience message deadlock under routing generated by a different engine, unless they repath. To verify that a torus fabric is routed free of credit loops, use `ibdmchk` to analyze data collected via ibdiagnet `–`vlr.

Torus-2QoS Configuration File Syntax

The file torus-2QoS.conf contains configuration information that is specific to the OpenSM routing engine torus-2QoS. Blank lines and lines where the first non-whitespace character is "#" are ignored. A token is any contiguous group of non-whitespace characters. Any tokens on a line following the recognized configuration tokens described below are ignored.

```
[torus|mesh] x_radix[m|M|t|T] y_radix[m|M|t|T] z_radix[m|M|t|T]
```

Either torus or mesh must be the first keyword in the configuration and sets the topology that torus-2QoS will try to construct. A 2D topology can be configured by specifying one of x_radix, y_radix, or z_radix as 1. An individual dimension can be configured as mesh (open) or torus (looped) by suffixing its radix specification with one of m, M, t, or T. Thus, "mesh 3T 4 5" and "torus 3 4M 5M"

both specify the same topology.
Note that although torus-2QoS can route mesh fabrics, its ability to route around failed components is severely compromised on such fabrics. A failed fabric components very likely to cause a disjoint ring; see UNICAST ROUTING in torus-2QoS(8).

```
xp_link sw0_GUID sw1_GUID
yp_link sw0_GUID sw1_GUID
zp_link sw0_GUID sw1_GUID
xm_link sw0_GUID sw1_GUID
ym_link sw0_GUID sw1_GUID
zm_link sw0_GUID sw1_GUID
```

These keywords are used to seed the torus/mesh topology. For example, "xp_link 0x2000 0x2001" specifies that a link from the switch with node GUID 0x2000 to the switch with node GUID 0x2001 would point in the positive x direction, while "xm_link 0x2000 0x2001" specifies that a link from the switch with node GUID 0x2000 to the switch with node GUID 0x2001 would point in the negative x direction. All the link keywords for a given seed must specify the same "from" switch.
In general, it is not necessary to configure both the positive and negative directions for a given coordinate; either is sufficient. However, the algorithm used for topology discovery needs extra information for torus dimensions of radix four (see TOPOLOGY DISCOVERY in torus-2QoS(8)). For such cases, both the positive and negative coordinate directions must be specified.
Based on the topology specified via the torus/mesh keyword, torus-2QoS will detect and log when it has insufficient seed configuration.

```
GUIDx_dateline position
y_dateline position
z_dateline position
```

In order for torus-2QoS to provide the guarantee that path SL values do not change under any conditions for which it can still route the fabric, its idea of dateline position must not change relative to physical switch locations. The dateline keywords provide the means to configure such behavior.

The dateline for a torus dimension is always between the switch with coordinate 0 and the switch with coordinate radix-1 for that dimension. By default, the common switch in a torus seed is taken as the origin of the coordinate system used to describe switch location. The position parameter for a dateline keyword moves the origin (and hence the dateline) the specified amount relative to the common switch in a torus seed.

```
next_seed
```

If any of the switches used to specify a seed were to fail torus-2QoS would be unable to complete topology discovery successfully. The next_seed keyword specifies that the following link and dateline keywords apply to a new seed specification.
For maximum resiliency, no seed specification should share a switch with any other seed specification. Multiple seed specifications should use dateline configuration to ensure that torus-2QoS can grant path SL values that are constant, regardless of which seed was used to initiate topology discovery.
portgroup_max_ports max_ports - This keyword specifies the maximum number of parallel inter-switch links, and also the maximum number of host ports per switch, that torus-2QoS can accommodate. The default value is 16. Torus-2QoS will log an error message during topology discovery if this parameter needs to be increased. If this keyword appears multiple times, the last instance prevails.
port_order p1 p2 p3 ... - This keyword specifies the order in which CA ports on a destination switch

are visited when computing routes. When the fabric contains switches connected with multiple parallel links, routes are distributed in a round-robin fashion across such links, and so changing the order that CA ports are visited changes the distribution of routes across such links. This may be advantageous for some specific traffic patterns.
The default is to visit CA ports in increasing port order on destination switches. Duplicate values in the list will be ignored.

Example:

```
# Look for a 2D (since x radix is one) 4x5 torus.
torus 1 4 5
# y is radix-4 torus dimension, need both
# ym_link and yp_link configuration.
yp_link 0x200000 0x200005 # sw @ y=0,z=0 -> sw @ y=1,z=0
ym_link 0x200000 0x20000f # sw @ y=0,z=0 -> sw @ y=3,z=0
# z is not radix-4 torus dimension, only need one of
# zm_link or zp_link configuration.
zp_link 0x200000 0x200001 # sw @ y=0,z=0 -> sw @ y=0,z=1
next_seed
yp_link 0x20000b 0x200010 # sw @ y=2,z=1 -> sw @ y=3,z=1
ym_link 0x20000b 0x200006 # sw @ y=2,z=1 -> sw @ y=1,z=1
zp_link 0x20000b 0x20000c # sw @ y=2,z=1 -> sw @ y=2,z=2
y_dateline -2 # Move the dateline for this seed
z_dateline -1 # back to its original position.
# If OpenSM failover is configured, for maximum resiliency
# one instance should run on a host attached to a switch
# from the first seed, and another instance should run
# on a host attached to a switch from the second seed.
# Both instances should use this torus-2QoS.conf to ensure
# path SL values do not change in the event of SM failover.
# port_order defines the order on which the ports would be
# chosen for routing.
port_order 7 10 8 11 9 12 25 28 26 29 27 30
```

Routing Chains

The routing chains feature is offering a solution that enables one to configure different parts of the fabric and define a different routing engine to route each of them. The routings are done in a sequence (hence the name "chains") and any node in the fabric that is configured in more than one part is left with the routing updated by the last routing engine it was a part of.

Configuring Routing Chains

To configure routing chains:

1. Define the port groups.
2. Define topologies based on previously defined port groups.
3. Define configuration files for each routing engine.
4. Define routing engine chains over previously defined topologies and configuration files.

Defining Port Groups

The basic idea behind the port groups is the ability to divide the fabric into sub-groups and give each group an identifier that can be used to relate to all nodes in this group. The port groups is a separate feature from the routing chains but is a mandatory prerequisite for it. In addition, it is used to define the participants in each of the routing algorithms.

Defining a Port Group Policy File

In order to define a port group policy file, set the parameter 'pgrp_policy_file' in the OpenSM configuration file.
pgrp_policy_file /etc/opensm/conf/port_groups_policy_file

Configuring a Port Group Policy

The port groups policy file details the port groups in the fabric. The policy file should be composed of one or more paragraphs that define a group. Each paragraph should begin with the line 'port-group' and end with the line 'end-port-group'.
For example:

```
port-group
…port group qualifiers…
end-port-group
```

Port Group Qualifiers

> ⚠ Unlike the port group's beginning and end which do not require a colon, all qualifiers must end with a colon (':'). Also - a colon is a predefined mark that must not be used inside qualifier values. The inclusion of a colon in the name or the use of a port group will result in the policy's failure.

Rule Qualifier

Parameter	Description	Example
`name`	Each group must have a name. Without a name qualifier, the policy fails.	`name: grp1`
`use`	'use' is an optional qualifier that one can define in order to describe the usage of this port group (if undefined, an empty string is used as a default).	`use: first port group`

There are several qualifiers used to describe a rule that determines which ports will be added to the group. Each port group may include one or more rules out of the rules described in the below table (at least one rule must be defined for each port group).

Parameter	Description	Example
`guid list`	Comma separated list of GUIDs to include in the group. If no specific physical ports were configured, all physical ports of the guid are chosen. However, for each guid, one can detail specific physical ports to be included in the group. This can be done using the following syntax: • Specify a specific port in a guid to be chosen port-guid: 0x283@3 • Specify a specific list of ports in a guid to be chosen port-guid: 0x286@1/5/7 • Specify a specific range of ports in a guid to be chosen port-guid: 0x289@2-5 • Specify a list of specific ports and ports ranges in a guid to be chosen port-guid: 0x289@2-5/7/9-13/18 • Complex rule port-guid: 0x283@5-8/12/14, 0x286, 0x289/6/ 8/12	`port-guid: 0x283, 0x286, 0x289`

Parameter	Description	Example
`port guid range`	It is possible to configure a range of guids to be chosen to the group. However, while using the range qualifier, it is impossible to detail specific physical ports. Note: A list of ranges cannot be specified. The below example is invalid and will cause the policy to fail: port-guid-range: 0x283-0x289, 0x290- 0x295	`port-guid-range: 0x283-0x289`
`port name`	One can configure a list of hostnames as a rule. Hosts with a node description that is built out of these hostnames will be chosen. Since the node description contains the network card index as well, one might also specify a network card index and a physical port to be chosen. For example, the given configuration will cause only physical port 2 of a host with the node description 'kuku HCA-1' to be chosen. port and hca_idx parameters are optional. If the port is unspecified, all physical ports are chosen. If hca_idx is unspecified, all card numbers are chosen. Specifying a hostname is mandatory. One can configure a list of hostname/ port/hca_idx sets in the same qualifier as follows: port-name: hostname=kuku; port=2; hca_idx=1 , hostname=host1; port=3, hostname=host2 Note: port-name qualifier is not relevant for switches, but for HCA's only.	`port-name: host-name=kuku; port=2; hca_idx=1`
`port regexp`	One can define a regular expression so that only nodes with a matching node description will be chosen to the group. Note: This example shows how to choose nodes which their node description starts with 'SW'.	`port-regexp: SW`
	It is possible to specify one physical port to be chosen for matching nodes (there is no option to define a list or a range of ports). The given example will cause only nodes that match physical port 3 to be added to the group.	`port-regexp: SW:3`
`union rule`	It is possible to define a rule that unites two different port groups. This means that all ports from both groups will be included in the united group.	`union-rule: grp1, grp2`
`subtract rule`	One can define a rule that subtracts one port group from another. The given rule, for example, will cause all the ports which are a part of grp1, but not included in grp2, to be chosen. In subtraction (unlike union), the order does matter, since the purpose is to subtract the second group from the first one. There is no option to define more than two groups for union/subtraction. However, one can unite/subtract groups which are a union or a subtraction themselves, as shown in the port groups policy file example.	`subtract-rule: grp1, grp2`

Predefined Port Groups

There are 3 predefined, automatically created port groups that are available for use, yet cannot be defined in the policy file (if a group in the policy is configured with the name of one of these predefined groups, the policy fails) -

- ALL - a group that includes all nodes in the fabric
- ALL_SWITCHES - a group that includes all switches in the fabric
- ALL_CAS - a group that includes all HCAs in the fabric
- ALL_ROUTERS - a group that includes all routers in the fabric (supported in OpenSM starting from v4.9.0)

Port Groups Policy Examples

```
port-group
name: grp3
use: Subtract of groups grp1 and grp2
subtract-rule: grp1, grp2
end-port-group

port-group
name: grp1
port-guid: 0x281, 0x282, 0x283
end-port-group

port-group
name: grp2
port-guid-range: 0x282-0x286
port-name: hostname=server1 port=1
end-port-group

port-group
name: grp4
port-name: hostname=kika port=1 hca_idx=1
end-port-group

port-group
name: grp3
union-rule: grp3, grp4
end-port-group
```

Defining a Topologies Policy File

In order to define a topology policy file, set the parameter 'topo_policy_file' in the OpenSM configuration file.

```
topo_policy_file /etc/opensm/conf/topo_policy_file.cfg
```

Configuring a Topology Policy

The topologies policy file details a list of topologies. The policy file should be composed of one or more paragraphs which define a topology. Each paragraph should begin with the line 'topol- ogy' and end with the line 'end-topology'.
For example:

```
topology
…topology qualifiers…
end-topology
```

Topology Qualifiers

Unlike topology and end-topology which do not require a colon, all qualifiers must end with a colon (':'). Also - a colon is a predefined mark that must not be used inside qualifier values. An inclusion of a column in the qualifier values will result in the policy's failure.

All topology qualifiers are mandatory. Absence of any of the below qualifiers will cause the policy parsing to fail.

Topology Qualifiers

Parameter	Description	Example
`id`	Topology ID. Legal Values - any positive value. Must be unique.	`id: 1`
`sw-grp`	Name of the port group that includes all switches and switch ports to be used in this topology.	`sw-grp: ys_switches`
`hca-grp`	Name of the port group that includes all HCA's to be used in this topology.	`hca-grp: ys_hosts`

Configuration File per Routing Engine

Each engine in the routing chain can be provided by its own configuration file. Routing engine configuration file is the fraction of parameters defined in the main OpenSM configuration file. Some rules should be applied when defining a particular configuration file for a routing engine:

- Parameters that are not specified in specific routing engine configuration file are inherited from the main OpenSM configuration file.
- The following configuration parameters are taking effect only in the main OpenSM configuration file:
  - qos and qos_* settings like (vl_arb, sl2vl, etc.)
  - lmc
  - routing_engine

Defining a Routing Chain Policy File

In order to define a port group policy file, set the parameter 'rch_policy_file' in the OpenSM configuration file.

```
rch_policy_file /etc/opensm/conf/chains_policy_file
```

First Routing Engine in the Chain

The first unicast engine in a routing chain must include all switches and HCAs in the fabric (topology id must be 0). The path-bit parameter value is path-bit 0 and it cannot be changed.

Configuring a Routing Chains Policy

The routing chains policy file details the routing engines (and their fallback engines) used for the fabric's routing. The policy file should be composed of one or more paragraphs which defines an engine (or a fallback engine). Each paragraph should begin with the line 'unicast-step' and end with the line 'end-unicast-step'.
For example:

```
unicast-step
…routing engine qualifiers…
end-unicast-step
```

Routing Engine Qualifiers

⚠ Unlike unicast-step and end-unicast-step which do not require a colon, all qualifiers must end with a colon (':'). Also - a colon is a predefined mark that must not be used inside qualifier values. An inclusion of a colon in the qualifier values will result in the policy's failure.

Parameter	Description	Example
`id`	'id' is mandatory. Without an ID qualifier for each engine, the policy fails. • Legal values - size_t value (0 is illegal). • The engines in the policy chain are set according to an ascending id order, so it is highly crucial to verify that the id that is given to the engines match the order in which you would like the engines to be set.	`is: 1`

Parameter	Description	Example
`engine`	This is a mandatory qualifier that describes the routing algorithm used within this unicast step. Currently, on the first phase of routing chains, legal values are minhop/ftree/updn.	`engine: minhop`
`use`	This is an optional qualifier that enables one to describe the usage of this unicast step. If undefined, an empty string is used as a default.	`use: ftree routing for for yellow stone nodes`
`config`	This is an optional qualifier that enables one to define a separate OpenSM config file for a specific unicast step. If undefined, all parameters are taken from main OpenSM configuration file.	`config: /etc/config/opensm2.cfg`
`topology`	Define the topology that this engine uses. • Legal value – id of an existing topology that is defined in topologies policy (or zero that represents the entire fabric and not a specific topology). • Default value – If unspecified, a routing engine will relate to the entire fabric (as if topology zero was defined). • Notice: The first routing engine (the engine with the lowest id) MUST be configured with topology: 0 (entire fabric) or else, the routing chain parser will fail.	`topology: 1`
`fallback-to`	This is an optional qualifier that enables one to define the current unicast step as a fallback to another unicast step. This can be done by defining the id of the unicast step that this step is a fallback to. • If undefined, the current unicast step is not a fallback. • If the value of this qualifier is a non-existent engine id, this step will be ignored. • A fallback step is meaningless if the step it is a fallback to did not fail. • It is impossible to define a fallback to a fall- back step (such definition will be ignored)	`-`
`path-bit`	This is an optional qualifier that enables one to define a specific lid offset to be used by the current unicast step. Setting lmc > 0 in main OpenSM configuration file is a prerequisite for assigning specific path-bit for the routing engine. Default value is 0 (if path-bit is not specified)	`Path-bit: 1`

Dump Files per Routing Engine

Each routing engine on the chain will dump its own data files if the appropriate log_flags is set (for instance 0x43).

The files that are dumped by each engine are:

- opensm-lid-matrix.dump
- opensm-lfts.dump
- opensm.fdbs
- opensm-subnet.lst

These files should contain the relevant data for each engine topology.

> ⚠ sl2vl and mcfdbs files are dumped only once for the entire fabric and NOT by every routing engine.

- Each engine concatenates its ID and routing algorithm name in its dump files names, as follows:
    - opensm-lid-matrix.2.minhop.dump
    - opensm.fdbs.3.ftree
    - opensm-subnet.4.updn.lst
- In case that a fallback routing engine is used, both the routing engine that failed and the fallback engine that replaces it, dump their data.
  If, for example, engine 2 runs ftree and it has a fallback engine with 3 as its id that runs minhop, one should expect to find 2 sets of dump files, one for each engine:
    - opensm-lid-matrix.2.ftree.dump
    - opensm-lid-matrix.3.minhop.dump
    - opensm.fdbs.2.ftree
    - opensm.fdbs.3.munhop

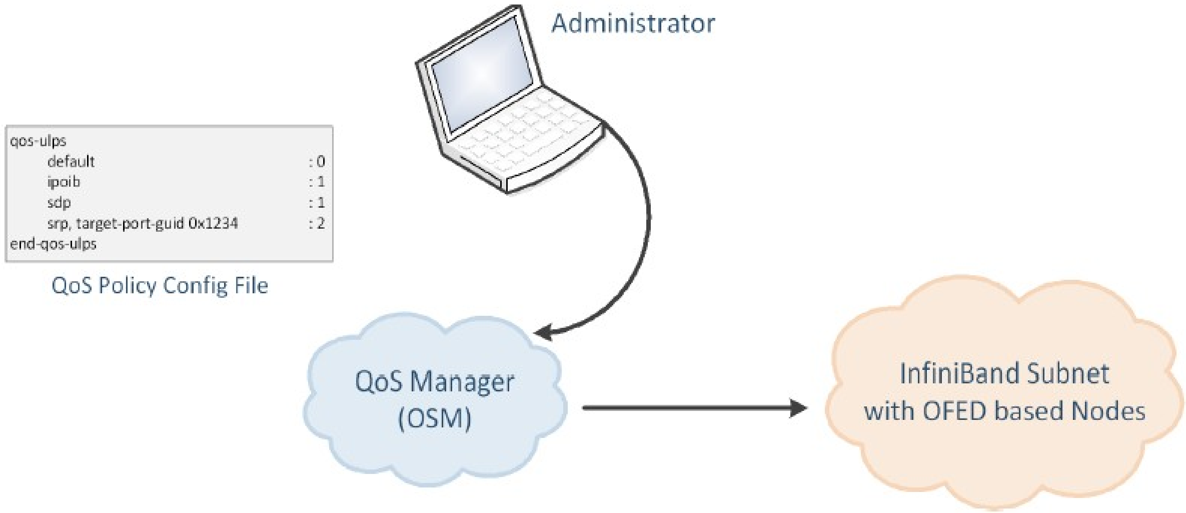
14.6.2.1.2.6 Unicast Routing Cache

Unicast routing cache prevents routing recalculation (which is a heavy task in a large cluster) when no topology change was detected during the heavy sweep, or when the topology change does not require new routing calculation (for example, when one or more CAs/RTRs/leaf switches going down, or one or more of these nodes coming back after being down).

14.6.2.1.2.7 Quality of Service Management in OpenSM

When Quality of Service (QoS) in OpenSM is enabled (using the ‘-Q’ or ‘--qos’ flags), OpenSM looks for a QoS Policy file. During fabric initialization and at every heavy sweep, OpenSM parses the QoS policy file, applies its settings to the discovered fabric elements, and enforces the provided policy on client requests. The overall flow for such requests is as follows:

- The request is matched against the defined matching rules such that the QoS Level definition is found
- Given the QoS Level, a path(s) search is performed with the given restrictions imposed by that level



```
qos-ulps
        default                                  : 0
        ipoib                                    : 1
        sdp                                      : 1
        srp, target-port-guid 0x1234             : 2
end-qos-ulps
```

There are two ways to define QoS policy:

- Advanced – the advanced policy file syntax provides the administrator various ways to match a PathRecord/MultiPathRecord (PR/MPR) request, and to enforce various QoS constraints on the requested PR/MPR
- Simple – the simple policy file syntax enables the administrator to match PR/MPR requests by various ULPs and applications running on top of these ULPs

Advanced QoS Policy File

The QoS policy file has the following sections:

1. Port Groups (denoted by port-groups) - this section defines zero or more port groups that can be referred later by matching rules (see below). Port group lists ports by:
   - Port GUID
   - Port name, which is a combination of NodeDescription and IB port number
   - PKey, which means that all the ports in the subnet that belong to partition with a given PKey belong to this port group
   - Partition name, which means that all the ports in the subnet that belong to partition with a given name belong to this port group
   - Node type, where possible node types are: CA, SWITCH, ROUTER, ALL, and SELF (SM's port).

2. QoS Setup (denoted by qos-setup) - this section describes how to set up SL2VL and VL Arbitration tables on various nodes in the fabric. However, this is not supported in OFED. SL2VL and VLArb tables should be configured in the OpenSM options file (default location - /var/cache/opensm/opensm.opts).

3. QoS Levels (denoted by qos-levels) - each QoS Level defines Service Level (SL) and a few optional fields:
   - MTU limit
   - Rate limit
   - PKey
   - Packet lifetime

   When path(s) search is performed, it is done with regards to restriction that these QoS Level parameters impose. One QoS level that is mandatory to define is a DEFAULT QoS level. It is applied to a PR/MPR query that does not match any existing match rule. Similar to any other QoS Level, it can also be explicitly referred by any match rule.

- QoS Matching Rules (denoted by qos-match-rules) - each PathRecord/MultiPathRecord query that OpenSM receives is matched against the set of matching rules. Rules are scanned in order of appearance in the QoS policy file such as the first match takes precedence.
  Each rule has a name of QoS level that will be applied to the matching query. A default QoS level is applied to a query that did not match any rule.
  Queries can be matched by:
  - Source port group (whether a source port is a member of a specified group)
  - Destination port group (same as above, only for destination port)
  - PKey
  - QoS class

- Service ID
To match a certain matching rule, PR/MPR query has to match ALL the rule's criteria.
However, not all the fields of the PR/MPR query have to appear in the matching rule.
For instance, if the rule has a single criterion - Service ID, it will match any query that has this Service ID, disregarding rest of the query fields. However, if a certain query has only Service ID (which means that this is the only bit in the PR/MPR component mask that is on), it will not match any rule that has other matching criteria besides Service ID.

Simple QoS Policy Definition

Simple QoS policy definition comprises of a single section denoted by qos-ulps. Similar to the advanced QoS policy, it has a list of match rules and their QoS Level, but in this case a match rule has only one criterion - its goal is to match a certain ULP (or a certain application on top of this ULP) PR/MPR request, and QoS Level has only one constraint - Service Level (SL).
The simple policy section may appear in the policy file in combine with the advanced policy, or as a stand-alone policy definition. See more details and list of match rule criteria below.

Policy File Syntax Guidelines

- Leading and trailing blanks, as well as empty lines, are ignored, so the indentation in the example is just for better readability.
- Comments are started with the pound sign (#) and terminated by EOL.
- Any keyword should be the first non-blank in the line, unless it's a comment.
- Keywords that denote section/subsection start have matching closing keywords.
- Having a QoS Level named "DEFAULT" is a must - it is applied to PR/MPR requests that did not match any of the matching rules.
- Any section/subsection of the policy file is optional.

Examples of Advanced Policy Files

As mentioned earlier, any section of the policy file is optional, and the only mandatory part of the policy file is a default QoS Level.
Here is an example of the shortest policy file:

```
qos-levels
    qos-level
        name: DEFAULT
        sl: 0
    end-qos-level
end-qos-levels
```

Port groups section is missing because there are no match rules, which means that port groups are not referred anywhere, and there is no need defining them. And since this policy file doesn't have any matching rules, PR/MPR query will not match any rule, and OpenSM will enforce default QoS level. Essentially, the above example is equivalent to not having a QoS policy file at all.
The following example shows all the possible options and keywords in the policy file and their syntax:

```
#
# See the comments in the following example.
# They explain different keywords and their meaning.
#
port-groups

    port-group # using port GUIDs
        name: Storage
        # "use" is just a description that is used for logging
        #  Other than that, it is just a comment
        use: SRP Targets
        port-guid: 0x10000000000001, 0x10000000000005-0x1000000000FFFA
        port-guid: 0x1000000000FFFF
    end-port-group

    port-group
        name: Virtual Servers
        # The syntax of the port name is as follows:
        #   "node_description/Pnum".
        # node_description is compared to the NodeDescription of the node,
        # and "Pnum" is a port number on that node.
        port-name: “vs1 HCA-1/P1, vs2 HCA-1/P1”
    end-port-group

    # using partitions defined in the partition policy
    port-group
        name: Partitions
        partition: Part1
        pkey: 0x1234
    end-port-group

    # using node types: CA, ROUTER, SWITCH, SELF (for node that runs SM)
    # or ALL (for all the nodes in the subnet)
    port-group
        name: CAs and SM
        node-type: CA, SELF
    end-port-group

end-port-groups

qos-setup
    # This section of the policy file describes how to set up SL2VL and VL
    # Arbitration tables on various nodes in the fabric.
    # However, this is not supported in OFED - the section is parsed
    # and ignored. SL2VL and VLArb tables should be configured in the
    # OpenSM options file (by default - /var/cache/opensm/opensm.opts).
end-qos-setup

qos-levels

    # Having a QoS Level named "DEFAULT" is a must - it is applied to
    # PR/MPR requests that didn't match any of the matching rules.
    qos-level
        name: DEFAULT
        use: default QoS Level
        sl: 0
    end-qos-level

    # the whole set: SL, MTU-Limit, Rate-Limit, PKey, Packet Lifetime
    qos-level
        name: WholeSet
        sl: 1
        mtu-limit: 4
        rate-limit: 5
        pkey: 0x1234
        packet-life: 8
    end-qos-level

end-qos-levels

# Match rules are scanned in order of their appearance in the policy file.
# First matched rule takes precedence.
qos-match-rules

    # matching by single criteria: QoS class
    qos-match-rule
        use: by QoS class
        qos-class: 7-9,11
        # Name of qos-level to apply to the matching PR/MPR
        qos-level-name: WholeSet
    end-qos-match-rule

    # show matching by destination group and service id
    qos-match-rule
        use: Storage targets
        destination: Storage
        service-id: 0x10000000000001, 0x10000000000008-0x10000000000FFF
        qos-level-name: WholeSet
    end-qos-match-rule

    qos-match-rule
        source: Storage
        use: match by source group only
        qos-level-name: DEFAULT
    end-qos-match-rule
    qos-match-rule
        use: match by all parameters
        qos-class: 7-9,11
        source: Virtual Servers
        destination: Storage
        service-id: 0x0000000000010000-0x000000000001FFFF
        pkey: 0x0F00-0x0FFF
        qos-level-name: WholeSet
    end-qos-match-rule
end-qos-match-rules
```

Simple QoS Policy - Details and Examples

Simple QoS policy match rules are tailored for matching ULPs (or some application on top of a ULP) PR/MPR requests. This section has a list of per-ULP (or per-application) match rules and the SL that

should be enforced on the matched PR/MPR query.
Match rules include:

- Default match rule that is applied to PR/MPR query that didn't match any of the other match rules
- IPoIB with a default PKey
- IPoIB with a specific PKey
- Any ULP/application with a specific Service ID in the PR/MPR query
- Any ULP/application with a specific PKey in the PR/MPR query
- Any ULP/application with a specific target IB port GUID in the PR/MPR query

Since any section of the policy file is optional, as long as basic rules of the file are kept (such as no referring to nonexistent port group, having default QoS Level, etc), the simple policy section (qos-ulps) can serve as a complete QoS policy file.
The shortest policy file in this case would be as follows:

```
qos-ulps
    default   : 0 #default SL
end-qos-ulps
```

It is equivalent to the previous example of the shortest policy file, and it is also equivalent to not having policy file at all. Below is an example of simple QoS policy with all the possible keywords:

```
qos-ulps
default               :0 # default SL
sdp, port-num 30000   :0 # SL for application running on
                           # top of SDP when a destination
                           # TCP/IPport is 30000
sdp, port-num 10000-20000     : 0
sdp                   :1 # default SL for any other
                           # application running on top of SDP
rds                   :2 # SL for RDS traffic
ipoib, pkey 0x0001    :0 # SL for IPoIB on partition with
                           # pkey 0x0001
ipoib                 :4 # default IPoIB partition,
                           # pkey=0x7FFF
any, service-id 0x6234:6 # match any PR/MPR query with a
                           # specific Service ID
any, pkey 0x0ABC      :6 # match any PR/MPR query with a
                           # specific PKey
srp, target-port-guid 0x1234  : 5 # SRP when SRP Target is located
                           # on a specified IB port GUID
any, target-port-guid 0x0ABC-0xFFFFF : 6 # match any PR/MPR query
                           # with a specific target port GUID
end-qos-ulps
```

Similar to the advanced policy definition, matching of PR/MPR queries is done in order of appearance in the QoS policy file such as the first match takes precedence, except for the "default" rule, which is applied only if the query didn't match any other rule. All other sections of the QoS policy file take precedence over the qos-ulps section. That is, if a policy file has both qos-match-rules and qos-ulps sections, then any query is matched first against the rules in the qos-match-rules section, and only if there was no match, the query is matched against the rules in qos-ulps section.
Note that some of these match rules may overlap, so in order to use the simple QoS definition effectively, it is important to understand how each of the ULPs is matched.

IPoIB

IPoIB query is matched by PKey or by destination GID, in which case this is the GID of the multicast group that OpenSM creates for each IPoIB partition.
Default PKey for IPoIB partition is 0x7fff, so the following three match rules are equivalent:

```
ipoib:<SL>ipoib, pkey 0x7fff : <SL>
any, pkey 0x7fff : <SL>
```

SRP

Service ID for SRP varies from storage vendor to vendor, thus SRP query is matched by the target IB port GUID. The following two match rules are equivalent:

```
srp, target-port-guid 0x1234 : <SL>
any, target-port-guid 0x1234 : <SL>
```

Note that any of the above ULPs might contain target port GUID in the PR query, so in order for these queries not to be recognized by the QoS manager as SRP, the SRP match rule (or any match rule that refers to the target port GUID only) should be placed at the end of the qos-ulps match rules.

MPI

SL for MPI is manually configured by an MPI admin. OpenSM is not forcing any SL on the MPI traffic, which explains why it is the only ULP that did not appear in the qos-ulps section.

SL2VL Mapping and VL Arbitration

OpenSM cached options file has a set of QoS related configuration parameters, that are used to configure SL2VL mapping and VL arbitration on IB ports. These parameters are:

- Max VLs: the maximum number of VLs that will be on the subnet
- High limit: the limit of High Priority component of VL Arbitration table (IBA 7.6.9)
- VLArb low table: Low priority VL Arbitration table (IBA 7.6.9) template
- VLArb high table: High priority VL Arbitration table (IBA 7.6.9) template
- SL2VL: SL2VL Mapping table (IBA 7.6.6) template. It is a list of VLs corresponding to SLs 0-15 (Note that VL15 used here means drop this SL).

There are separate QoS configuration parameters sets for various target types: CAs, routers, switch external ports, and switch's enhanced port 0. The names of such parameters are prefixed by "qos_<type>_" string. Here is a full list of the currently supported sets:

- qos_ca_ —QoS configuration parameters set for CAs.
- qos_rtr_ —parameters set for routers.
- qos_sw0_ —parameters set for switches' port 0.
- qos_swe_ —parameters set for switches' external ports.

Here's the example of typical default values for CAs and switches' external ports (hard-coded in OpenSM initialization):

```
qos_ca_max_vls 15
qos_ca_high_limit 0
qos_ca_vlarb_high 0:4,1:0,2:0,3:0,4:0,5:0,6:0,7:0,8:0,9:0,10:0,11:0,12:0,13:0,14:0
qos_ca_vlarb_low 0:0,1:4,2:4,3:4,4:4,5:4,6:4,7:4,8:4,9:4,10:4,11:4,12:4,13:4,14:4
qos_ca_sl2vl 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,7
qos_swe_max_vls 15
qos_swe_high_limit 0
qos_swe_vlarb_high 0:4,1:0,2:0,3:0,4:0,5:0,6:0,7:0,8:0,9:0,10:0,11:0,12:0,13:0,14:0
qos_swe_vlarb_low 0:0,1:4,2:4,3:4,4:4,5:4,6:4,7:4,8:4,9:4,10:4,11:4,12:4,13:4,14:4
qos_swe_sl2vl 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,7
```

VL arbitration tables (both high and low) are lists of VL/Weight pairs. Each list entry contains a VL number (values from 0-14), and a weighting value (values 0-255), indicating the number of 64 byte

units (credits) which may be transmitted from that VL when its turn in the arbitration occurs. A weight of 0 indicates that this entry should be skipped. If a list entry is programmed for VL15 or for a VL that is not supported or is not currently configured by the port, the port may either skip that entry or send from any supported VL for that entry.
Note, that the same VLs may be listed multiple times in the High or Low priority arbitration tables, and, further, it can be listed in both tables. The limit of high-priority VLArb table (qos_<type>_high_limit) indicates the number of high-priority packets that can be transmitted without an opportunity to send a low-priority packet. Specifically, the number of bytes that can be sent is high_limit times 4K bytes.
A high_limit value of 255 indicates that the byte limit is unbounded.

If the 255 value is used, the low priority VLs may be starved.

A value of 0 indicates that only a single packet from the high-priority table may be sent before an opportunity is given to the low-priority table.
Keep in mind that ports usually transmit packets of size equal to MTU. For instance, for 4KB MTU a single packet will require 64 credits, so in order to achieve effective VL arbitration for packets of 4KB MTU, the weighting values for each VL should be multiples of 64.
Below is an example of SL2VL and VL Arbitration configuration on subnet:

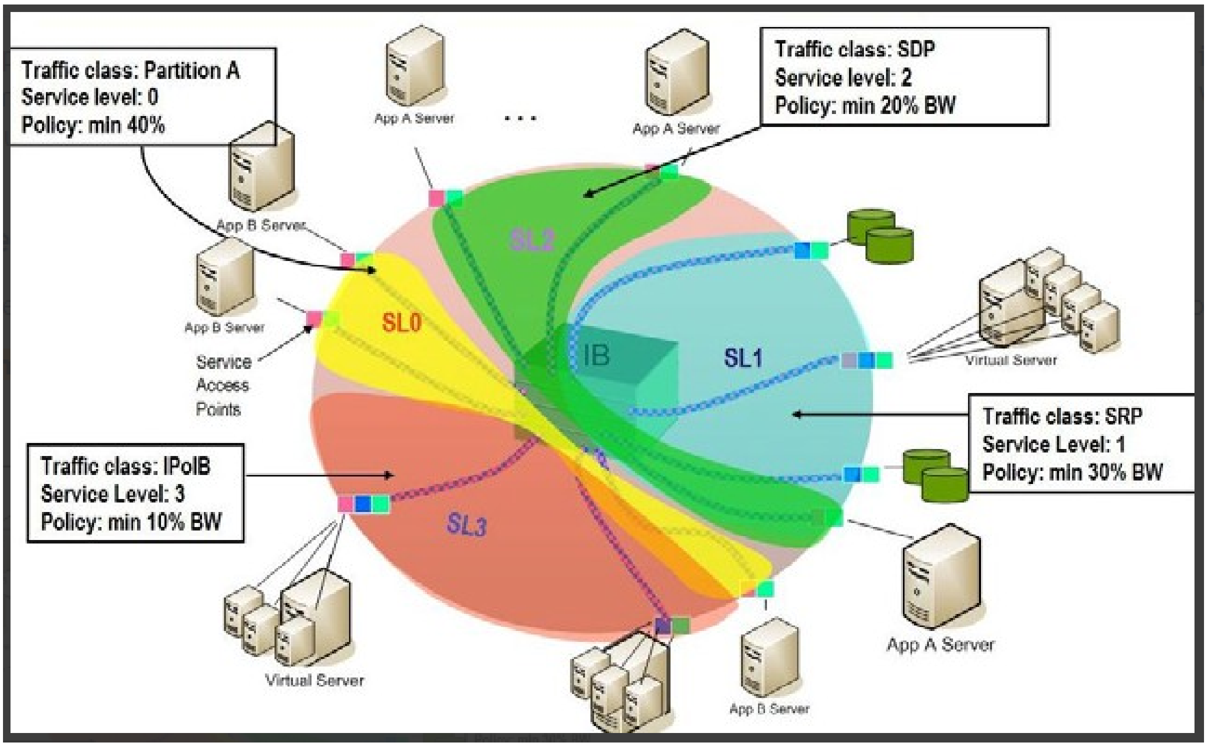
```
qos_ca_max_vls 15
qos_ca_high_limit 6
qos_ca_vlarb_high 0:4
qos_ca_vlarb_low 0:0,1:64,2:128,3:192,4:0,5:64,6:64,7:64
qos_ca_sl2vl 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,7
qos_swe_max_vls 15
qos_swe_high_limit 6
qos_swe_vlarb_high 0:4
qos_swe_vlarb_low 0:0,1:64,2:128,3:192,4:0,5:64,6:64,7:64
qos_swe_sl2vl 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,7
```

In this example, there are 8 VLs configured on subnet: VL0 to VL7. VL0 is defined as a high priority VL, and it is limited to 6 x 4KB = 24KB in a single transmission burst. Such configuration would suilt VL that needs low latency and uses small MTU when transmitting packets. Rest of VLs are defined as low priority VLs with different weights, while VL4 is effectively turned off.

Deployment Example

The figure below shows an example of an InfiniBand subnet that has been configured by a QoS manager to provide different service levels for various ULPs.

QoS Deployment on InfiniBand Subnet Example



QoS Configuration Examples

The following are examples of QoS configuration for different cluster deployments. Each example provides the QoS level assignment and their administration via OpenSM configuration files.

Typical HPC Example: MPI and Lustre

Assignment of QoS Levels

- MPI
  - Separate from I/O load
  - Min BW of 70%
- Storage Control (Lustre MDS)
  - Low latency
- Storage Data (Lustre OST)
  - Min BW 30%

Administration

- MPI is assigned an SL via the command line
  ```
  host1# mpirun -sl 0
  ```
- OpenSM QoS policy file

```
qos-ulps
   default                                        :0 # default SL (for MPI)
   any, target-port-guid OST1,OST2,OST3,OST4      :1 # SL for Lustre OST
   any, target-port-guid MDS1,MDS2                :2 # SL for Lustre MDS
end-qos-ulps
```

Note: In this policy file example, replace OST* and MDS* with the real port GUIDs.

- OpenSM options file

```
qos_max_vls 8
qos_high_limit 0
qos_vlarb_high 2:1
qos_vlarb_low 0:96,1:224
qos_sl2vl 0,1,2,3,4,5,6,7,15,15,15,15,15,15,15,15
```

EDC SOA (2-tier): IPoIB and SRP

The following is an example of QoS configuration for a typical enterprise data center (EDC) with service oriented architecture (SOA), with IPoIB carrying all application traffic and SRP used for storage.

QoS Levels

- Application traffic
  - IPoIB (UD and CM) and SDP
  - Isolated from storage
  - Min BW of 50%
- SRP
  - Min BW 50%
  - Bottleneck at storage nodes

Administration

- OpenSM QoS policy file

```
 qos-ulps
    default                                 :0
    ipoib                                   :1
    sdp                                     :1
    srp, target-port-guid SRPT1,SRPT2,SRPT3     :2
 end-qos-ulps
```

Note: In this policy file example, replace SRPT* with the real SRP Target port GUIDs.

- OpenSM options file

```
qos_max_vls 8
qos_high_limit 0
qos_vlarb_high 1:32,2:32
qos_vlarb_low 0:1,
qos_sl2vl 0,1,2,3,4,5,6,7,15,15,15,15,15,15,15,15
```

EDC (3-tier): IPoIB, RDS, SRP

The following is an example of QoS configuration for an enterprise data center (EDC), with IPoIB carrying all application traffic, RDS for database traffic, and SRP used for storage.

QoS Levels

- Management traffic (ssh)
  - IPoIB management VLAN (partition A)
  - Min BW 10%
- Application traffic

    - IPoIB application VLAN (partition B)
    - Isolated from storage and database
    - Min BW of 30%
- Database Cluster traffic
    - RDS
    - Min BW of 30%
- SRP
    - Min BW 30%
    - Bottleneck at storage nodes

Administration

- OpenSM QoS policy file

```
qos-ulps
   default                                       :0
   ipoib, pkey 0x8001                            :1
   ipoib, pkey 0x8002                            :2
   rds                                           :3
   srp, target-port-guid SRPT1, SRPT2, SRPT3    :4
end-qos-ulps
```

Note: In the following policy file example, replace SRPT* with the real SRP Initiator port GUIDs.

- OpenSM options file

```
qos_max_vls 8
qos_high_limit 0
qos_vlarb_high 1:32,2:96,3:96,4:96
qos_vlarb_low 0:1
qos_sl2vl 0,1,2,3,4,5,6,7,15,15,15,15,15,15,15,15
```

- Partition configuration file

```
Default=0x7fff,ipoib : ALL=full;PartA=0x8001, sl=1, ipoib : ALL=full;
```

Enhanced QoS

Enhanced QoS provides a higher resolution of QoS at the service level (SL). Users can configure rate limit values per SL for physical ports, virtual ports, and port groups, using enhanced_qos_policy_file configuration parameter.
Valid values of this parameter:

- Full path to the policy file through which Enhanced QoS Manager is configured
- "null" - to disable the Enhanced QoS Manager (default value)

⚠ To enable Enhanced QoS Manager, QoS must be enabled in OpenSM.

Enhanced QoS Policy File

The policy file is comprised of three sections:

- BW_NAMES: Used to define bandwidth setting and name (currently, rate limit is the only setting). Bandwidth names can be used in BW_RULES and VPORT_BW_RULES sections.
  Bandwidth names are defined using the syntax:
  `<name> = <rate limit in 1Mbps units>`
  Example: `My_bandwidth = 50`
- BW_RULES: Used to define the rules that map the bandwidth setting to a specific SL of a specific GUID.
  Bandwidth rules are defined using the syntax:
  `<guid>|<port group name> = <sl id>:<bandwidth name>, <sl id>:<bandwidth name>…`
  Examples:
  `0x2c90000000025 = 5:My_bandwidth, 7:My_bandwidth`
  `Port_grp1 = 3:My_bandwidth, 9:My_bandwidth`
- VPORT_BW_RULES: Used to define the rules that map the bandwidth setting to a specific SL of a specific virtual port GUID.
  Bandwidth rules are defined using the syntax:
  `<guid>= <sl id>:<bandwidth name>, <sl id>:<bandwidth name>…`
  Examples:
  `0x2c90000000026= 5:My_bandwidth, 7:My_bandwidth`

Special Keywords

- Keyword “all” allows setting a rate limit of all SLs to some BW for a specific physical or virtual port. It is possible to combine “all” with specific SL rate limits.
  Example:
  `0x2c90000000025 = all:BW1,SL3:BW2`
  In this case, SL3 will be assigned BW2 rate limit, while the rest of SLs get BW1 rate limit.
- "default" is a well-known name which can be used to define a default rule used for any GUID with no defined rule.
  If no default rule is defined, any GUID without a specific rule will be configured with unlimited rate limit for all SLs.
  Keyword “all” is also applicable to the default rule. Default rule is local to each section.

Special Subnet Manager Configuration Options

New SM configuration option enhanced_qos_vport0_unlimit_default_rl was added to opensm.conf.

The possible values for this configuration option are:

- TRUE: For specific virtual port0 GUID, SLs not mentioned in bandwidth rule will be set to unlimited bandwidth (0) regardless of the default rule of the VPORT_BW_RULES section.
  Virtual port0 GUIDs not mentioned in VPORT_BW_SECTION will be set to unlimited BW on all SLs.
- FALSE: The GUID of virtual port0 is treated as any other virtual port in VPORT_BW_SECTION.
  SM should be signaled by HUP once the option is changed.

Default: TRUE

Notes

- When rate limit is set to 0, it means that the bandwidth is unlimited.
- Any unspecified SL in a rule will be set to 0 (unlimited) rate limit automatically if no default rule is specified.
- Failure to complete policy file parsing leads to an undefined behavior. User must confirm no relevant error messages in SM log in order to ensure Enhanced QoS Manager is configured properly.
- A file with only 'BW_NAMES' and 'BW_RULES' keywords configures the network with an unlimited rate limit.
- HCA physical port GUID can be specified in BW_RULES and VPORT_BW_RULES sections.
- In BW_RULES section, the rate limit assigned to a specific SL will limit the total BW that can be sent through the PF on a given SL.
- In VPORT_BW_RULES section, the rate limit assigned to a specific SL will limit only the traffic sent from the IB interface corresponding to the physical port GUID (virtual port0 IB interface). The traffic sent from other virtual IB interfaces will not be limited if no specific rules are defined.

Policy File Example

All physical ports in the fabric are with a rate limit of 50Mbps on SL1, except for GUID 0x2c90000000025, which is configured with rate limit of 25Mbps on SL1. In this example, the traffic on SLs (other than SL1) is unlimited.
All virtual ports in the fabric (except virtual port0 of all physical ports) will be rate-limited to 15Mbps for all SLs because of the default rule of VPORT_BW_RULES section.
Virtual port GUID 0x2c90000000026 is configured with a rate limit of 10Mbps on SL3. The rest of the SLs on this virtual port will get a rate limit of 15 Mbps because of the default rule of VPORT_BW_RULES section.

```
-------------------------------------------------------------------------
BW_NAMES
bw1 = 50
bw2 = 25
bw3 = 15
bw4 = 10

BW_RULES
default= 1:bw1
0x2c90000000025= 1:bw2

VPORT_BW_RULES
default= all:bw3
0x2c90000000026= 3:bw4

--------------------------------------------------------------------------
```

14.6.2.1.2.8 Adaptive Routing Manager and Self-Healing Networking

Adaptive Routing Manager supports advanced InfiniBand features; Adaptive Routing (AR) and Self-Healing Networking.

For information on how to set up AR and Self-Healing Networking, please refer to HowTo Configure Adaptive Routing and Self-Healing Networking Community post.

DOS MAD Prevention

DOS MAD prevention is achieved by assigning a threshold for each agent's RX. Agent's RX threshold provides a protection mechanism to the host memory by limiting the agents' RX with a threshold. Incoming MADs above the threshold are dropped and are not queued to the agent's RX.

To enable DOS MAD Prevention:

1. Go to /etc/modprobe.d/mlnx.conf.
2. Add to the file the option below.

```
ib_umad enable_rx_threshold 1
```

The threshold value can be controlled from the user-space via libibumad.

To change the value, use the following API:

```
int umad_update_threshold(int fd, int threshold);

@fd: file descriptor, agent's RX associated to this fd.
@threshold: new threshold value
```

14.6.2.1.2.9 IB Router Support in OpenSM

In order to enable the IB router in OpenSM, the following parameters should be configured:

IB Router Parameters for OpenSM

Parameter	Description	Default Value
`rtr_pr_flow_label`	Defines whether the SM should create alias GUIDs required for router support for each port. Defines flow label value to use in response for path records related to the router.	0 (Disabled)
rtr_pr_tclass	Defines TClass value to use in response for path records related to the router	0
rtr_pr_sl	Defines sl value to use in response for path records related to router.	0
rtr_p_mtu	Defines MTU value to use in response for path records related to the router.	4 (IB_MTU_LEN_2048)
rtr_pr_rate	Defines rate value to use in response for path records related to the router.	16 (IB_PATH_RECORD_RATE_100_GBS)

14.6.2.1.2.10 OpenSM Activity Report

OpenSM can produce an activity report in a form of a dump file which details the different activities done in the SM. Activities are divided into subjects. The OpenSM Supported Activities table below specifies the different activities currently supported in the SM activity report.
Reporting of each subject can be enabled individually using the configuration parameter `activity_report_subjects`:

- Valid values:
  Comma separated list of subjects to dump. The current supported subjects are:
  - "mc" - activity IDs 1, 2 and 8
  - "prtn" - activity IDs 3, 4, and 5
  - "virt" - activity IDs 6 and 7
  - "routing" - activity IDs 8-12

Two predefined values can be configured as well:

  - "all" - dump all subjects
  - "none" - disable the feature by dumping none of the subjects
- Default value: "none"

OpenSM Supported Activities

ACtivity ID	Activity Name	Additional Fields	Comments	Description
1	mcm_member	• MLid • MGid • Port Guid • Join State	Join state: 1 - Join -1 - Leave	Member joined/ left MC group
2	mcg_change	• MLid • MGid • Change	Change: 0 - Create 1 - Delete	MC group created/ deleted
3	prtn_guid_add	• Port Guid • PKey • Block index • Pkey Index		Guid added to partition
4	prtn_create	-PKey • Prtn Name		Partition created
5	prtn_delete	• PKey • Delete Reason	Delete Reason: 0 - empty prtn 1 - duplicate prtn 2 - sm shutdown	Partition deleted
6	port_virt_discover	• Port Guid • Top Index		Port virtualization discovered
7	vport_state_change	• Port Guid • VPort Guid • VPort Index • VNode Guid • VPort State	VPort State: 1 - Down 2 - Init 3 - ARMED 4 - Active	Vport state changed
8	mcg_tree_calc	mlid		MCast group tree calculated
9	routing_succeed	routing engine name		Routing done successfully
10	routing_failed	routing engine name		Routing failed
11	ucast_cache_invalidated			ucast cache invalidated
12	ucast_cache_routing_done			ucast cache routing done

14.6.2.1.2.11 Offsweep Balancing

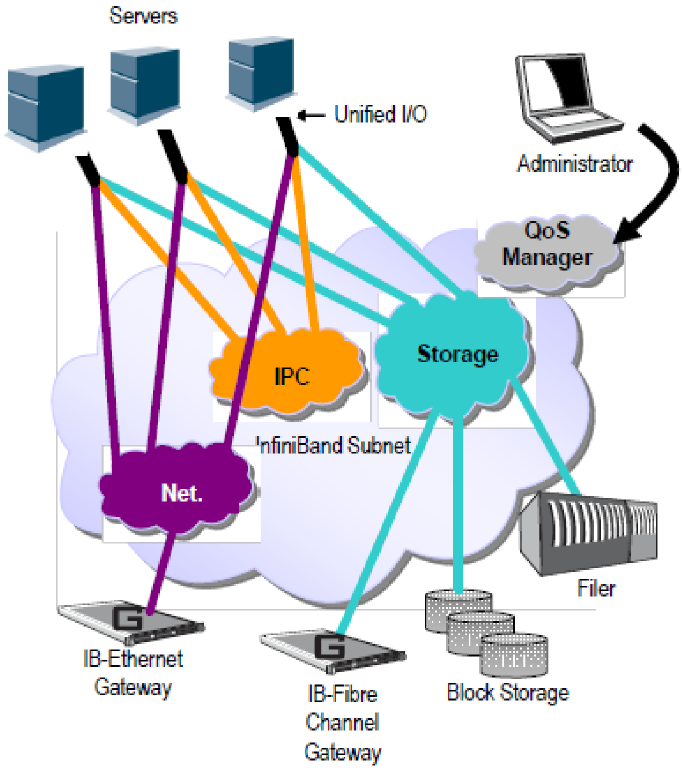
When working with minhop/dor/updn, subnet manager can re-balance routing during idle time (between sweeps).

- offsweep_balancing_enabled - enables/disables the feature. Examples:
  - offsweep_balancing_enabled = TRUE

    - offsweep_balancing_enabled = FALSE (default)
- offsweep_balancing_window - defines window of seconds to wait after sweep before starting the re-balance process. Applicable only if offsweep_balancing_enabled=TRUE. Example: offsweep_balancing_window = 180 (default)

14.6.2.1.3 QoS - Quality of Service

Quality of Service (QoS) requirements stem from the realization of I/O consolidation over an IB network. As multiple applications and ULPs share the same fabric, a means is needed to control their use of network resources.



The basic need is to differentiate the service levels provided to different traffic flows, such that a policy can be enforced and can control each flow utilization of fabric resources.
The InfiniBand Architecture Specification defines several hardware features and management interfaces for supporting QoS:

Up to 15 Virtual Lanes (VL) carry traffic in a non-blocking manner

- Arbitration between traffic of different VLs is performed by a two-priority-level weighted round robin arbiter. The arbiter is programmable with a sequence of (VL, weight) pairs and a maximal number of high priority credits to be processed before low priority is served
- Packets carry class of service marking in the range 0 to 15 in their header SL field
- Each switch can map the incoming packet by its SL to a particular output VL, based on a programmable table VL=SL-to-VL-MAP(in-port, out-port, SL)
- The Subnet Administrator controls the parameters of each communication flow by providing them as a response to Path Record (PR) or MultiPathRecord (MPR) queries

DiffServ architecture (IETF RFC 2474 & 2475) is widely used in highly dynamic fabrics. The following subsections provide the functional definition of the various software elements that enable a DiffServ-like architecture over the NVIDIA OFED software stack.

14.6.2.1.3.1 **QoS Architecture**

QoS functionality is split between the SM/SA, CMA and the various ULPs. We take the "chronology approach" to describe how the overall system works.

1. The network manager (human) provides a set of rules (policy) that define how the network is being configured and how its resources are split to different QoS-Levels. The policy also define how to decide which QoS-Level each application or ULP or service use.
2. The SM analyzes the provided policy to see if it is realizable and performs the necessary fabric setup. Part of this policy defines the default QoS-Level of each partition. The SA is enhanced to match the requested Source, Destination, QoS-Class, Service-ID, PKey against the policy, so clients (ULPs, programs) can obtain a policy enforced QoS. The SM may also set up partitions with appropriate IPoIB broadcast group. This broadcast group carries its QoS attributes: SL, MTU, RATE, and Packet Lifetime.
3. IPoIB is being setup. IPoIB uses the SL, MTU, RATE and Packet Lifetime available on the multicast group which forms the broadcast group of this partition.
4. MPI which provides non IB based connection management should be configured to run using hard coded SLs. It uses these SLs for every QP being opened.
5. ULPs that use CM interface (like SRP) have their own pre-assigned Service-ID and use it while obtaining PathRecord/MultiPathRecord (PR/MPR) for establishing connections. The SA receiving the PR/MPR matches it against the policy and returns the appropriate PR/MPR including SL, MTU, RATE and Lifetime.
6. ULPs and programs (e.g. SDP) use CMA to establish RC connection provide the CMA the target IP and port number. ULPs might also provide QoS-Class. The CMA then creates Service-ID for the ULP and passes this ID and optional QoS-Class in the PR/MPR request. The resulting PR/MPR is used for configuring the connection QP.

PathRecord and Multi Path Record Enhancement for QoS:

As mentioned above, the PathRecord and MultiPathRecord attributes are enhanced to carry the Service-ID which is a 64bit value. A new field QoS-Class is also provided.
A new capability bit describes the SM QoS support in the SA class port info. This approach provides an easy migration path for existing access layer and ULPs by not introducing new set of PR/MPR attributes.

14.6.2.1.3.2 Supported Policy

The QoS policy, which is specified in a stand-alone file, is divided into the following four subsections:

Port Group

A set of CAs, Routers or Switches that share the same settings. A port group might be a partition defined by the partition manager policy, list of GUIDs, or list of port names based on NodeDescription.

Fabric Setup

Defines how the SL2VL and VLArb tables should be set up.

> ⚠ In OFED this part of the policy is ignored. SL2VL and VLArb tables should be configured in the OpenSM options file (opensm.opts).

QoS-Levels Definition

This section defines the possible sets of parameters for QoS that a client might be mapped to. Each set holds SL and optionally: Max MTU, Max Rate, Packet Lifetime and Path Bits.

> ⚠ Path Bits are not implemented in OFED.

Matching Rules

A list of rules that match an incoming PR/MPR request to a QoS-Level. The rules are processed in order such as the first match is applied. Each rule is built out of a set of match expressions which should all match for the rule to apply. The matching expressions are defined for the following fields:

- SRC and DST to lists of port groups
- Service-ID to a list of Service-ID values or ranges
- QoS-Class to a list of QoS-Class values or ranges

14.6.2.1.3.3 CMA Features

The CMA interface supports Service-ID through the notion of port space as a prefix to the port number, which is part of the sockaddr provided to rdma_resolve_add(). The CMA also allows the ULP (like SDP) to propagate a request for a specific QoS-Class. The CMA uses the provided QoS-Class and Service-ID in the sent PR/MPR.

IPoIB

IPoIB queries the SA for its broadcast group information and uses the SL, MTU, RATE and Packet Lifetime available on the multicast group which forms this broadcast group.

SRP

The current SRP implementation uses its own CM callbacks (not CMA). So SRP fills in the Service-ID in the PR/MPR by itself and use that information in setting up the QP.
SRP Service-ID is defined by the SRP target I/O Controller (it also complies with IBTA Service- ID rules). The Service-ID is reported by the I/O Controller in the ServiceEntries DMA attribute and should be used in the PR/MPR if the SA reports its ability to handle QoS PR/MPRs.

14.6.2.1.4 IP over InfiniBand (IPoIB)

14.6.2.1.4.1 Upper Layer Protocol (ULP)

The IP over IB (IPoIB) ULP driver is a network interface implementation over InfiniBand. IPoIB encapsulates IP datagrams over an InfiniBand Datagram transport service. The IPoIB driver, ib_ipoib, exploits the following capabilities:

- VLAN simulation over an InfiniBand network via child interfaces
- High Availability via Bonding

- Varies MTU values:
    - up to 4k in Datagram mode
- Uses any ConnectX® IB ports (one or two)
- Inserts IP/UDP/TCP checksum on outgoing packets
- Calculates checksum on received packets
- Support net device TSO through ConnectX® LSO capability to defragment large data- grams to MTU quantas.

IPoIB also supports the following software based enhancements:

- Giant Receive Offload
- NAPI
- Ethtool support

14.6.2.1.4.2 Enhanced IPoIB

Enhanced IPoIB feature enables offloading ULP basic capabilities to a lower vendor specific driver, in order to optimize IPoIB data path. This will allow IPoIB to support multiple stateless offloads, such as RSS/TSS, and better utilize the features supported, enabling IPoIB datagram to reach peak performance in both bandwidth and latency.

Enhanced IPoIB supports/performs the following:

- Stateless offloads (RSS, TSS)
- Multi queues
- Interrupt moderation
- Multi partitions optimizations
- Sharing send/receive Work Queues
- Vendor specific optimizations
- UD mode only

14.6.2.1.4.3 Port Configuration

The physical port MTU (indicates the port capability) default value is 4k, whereas the IPoIB port MTU ("logical" MTU) default value is 2k as it is set by the OpenSM.
To change the IPoIB MTU to 4k, edit the OpenSM partition file in the section of IPoIB setting as follow:

```
Default=0xffff, ipoib, mtu=5 : ALL=full;
```

where:

"mtu=5" indicates that all IPoIB ports in the fabric are using 4k MTU, ("mtu=4" indi- cates 2k MTU)

14.6.2.1.4.4 IPoIB Configuration

Unless you have run the installation script mlnxofedinstall with the flag '-n', then IPoIB has not been configured by the installation. The configuration of IPoIB requires assigning an IP address and a subnet mask to each HCA port, like any other network adapter card (i.e., you need to prepare a file called ifcfg-ib<n> for each port). The first port on the first HCA in the host is called interface ib0, the second port is called ib1, and so on.

IPoIB configuration can be based on DHCP or on a static configuration that you need to supply (see below). You can also apply a manual configuration that persists only until the next reboot or driver restart (see below).

IPoIB Configuration Based on DHCP

Setting an IPoIB interface configuration based on DHCP is performed similarly to the configuration of Ethernet interfaces. In other words, you need to make sure that IPoIB configuration files include the following line:

- For RedHat:

```
BOOTPROTO=dhcp
```

- For SLES:

```
BOOTPROTO='dchp'
```

> If IPoIB configuration files are included, ifcfg-ib<n> files will be installed under:
>
> /etc/sysconfig/network-scripts/ on a RedHat machine
> /etc/sysconfig/network/ on a SuSE machine.

> A patch for DHCP may be required for supporting IPoIB. For further information, please see the REAME file available under the docs/dhcp/ directory.

> Red Hat Enterprise Linux 7 supports assigning static IP addresses to InfiniBand IPoIB interfaces. However, as these interfaces do not have a normal hardware Ethernet address, a different method of specifying a unique identifier for the IPoIB interface must be used. The standard is to use the option dhcp-client-identifier= construct to specify the IPoIB interface's dhcp-client-identifier field. The DHCP server host construct supports at most one hardware Ethernet and one dhcp-client-identifier entry per host stanza. However, there may be more than one fixed-address entry and the DHCP server will automatically respond with an address that is appropriate for the network that the DHCP request was received on.

Standard DHCP fields holding MAC addresses are not large enough to contain an IPoIB hardware address. To overcome this problem, DHCP over InfiniBand messages convey a client identifier field used to identify the DHCP session. This client identifier field can be used to associate an IP address with a client identifier value, such that the DHCP server will grant the same IP address to any client that conveys this client identifier.
The length of the client identifier field is not fixed in the specification. For the *NVIDIA OFED for Linux* package, it is recommended to have IPoIB use the same format that FlexBoot uses for this client identifier.

DHCP Server

In order for the DHCP server to provide configuration records for clients, an appropriate configuration file needs to be created. By default, the DHCP server looks for a configuration file called dhcpd.conf under /etc. You can either edit this file or create a new one and provide its full

path to the DHCP server using the -cf flag (See a file example at docs/dhcpd.conf).
The DHCP server must run on a machine which has loaded the IPoIB module. To run the DHCP server from the command line, enter:

```
dhcpd <IB network interface name> -d
```

Example:

```
host1# dhcpd ib0 -d
```

DHCP Client (Optional)

A DHCP client can be used if you need to prepare a diskless machine with an IB driver.

In order to use a DHCP client identifier, you need to first create a configuration file that defines the DHCP client identifier.

Then run the DHCP client with this file using the following command:

```
dhclient -cf <client conf file> <IB network interface name>
```

Example of a configuration file for the ConnectX (PCI Device ID 26428), called `dhclient.conf`:

```
The value indicates a hexadecimal number interface "ib1" {
send dhcp-client-identifier
ff:00:00:00:00:00:02:00:00:02:c9:00:00:02:c9:03:00:00:10:39;
}
```

Example of a configuration file for InfiniHost III Ex (PCI Device ID 25218), called `dhclient.conf`:

```
The value indicates a hexadecimal number interface "ib1" {
send dhcp-client-identifier
20:00:55:04:01:fe:80:00:00:00:00:00:00:00:02:c9:02:00:23:13:92;
}
```

In order to use the configuration file, run:

```
host1# dhclient -cf dhclient.conf ib1
```

Static IPoIB Configuration

If you wish to use an IPoIB configuration that is not based on DHCP, you need to supply the installation script with a configuration file (using the ‘-n’ option) containing the full IP configuration. The IPoIB configuration file can specify either or both of the following data for an IPoIB interface:

- A static IPoIB configuration
- An IPoIB configuration based on an Ethernet configuration
  See your Linux distribution documentation for additional information about configuring IP addresses.

The following code lines are an excerpt from a sample IPoIB configuration file:

```
# Static settings; all values provided by this file
IPADDR_ib0=10.4.3.175
NETMASK_ib0=255.255.0.0
NETWORK_ib0=10.4.0.0
BROADCAST_ib0=10.4.255.255
ONBOOT_ib0=1
# Based on eth0; each '*' will be replaced with a corresponding octet
# from eth0.
LAN_INTERFACE_ib0=eth0
IPADDR_ib0=10.4.'*'.'*'
NETMASK_ib0=255.255.0.0
NETWORK_ib0=10.4.0.0
BROADCAST_ib0=10.4.255.255
ONBOOT_ib0=1
# Based on the first eth<n> interface that is found (for n=0,1,...);
# each '*' will be replaced with a corresponding octet from eth<n>.
LAN_INTERFACE_ib0=
IPADDR_ib0=10.4.'*'.'*'
NETMASK_ib0=255.255.0.0
NETWORK_ib0=10.4.0.0
BROADCAST_ib0=10.4.255.255
ONBOOT_ib0=1
```

Manually Configuring IPoIB

> This manual configuration persists only until the next reboot or driver restart.

To manually configure IPoIB for the default IB partition (VLAN), perform the following steps:

1. Configure the interface by entering the ifconfig command with the following items:
   - The appropriate IB interface (ib0, ib1, etc.)
   - The IP address that you want to assign to the interface
   - The netmask keyword
   - The subnet mask that you want to assign to the interface
   The following example shows how to configure an IB interface:

   ```
   host1$ ifconfig ib0 10.4.3.175 netmask 255.255.0.0
   ```

2. (Optional) Verify the configuration by entering the ifconfig command with the appropriate interface identifier *ib#* argument.
   The following example shows how to verify the configuration:

   ```
   host1$ ifconfig ib0
   b0  Link encap:UNSPEC  HWaddr 80-00-04-04-FE-80-00-00-00-00-00-00-00-00-00-00
   inet addr:10.4.3.175  Bcast:10.4.255.255  Mask:255.255.0.0
   UP BROADCAST MULTICAST  MTU:65520  Metric:1
   RX packets:0 errors:0 dropped:0 overruns:0 frame:0
   TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
   collisions:0 txqueuelen:128
   RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
   ```

3. Repeat the first two steps on the remaining interface(s).

14.6.2.1.4.5 Sub-interfaces

You can create sub-interfaces for a primary IPoIB interface to provide traffic isolation. Each such sub-interface (also called a child interface) has a different IP and network addresses from the primary (parent) interface. The default Partition Key (PKey), ff:ff, applies to the primary (parent) interface.

This section describes how to:

- Create a subinterface
- Remove a subinterface

Creating a Subinterface

In the following procedure, ib0 is used as an example of an IB sub-interface.

➢ *To create a child interface (sub-interface), follow this procedure:*

1. Decide on the PKey to be used in the subnet (valid values can be 0 or any 16-bit unsigned value). The actual PKey used is a 16-bit number with the most significant bit set. For example, a value of 1 will give a PKey with the value 0x8001.
2. Create a child interface by running:

   ```
   host1$ echo <PKey> > /sys/class/net/<IB subinterface>/create_child
   ```

   Example:

   ```
   host1$ echo 1 > /sys/class/net/ib0/create_child
   ```

   This will create the interface ib0.8001.
3. Verify the configuration of this interface by running:

   ```
   host1$ ifconfig <subinterface>.<subinterface PKey>
   ```

   Using the example of the previous step:

   ```
   host1$ ifconfig ib0.8001
   ib0.8001  Link encap:UNSPEC  HWaddr 80-00-00-4A-FE-80-00-00-00-00-00-00-00-00-00-00
   BROADCAST MULTICAST  MTU:2044  Metric:1
   RX packets:0 errors:0 dropped:0 overruns:0 frame:0
   TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
   collisions:0 txqueuelen:128
   RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
   ```
4. As can be seen, the interface does not have IP or network addresses. To configure those, you should follow the manual configuration procedure described in "Manually Configuring IPoIB" section above.
5. To be able to use this interface, a configuration of the Subnet Manager is needed so that the PKey chosen, which defines a broadcast address, be recognized.

Removing a Subinterface

➢ *To remove a child interface (subinterface), run:*

```
echo <subinterface PKey> /sys/class/net/<ib_interface>/delete_child
```

Using the example of the second step from the previous chapter:

```
echo 0x8001 > /sys/class/net/ib0/delete_child
```

Note that when deleting the interface you must use the PKey value with the most significant bit set (e.g., 0x8000 in the example above).

14.6.2.1.4.6 Verifying IPoIB Functionality

➢ To verify your configuration and IPoIB functionality are successful, perform the following steps:

1. Verify the IPoIB functionality by using the `ifconfig` command.
   The following example shows how two IB nodes are used to verify IPoIB functionality. In the following example, IB node 1 is at 10.4.3.175, and IB node 2 is at 10.4.3.176:

```
host1# ifconfig ib0 10.4.3.175 netmask 255.255.0.0
host2# ifconfig ib0 10.4.3.176 netmask 255.255.0.0
```

2. Enter the ping command from 10.4.3.175 to 10.4.3.176.
3. The following example shows how to enter the ping command:

```
host1# ping -c 5 10.4.3.176
PING 10.4.3.176 (10.4.3.176) 56(84) bytes of data.
64 bytes from 10.4.3.176: icmp_seq=0 ttl=64 time=0.079 ms
64 bytes from 10.4.3.176: icmp_seq=1 ttl=64 time=0.044 ms
64 bytes from 10.4.3.176: icmp_seq=2 ttl=64 time=0.055 ms
64 bytes from 10.4.3.176: icmp_seq=3 ttl=64 time=0.049 ms
64 bytes from 10.4.3.176: icmp_seq=4 ttl=64 time=0.065 ms
--- 10.4.3.176 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 3999ms rtt min/avg/max/mdev = 0.044/0.058/0.079/
0.014 ms, pipe 2
```

14.6.2.1.4.7 Bonding IPoIB

To create an interface configuration script for the ibX and bondX interfaces, you should use the standard syntax (depending on your OS).
Bonding of IPoIB interfaces is accomplished in the same manner as would bonding of Ethernet interfaces: via the Linux Bonding Driver.

- Network Script files for IPoIB slaves are named after the IPoIB interfaces (e.g: ifcfg- ib0)
- The only meaningful bonding policy in IPoIB is High-Availability (bonding mode number 1, or active-backup)
- Bonding parameter "fail_over_mac" is meaningless in IPoIB interfaces, hence, the only supported value is the default: 0

For a persistent bonding IPoIB Network configuration, use the same Linux Network Scripts semantics, with the following exceptions/ additions:

- In the bonding master configuration file (e.g: ifcfg-bond0), in addition to Linux bonding semantics, use the following parameter: MTU=65520
  COND

  > ⚠ For IPoIB slaves, use MTU=2044. If you do not set the correct MTU or do not set MTU at all, performance of the interface might decrease.

  Dynamically Connected Transport (DCT)
- In the bonding slave configuration file (e.g: ifcfg-ib0), use the same Linux Network Scripts semantics. In particular: DEVICE=ib0
- In the bonding slave configuration file (e.g: ifcfg-ib0.8003), the line TYPE=InfiniBand is necessary when using bonding over devices configured with partitions (p_key)
- For RHEL users:
  In /etc/modprobe.b/bond.conf add the following lines:

```
alias bond0 bonding
```

- For SLES users:
  It is necessary to update the MANDATORY_DEVICES environment variable in /etc/sysconfig/network/config with the names of the IPoIB slave devices (e.g. ib0, ib1, etc.). Otherwise, bonding master may be created before IPoIB slave interfaces at boot time.
  It is possible to have multiple IPoIB bonding masters and a mix of IPoIB bonding master and Ethernet bonding master. However, It is NOT possible to mix Ethernet and IPoIB slaves under the same bonding master.

⚠ Restarting openibd does no keep the bonding configuration via Network Scripts. You have to restart the network service in order to bring up the bonding master. After the configuration is saved, restart the network service by running: /etc/init.d/network restart.

14.6.2.1.4.8 Dynamic PKey Change

Dynamic PKey change means the PKey can be changed (add/removed) in the SM database and the interface that is attached to that PKey is updated immediately without the need to restart the driver.
If the PKey is already configured in the port by the SM, the child-interface can be used immediately. If not, the interface will be ready to use only when SM adds the relevant PKey value to the port after the creation of the child interface. No additional configuration is required once the child-interface is created.

14.6.2.1.4.9 Precision Time Protocol (PTP) over IPoIB

This feature allows for accurate synchronization between the distributed entities over the network. The synchronization is based on symmetric Round Trip Time (RTT) between the master and slave devices.

This feature is enabled by default, and is also supported over PKey interfaces.

For more on the PTP feature, refer to Running Linux PTP with ConnectX-4/ConnectX-5/ConnectX-6 Community post.

For further information on Time-Stamping, follow the steps in "Time-Stamping Service".

14.6.2.1.4.10 One Pulse Per Second (1PPS) over IPoIB

1PPS is a time synchronization feature that allows the adapter to be able to send or receive 1 pulse per second on a dedicated pin on the adapter card using an SMA connector (SubMiniature version A). Only one pin is supported and could be configured as 1PPS in or 1PPS out.
For further information, refer to HowTo Test 1PPS on NVIDIA Adapters Community post.

14.6.2.1.5 Advanced Transport

14.6.2.1.5.1 Atomic Operations

Atomic Operations in mlx5 Driver

To enable atomic operation with this endianness contradiction, use the `ibv_create_qp` to create the QP and set the `IBV_QP_CREATE_ATOMIC_BE_REPLY` flag on `create_flags` .

14.6.2.1.5.2 XRC - eXtended Reliable Connected Transport Service for InfiniBand

XRC allows significant savings in the number of QPs and the associated memory resources required to establish all to all process connectivity in large clusters.
It significantly improves the scalability of the solution for large clusters of multicore end-nodes by reducing the required resources.
For further details, please refer to the "Annex A14 Supplement to InfiniBand Architecture Specification Volume 1.2.1"
A new API can be used by user space applications to work with the XRC transport. The legacy API is currently supported in both binary and source modes, however it is deprecated. Thus we recommend using the new API.
The new verbs to be used are:

- ibv_open_xrcd/ibv_close_xrcd
- ibv_create_srq_ex
- ibv_get_srq_num
- ibv_create_qp_ex
- ibv_open_qp

Please use `ibv_xsrq_pingpong` for basic tests and code reference. For detailed information regarding the various options for these verbs, please refer to their appropriate man pages.

14.6.2.1.5.3 Dynamically Connected Transport (DCT)

Dynamically Connected transport (DCT) service is an extension to transport services to enable a higher degree of scalability while maintaining high performance for sparse traffic. Utilization of DCT reduces the total number of QPs required system wide by having Reliable type QPs dynamically connect and disconnect from any remote node. DCT connections only stay connected while they are active. This results in smaller memory footprint, less overhead to set connections and higher on-chip cache utilization and hence increased performance. DCT is supported only in mlx5 driver.

⚠ Please note that ConnectX-4 supports DCT v0 and ConnectX-5 and above support DCT v1. DCTv0 and DCT v1 are not interoperable.

14.6.2.1.5.4 MPI Tag Matching and Rendezvous Offloads

⚠ Supported in ConnectX®-5 and above adapter cards.

Tag Matching and Rendezvous Offloads is a technology employed by NVIDIA to offload the processing of MPI messages from the host machine onto the network card. Employing this technology enables a zero copy of MPI messages, i.e. messages are scattered directly to the user's buffer without intermediate buffering and copies. It also provides a complete rendezvous progress by NVIDIA devices. Such overlap capability enables the CPU to perform the application's computational tasks while the remote data is gathered by the adapter.

For more information Tag Matching Offload, please refer to the Understanding MPI Tag Matching and Rendezvous Offloads (ConnectX-5) Community post.

14.6.2.1.6 Optimized Memory Access

14.6.2.1.6.1 Memory Region Re-registration

Memory Region Re-registration allows the user to change attributes of the memory region. The user may change the PD, access flags or the address and length of the memory region. Memory region supports contagious pages allocation. Consequently, it de-registers memory region followed by register memory region. Where possible, resources are reused instead of de-allocated and reallocated.

Example:

```
int ibv_rereg_mr(struct ibv_mr *mr, int flags, struct ibv_pd *pd, void *addr, size_t length, uint64_t access,
struct ibv_rereg_mr_attr *attr);
```

@mr:	The memory region to modify.
@flags:	A bit-mask used to indicate which of the following properties of the memory region are being modified. Flags should be one of: IBV_REREG_MR_CHANGE_TRANSLATION /* Change translation (location and length) */ IBV_REREG_MR_CHANGE_PD/* Change protection domain*/ IBV_REREG_MR_CHANGE_ACCESS/* Change access flags*/
@pd:	If IBV_REREG_MR_CHANGE_PD is set in flags, this field specifies the new protection domain to associated with the memory region, otherwise, this parameter is ignored.
@addr:	If IBV_REREG_MR_CHANGE_TRANSLATION is set in flags, this field specifies the start of the virtual address to use in the new translation, otherwise, this parameter is ignored.
@length:	If IBV_REREG_MR_CHANGE_TRANSLATION is set in flags, this field specifies the length of the virtual address to use in the new translation, otherwise, this parameter is ignored.
@access:	If IBV_REREG_MR_CHANGE_ACCESS is set in flags, this field specifies the new memory access rights, otherwise, this parameter is ignored. Could be one of the following: IBV_ACCESS_LOCAL_WRITE IBV_ACCESS_REMOTE_WRITE IBV_ACCESS_REMOTE_READ IBV_ACCESS_ALLOCATE_MR /* Let the library allocate the memory for * the user, tries to get contiguous pages */
@attr:	Future extensions

ibv_rereg_mr returns 0 on success, or the value of an errno on failure (which indicates the error reason). In case of an error, the MR is in undefined state. The user needs to call ibv_dereg_mr in order to release it.

Please note that if the MR (Memory Region) is created as a Shared MR and a translation is requested, after the call, the MR is no longer a shared MR. Moreover, Re-registration of MRs that uses NVIDIA PeerDirect™ technology are not supported.

14.6.2.1.6.2 Memory Window

Memory Window allows the application to have a more flexible control over remote access to its memory. It is available only on physical functions/native machines The two types of Memory Windows supported are: type 1 and type 2B.
Memory Windows are intended for situations where the application wants to:

- Grant and revoke remote access rights to a registered region in a dynamic fashion with less of a performance penalty
- Grant different remote access rights to different remote agents and/or grant those rights over different ranges within registered region

For further information, please refer to the InfiniBand specification document.

> ⚠ Memory Windows API cannot co-work with peer memory clients (PeerDirect).

Query Capabilities

Memory Windows are available if and only the hardware supports it. To verify whether Memory Windows are available, run `ibv_query_device`.
For example:

```
struct ibv_device_attr device_attr = {.comp_mask = IBV_DEVICE_ATTR_RESERVED - 1};
ibv_query_device(context, & device_attr);
if (device_attr.exp_device_cap_flags & IBV_DEVICE_MEM_WINDOW ||
                 device_attr.exp_device_cap_flags & IBV_DEVICE_MW_TYPE_2B) {
/* Memory window is supported */
```

Memory Window Allocation

Allocating memory window is done by calling the `ibv_alloc_mw` verb.

```
type_mw = IBV_MW_TYPE_2/ IBV_MW_TYPE_1
mw = ibv_alloc_mw(pd, type_mw);
```

Binding Memory Windows

After being allocated, memory window should be bound to a registered memory region. Memory Region should have been registered using the IBV_ACCESS_MW_BIND access flag.

For further information on how to bind memory windows, please see rdma-core man page.

Invalidating Memory Window

Before rebinding Memory Window type 2, it must be invalidated using `ibv_post_send` - see here.

Deallocating Memory Window

Deallocating memory window is done using the ibv_dealloc_mw verb.

```
ibv_dealloc_mw(mw);
```

14.6.2.1.6.3 User-Mode Memory Registration (UMR)

User-mode Memory Registration (UMR) is a fast registration mode which uses send queue. The UMR support enables the usage of RDMA operations and scatters the data at the remote side through the definition of appropriate memory keys on the remote side.
UMR enables the user to:

- Create indirect memory keys from previously registered memory regions, including creation of KLM's from previous KLM's. There are not data alignment or length restrictions associated with the memory regions used to define the new KLM's.
- Create memory regions, which support the definition of regular non-contiguous memory regions.

14.6.2.1.6.4 On-Demand-Paging (ODP)

On-Demand-Paging (ODP) is a technique to alleviate much of the shortcomings of memory registration. Applications no longer need to pin down the underlying physical pages of the address space, and track the validity of the mappings. Rather, the HCA requests the latest translations from the OS when pages are not present, and the OS invalidates translations which are no longer valid due to either non-present pages or mapping changes. ODP does not support contiguous pages.
ODP can be further divided into 2 subclasses: Explicit and Implicit ODP.

- Explicit ODP
  In Explicit ODP, applications still register memory buffers for communication, but this operation is used to define access control for IO rather than pin-down the pages. ODP Memory Region (MR) does not need to have valid mappings at registration time.
- Implicit ODP
  In Implicit ODP, applications are provided with a special memory key that represents their complete address space. This all IO accesses referencing this key (subject to the access rights associated with the key) does not need to register any virtual address range.

Query Capabilities

On-Demand Paging is available if both the hardware and the kernel support it. To verify whether ODP is supported, run `ibv_query_device.`

For further information, please refer to the `ibv_query_device` manual page.

Registering ODP Explicit and Implicit MR

ODP Explicit MR is registered after allocating the necessary resources (e.g. PD, buffer), while ODP implicit MR registration provides an implicit lkey that represents the complete address space.

For further information, please refer to the `ibv_reg_mr` manual page.

De-registering ODP MR

ODP MR is deregistered the same way a regular MR is deregistered:

```
ibv_dereg_mr(mr);
```

Advice MR Verb

The driver can pre-fetch a given range of pages and map them for access from the HCA. The advice MR verb is applicable for ODP MRs only.
For further information, please refer to the `ibv_advise_mr` manual page.

ODP Statistics

To aid in debugging and performance measurements and tuning, ODP support includes an extensive set of statistics.

For further information, please refer to rdma-statistics manual page.

14.6.2.1.6.5 Inline-Receive

The HCA may write received data to the Receive CQE. Inline-Receive saves PCIe Read transaction since the HCA does not need to read the scatter list. Therefore, it improves performance in case of short receive-messages.

On poll CQ, the driver copies the received data from CQE to the user's buffers.

Inline-Receive is enabled by default and is transparent to the user application. To disable it globally, set MLX5_SCATTER_TO_CQE environment variable to the value of 0. Otherwise, disable it on a specific QP using mlx5dv_create_qp() with MLX5DV_QP_CREATE_DISABLE_SCATTER_TO_CQE.

For further information, please refer to the manual page of mlx5dv_create_qp().

14.6.2.1.7 NVIDIA PeerDirect

NVIDIA PeerDirect™ uses an API between IB CORE and peer memory clients, (e.g. GPU cards) to provide access to an HCA to read/write peer memory for data buffers. As a result, it allows RDMA-based (over InfiniBand/RoCE) application to use peer device computing power, and RDMA interconnect at the same time without copying the data between the P2P devices.

For example, PeerDirect is being used for GPUDirect RDMA.

Detailed description for that API exists under MLNX OFED installation, please see `docs/readme_and_user_manual/PEER_MEMORY_API.txt`.

14.6.2.1.7.1 PeerDirect Async

Mellanox PeerDirect Async sub-system gives PeerDirect hardware devices, such as GPU cards, dedicated AS accelerators, and so on, the ability to take control over HCA in critical path offloading CPU. To achieve this, there is a set of verb calls and structures providing application with abstract description of operation sequences intended to be executed by peer device.

14.6.2.1.7.2 Relaxed Ordering (RSYNC)

> ⚠ This feature is only supported on ConnectX-5 adapter cards and above.

In GPU systems with relaxed ordering, RSYNC callback will be invoked to ensure memory consistency. The registration and implementation of the callback will be done using an external module provided by the system vendor. Loading the module will register the callback in MLNX_OFED to be used later to guarantee memory operations order.

14.6.2.1.8 CPU Overhead Distribution

When creating a CQ using the `ibv_create_cq()` API, a "`comp_vector`" argument is sent. If the value set for this argument is 0, while the CPU core executing this verb is not equal to zero, the driver assigns a completion EQ with the least CQs reporting to it. This method is used to distribute CQs amongst available completions EQ. To assign a CQ to a specific EQ, the EQ needs to be specified in the `comp_vector` argument.

14.6.2.1.9 Out-of-Order (OOO) Data Placement

⚠ This feature is only supported on:

- ConnectX-5 adapter cards and above
- RC and XRC QPs
- DC transport

14.6.2.1.9.1 Overview

In certain fabric configurations, InfiniBand packets for a given QP may take up different paths in a network from source to destination. This results into packets being received in an out-of-order manner. These packets can now be handled instead of being dropped, in order to avoid retransmission, by:

- Achieving better network utilization
- Decreasing latency

Data will be placed into host memory in an out-of-order manner when out-of-order messages are received.

For information on how to set up out-of-order processing by the QP, please refer to HowTo Configure Adaptive Routing and SHIELD Community post.

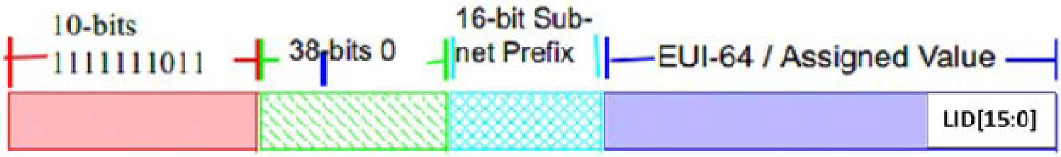
14.6.2.1.10 IB Router

IB router provides the ability to send traffic between two or more IB subnets thereby potentially expanding the size of the network to over 40k end-ports, enabling separation and fault resilience between islands and IB subnets, and enabling connection to different topologies used by different subnets.
The forwarding between the IB subnets is performed using GRH lookup. The IB router's basic functionality includes:

- Removal of current L2 LRH (local routing header)
- Routing
- table lookup - using GID from GRH
- Building new LRH according to the destination according to the routing table

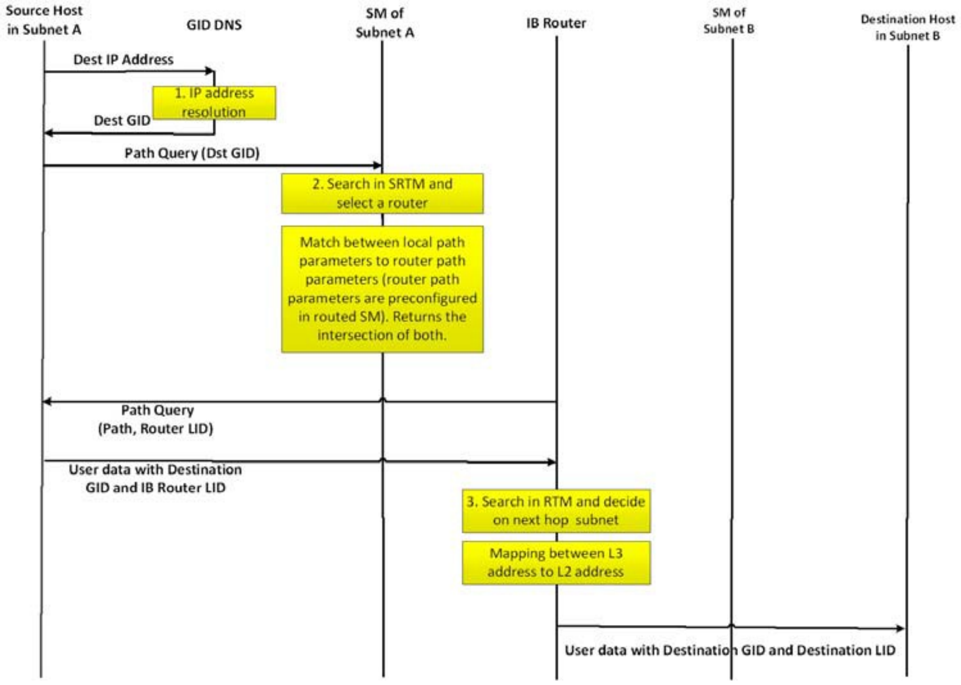
The DLID in the new LRH is built using simplified GID-to-LID mapping (where LID = 16 LSB bits of GID) thereby not requiring to send for ARP query/lookup.

Local Unicast GID Format



For this to work, the SM allocates an alias GID for each host in the fabric where the alias GID = {subnet prefix[127:64], reserved[63:16], LID[15:0}. Hosts should use alias GIDs in order to transmit traffic to peers on remote subnets.

Host-to-Host IB Router Unicast Flow


Source Host in Subnet A
GID DNS
SM of Subnet A
IB Router
SM of Subnet B
Destination Host in Subnet B
Dest IP Address
1. IP address resolution
Dest GID
Path Query (Dst GID)
2. Search in SRTM and select a router
Match between local path parameters to router path parameters (router path parameters are preconfigured in routed SM). Returns the intersection of both.
Path Query (Path, Router LID)
User data with Destination GID and IB Router LID
3. Search in RTM and decide on next hop subnet
Mapping between L3 address to L2 address
User data with Destination GID and Destination LID

- For information on the architecture and functionality of IB Router, refer to IB Router Architecture and Functionality Community post.
- For information on IB Router configuration, refer to HowTo Configure IB Routers Community post.

14.6.2.1.11 MAD Congestion Control

The SA Management Datagrams (MAD) are General Management Packets (GMP) used to communicate with the SA entity within the InfiniBand subnet. SA is normally part of the subnet manager, and it is contained within a single active instance. Therefore, congestion on the SA communication level may occur.

Congestion control is done by allowing max_outstanding MADs only, where outstanding MAD means that is has no response yet. It also holds a FIFO queue that holds the SA MADs that their sending is delayed due to max_outstanding overflow.

The length of the queue is queue_size and meant to limit the FIFO growth beyond the machine

memory capabilities. When the FIFO is full, SA MADs will be dropped, and the drops counter will increment accordingly.
When time expires (time_sa_mad) for a MAD in the queue, it will be removed from the queue and the user will be notified of the item expiration.
This features is implemented per CA port.
The SA MAD congestion control values are configurable using the following sysfs entries:

```
/sys/class/infiniband/mlx5_0/mad_sa_cc/
    1
      drops
      max_outstanding
      queue_size
      time_sa_mad
    2
    drops
    max_outstanding
    queue_size
    time_sa_mad
```

To print the current value:

```
cat  /sys/class/infiniband/mlx5_0/mad_sa_cc/1/max_outstanding 16
```

To change the current value:

```
echo 32 > /sys/class/infiniband/mlx5_0/mad_sa_cc/1/max_outstanding
cat      /sys/class/infiniband/mlx5_0/mad_sa_cc/1/max_outstanding
32
```

To reset the drops counter:

```
echo 0 > /sys/class/infiniband/mlx5_0/mad_sa_cc/1/drops
```

Parameters' Valid Ranges

Parameter	Range		Default Values
	MIN	MAX	
max_oustanding	1	2^20	16
queue_size	16	2^20	16
time_sa_mad	1 milliseconds	10000	20 milliseconds

14.6.2.2 Storage Protocols

There are several storage protocols that use the advantage of InfiniBand and RDMA for performance reasons (high throughput, low latency and low CPU utilization). In this chapter we will discuss the following protocols:

- SCSI RDMA Protocol (SRP) is designed to take full advantage of the protocol off-load and RDMA features provided by the InfiniBand architecture.
- iSCSI Extensions for RDMA (iSER) is an extension of the data transfer model of iSCSI, a storage networking standard for TCP/IP. It uses the iSCSI components while taking the advantage of the RDMA protocol suite. ISER is implemented on various stor- age targets such as TGT, LIO, SCST and out of scope of this manual.
  For various ISER targets configuration steps, troubleshooting and debugging, as well as other implementation of storage protocols over RDMA (such as Ceph over RDMA, nbdX and more) refer to Storage Solutions on the Community website.
- Lustre is an open-source, parallel distributed file system, generally used for large-scale cluster computing that supports many requirements of leadership class HPC simulation environments.
- NVM Express™ over Fabrics (NVME-oF)
  - NVME-oF is a technology specification for networking storage designed to enable NVMe message-based commands to transfer data between a host computer and a target solid-state storage device or system over a network such as Ethernet, Fibre Channel, and InfiniBand. Tunneling NVMe commands through an RDMA fabric provides a high throughput and a low latency. This is an alternative to the SCSi based storage networking protocols.
  - NVME-oF Target Offload is an implementation of the new NVME-oF standard Target (server) side in hardware. Starting from ConnectX-5 family cards, all regular IO requests can be processed by the HCA, with the HCA sending IO requests directly to a real NVMe PCI device, using peer-to-peer PCI communications. This means that excluding connection management and error flows, no CPU utilization will be observed during NVME-oF traffic.
    For further information, please refer to Storage Solutions on the Community website (enterprise-support.nvidia.com/s/).

14.6.2.2.1 SRP - SCSI RDMA Protocol

The SCSI RDMA Protocol (SRP) is designed to take full advantage of the protocol offload and RDMA features provided by the InfiniBand architecture. SRP allows a large body of SCSI software to be readily used on InfiniBand architecture. The SRP Initiator controls the connection to an SRP Target in order to provide access to remote storage devices across an InfiniBand fabric. The kSRP Target resides in an IO unit and provides storage services.

14.6.2.2.1.1 SRP Initiator

This SRP Initiator is based on open source from OpenFabrics (www.openfabrics.org) that implements the SCSI RDMA Protocol-2 (SRP-2). SRP-2 is described in Document # T10/1524-D available from http://www.t10.org.

The SRP Initiator supports

- Basic SCSI Primary Commands -3 (SPC-3)
  (www.t10.org/ftp/t10/drafts/spc3/spc3r21b.pdf)
- Basic SCSI Block Commands -2 (SBC-2)
  (www.t10.org/ftp/t10/drafts/sbc2/sbc2r16.pdf)

- Basic functionality, task management and limited error handling

⚠ This package, however, does not include an SRP Target.

Loading SRP Initiator

➤ *To load the SRP module either:*

- Execute the `modprobe ib_srp` command after the OFED driver is up.

or

1. Change the value of `SRP_LOAD` in `/etc/infiniband/openib.conf` to “`yes`”.
2. Run `/etc/init.d/openibd restart` for the changes to take effect.

⚠ When loading the `ib_srp` module, it is possible to set the module parameter `srp_sg_tablesize.` This is the maximum number of gather/scatter entries per I/O (default: 12).

SRP Module Parameters

When loading the SRP module, the following parameters can be set (viewable by the "modinfo ib_srp" command):

`cmd_sg_entries`	Default number of gather/scatter entries in the SRP command (default is 12, max 255)
`allow_ext_sg`	Default behavior when there are more than cmd_sg_entries S/G entries after mapping; fails the request when false (default false)
`topspin_workarounds`	Enable workarounds for Topspin/Cisco SRP target bugs
`reconnect_delay`	Time between successive reconnect attempts. Time between successive reconnect attempts of SRP initiator to a disconnected target until dev_loss_tmo timer expires (if enabled), after that the SCSI target will be removed
`fast_io_fail_tmo`	Number of seconds between the observation of a transport layer error and failing all I/O. Increasing this timeout allows more tolerance to transport errors, however, doing so increases the total failover time in case of serious transport failure. Note: fast_io_fail_tmo value must be smaller than the value of recon- nect_delay
`dev_loss_tmo`	Maximum number of seconds that the SRP transport should insulate transport layer errors. After this time has been exceeded the SCSI target is removed. Normally it is advised to set this to -1 (disabled) which will never remove the scsi_host. In deployments where different SRP targets are connected and disconnected frequently, it may be required to enable this timeout in order to clean old scsi_hosts representing targets that no longer exists

Constraints between parameters:

- dev_loss_tmo, fast_io_fail_tmo, reconnect_delay cannot be all disabled or negative values.
- reconnect_delay must be positive number.

- fast_io_fail_tmo must be smaller than SCSI block device timeout.
- fast_io_fail_tmo must be smaller than dev_loss_tmo.

SRP Remote Ports Parameters

Several SRP remote ports parameters are modifiable online on existing connection.

➤ *To modify dev_loss_tmo to 600 seconds:*

```
echo 600 > /sys/class/srp_remote_ports/port-xxx/dev_loss_tmo
```

➤ *To modify fast_io_fail_tmo to 15 seconds:*

```
echo 15 > /sys/class/srp_remote_ports/port-xxx/fast_io_fail_tmo
```

➤ *To modify reconnect_delay to 10 seconds:*

```
echo 20 > /sys/class/srp_remote_ports/port-xxx/reconnect_delay
```

Manually Establishing an SRP Connection

The following steps describe how to manually load an SRP connection between the Initiator and an SRP Target. “Automatic Discovery and Connection to Targets” section explains how to do this automatically.

- Make sure that the ib_srp module is loaded, the SRP Initiator is reachable by the SRP Target, and that an SM is running.
- To establish a connection with an SRP Target and create an SRP (SCSI) device for that target under /dev, use the following command:

  ```
  echo -n id_ext=[GUID value],ioc_guid=[GUID value],dgid=[port GID value],\
  pkey=ffff,service_id=[service[0] value] > \
  /sys/class/infiniband_srp/srp-mlx[hca number]-[port number]/add_target
  ```

See “SRP Tools - ibsrpdm, srp_daemon and srpd Service Script” section for instructions on how the parameters in this echo command may be obtained.

Notes:

- Execution of the above “echo” command may take some time
- The SM must be running while the command executes
- It is possible to include additional parameters in the echo command:
  - max_cmd_per_lun - Default: 62
  - max_sect (short for max_sectors) - sets the request size of a command
  - io_class - Default: 0x100 as in rev 16A of the specification (In rev 10 the default was 0xff00)
  - tl_retry_count - a number in the range 2..7 specifying the IB RC retry count. Default: 2
  - comp_vector, a number in the range 0..n-1 specifying the MSI-X completion vector. Some HCA's allocate multiple (n) MSI-X vectors per HCA port. If the IRQ affinity masks of these interrupts have been configured such that each MSI-X interrupt is handled by a

  different CPU then the comp_vector parameter can be used to spread the SRP completion workload over multiple CPU's.
  - cmd_sg_entries, a number in the range 1..255 that specifies the maximum number of data buffer descriptors stored in the SRP_CMD information unit itself. With allow_ext_sg=0 the parameter cmd_sg_entries defines the maximum S/G list length for a single SRP_CMD, and commands whose S/G list length exceeds this limit after S/G list collapsing will fail.
  - initiator_ext - see "Multiple Connections from Initiator InfiniBand Port to the Target" section.
- To list the new SCSI devices that have been added by the echo command, you may use either of the following two methods:
  - Execute “fdisk -l”. This command lists all devices; the new devices are included in this listing.
  - Execute “dmesg” or look at /var/log/messages to find messages with the names of the new devices.

SRP sysfs Parameters

Interface for making ib_srp connect to a new target. One can request ib_srp to connect to a new target by writing a comma-separated list of login parameters to this sysfs attribute. The supported parameters are:

id_ext	A 16-digit hexadecimal number specifying the eight byte identifier extension in the 16-byte SRP target port identifier. The target port identifier is sent by ib_srp to the target in the SRP_LOGIN_REQ request.
ioc_guid	A 16-digit hexadecimal number specifying the eight byte I/O controller GUID portion of the 16-byte target port identifier.
dgid	A 32-digit hexadecimal number specifying the destination GID.
pkey	A four-digit hexadecimal number specifying the InfiniBand partition key.
service_id	A 16-digit hexadecimal number specifying the InfiniBand service ID used to establish communication with the SRP target. How to find out the value of the service ID is specified in the documentation of the SRP target.
max_sect	A decimal number specifying the maximum number of 512-byte sectors to be transferred via a single SCSI command.
max_cmd_per_lun	A decimal number specifying the maximum number of outstanding commands for a single LUN.
io_class	A hexadecimal number specifying the SRP I/O class. Must be either 0xff00 (rev 10) or 0x0100 (rev 16a). The I/O class defines the format of the SRP initiator and target port identifiers.
initiator_ext	A 16-digit hexadecimal number specifying the identifier extension portion of the SRP initiator port identifier. This data is sent by the initiator to the target in the SRP_LOGIN_REQ request.
cmd_sg_entries	A number in the range 1..255 that specifies the maximum number of data buffer descriptors stored in the SRP_CMD information unit itself. With allow_ext_sg=0 the parameter cmd_sg_entries defines the maxi- mum S/G list length for a single SRP_CMD, and commands whose S/G list length exceeds this limit after S/G list collapsing will fail.
allow_ext_sg	Whether ib_srp is allowed to include a partial memory descriptor list in an SRP_CMD instead of the entire list. If a partial memory descriptor list has been included in an SRP_CMD the remaining memory descriptors are communicated from initiator to target via an additional RDMA transfer. Setting allow_ext_sg to 1 increases the maximum amount of data that can be transferred between initiator and target via a single SCSI command. Since not all SRP target implementations support partial memory descriptor lists the default value for this option is 0.

sg_tablesize	A number in the range 1..2048 specifying the maximum S/G list length the SCSI layer is allowed to pass to ib_srp. Specifying a value that exceeds cmd_sg_entries is only safe with partial memory descriptor list support enabled (allow_ext_sg=1).
comp_vector	A number in the range 0..n-1 specifying the MSI-X completion vector. Some HCA's allocate multiple (n) MSI-X vectors per HCA port. If the IRQ affinity masks of these interrupts have been configured such that each MSI-X interrupt is handled by a different CPU then the comp_vector parameter can be used to spread the SRP completion workload over multiple CPU's.
tl_retry_count	A number in the range 2..7 specifying the IB RC retry count.

SRP Tools - ibsrpdm, srp_daemon and srpd Service Script

The OFED distribution provides two utilities: ibsrpdm and srp_daemon:

- They detect targets on the fabric reachable by the Initiator (Step 1)
- Output target attributes in a format suitable for use in the above “echo” command (Step 2)
- A service script srpd which may be started at stack startup

The utilities can be found under /usr/sbin/, and are part of the srptools RPM that may be installed using the OFED installation. Detailed information regarding the various options for these utilities are provided by their man pages.
Below, several usage scenarios for these utilities are presented.

ibsrpdm

ibsrpdm has the following tasks:

1. Detecting reachable targets.
    a. To detect all targets reachable by the SRP initiator via the default umad device (/sys/class/infiniband_mad/umad0), execute the following command:

    ```
    ibsrpdm
    ```

    This command will result into readable output information on each SRP Target detected. Sample:

    ```
         IO Unit Info:
              port LID:        0103
              port GID:        fe800000000000000002c90200402bd5
              change ID:       0002
              max controllers: 0x10
           controller[  1]
               GUID:      0002c90200402bd4
               vendor ID: 0002c9
               device ID: 005a44
               IO class : 0100
               ID:        LSI Storage Systems SRP Driver 200400a0b81146a1
               service entries: 1
               service[  0]: 200400a0b81146a1 / SRP.T10:200400A0B81146A1
    ```

    b. To detect all the SRP Targets reachable by the SRP Initiator via another umad device, use the following command:

    ```
    ibsrpdm -d <umad device>
    ```

2. Assisting in SRP connection creation.
    a. To generate an output suitable for utilization in the “echo” command in “Manually Establishing an SRP Connection” section, add the ‘-c’ option to ibsrpdm:

```
ibsrpdm -c
```

Sample output:

```
id_ext=200400A0B81146A1,ioc_guid=0002c90200402bd4,
dgid=fe800000000000000002c90200402bd5,pkey=ffff,service_id=200400a0b81146a1
```

b. To establish a connection with an SRP Target using the output from the ‘ibsrpdm -c’ example above, execute the following command:

```
echo -n id_ext=200400A0B81146A1,ioc_guid=0002c90200402bd4,
dgid=fe800000000000000002c90200402bd5,pkey=ffff,service_id=200400a0b81146a1 > /sys/
class/infiniband_srp/srp-mlx5_0-1/add_target
```

The SRP connection should now be up; the newly created SCSI devices should appear in the listing obtained from the ‘`fdisk -l`’ command.

3. Discover reachable SRP Targets given an InfiniBand HCA name and port, rather than by just running `/sys/class/infiniband_mad/umad<N>` where `<N>` is a digit.

srpd

The srpd service script allows automatic activation and termination of the srp_daemon utility on all system live InfiniBand ports.

srp_daemon

srp_daemon utility is based on ibsrpdm and extends its functionality. In addition to the ibsrpdm functionality described above, srp_daemon can:

- Establish an SRP connection by itself (without the need to issue the “echo” command described in “Manually Establishing an SRP Connection” section)
- Continue running in background, detecting new targets and establishing SRP connections with them (daemon mode)
- Discover reachable SRP Targets given an infiniband HCA name and port, rather than just by `/dev/umad<N>` where `<N>` is a digit
- Enable High Availability operation (together with Device-Mapper Multipath)
- Have a configuration file that determines the targets to connect to:

1. srp_daemon commands equivalent to ibsrpdm:

```
"srp_daemon -a -o" is equivalent to "ibsrpdm"
"srp_daemon -c -a -o" is equivalent to "ibsrpdm -c"
```

Note: These srp_daemon commands can behave differently than the equivalent ibsrpdm command when /etc/srp_daemon.conf is not empty.

2. srp_daemon extensions to ibsrpdm.

    - To discover SRP Targets reachable from the HCA device <InfiniBand HCA name> and the port <port num>, (and to generate output suitable for 'echo'), execute:

```
host1# srp_daemon -c -a -o -i <InfiniBand HCA name> -p <port number>
```

Note: To obtain the list of InfiniBand HCA device names, you can either use the ibstat tool or run ‘ls /sys/class/infiniband’.

- To both discover the SRP Targets and establish connections with them, just add the -e option to the above command.
- Executing srp_daemon over a port without the -a option will only display the reachable targets via the port and to which the initiator is not connected. If executing with the -e option it is better to omit -a.
- It is recommended to use the -n option. This option adds the initiator_ext to the connecting string (see "Multiple Connections from Initiator InfiniBand Port to the Target" section).
- srp_daemon has a configuration file that can be set, where the default is /etc/srp_daemon.conf. Use the -f to supply a different configuration file that configures the targets srp_daemon is allowed to connect to. The configuration file can also be used to set values for additional parameters (e.g., max_cmd_per_lun, max_sect).
- A continuous background (daemon) operation, providing an automatic ongoing detection and connection capability. See "Automatic Discovery and Connection to Targets" section.

Automatic Discovery and Connection to Targets

- Make sure the ib_srp module is loaded, the SRP Initiator can reach an SRP Target, and that an SM is running.
- To connect to all the existing Targets in the fabric, run "`srp_daemon -e -o`". This utility will scan the fabric once, connect to every Target it detects, and then exit.

  > ⚠ srp_daemon will follow the configuration it finds in /etc/srp_daemon.conf. Thus, it will ignore a target that is disallowed in the configuration file.

- To connect to all the existing Targets in the fabric and to connect to new targets that will join the fabric, execute srp_daemon -e. This utility continues to execute until it is either killed by the user or encounters connection errors (such as no SM in the fabric).
- To execute SRP daemon as a daemon on all the ports:
  - srp_daemon.sh (found under /usr/sbin/). srp_daemon.sh sends its log to /var/log/srp_daemon.log.
  - Start the srpd service script, run service srpd start

For the changes in openib.conf to take effect, run:

```
/etc/init.d/openibd restart
```

Multiple Connections from Initiator InfiniBand Port to the Target

Some system configurations may need multiple SRP connections from the SRP Initiator to the same SRP Target: to the same Target IB port, or to different IB ports on the same Target HCA.
In case of a single Target IB port, i.e., SRP connections use the same path, the configuration is enabled using a different initiator_ext value for each SRP connection. The initiator_ext value is a 16-hexadecimal-digit value specified in the connection command.

Also in case of two physical connections (i.e., network paths) from a single initiator IB port to two different IB ports on the same Target HCA, there is need for a different initiator_ext value on each path. The conventions is to use the Target port GUID as the initiator_ext value for the relevant path.

If you use srp_daemon with -n flag, it automatically assigns initiator_ext values according to this convention. For example:

```
id_ext=200500A0B81146A1,ioc_guid=0002c90200402bec,\
dgid=fe800000000000000002c90200402bed,pkey=ffff,\ service_id=200500a0b81146a1,initiator_ext=ed2b400002c90200
```

Notes:

- It is recommended to use the -n flag for all srp_daemon invocations.
- ibsrpdm does not have a corresponding option.
- srp_daemon.sh always uses the -n option (whether invoked manually by the user, or automatically at startup by setting SRP_DAEMON_ENABLE to yes).

High Availability (HA)

High Availability works using the Device-Mapper (DM) multipath and the SRP daemon. Each initiator is connected to the same target from several ports/HCAs. The DM multipath is responsible for joining together different paths to the same target and for failover between paths when one of them goes offline. Multipath will be executed on newly joined SCSI devices.
Each initiator should execute several instances of the SRP daemon, one for each port. At startup, each SRP daemon detects the SRP Targets in the fabric and sends requests to the ib_srp module to connect to each of them. These SRP daemons also detect targets that subsequently join the fabric, and send the ib_srp module requests to connect to them as well.

Operation

When a path (from port1) to a target fails, the ib_srp module starts an error recovery process. If this process gets to the reset_host stage and there is no path to the target from this port, ib_srp will remove this scsi_host. After the scsi_host is removed, multipath switches to another path to this target (from another port/HCA).
When the failed path recovers, it will be detected by the SRP daemon. The SRP daemon will then request ib_srp to connect to this target. Once the connection is up, there will be a new scsi_host for this target. Multipath will be executed on the devices of this host, returning to the original state (prior to the failed path).

Manual Activation of High Availability

Initialization - execute after each boot of the driver:

1. Execute `modprobe dm-multipath`
2. Execute `modprobe ib-srp`
3. Make sure you have created file /etc/udev/rules.d/91-srp.rules as described above
4. Execute for each port and each HCA:

   ```
   srp_daemon -c -e -R 300 -i <InfiniBand HCA name> -p <port number>
   ```

   This step can be performed by executing srp_daemon.sh, which sends its log to /var/log/srp_daemon.log.

Now it is possible to access the SRP LUNs on /dev/mapper/.

⚠ It is possible for regular (non-SRP) LUNs to also be present; the SRP LUNs may be identified by their names. You can configure the /etc/multipath.conf file to change multipath behavior.

⚠ It is also possible that the SRP LUNs will not appear under /dev/mapper/. This can occur if the SRP LUNs are in the black-list of multipath. Edit the ‘blacklist’ section in /etc/multipath.conf and make sure the SRP LUNs are not blacklisted.

Automatic Activation of High Availability

- Start srpd service, run:

  ```
  service srpd start
  ```

- From the next loading of the driver it will be possible to access the SRP LUNs on /dev/mapper/

  ⚠ It is possible that regular (not SRP) LUNs are also present. SRP LUNs may be identified by their name.

- It is possible to see the output of the SRP daemon in /var/log/srp_daemon.log

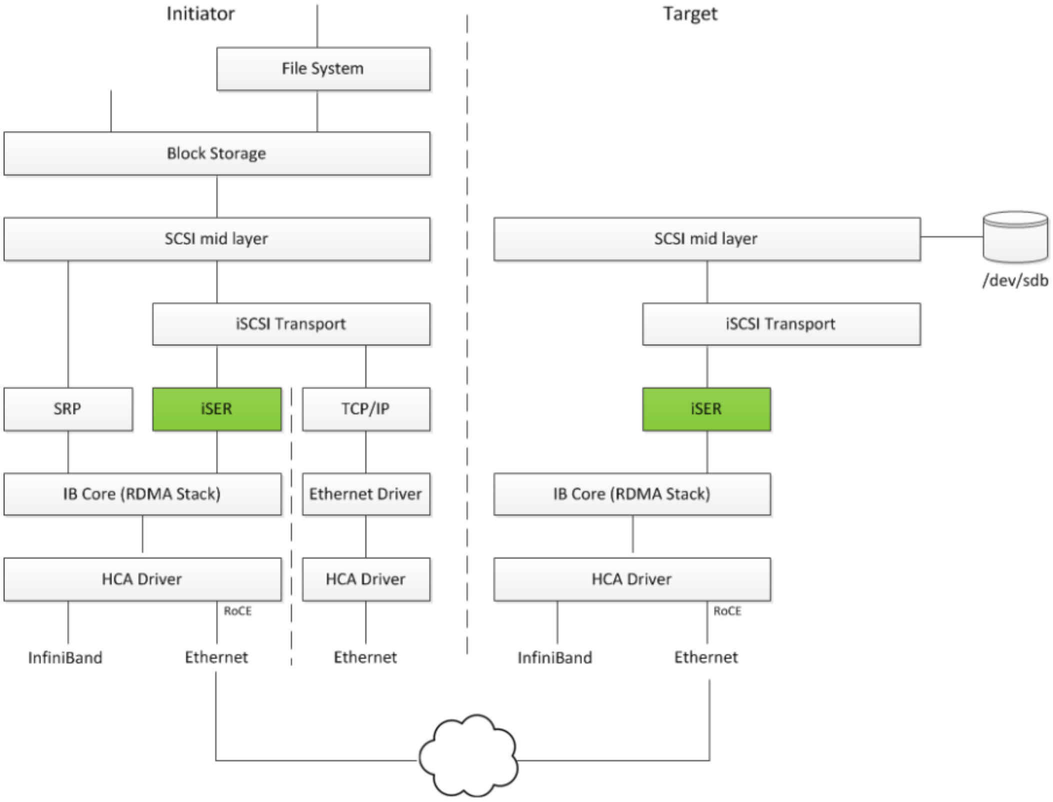
14.6.2.2.1.2 Shutting Down SRP

SRP can be shutdown by using “rmmod ib_srp”, or by stopping the OFED driver (“/etc/init.d/openibd stop”), or as a by-product of a complete system shutdown.
Prior to shutting down SRP, remove all references to it. The actions you need to take depend on the way SRP was loaded. There are three cases:

1. Without High Availability
   When working without High Availability, you should unmount the SRP partitions that were mounted prior to shutting down SRP.
2. After Manual Activation of High Availability
   If you manually activated SRP High Availability, perform the following steps:
   a. Unmount all SRP partitions that were mounted.
   b. Stop service srpd (Kill the SRP daemon instances).
   c. Make sure there are no multipath instances running. If there are multiple instances, wait for them to end or kill them.
   d. Run: `multipath -F`
3. After Automatic Activation of High Availability
   If SRP High Availability was automatically activated, SRP shutdown must be part of the driver shutdown ("/etc/init.d/openibd stop") which performs Steps 2-4 of case b above. However, you still have to unmount all SRP partitions that were mounted before driver shutdown.

14.6.2.2.2 iSCSI Extensions for RDMA (iSER)

iSCSI Extensions for RDMA (iSER) extends the iSCSI protocol to RDMA. It permits data to be transferred directly into and out of SCSI buffers without intermediate data copies.
iSER uses the RDMA protocol suite to supply higher bandwidth for block storage transfers (zero time copy behavior). To that fact, it eliminates the TCP/IP processing overhead while preserving the compatibility with iSCSI protocol.



There are three target implementation of ISER:

- Linux SCSI target framework (tgt)
- Linux-IO target (LIO)
- Generic SCSI target subsystem for Linux (SCST)

Each one of those targets can work in TCP or iSER transport modes.
iSER also supports RoCE without any additional configuration required. To bond the RoCE interfaces, set the fail_over_mac option in the bonding driver (see "Bonding IPoIB").
RDMA/RoCE is located below the iSER block on the network stack. In order to run iSER, the RDMA layer should be configured and validated (over Ethernet or InfiniBand). For troubleshooting RDMA, please refer to "HowTo Enable, Verify and Troubleshoot RDMA" on the Community website.

14.6.2.2.2.1 iSER Initiator

The iSER initiator is controlled through the iSCSI interface available from the iscsi-initiator-utils package.

To discover and log into iSCSI targets, as well as access and manage the open-iscsi database use the `iscasiadm` utility, a command-line tool.

To enable iSER as a transport protocol use "`I iser`" as a parameter of the `iscasiadm` command.

Example for discovering and connecting targets over iSER:

```
iscsiadm -m discovery -o new -o old -t st -I iser -p <ip:port> -l
```

Note that the target implementation (e.g. LIO, SCST, TGT) does not affect he initiator process and configuration.

14.6.2.2.2.2 iSER Targets

> Setting the iSER target is out of scope of this manual. For guidelines of how to do so, please refer to the relevant target documentation (e.g. stgt, targetcli).

Targets settings such as timeouts and retries are set the same as any other iSCSI targets.

> If targets are set to auto connect on boot, and targets are unreachable, it may take a long time to continue the boot process if timeouts and max retries are set too high.

For various configuration, troubleshooting and debugging examples, refer to Storage Solutions on the Community website.

14.6.2.2.3 Lustre

Lustre is an open-source, parallel distributed file system, generally used for large-scale cluster computing that supports many requirements of leadership class HPC simulation environments.

Lustre Compilation for MLNX_OFED:

> This procedure applies to RHEL/SLES OSs supported by Lustre. For further information, please refer to Lustre Release Notes.

➢ *To compile Lustre version 2.4.0 and higher:*

```
$ ./configure --with-o2ib=/usr/src/ofa_kernel/default/
$ make rpms
```

➢ *To compile older Lustre versions:*

```
$ EXTRA_LNET_INCLUDE="-I/usr/src/ofa_kernel/default/include/ -include /usr/src/ofa_kernel/default/include/linux/
compat-2.6.h" ./configure --with-o2ib=/usr/src/ofa_kernel/default/
$ EXTRA_LNET_INCLUDE="-I/usr/src/ofa_kernel/default/include/ -include /usr/src/ofa_kernel/default/include/linux/
compat-2.6.h" make rpms
```

For full installation example, refer to HowTo Install NVIDIA OFED driver for Lustre Community post.

14.6.2.2.4 NVME-oF - NVM Express over Fabrics

14.6.2.2.4.1 NVME-oF

NVME-oF enables NVMe message-based commands to transfer data between a host computer and a target solid-state storage device or system over a network such as Ethernet, Fibre Channel, and InfiniBand. Tunneling NVMe commands through an RDMA fabric provides a high throughput and a low latency.
For information on how to configure NVME-oF, please refer to the HowTo Configure NVMe over Fabrics Community post.

> ⚠ The --with-nvmf installation option should not be specified, if nvme-tcp kernel module is used. In this case, the native Inbox nvme-tcp kernel module will be loaded.

14.6.2.2.4.2 NVME-oF Target Offload

> ⚠ This feature is only supported for ConnectX-5 adapter cards family and above.

NVME-oF Target Offload is an implementation of the new NVME-oF standard Target (server) side in hardware. Starting from ConnectX-5 family cards, all regular IO requests can be processed by the HCA, with the HCA sending IO requests directly to a real NVMe PCI device, using peer-to-peer PCI communications. This means that excluding connection management and error flows, no CPU utilization will be observed during NVME-oF traffic.

- For instructions on how to configure NVME-oF target offload, refer to HowTo Configure NVME-oF Target Offload Community post.
- For instructions on how to verify that NVME-oF target offload is working properly, refer to Simple NVMe-oF Target Offload Benchmark Community post.

14.6.2.3 Virtualization

The chapter contains the following sections:

- Single Root IO Virtualization (SR-IOV)
- Enabling Paravirtualization
- VXLAN Hardware Stateless Offloads
- Q-in-Q Encapsulation per VF in Linux (VST)
- 802.1Q Double-Tagging
- Scalable Functions

14.6.2.3.1 Single Root IO Virtualization (SR-IOV)

Single Root IO Virtualization (SR-IOV) is a technology that allows a physical PCIe device to present itself multiple times through the PCIe bus. This technology enables multiple virtual instances of the device with separate resources. NVIDIA adapters are capable of exposing up to 127 virtual instances (Virtual Functions (VFs) for each port in the NVIDIA ConnectX® family cards. These virtual functions can then be provisioned separately. Each VF can be seen as an additional device connected to the

Physical Function. It shares the same resources with the Physical Function, and its number of ports equals those of the Physical Function.

SR-IOV is commonly used in conjunction with an SR-IOV enabled hypervisor to provide virtual machines direct hardware access to network resources hence increasing its performance.

In this chapter we will demonstrate setup and configuration of SR-IOV in a Red Hat Linux environment using ConnectX® VPI adapter cards.

14.6.2.3.1.1 System Requirements

To set up an SR-IOV environment, the following is required:

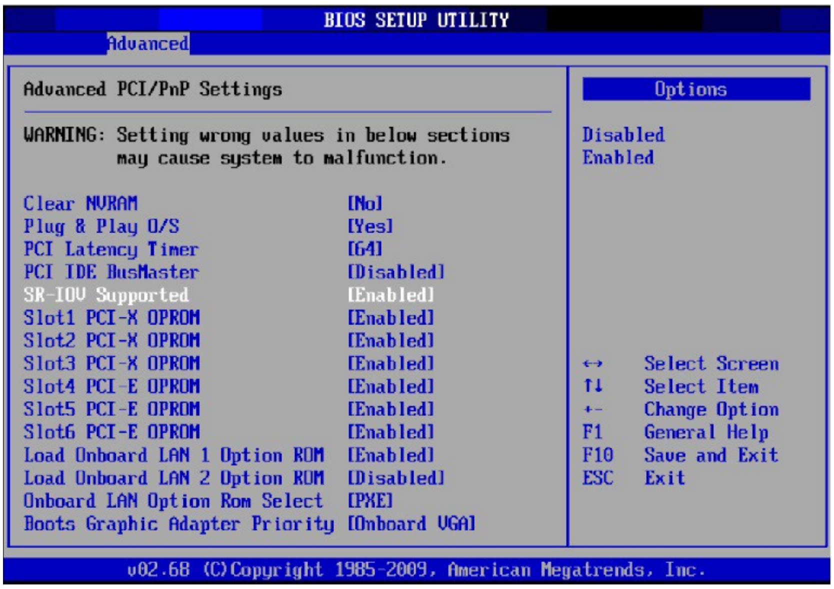
- MLNX_OFED Driver

- A server/blade with an SR-IOV-capable motherboard BIOS
- Hypervisor that supports SR-IOV such as: Red Hat Enterprise Linux Server Version 6
- NVIDIA ConnectX® VPI Adapter Card family with SR-IOV capability

14.6.2.3.1.2 Setting Up SR-IOV

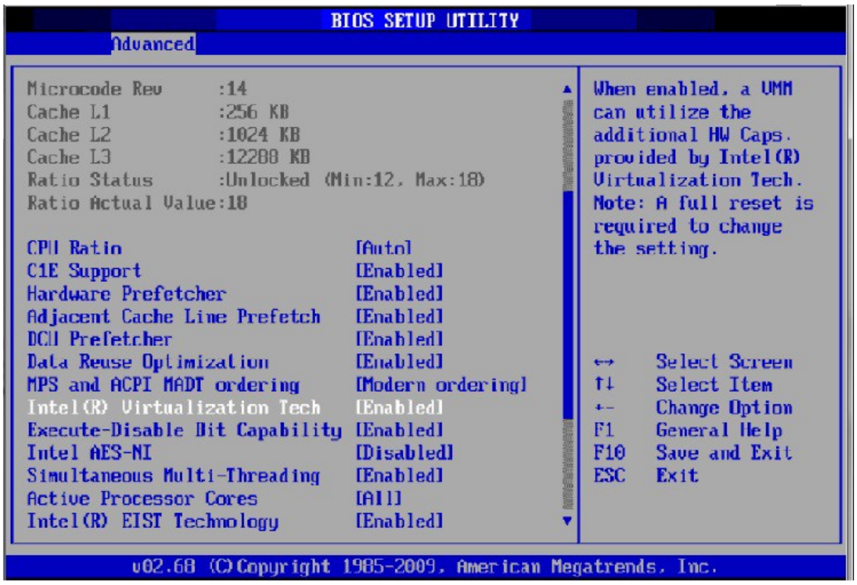
Depending on your system, perform the steps below to set up your BIOS. The figures used in this section are for illustration purposes only. For further information, please refer to the appropriate BIOS User Manual:

1. Enable "SR-IOV" in the system BIOS.



```
                         BIOS SETUP UTILITY
      Advanced

 Advanced PCI/PnP Settings                            Options

 WARNING: Setting wrong values in below sections      Disabled
          may cause system to malfunction.            Enabled

 Clear NVRAM                     [No]
 Plug & Play O/S                 [Yes]
 PCI Latency Timer               [64]
 PCI IDE BusMaster               [Disabled]
 SR-IOV Supported                [Enabled]
 Slot1 PCI-X OPROM               [Enabled]
 Slot2 PCI-X OPROM               [Enabled]
 Slot3 PCI-X OPROM               [Enabled]            ↔     Select Screen
 Slot4 PCI-E OPROM               [Enabled]            ↑↓    Select Item
 Slot5 PCI-E OPROM               [Enabled]            +-    Change Option
 Slot6 PCI-E OPROM               [Enabled]            F1    General Help
 Load Onboard LAN 1 Option ROM   [Enabled]            F10   Save and Exit
 Load Onboard LAN 2 Option ROM   [Disabled]           ESC   Exit
 Onboard LAN Option Rom Select   [PXE]
 Boots Graphic Adapter Priority  [Onboard VGA]

          v02.68 (C)Copyright 1985-2009, American Megatrends, Inc.
```

2. Enable "Intel Virtualization Technology".



```
                         BIOS SETUP UTILITY
     Advanced

  Microcode Rev       :14                                 When enabled, a VMM
  Cache L1            :256 KB                             can utilize the
  Cache L2            :1024 KB                            additional HW Caps.
  Cache L3            :12288 KB                           provided by Intel(R)
  Ratio Status        :Unlocked (Min:12, Max:18)          Virtualization Tech.
  Ratio Actual Value:18                                   Note: A full reset is
                                                          required to change
  CPU Ratio                         [Auto]                the setting.
  C1E Support                       [Enabled]
  Hardware Prefetcher               [Enabled]
  Adjacent Cache Line Prefetch      [Enabled]
  DCU Prefetcher                    [Enabled]
  Data Reuse Optimization           [Enabled]             ↔     Select Screen
  MPS and ACPI MADT ordering        [Modern ordering]     ↑↓    Select Item
  Intel(R) Virtualization Tech      [Enabled]             +-    Change Option
  Execute-Disable Bit Capability    [Enabled]             F1    General Help
  Intel AES-NI                      [Disabled]            F10   Save and Exit
  Simultaneous Multi-Threading      [Enabled]             ESC   Exit
  Active Processor Cores            [All]
  Intel(R) EIST Technology          [Enabled]

          v02.68 (C)Copyright 1985-2009, American Megatrends, Inc.
```

3. Install a hypervisor that supports SR-IOV.
4. Depending on your system, update the /boot/grub/grub.conf file to include a similar command line load parameter for the Linux kernel.
   For example, to Intel systems, add:

```
default=0
timeout=5
splashimage=(hd0,0)/grub/splash.xpm.gz
hiddenmenu
title Red Hat Enterprise Linux Server (4.x.x)
        root (hd0,0)
        kernel /vmlinuz-4.x.x ro root=/dev/VolGroup00/LogVol00 rhgb quiet
        intel_iommu=on        initrd /initrd-4.x.x.img
```

   Note: Please make sure the parameter "intel_iommu=on" exists when updating the /boot/grub/grub.conf file, otherwise SR-IOV cannot be loaded.
   Some OSs use /boot/grub2/grub.cfg file. If your server uses such file, please edit this file instead (add “intel_iommu=on” for the relevant menu entry at the end of the line that starts with "linux16").

14.6.2.3.1.3 Configuring SR-IOV (Ethernet)

To set SR-IOV in Ethernet mode, refer to HowTo Configure SR-IOV for ConnectX-4/ConnectX- 5/ConnectX-6 with KVM (Ethernet) Community Post.

14.6.2.3.1.4 Configuring SR-IOV (InfiniBand)

1. Install the MLNX_OFED driver for Linux that supports SR-IOV.
2. Check if SR-IOV is enabled in the firmware.

```
mlxconfig -d /dev/mst/mt4115_pciconf0 q

  Device #1:
  ----------

  Device type:    Connect4
  PCI device:     /dev/mst/mt4115_pciconf0
  Configurations:           Current
```

```
        SRIOV_EN                    1
        NUM_OF_VFS                  8
```

⚠ If needed, use mlxconfig to set the relevant fields:
`mlxconfig -d /dev/mst/mt4115_pciconf0 set SRIOV_EN=1 NUM_OF_VFS=16`

3. Reboot the server.
4. Write to the sysfs file the number of Virtual Functions you need to create for the PF. You can use one of the following equivalent files:
   You can use one of the following equivalent files:
   - A standard Linux kernel generated file that is available in the new kernels.

   ```
   echo [num_vfs] > /sys/class/infiniband/mlx5_0/device/sriov_numvfs
   ```

   Note: This file will be generated only if IOMMU is set in the grub.conf file (by adding intel_iommu=on, as seen in the fourth step under "Setting Up SR-IOV").
   - A file generated by the mlx5_core driver with the same functionality as the kernel generated one.

   ```
   echo [num_vfs] > /sys/class/infiniband/mlx5_0/device/mlx5_num_vfs
   ```

   Note: This file is used by old kernels that do not support the standard file. In such kernels, using sriov_numvfs results in the following error: "bash: echo: write error: Function not implemented".
   The following rules apply when writing to these files:
   - If there are no VFs assigned, the number of VFs can be changed to any valid value (0 - max #VFs as set during FW burning)
   - If there are VFs assigned to a VM, it is not possible to change the number of VFs
   - If the administrator unloads the driver on the PF while there are no VFs assigned, the driver will unload and SRI-OV will be disabled
   - If there are VFs assigned while the driver of the PF is unloaded, SR-IOV will not be disabled. This means that VFs will be visible on the VM. However, they will not be operational. This is applicable to OSs with kernels that use pci_stub and not vfio.
     - The VF driver will discover this situation and will close its resources
     - When the driver on the PF is reloaded, the VF becomes operational. The administrator of the VF will need to restart the driver in order to resume working with the VF.
5. Load the driver. To verify that the VFs were created. Run:

   ```
   lspci | grep Mellanox
   08:00.0 Infiniband controller: Mellanox Technologies MT27700 Family [ConnectX-4]
   08:00.1 Infiniband controller: Mellanox Technologies MT27700 Family [ConnectX-4]
   08:00.2 Infiniband controller: Mellanox Technologies MT27700 Family [ConnectX-4 Virtual Function]
   08:00.3 Infiniband controller: Mellanox Technologies MT27700 Family [ConnectX-4 Virtual Function]
   08:00.4 Infiniband controller: Mellanox Technologies MT27700 Family [ConnectX-4 Virtual Function]
   08:00.5 Infiniband controller: Mellanox Technologies MT27700 Family [ConnectX-4 Virtual Function]
   ```

6. Configure the VFs.
   After VFs are created, 3 sysfs entries per VF are available under /sys/class/infiniband/mlx5_<PF INDEX>/device/sriov (shown below for VFs 0 to 2):

   ```
   +-- 0
   |   +-- node
   |   +-- policy
   ```

```
|    +-- port
+-- 1
|    +-- node
|    +-- policy
|    +-- port
+-- 2
     +-- node
     +-- policy
     +-- port
```

For each Virtual Function, the following files are available:
- Node - Node’s GUID:
The user can set the node GUID by writing to the /sys/class/infiniband/<PF>/device/sriov/<index>/node file. The example below, shows how to set the node GUID for VF 0 of mlx5_0.

```
echo 00:11:22:33:44:55:1:0 > /sys/class/infiniband/mlx5_0/device/sriov/0/node
```

- Port - Port’s GUID:
The user can set the port GUID by writing to the /sys/class/infiniband/<PF>/device/sriov/<index>/port file. The example below, shows how to set the port GUID for VF 0 of mlx5_0.

```
echo 00:11:22:33:44:55:2:0 > /sys/class/infiniband/mlx5_0/device/sriov/0/port
```

- Policy - The vport's policy. The user can set the port GUID by writing to the /sys/class/infiniband/<PF>/device/sriov/<index>/port file. The policy can be one of:
  - Down - the VPort PortState remains 'Down'
  - Up - if the current VPort PortState is 'Down', it is modified to 'Initialize'. In all other states, it is unmodified. The result is that the SM may bring the VPort up.
  - Follow - follows the PortState of the physical port. If the PortState of the physical port is 'Active', then the VPort implements the 'Up' policy. Otherwise, the VPort PortState is 'Down'.

Notes:
- The policy of all the vports is initialized to “Down” after the PF driver is restarted except for VPort0 for which the policy is modified to 'Follow' by the PF driver.
- To see the VFs configuration, you must unbind and bind them or reboot the VMs if the VFs were assigned.

7. Make sure that OpenSM supports Virtualization (Virtualization must be enabled).
The /etc/opensm/opensm.conf file should contain the following line:

```
virt_enabled 2
```

Note: OpenSM and any other utility that uses SMP MADs (ibnetdiscover, sminfo, iblink- info, smpdump, ibqueryerr, ibdiagnet and smpquery) should run on the PF and not on the VFs. In case of multi PFs (multi-host), OpenSM should run on Host0.

VFs Initialization Note

Since the same mlx5_core driver supports both Physical and Virtual Functions, once the Virtual Functions are created, the driver of the PF will attempt to initialize them so they will be available to the OS owning the PF. If you want to assign a Virtual Function to a VM, you need to make sure the VF is not used by the PF driver. If a VF is used, you should first unbind it before assigning to a VM.

➤ *To unbind a device use the following command:*

1. Get the full PCI address of the device.

```
lspci -D
```

Example:

```
0000:09:00.2
```

2. Unbind the device.

```
echo 0000:09:00.2 > /sys/bus/pci/drivers/mlx5_core/unbind
```

3. Bind the unbound VF.

```
echo 0000:09:00.2 > /sys/bus/pci/drivers/mlx5_core/bind
```

PCI BDF Mapping of PFs and VFs

PCI addresses are sequential for both of the PF and their VFs. Assuming the card's PCI slot is 05:00 and it has 2 ports, the PFs PCI address will be 05:00.0 and 05:00.1.
Given 3 VFs per PF, the VFs PCI addresses will be:

```
05:00.2-4 for VFs 0-2 of PF 0 (mlx5_0)
05:00.5-7 for VFs 0-2 of PF 1 (mlx5_1)
```

14.6.2.3.1.5 Additional SR-IOV Configurations

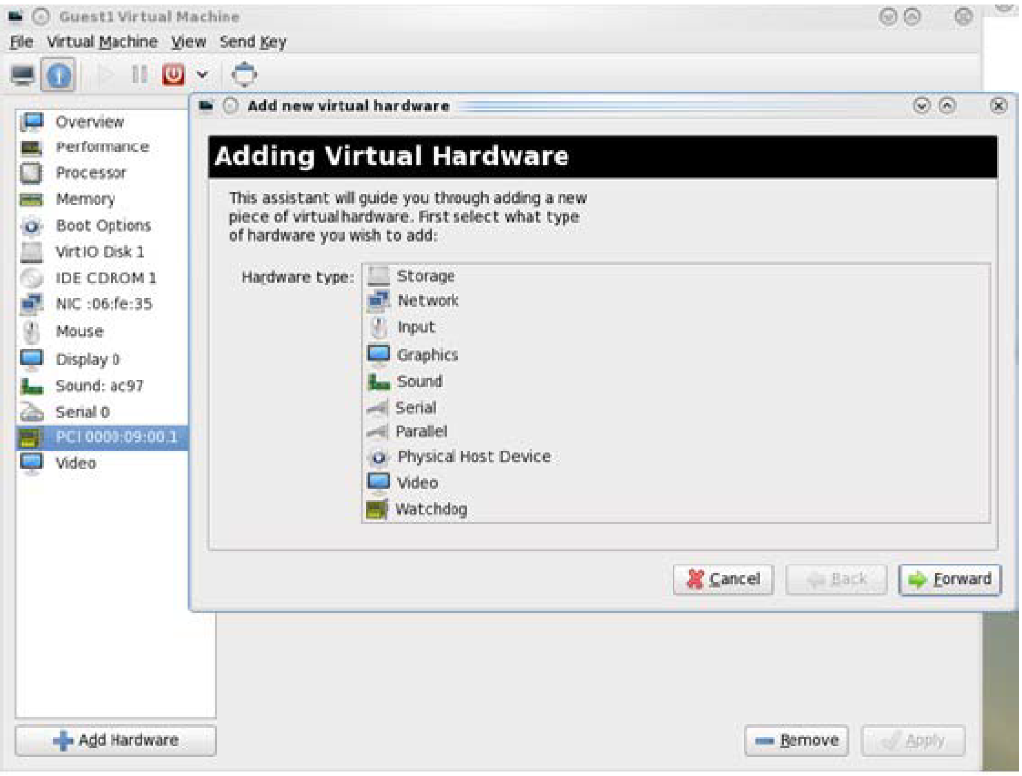
Assigning a Virtual Function to a Virtual Machine

This section describes a mechanism for adding a SR-IOV VF to a Virtual Machine.

Assigning the SR-IOV Virtual Function to the Red Hat KVM VM Server

1. Run the virt-manager.
2. Double click on the virtual machine and open its Properties.

3. Go to Details → Add hardware → PCI host device.



Adding Virtual Hardware

This assistant will guide you through adding a new piece of virtual hardware. First select what type of hardware you wish to add:

Hardware type:

4. Choose a NVIDIA virtual function according to its PCI device (e.g., 00:03.1)
5. If the Virtual Machine is up reboot it, otherwise start it.
6. Log into the virtual machine and verify that it recognizes the NVIDIA card. Run:

```
lspci | grep Mellanox
```

Example:

```
lspci | grep Mellanox
01:00.0 Infiniband controller: Mellanox Technologies MT28800 Family [ConnectX-5 Ex]
```

7. Add the device to the `/etc/sysconfig/network-scripts/ifcfg-ethX` configuration file. The MAC address for every virtual function is configured randomly, therefore it is not necessary to add it.

Ethernet Virtual Function Configuration when Running SR-IOV

SR-IOV Virtual function configuration can be done through Hypervisor iprout2/netlink tool, if present. Otherwise, it can be done via sysfs.

```
ip link set { dev DEVICE | group DEVGROUP } [ { up | down } ]
...
[ vf NUM [ mac LLADDR ] [ vlan VLANID [ qos VLAN-QOS ] ]
...
[ spoofchk { on | off} ] ]
...
sysfs configuration (ConnectX-4):
/sys/class/net/enp8s0f0/device/sriov/[VF]
+-- [VF]
```

```
| +-- config
| +-- link_state
| +-- mac
| +-- mac_list
| +-- max_tx_rate
| +-- min_tx_rate
| +-- spoofcheck
| +-- stats
| +-- trunk
| +-- trust
| +-- vlan
```

VLAN Guest Tagging (VGT) and VLAN Switch Tagging (VST)

When running ETH ports on VGT, the ports may be configured to simply pass through packets as is from VFs (VLAN Guest Tagging), or the administrator may configure the Hypervisor to silently force packets to be associated with a VLAN/Qos (VLAN Switch Tagging).
In the latter case, untagged or priority-tagged outgoing packets from the guest will have the VLAN tag inserted, and incoming packets will have the VLAN tag removed.
The default behavior is VGT.

To configure VF VST mode, run:

```
ip link set dev <PF device> vf <NUM> vlan <vlan_id> [qos <qos>]
```

where:

- NUM = 0..max-vf-num
- vlan_id = 0..4095
- qos = 0..7

For example:

- ip link set dev eth2 vf 2 vlan 10 qos 3 - sets VST mode for VF #2 belonging to PF eth2, with vlan_id = 10 and qos = 3
- ip link set dev eth2 vf 2 vlan 0 - sets mode for VF 2 back to VGT

Additional Ethernet VF Configuration Options

- Guest MAC configuration - by default, guest MAC addresses are configured to be all zeroes. If the administrator wishes the guest to always start up with the same MAC, he/she should configure guest MACs before the guest driver comes up. The guest MAC may be configured by using:

  ```
  ip link set dev <PF device> vf <NUM> mac <LLADDR>
  ```

  For legacy and ConnectX-4 guests, which do not generate random MACs, the administrator should always configure their MAC addresses via IP link, as above.

- Spoof checking - Spoof checking is currently available only on upstream kernels newer than 3.1.

  ```
  ip link set dev <PF device> vf <NUM> spoofchk [on | off]
  ```

- Guest Link State

```
ip link set dev <PF device> vf <UM> state [enable| disable| auto]
```

Virtual Function Statistics

Virtual function statistics can be queried via sysfs:

```
cat /sys/class/infiniband/mlx5_2/device/sriov/2/stats tx_packets : 5011
tx_bytes : 4450870
tx_dropped : 0
rx_packets : 5003
rx_bytes : 4450222
rx_broadcast : 0
rx_multicast : 0
tx_broadcast : 0
tx_multicast : 8
rx_dropped : 0
```

Mapping VFs to Ports

To view the VFs mapping to ports:

Use the ip link tool v2.6.34~3 and above.

```
ip link
```

Output:

```
61: p1p1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 00:02:c9:f1:72:e0 brd ff:ff:ff:ff:ff:ff
    vf 0 MAC 00:00:00:00:00:00, vlan 4095, spoof checking off, link-state auto
    vf 37 MAC 00:00:00:00:00:00, vlan 4095, spoof checking off, link-state auto
    vf 38 MAC ff:ff:ff:ff:ff:ff, vlan 65535, spoof checking off, link-state disable
    vf 39 MAC ff:ff:ff:ff:ff:ff, vlan 65535, spoof checking off, link-state disable
```

When a MAC is ff:ff:ff:ff:ff:ff, the VF is not assigned to the port of the net device it is listed under. In the example above, vf38 is not assigned to the same port as p1p1, in contrast to vf0. However, even VFs that are not assigned to the net device, could be used to set and change its settings. For example, the following is a valid command to change the spoof check:

```
ip link set dev p1p1 vf 38 spoofchk on
```

This command will affect only the vf38. The changes can be seen in ip link on the net device that this device is assigned to.

RoCE Support

RoCE is supported on Virtual Functions and VLANs may be used with it. For RoCE, the hypervisor GID table size is of 16 entries while the VFs share the remaining 112 entries. When the number of VFs is larger than 56 entries, some of them will have GID table with only a single entry which is inadequate if VF's Ethernet device is assigned with an IP address.

Virtual Guest Tagging (VGT+)

VGT+ is an advanced mode of Virtual Guest Tagging (VGT), in which a VF is allowed to tag its own packets as in VGT, but is still subject to an administrative VLAN trunk policy. The policy determines which VLAN IDs are allowed to be transmitted or received. The policy does not determine the user priority, which is left unchanged.

Packets can be sent in one of the following modes: when the VF is allowed to send/receive untagged and priority tagged traffic and when it is not. No default VLAN is defined for VGT+ port. The send packets are passed to the eSwitch only if they match the set, and the received packets are forwarded to the VF only if they match the set.

Configuration

> ⚠ When working in SR-IOV, the default operating mode is VGT.

➢ *To enable VGT+ mode:*

Set the corresponding port/VF (in the example below port eth5, VF0) range of allowed VLANs.

```
echo "<add> <start_vid> <end_vid>" > /sys/class/net/eth5/device/sriov/0/trunk
```

Examples:

- Adding VLAN ID range (4-15) to trunk:

  ```
  echo add 4 15 > /sys/class/net/eth5/device/sriov/0/trunk
  ```

- Adding a single VLAN ID to trunk:

  ```
  echo add 17 17 > /sys/class/net/eth5/device/sriov/0/trunk
  ```

Note: When VLAN ID = 0, it indicates that untagged and priority-tagged traffics are allowed

➢ *To disable VGT+ mode, make sure to remove all VLANs.*

```
echo rem 0 4095 > /sys/class/net/eth5/device/sriov/0/trunk
```

➢ *To remove selected VLANs.*

- Remove VLAN ID range (4-15) from trunk:

  ```
  echo rem 4 15 > /sys/class/net/eth5/device/sriov/0/trunk
  ```

- Remove a single VLAN ID from trunk:

  ```
  echo rem 17 17 > /sys/class/net/eth5/device/sriov/0/trunk
  ```

SR-IOV Advanced Security Features

SR-IOV MAC Anti-Spoofing

Normally, MAC addresses are unique identifiers assigned to network interfaces, and they are fixed addresses that cannot be changed. MAC address spoofing is a technique for altering the MAC address to serve different purposes. Some of the cases in which a MAC address is altered can be legal, while others can be illegal and abuse security mechanisms or disguises a possible attacker.

The SR-IOV MAC address anti-spoofing feature, also known as MAC Spoof Check provides protection against malicious VM MAC address forging. If the network administrator assigns a MAC address to a VF (through the hypervisor) and enables spoof check on it, this will limit the end user to send traffic only from the assigned MAC address of that VF.

MAC Anti-Spoofing Configuration

> MAC anti-spoofing is disabled by default.

In the configuration example below, the VM is located on VF-0 and has the following MAC address: 11:22:33:44:55:66.
There are two ways to enable or disable MAC anti-spoofing:

1. Use the standard IP link commands - available from Kernel 3.10 and above.
    a. To enable MAC anti-spoofing, run:

    ```
    ip link set ens785f1 vf 0 spoofchk on
    ```

    b. To disable MAC anti-spoofing, run:

    ```
    ip link set ens785f1 vf 0 spoofchk off
    ```

2. Specify echo "ON" or "OFF" to the file located under /sys/class/net/<ifname / device/sriov/<VF index>/spoofcheck.
    a. To enable MAC anti-spoofing, run:

    ```
    echo "ON" > /sys/class/net/ens785f1/vf/0/spoofchk
    ```

    b. To disable MAC anti-spoofing, run:

    ```
    echo "OFF" > /sys/class/net/ens785f1/vf/0/spoofchk
    ```

> This configuration is non-persistent and does not survive driver restart.

Limit and Bandwidth Share Per VF

This feature enables rate limiting traffic per VF in SR-IOV mode. For details on how to configure rate limit per VF for ConnectX-4 and above adapter cards, please refer to HowTo Configure Rate Limit per VF for ConnectX-4/ConnectX-5/ConnectX-6 Community post.

Limit Bandwidth per Group of VFs

VFs Rate Limit for vSwitch (OVS) feature allows users to join available VFs into groups and set a rate limitation on each group. Rate limitation on a VF group ensures that the total Tx bandwidth that the VFs in this group get (altogether combined) will not exceed the given value.
With this feature, a VF can still be configured with an individual rate limit as in the past (under /sys/class/net/<ifname>/device/sriov/<vf_num>/max_tx_rate). However, the actual bandwidth limit on the VF will eventually be determined considering the VF group limitation and how many VFs are

in the same group.
For example: 2 VFs (0 and 1) are attached to group 3.

Case 1: The rate limitation on the group is set to 20G. Rate limit of each VF is 15G
Result: Each VF will have a rate limit of 10G

Case 2: Group’s max rate limitation is still set to 20G. VF 0 is configured to 30G limit, while VF 1 is configured to 5G rate limit
Result: VF 0 will have 15G de-facto. VF 1 will have 5G

The rule of thumb is that the group’s bandwidth is distributed evenly between the number of VFs in the group. If there are leftovers, they will be assigned to VFs whose individual rate limit has not been met yet.

VFs Rate Limit Feature Configuration

1. When VF rate group is supported by FW, the driver will create a new hierarchy in the SRI-OV sysfs named “groups” (/sys/class/net/<ifname>/device/sriov/groups/). It will contain all the info and the configurations allowed for VF groups.
2. All VFs are placed in group 0 by default since it is the only existing group following the initial driver start. It would be the only group available under /sys/class/net/<ifname>/device/sriov/groups/
3. The VF can be moved to a different group by writing to the group file -> echo $GROUP_ID > /sys/class/net/<ifname>/device/sriov/<vf_id>/group
4. The group IDs allowed are 0-255
5. Only when there is at least 1 VF in a group, there will be a group configuration available under /sys/class/net/<ifname>/device/sriov/groups/ (Except for group 0, which is always available even when it’s empty).
6. Once the group is created (by moving at least 1 VF to that group), users can configure the group’s rate limit. For example:
    a. echo 10000 > /sys/class/net/<ifname>/device/sriov/5/max_tx_rate – setting individual rate limitation of VF 5 to 10G (Optional)
    b. echo 7 > /sys/class/net/<ifname>/device/sriov/5/group – moving VF 5 to group 7
    c. echo 5000 > /sys/class/net/<ifname>/device/sriov/groups/7/max_tx_rate – setting group 7 with rate limitation of 5G
    d. When running traffic via VF 5 now, it will be limited to 5G because of the group rate limit even though the VF itself is limited to 10G
    e. echo 3 > /sys/class/net/<ifname>/device/sriov/5/group – moving VF 5 to group 3
    f. Group 7 will now disappear from /sys/class/net/<ifname>/device/sriov/groups since there are 0 VFs in it. Group 3 will now appear. Since there’s no rate limit on group 3, VF 5 can transmit at 10G (thanks to its individual configuration)

Notes:

- You can see to which group the VF belongs to in the ‘stats’ sysfs (cat /sys/class/net/<ifname>/device/sriov/<vf_num>/stats)
- You can see the current rate limit and number of attached VFs to a group in the group’s ‘config’ sysfs (cat /sys/class/net/<ifname>/device/sriov/groups/<group_id>/config)

Bandwidth Guarantee per Group of VFs

Bandwidth guarantee (minimum BW) can be set on a group of VFs to ensure this group is able to transmit at least the amount of bandwidth specified on the wire.

Note the following:

- The minimum BW settings on VF groups determine how the groups share the total BW between themselves. It does not impact an individual VF’s rate settings.
- The total minimum BW that is set on the VF groups should not exceed the total line rate. Otherwise, results are unexpected.
- It is still possible to set minimum BW on the individual VFs inside the group. This will determine how the VFs share the group’s minimum BW between themselves. The total minimum BW of the VF member should not exceed the minimum BW of the group.

For instruction on how to create groups of VFs, see Limit Bandwidth per Group of VFs above.

Example

With a 40Gb link speed, assuming 4 groups and default group 0 have been created:

```
echo 20000 > /sys/class/net/<ifname>/device/sriov/group/1/min_tx_rate
echo 5000 > /sys/class/net/<ifname>/device/sriov/group/2/min_tx_rate
echo 15000 > /sys/class/net/<ifname>/device/sriov/group/3/min_tx_rate
```

```
Group 0(default) : 0 - No BW guarantee is configured.
Group 1 : 20000 - This is the maximum min rate among groups
Group 2 : 5000 which is 25% of the maximum min rate
Group 3 : 15000 which is 75% of the maximum min rate
Group 4 : 0 - No BW guarantee is configured.
```

Assuming there are VFs attempting to transmit in full line rate in all groups, the results would look like: In which case, the minimum BW allocation would be:

```
Group0 - Will have no BW to use since no BW guarantee was set on it while other groups do have such settings.
Group1 - Will transmit at 20Gb/s
Group2 - Will transmit at 5Gb/s
Group3 - Will transmit at 15Gb/s
Group4 - Will have no BW to use since no BW guarantee was set on it while other groups do have such settings.
```

Privileged VFs

In case a malicious driver is running over one of the VFs, and in case that VF's permissions are not restricted, this may open security holes. However, VFs can be marked as trusted and can thus receive an exclusive subset of physical function privileges or permissions. For example, in case of allowing all VFs, rather than specific VFs, to enter a promiscuous mode as a privilege, this will enable malicious users to sniff and monitor the entire physical port for incoming traffic, including traffic targeting other VFs, which is considered a severe security hole.

Privileged VFs Configuration

In the configuration example below, the VM is located on VF-0 and has the following MAC address: 11:22:33:44:55:66.
There are two ways to enable or disable trust:

1. Use the standard IP link commands - available from Kernel 4.5 and above.
    a. To enable trust for a specific VF, run:

```
    ip link set ens785f1 vf 0 trust on
    ```

    b. To disable trust for a specific VF, run:

    ```
    ip link set ens785f1 vf 0 trust off
    ```

2. Specify echo "ON" or "OFF" to the file located under /sys/class/net/<ETH_IF_NAME> / device/ sriov/<VF index>/trust.

    a. To enable trust for a specific VF, run:

    ```
    echo "ON" > /sys/class/net/ens785f1/device/sriov/0/trust
    ```

    b. To disable trust for a specific VF, run:

    ```
    echo "OFF" > /sys/class/net/ens785f1/device/sriov/0/trust
    ```

Probed VFs

Probing Virtual Functions (VFs) after SR-IOV is enabled might consume the adapter cards' resources. Therefore, it is recommended not to enable probing of VFs when no monitoring of the VM is needed. VF probing can be disabled in two ways, depending on the kernel version installed on your server:

1. If the kernel version installed is v4.12 or above, it is recommended to use the PCI sysfs interface `sriov_drivers_autoprobe`. For more information, see linux-next branch.
2. If the kernel version installed is older than v4.12, it is recommended to use the mlx5_core module parameter probe_vf with driver version 4.1 or above.

Example:

```
echo 0 > /sys/module/mlx5_core/parameters/probe_vf
```

For more information on how to probe VFs, see HowTo Configure and Probe VFs on mlx5 Drivers Community post.

VF Promiscuous Rx Modes

VF Promiscuous Mode

VFs can enter a promiscuous mode that enables receiving the unmatched traffic and all the multicast traffic that reaches the physical port in addition to the traffic originally targeted to the VF. The unmatched traffic is any traffic's DMAC that does not match any of the VFs' or PFs' MAC addresses.
Note: Only privileged/trusted VFs can enter the VF promiscuous mode.

➤ *To set the promiscuous mode on for a VF, run:*

```
ifconfig eth2 promisc
```

➤ *To exit the promiscuous mode, run:*

```
ifconfig eth2 -promisc
```

VF All-Multi Mode

VFs can enter an all-multi mode that enables receiving all the multicast traffic sent from/to the other functions on the same physical port in addition to the traffic originally targeted to the VF.
Note: Only privileged/trusted VFs can enter the all-multi RX mode.

➤ *To set the all-multi mode on for a VF, run:*

```
ifconfig eth2 allmulti
```

➤ *To exit the all-multi mode, run:*

```
#ifconfig eth2 -allmulti
```

14.6.2.3.1.6 Uninstalling the SR-IOV Driver

➤ *To uninstall SR-IOV driver, perform the following:*

1. For Hypervisors, detach all the Virtual Functions (VF) from all the Virtual Machines (VM) or stop the Virtual Machines that use the Virtual Functions.
   Please be aware that stopping the driver when there are VMs that use the VFs, will cause machine to hang.
2. Run the script below. Please be aware, uninstalling the driver deletes the entire driver's file, but does not unload the driver.

   ```
   [root@swl022 ~]# /usr/sbin/ofed_uninstall.sh
   This program will uninstall all OFED packages on your machine.
   Do you want to continue?[y/N]:y
   Running /usr/sbin/vendor_pre_uninstall.sh
   Removing OFED Software installations
   Running /bin/rpm -e --allmatches kernel-ib kernel-ib-devel libibverbs libibverbs-devel libibverbs-
   devel-static libibverbs-utils libmlx4 libmlx4-devel libibcm libibcm-devel libibumad libibumad-devel
   libibumad-static libibmad libibmad-devel libibmad-static librdmacm librdmacm-utils librdmacm-devel ibacm
   opensm-libs opensm-devel perftest compat-dapl compat-dapl-devel dapl dapl-devel dapl-devel-static dapl-
   utils srptools infiniband-diags-guest ofed-scripts opensm-devel
   warning: /etc/infiniband/openib.conf saved as /etc/infiniband/openib.conf.rpmsave
   Running /tmp/2818-ofed_vendor_post_uninstall.sh
   ```
3. Restart the server.

14.6.2.3.1.7 SR-IOV Live Migration

⚠ This feature is supported in Ethernet mode only.

Live migration refers to the process of moving a guest virtual machine (VM) running on one physical host to another host without disrupting normal operations or causing other adverse effects for the end user.

Using the Migration process is useful for:

- load balancing
- hardware independence

- energy saving
- geographic migration
- fault tolerance

Migration works by sending the state of the guest virtual machine's memory and any virtualized devices to a destination host physical machine. Migrations can be performed live or not, in the live case, the migration will not disrupt the user operations and it will be transparent to it as explained in the sections below.

Non-Live Migration

When using the non-live migration process, the Hypervisor suspends the guest virtual machine, then moves an image of the guest virtual machine's memory to the destination host physical machine. The guest virtual machine is then resumed on the destination host physical machine, and the memory the guest virtual machine used on the source host physical machine is freed. The time it takes to complete such a migration depends on the network bandwidth and latency. If the network is experiencing heavy use or low bandwidth, the migration will take longer then desired.

Live Migration

When using the Live Migration process, the guest virtual machine continues to run on the source host physical machine while its memory pages are transferred to the destination host physical machine. During migration, the Hypervisor monitors the source for any changes in the pages it has already transferred and begins to transfer these changes when all of the initial pages have been transferred.

It also estimates transfer speed during migration, so when the remaining amount of data to transfer will take a certain configurable period of time, it will suspend the original guest virtual machine, transfer the remaining data, and resume the same guest virtual machine on the destination host physical machine.

MLX5 VF Live Migration

The purpose of this section is to demonstrate how to perform basic live migration of a QEMU VM with an MLX5 VF assigned to it. This section does not explains how to create VMs either using libvirt or directly via QEMU.

Requirements

The below are the requirements for working with MLX5 VF Live Migration.

Components	Description
Adapter Cards	• ConnectX-7 ETH • BlueField-3 ETH ⚠ The same PSID must be used on both the source and the target hosts (identical cards, same CAPs and features are needed), and have the same firmware version.
Firmware	• 28.41.1000 • 32.41.1000
Kernel	Linux v6.7 or newer
User Space Tools	iproute2 version 6.2 or newer

Components	Description
QEMU	QEMU 8.1 or newer
Libvirt	Libvirt 8.6 or newer

Setup

NVCONFIG

SR-IOV should be enabled and be configured to support the required number of VFs as of enabling live migration. This can be achieved by the below command:

```
mlxconfig -d *<PF_BDF>* s SRIOV_EN=1 NUM_OF_VFS=4 VF_MIGRATION_MODE=2
```

where:

SRIOV_EN	Enable Single-Root I/O Virtualization (SR-IOV)
NUM_OF_VFS	The total number of Virtual Functions (VFs) that can be supported, for each PF.
VF_MIGRATION_MODE	Defines support for VF migration. • 0x0: DEVICE_DEFAULT • 0x1: MIGRATION_DISABLED • 0x2: MIGRATION_ENABLED

Kernel Configuration

Needs to be compiled with driver MLX5_VFIO_PCI enabled. (i.e. CONFIG_MLX5_VFIO_PCI).

To load the driver, run:

```
modprobe mlx5_vfio_pci
```

QEMU

Needs to be compiled with VFIO_PCI enabled (this is enabled by default).

Host Preparation

As stated earlier, creating the VMs is beyond the scope of this guide and we assume that they are already created. However, the VM configuration should be a migratable configuration, similarly to how it is done without SRIOV VFs.

> ⚠ The below steps should be done before running the VMs.

Over libvirt

1. Set the PF in the "switchdev" mode.

   ```
   devlink dev eswitch set pci/<PF_BDF> mode switchdev
   ```

2. Create the VFs that will be assigned to the VMs.

```
echo "1" > /sys/bus/pci/devices/<PF_BDF>/sriov_numvfs
```

3. Set the VFs as migration capable.
    a. See the name of the VFs, run:

    ```
    devlink port show
    ```

    b. Unbind the VFs from mlx5_core, run:

    ```
    echo '<VF_BDF>' > /sys/bus/pci/drivers/mlx5_core/unbind
    ```

    c. Use devlink to set each VF as migration capable, run:

    ```
    devlink port function set pci/<PF_BDF>/1 migratable enable
    ```

4. Assign the VFs to the VMs.

    a. Edit the VMs XML file, run:

    ```
    virsh edit <VM_NAME>
    ```

    b. Assign the VFs to the VM by adding the following under the "devices" tag:

    ```
    <hostdev mode='subsystem' type='pci' managed='no'>
      <driver name='vfio'/>
        <source>
          <address domain='0x0000' bus='0x08' slot='0x00' function='0x2'/>
        </source>
      <address type='pci' domain='0x0000' bus='0x09' slot='0x00' function='0x0'/>
    </hostdev>
    ```

    > The domain, bus, slot and function values above are dummy values, replace them with your VFs values.

5. Set the destination VM in incoming mode.
    a. Edit the destination VM XML file, run:

    ```
    virsh edit <VM_NAME>
    ```

    b. Set the destination VM in migration incoming mode by adding the following under "domain" tag:

    ```
    <domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
      [...]
      <qemu:commandline>
        <qemu:arg value='--incoming'/>
        <qemu:arg value='tcp:<DEST_IP>:<DEST_PORT>'/>
      </qemu:commandline>
    </domain>
    ```

    > To be able to save the file, the above `"xmlns:qemu"` attribute of the "domain" tag must be added as well.

6. Bind the VFs to mlx5_vfio_pci driver.

    a. Detach the VFs from libvirt management, run:

```
virsh nodedev-detach pci_<VF_BDF>
```

   b. Unbind the VFs from vfio-pci driver (the VFs are automatically bound to it after running "virsh nodedev-detach"), run:

```
echo '<VF_BDF>' > /sys/bus/pci/drivers/vfio-pci/unbind
```

   c. Set driver override, run:

```
echo 'mlx5_vfio_pci' > /sys/bus/pci/devices/<VF_BDF>/driver_override
```

   d. Bind the VFs to mlx5_vfio_pci driver, run:

```
echo '<VF_BDF>' > /sys/bus/pci/drivers/mlx5_vfio_pci/bind
```

Directly over QEMU

1. Set the PF in "switchdev" mode.

```
devlink dev eswitch set pci/<PF_BDF> mode switchdev
```

2. Create the VFs that will be assigned to the VMs.

```
echo "1" > /sys/bus/pci/devices/<PF_BDF>/sriov_numvfs
```

3. Set the VFs as migration capable.
   a. See the name of the VFs, run:

```
devlink port show
```

   b. Unbind the VFs from mlx5_core, run:

```
echo '<VF_BDF>' > /sys/bus/pci/drivers/mlx5_core/unbind
```

   c. Use devlink to set each VF as migration capable, run:

```
devlink port function set pci/<PF_BDF>/1 migratable enable
```

4. Bind the VFs to mlx5_vfio_pci driver:
   a. Set driver override, run:

```
echo 'mlx5_vfio_pci' > /sys/bus/pci/devices/<VF_BDF>/driver_override
```

   b. Bind the VFs to mlx5_vfio_pci driver, run:

```
echo '<VF_BDF>' > /sys/bus/pci/drivers/mlx5_vfio_pci/bind
```

Running the Migration

Over libvirt

1. Start the VMs in source and in destination, run:

```
virsh start <VM_NAME>
```

2. Enable switchover-ack QEMU migration capability. Run the following commands both in source and destination:

```
virsh qemu-monitor-command <VM_NAME> --hmp "migrate_set_capability return-path on"
```

```
virsh qemu-monitor-command <VM_NAME> --hmp "migrate_set_capability switchover-ack on"
```

3. [Optional] Configure the migration bandwidth and downtime limit in source side:

```
virsh qemu-monitor-command <VM_NAME> --hmp "migrate_set_parameter max-bandwidth <VALUE>"
virsh qemu-monitor-command <VM_NAME> --hmp "migrate_set_parameter downtime-limit <VALUE>"
```

4. Start migration by running the migration command in source side:

```
virsh qemu-monitor-command <VM_NAME> --hmp "migrate -d tcp:<DEST_IP>:<DEST_PORT>"
```

5. Check the migration status by running the info command in source side:

```
virsh qemu-monitor-command <VM_NAME> --hmp "info migrate"
```

> When the migration status is "completed" it means the migration has finished successfully.

Directly over QEMU

1. Start the VM in source with the VF assigned to it:

```
qemu-system-x86_64 [...] -device vfio-pci,host=<VF_BDF>,id=mlx5_1
```

2. Start the VM in destination with the VF assigned to it and with the "incoming" parameter:

```
qemu-system-x86_64 [...] -device vfio-pci,host=<VF_BDF>,id=mlx5_1 -incoming tcp:<DEST_IP>:<DEST_PORT>
```

3. Enable switchover-ack QEMU migration capability. Run the following commands in QEMU monitor, both in source and destination:

```
migrate_set_capability return-path on
```

```
migrate_set_capability switchover-ack on
```

4. [Optional] Configure the migration bandwidth and downtime limit in source side:

```
migrate_set_parameter max-bandwidth <VALUE>
migrate_set_parameter downtime-limit <VALUE>
```

5. Start migration by running the migration command in QEMU monitor in source side:

```
migrate -d tcp:<DEST_IP>:<DEST_PORT>
```

6. Check the migration status by running the info command in QEMU monitor in source side:

```
info migrate
```

> ⚠ When the migration status is "completed" it means the migration has finished successfully.

Migration with MultiPort vHCA

Enables the usage of a dual port Virtual HCA (vHCA) to share RDMA resources (e.g., MR, CQ, SRQ, PDs) across the two Ethernet (RoCE) NIC network ports and display the NIC as a dual port device.

MultiPort vHCA (MPV) VF is made of 2 "regular" VFs, one VF of each port. Creating a migratable MPV VF requires the same steps as regular VF (see steps in section Over libvirt). The steps should be performed on each of the NIC ports. MPV VFs traffic cannot be configured with OVS. TC rules must be defined to configure the MPV VFs traffic.

Notes

> ⚠ In ConnectX-7 adapter cards, migration cannot run in parallel on more than 4 VFs. It is the administrator's responsibility to control that.

> ⚠ Live migration requires same firmware version on both the source and the target hosts.

14.6.2.3.2 Enabling Paravirtualization

➤ To enable Paravirtualization:

> ⚠ The example below works on RHEL7.* without a Network Manager.

1. Create a bridge.

```
vim /etc/sysconfig/network-scripts/ifcfg-bridge0
DEVICE=bridge0
TYPE=Bridge
IPADDR=12.195.15.1
NETMASK=255.255.0.0
BOOTPROTO=static
ONBOOT=yes
NM_CONTROLLED=no
DELAY=0
```

2. Change the related interface (in the example below bridge0 is created over eth5).

```
DEVICE=eth5
BOOTPROTO=none
STARTMODE=on
HWADDR=00:02:c9:2e:66:52
TYPE=Ethernet
NM_CONTROLLED=no
ONBOOT=yes
BRIDGE=bridge0
```

3. Restart the service network.

4. Attach a bridge to VM.

```
ifconfig -a
…
eth6      Link encap:Ethernet  HWaddr 52:54:00:E7:77:99
          inet addr:13.195.15.5  Bcast:13.195.255.255  Mask:255.255.0.0
          inet6 addr: fe80::5054:ff:fee7:7799/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:481 errors:0 dropped:0 overruns:0 frame:0
          TX packets:450 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:22440 (21.9 KiB)  TX bytes:19232 (18.7 KiB)
          Interrupt:10 Base address:0xa000
…
```

14.6.2.3.3 VXLAN Hardware Stateless Offloads

VXLAN technology provides scalability and security challenges solutions. It requires extension of the traditional stateless offloads to avoid performance drop. ConnectX family cards offer the following stateless offloads for a VXLAN packet, similar to the ones offered to non-encapsulated packets. VXLAN protocol encapsulates its packets using outer UDP header.

Available hardware stateless offloads:

- Checksum generation (Inner IP and Inner TCP/UDP)
- Checksum validation (Inner IP and Inner TCP/UDP)
- TSO support for inner TCP packets
- RSS distribution according to inner packets attributes
- Receive queue selection - inner frames may be steered to specific QPs

14.6.2.3.3.1 Enabling VXLAN Hardware Stateless Offloads

VXLAN offload is enabled by default for ConnectX-4 family devices running the minimum required firmware version and a kernel version that includes VXLAN support.

➤ *To confirm if the current setup supports VXLAN, run:*

```
ethtool -k $DEV | grep udp_tnl
```

Example:

```
ethtool -k ens1f0 | grep udp_tnl
tx-udp_tnl-segmentation: on
```

ConnectX-4 family devices support configuring multiple UDP ports for VXLAN offload. Ports can be added to the device by configuring a VXLAN device from the OS command line using the "ip" command.

Note: If you configure multiple UDP ports for offload and exceed the total number of ports supported by hardware, then those additional ports will still function properly, but will not benefit from any of the stateless offloads.

Example:

```
ip link add vxlan0 type vxlan id 10 group 239.0.0.10 ttl 10 dev ens1f0 dstport 4789
ip addr add 192.168.4.7/24 dev vxlan0
ip link set up vxlan0
```

Note: dstport' parameters are not supported in Ubuntu 14.4.

The VXLAN ports can be removed by deleting the VXLAN interfaces.

Example:

```
ip link delete vxlan0
```

14.6.2.3.3.2 Important Note

VXLAN tunneling adds 50 bytes (14-eth + 20-ip + 8-udp + 8-vxlan) to the VM Ethernet frame. Please verify that either the MTU of the NIC who sends the packets, e.g. the VM virtio-net NIC or the host side veth device or the uplink takes into account the tunneling overhead. Meaning, the MTU of the sending NIC has to be decremented by 50 bytes (e.g 1450 instead of 1500), or the uplink NIC MTU has to be incremented by 50 bytes (e.g 1550 instead of 1500)

14.6.2.3.4 Q-in-Q Encapsulation per VF in Linux (VST)

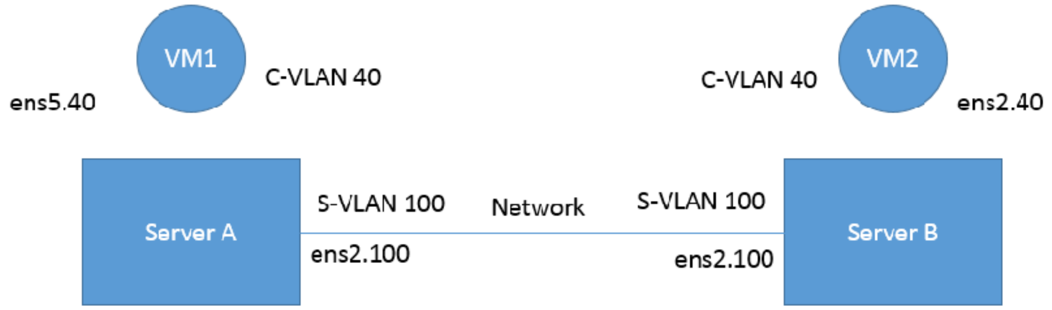
⚠ This feature is supported on ConnectX-5 and ConnectX-6 adapter cards only.

⚠ ConnectX-4 and ConnectX-4 Lx adapter cards support 802.1Q double-tagging (C-tag stacking on C-tag), refer to "802.1Q Double-Tagging" section.

This section describes the configuration of IEEE 802.1ad QinQ VLAN tag (S-VLAN) to the hypervisor per Virtual Function (VF). The Virtual Machine (VM) attached to the VF (via SR- IOV) can send traffic with or without C-VLAN. Once a VF is configured to VST QinQ encapsulation (VST QinQ), the adapter's hardware will insert S-VLAN to any packet from the VF to the physical port. On the receive side, the adapter hardware will strip the S-VLAN from any packet coming from the wire to that VF.

14.6.2.3.4.1 Setup

The setup assumes there are two servers equipped with ConnectX-5/ConnectX-6 adapter cards.



14.6.2.3.4.2 Prerequisites

- Kernel must be of v3.10 or higher, or custom/inbox kernel must support vlan-stag
- Firmware version 16/20.21.0458 or higher must be installed for ConnectX-5/ConnectX-6 HCAs
- The server should be enabled in SR-IOV and the VF should be attached to a VM on the hypervisor.
    - In order to configure SR-IOV in Ethernet mode for ConnectX-5/ConnectX-6 adapter cards, please refer to "Configuring SR-IOV for ConnectX-4/ConnectX-5 (Ethernet)" section. In the following configuration example, the VM is attached to VF0.
- Network Considerations - the network switches may require increasing the MTU (to support 1522 MTU size) on the relevant switch ports.

14.6.2.3.4.3 Configuring Q-in-Q Encapsulation per Virtual Function for ConnectX-5/ConnectX-6

1. Add the required S-VLAN (QinQ) tag (on the hypervisor) per port per VF. There are two ways to add the S-VLAN:
    a. By using sysfs:

    ```
    echo '100:0:802.1ad' > /sys/class/net/ens1f0/device/sriov/0/vlan
    ```

    b. By using the ip link command (available only when using the latest Kernel version):

    ```
    ip link set dev ens1f0 vf 0 vlan 100 proto 802.1ad
    ```

    Check the configuration using the ip link show command:

    ```
    # ip link show ens1f0
     ens1f0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT qlen 1000
        link/ether ec:0d:9a:44:37:84 brd ff:ff:ff:ff:ff:ff
        vf 0 MAC 00:00:00:00:00:00, vlan 100, vlan protocol 802.1ad, spoof checking off, link-state
    auto, trust off
        vf 1 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
        vf 2 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
        vf 3 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
        vf 4 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
    ```

2. **Optional:** Add S-VLAN priority. Use the qos parameter in the ip link command (or sysfs):

    ```
    ip link set dev ens1f0 vf 0 vlan 100 qos 3 proto 802.1ad
    ```

    Check the configuration using the ip link show command:

    ```
    # ip link show ens1f0
    ens1f0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT qlen 1000
        link/ether ec:0d:9a:44:37:84 brd ff:ff:ff:ff:ff:ff
        vf 0 MAC 00:00:00:00:00:00, vlan 100, qos 3, vlan protocol 802.1ad, spoof checking off, link-state
    auto, trust off
        vf 1 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
        vf 2 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
        vf 3 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
        vf 4 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
    ```

3. Create a VLAN interface on the VM and add an IP address.

    ```
    ip link add link ens5 ens5.40 type vlan protocol 802.1q id 40
    ip addr add 42.134.135.7/16 brd 42.134.255.255 dev ens5.40
    ip link set dev ens5.40 up
    ```

4. To verify the setup, run ping between the two VMs and open Wireshark or tcpdump to capture the packet.

14.6.2.3.5 802.1Q Double-Tagging

This section describes the configuration of 802.1Q double-tagging support to the hypervisor per Virtual Function (VF). The Virtual Machine (VM) attached to the VF (via SR-IOV) can send traffic with or without C-VLAN. Once a VF is configured to VST encapsulation, the adapter's hardware will insert C-VLAN to any packet from the VF to the physical port. On the receive side, the adapter hardware will strip the C-VLAN from any packet coming from the wire to that VF.

14.6.2.3.5.1 Configuring 802.1Q Double-Tagging per Virtual Function

1. Add the required C-VLAN tag (on the hypervisor) per port per VF. There are two ways to add the C-VLAN:
    a. By using sysfs:

    ```
    echo '100:0:802.1q' > /sys/class/net/ens1f0/device/sriov/0/vlan
    ```

    b. By using the ip link command (available only when using the latest Kernel version):

    ```
    ip link set dev ens1f0 vf 0 vlan 100
    ```

    Check the configuration using the ip link show command:

    ```
    # ip link show ens1f0
     ens1f0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT qlen 1000
        link/ether ec:0d:9a:44:37:84 brd ff:ff:ff:ff:ff:ff
        vf 0 MAC 00:00:00:00:00:00, vlan 100, spoof checking off, link-state auto, trust off
        vf 1 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
        vf 2 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
        vf 3 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
        vf 4 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
    ```

2. Create a VLAN interface on the VM and add an IP address.

    ```
    # ip link add link ens5 ens5.40 type vlan protocol 802.1q id 40
    # ip addr add 42.134.135.7/16 brd 42.134.255.255 dev ens5.40
    # ip link set dev ens5.40 up
    ```

3. To verify the setup, run ping between the two VMs and open Wireshark or tcpdump to capture the packet.

14.6.2.3.6 Scalable Functions

Scalable function is a lightweight function that has a parent PCI function on which it is deployed. Scalable functions are useful for containers where netdevice and RDMA devices of a scalable function can be assigned to a container. This way, the container can get complete offload capabilities of an eswitch, isolation and dedicated accelerated network device. For Step-by-Step Configuration instructions, follow the User Guide here.

14.6.2.4 Resiliency

The chapter contains the following sections:

- Reset Flow

14.6.2.4.1 Reset Flow

Reset Flow is activated by default. Once a "fatal device" error is recognized, both the HCA and the software are reset, the ULPs and user application are notified about it, and a recovery process is performed once the event is raised.

Currently, a reset flow can be triggered by a firmware assert with Recover Flow Request (RFR) only. Firmware RFR support should be enabled explicitly using mlxconfig commands.

To query the current value, run:

```
mlxconfig -d /dev/mst/mt4115_pciconf0 query | grep SW_RECOVERY_ON_ERRORS
```

To enable RFR bit support, run:

```
mlxconfig -d /dev/mst/mt4115_pciconf0 set SW_RECOVERY_ON_ERRORS=true
```

14.6.2.4.1.1 Kernel ULPs

Once a "fatal device" error is recognized, an IB_EVENT_DEVICE_FATAL event is created, ULPs are notified about the incident, and outstanding WQEs are simulated to be returned with "flush in error" message to enable each ULP to close its resources and not get stuck via calling its "remove_one" callback as part of "Reset Flow".
Once the unload part is terminated, each ULP is called with its " `add_one` " callback, its resources are re-initialized and it is re-activated.

14.6.2.4.1.2 User Space Applications (IB/RoCE)

Once a "fatal device" error is recognized an IB_EVENT_DEVICE_FATAL event is created, applications are notified about the incident and relevant recovery actions are taken.
Applications that ignore this event enter a zombie state, where each command sent to the kernel is returned with an error, and no completion on outstanding WQEs is expected.
The expected behavior from the applications is to register to receive such events and recover once the above event is raised. Same behavior is expected in case the NIC is unbounded from the PCI and later is rebounded. Applications running over RDMA CM should behave in the same manner once the RDMA_CM_EVENT_DEVICE_REMOVAL event is raised.
The below is an example of using the unbind/bind for NIC defined by "0000:04:00.0"

```
echo 0000:04:00.0 > /sys/bus/pci/drivers/mlx5_core/unbind
echo 0000:04:00.0 > /sys/bus/pci/drivers/mlx5_core/bind
```

14.6.2.4.1.3 SR-IOV

If the Physical Function recognizes the error, it notifies all the VFs about it by marking their communication channel with that information, consequently, all the VFs and the PF are reset.
If the VF encounters an error, only that VF is reset, whereas the PF and other VFs continue to work unaffected.

14.6.2.4.1.4 Forcing the VF to Reset

If an outside "reset" is forced by using the PCI sysfs entry for a VF, a reset is executed on that VF once it runs any command over its communication channel.
For example, the below command can be used on a hypervisor to reset a VF defined by 0000:04:00.1:

```
echo 1 >/sys/bus/pci/devices/0000:04:00.1/reset
```

14.6.2.4.1.5 Extended Error Handling (EEH)

Extended Error Handling (EEH) is a PowerPC mechanism that encapsulates AER, thus exposing AER events to the operating system as EEH events.
The behavior of ULPs and user space applications is identical to the behavior of AER.

14.6.2.4.1.6 CRDUMP

CRDUMP feature allows for taking an automatic snapshot of the device CR-Space in case the device's FW/HW fails to function properly.

Snapshots Triggers:

The snapshot is triggered after firmware detects a critical issue, requiring a recovery flow.

This snapshot can later be investigated and analyzed to track the root cause of the failure.
Currently, only the first snapshot is stored, and is exposed using a temporary virtual file. The virtual file is cleared upon driver reset.
When a critical event is detected, a message indicating CRDUMP collection will be printed to the Linux log. User should then back up the file pointed to in the printed message. The file location format is: /proc/driver/mlx5_core/crdump/<pci address>

Snapshot should be copied by Linux standard tool for future investigation.

14.6.2.4.1.7 Firmware Tracer

This mechanism allows for the device's FW/HW to log important events into the event tracing system (/sys/kernel/debug/tracing) without requiring any NVIDIA tool.

> ⚠ To be able to use this feature, trace points must be enabled in the kernel.

This feature is enabled by default, and can be controlled using sysfs commands.

➤ *To disable the feature:*

```
echo 0 > /sys/kernel/debug/tracing/events/mlx5/fw_tracer/enable
```

➤ *To enable the feature:*

```
echo 1 > /sys/kernel/debug/tracing/events/mlx5/fw_tracer/enable
```

➤ *To view FW traces using vim text editor:*

```
vim /sys/kernel/debug/tracing/trace
```

14.6.2.5 Docker Containers

On Linux, Docker uses resource isolation of the Linux kernel, to allow independent "containers" to run within a single Linux kernel instance.
Docker containers are supported on MLNX_OFEDusing Docker runtime. Virtual RoCE and InfiniBand devices are supported using SR-IOV mode.

Currently, RDMA/RoCE devices are supported in the modes listed in the following table.

Linux Containers Networking Modes

Orchestration and Clustering Tool	Version	Networking Mode	Link Layer	Virtualization Mode
Docker	Docker Engine 17.03 or higher	SR-IOV using sriov-plugin along with docker run wrapper tool	InfiniBand and Ethernet	SR-IOV
Kubernetes	Kubernetes 1.10.3 or higher	SR-IOV using device plugin, and using SR- IOV CNI plugin	InfiniBand and Ethernet	SR-IOV
		VXLAN using IPoIB bridge	InfiniBand	Shared HCA

14.6.2.5.1 Docker Using SR-IOV

In this mode, Docker engine is used to run containers along with SR-IOV networking plugin. To isolate the virtual devices, docker_rdma_sriov tool should be used. This mode is applicable to both InfiniBand and Ethernet link layers.
To obtain the plugin, visit: hub.docker.com/r/rdma/sriov-plugin
To install the docker_rdma_sriov tool, use the container tools installer available via hub.docker.com/r/rdma/container_tools_installer
For instructions on how to use Docker with SR-IOV, refer to Docker RDMA SRIOV Networking with ConnectX4/ConnectX5/ConnectX6 Community post.

14.6.2.5.2 Kubernetes Using SR-IOV

In order to use RDMA in Kubernetes environment with SR-IOV networking mode, two main components are required:

1. RDMA device plugin - this plugin allows for exposing RDMA devices in a Pod
2. SR-IOV CNI plugin - this plugin provisions VF net device in a Pod

When used in SR-IOV mode, this plugin enables SR-IOV and performs necessary configuration including setting GUID, MAC, privilege mode, and Trust mode.
The plugin also allocates the VF devices when Pods are scheduled and requested by Kubernetes framework.

14.6.2.5.3 Kubernetes with Shared HCA

One RDMA device (HCA) can be shared among multiple Pods running in a Kubernetes worker nodes. User defined networks are created using VXLAN or VETH networking devices. RDMA device (HCA) can be shared among multiple Pods running in a Kubernetes worker nodes.

14.6.2.6 HPC-X

For information on HPC-X®, please refer to HPC-X User Manual at developer.nvidia.com/networking/hpc-x.

14.6.2.7 Fast Driver Unload

This feature enables optimizing mlx5 driver teardown time in shutdown and kexec flows.

The fast driver unload is disabled by default. To enable it, the `prof_sel` module parameter of mlx5_core module should be set to 3.

15 DOCA Applications

This page provides an overview of the example DOCA applications implemented on top of NVIDIA® BlueField® DPU.

All of the DOCA reference applications described in this section are governed under the BSD-3 software license agreement.

15.1 Introduction

DOCA applications are an educational resource provided as a guide on how to program on the NVIDIA BlueField networking platform using DOCA API.

For instructions regarding the development environment and installation, refer to the NVIDIA DOCA Developer Guide and the NVIDIA DOCA Installation Guide for Linux respectively.

> ⓘ For questions, comments, and feedback, please contact us at DOCA-Feedback@exchange.nvidia.com.

15.1.1 Installation

DOCA applications are installed under `/opt/mellanox/doca/applications` with each application having its own dedicated folder. Each directory contains the source code and compilation files for the matching application.

15.1.2 Prerequisites

The DOCA SDK references (samples and applications) require the use of meson, with a minimal version requirement of 0.61.2. Since this version is usually more advanced than what is provided by the distribution provider, it is recommended to install meson directly through pip instead of through upstream packages:

```
$ sudo pip3 install meson==0.61.2
```

15.1.3 Compilation

As applications are shipped alongside their sources, developers may want to modify some of the code during their development process and then recompile the applications. The files required for the compilation are the following:

- `/opt/mellanox/doca/applications/meson.build` - main compilation file for a project that contains all the applications
- `/opt/mellanox/doca/applications/meson_options.txt` - configuration file for the compilation process
- `/opt/mellanox/doca/applications/<application_name>/meson.build` - application-specific compilation definitions

To recompile all the reference applications:

1. Move to the applications directory:

   ```
   cd /opt/mellanox/doca/applications
   ```

2. Prepare the compilation definitions:

   ```
   meson /tmp/build
   ```

3. Compile all the applications:

   ```
   ninja -C /tmp/build
   ```

   ⓘ The generated applications are located under the `/tmp/build/` directory, using the following path `/tmp/build/<application_name>/doca_<application_name>`.

⚠ Compilation against DOCA's SDK relies on environment variables which are automatically defined per user session upon login. For more information, please refer to section "Meson Complains About Missing Dependencies" in the NVIDIA DOCA Troubleshooting Guide.

15.1.4 Developer Configurations

When recompiling the applications, meson compiles them by default in "debug" mode. Therefore, the binaries would not be optimized for performance as they would include the debug symbol. For comparison, the application binaries shipped as part of DOCA's installation are compiled in "release" mode. To compile the applications in something other than debug, please consult Meson's configuration guide.

The applications also offer developers the ability to use the DOCA log's `TRACE` level (`DOCA_LOG_TRC`) on top of the existing DOCA log levels. Enabling the `TRACE` log level during compilation activates various developer log messages left out of the release compilation. Activating the `TRACE` log level may be done through `enable_trace_log` in the `meson_options.txt` file, or directly from the command line:

1. Prepare the compilation definitions to use the trace log level:

   ```
   meson /tmp/build -Denable_trace_log=true
   ```

2. Compile the applications:

   ```
   ninja -C /tmp/build
   ```

15.2 Application Use of DOCA Libs

The following table maps DOCA reference applications to the libraries they make use of.

Application Category	Application	Library Category											
		BareMetal/Virtualized Cloud					Secure Cloud Gateway		Cloud Storage	Monitoring	Streaming	HPC	
		Flow	DPA	DMA	FlexIO SDK	PCC	App Shield	SHA	Compress	Telemetry	GPUNetIO	Comch	UCX
Network	Ethernet L2 Forwarding												
	GPU Packet Processing										✓		
	Simple Forward VNF	✓											
	Switch	✓											
Security	App Shield Agent						✓			✓			
	East-west Overlay Encryption												
	IPsec Security Gateway	✓											
	PSP Gateway	✓											
	Secure Channel											✓	
	YARA Inspection						✓			✓			
Data Path Acceleration	DPA All-to-all		✓										
	DPA L2 Reflector				✓								
	PCC					✓							
Storage	DMA Copy			✓								✓	
	File Compression								✓			✓	
	File Integrity							✓				✓	
HPC	UROM RDMO												✓

15.3 Applications

15.3.1 App Shield Agent

The DOCA App Shield Agent application describes how to build secure process monitoring and is based on the DOCA APSH library, which leverages DPU capabilities such as regular expression (RXP) acceleration engine, hardware-based DMA, and more.

15.3.2 DMA Copy

The DOCA DMA Copy application describes how to transfer files between the DPU and the host. The application is based on the direct memory access (DMA) library, which leverages hardware acceleration for data copy for both local and remote memory.

15.3.3 DPA All-to-all

The DOCA DPA All-to-all application is a collective operation that allows data to be copied between multiple processes. This application is implemented using DOCA DPA, which leverages the data path accelerator (DPA) inside of the BlueField-3 which offloads the copying of the data to the DPA and leaves the CPU free for other computations.

15.3.4 DPA L2 Reflector

The DOCA DPA L2 Reflector application uses the data path accelerator (DPA) engine to intercept network traffic and swap the source and destination MAC addresses of each packet. It is based on the FlexIO API which leverages DPU capabilities such as high-speed DPA.

15.3.5 East-West Overlay Encryption

The DOCA East-West Overlay Encryption application (IPsec) sets up encrypted connections between different devices and works by encrypting IP packets and authenticating the packets' originator. It is based on a strongSwan solution which is an open-source IPsec-based VPN solution.

15.3.6 Ethernet L2 Forwarding

The DOCA Ethernet L2 Forwarding application is a DOCA Ethernet based application that forwards traffic from a single RX port to a single TX port and vice versa, leveraging DOCA's task/event batching feature for enhanced performance.

15.3.7 File Compression

The DOCA File Compression application shows how to compress and decompress data using hardware acceleration and to send and receive it. The application is based on the DOCA Compress and DOCA Comm-Channel libraries.

15.3.8 File Integrity

The DOCA File Integrity application shows how to send and receive files in a secure way using the hardware Crypto engine. It is based on the DOCA SHA and DOCA Comm-Channel libraries.

15.3.9 GPU Packet Processing

The DOCA GPU Packet Processing application shows how to combine DOCA GPUNetIO, DOCA Ethernet, and DOCA Flow to manage ICMP, UDP, TCP and HTTP connections with a GPU-centric approach using CUDA kernels without involving the CPU in the main data path.

15.3.10 IPsec Gateway

The DOCA IPsec Gateway application demonstrates how to insert rules related to IPsec encryption and decryption based on the DOCA Flow and IPsec libraries, which leverage the DPU's hardware capability for secure network communication.

15.3.11 Programmable Congestion Control

The DOCA Programmable Congestion Control application, programmable congestion control, is based on the DOCA PCC library and allows users to design and implement their own congestion control algorithm, giving them good flexibility to work out an optimal solution to handle congestion in their clusters.

15.3.12 PSP Gateway

The DOCA PSP Gateway application demonstrates how to exchange keys between application instances and insert rules controlling PSP encryption and decryption using the DOCA Flow library.

15.3.13 Secure Channel

The DOCA Secure Channel application is used to establish a secure, network-independent communication channel between the host and the DPU based on the DOCA Comm Channel library.

15.3.14 Simple Forward VNF

The DOCA Simple Forward VNF application is a forwarding application that takes VXLAN traffic from a single RX port and transmits it on a single TX port. It is based on the DOCA Flow library which leverages DPU capabilities such as building generic execution pipes in the hardware, and more.

15.3.15 Switch

The DOCA Switch application is used to establish internal switching between representor ports on the DPU. It is based on the DOCA Flow library which leverages DPU capabilities such as building generic execution pipes in the hardware, and more.

15.3.16 UROM RDMO

The DOCA UROM RDMO application demonstrates how to execute an Active Message outside the context of the target process. It is based on the DOCA UROM (Unified Resources and Offload Manager) library as a framework to launch UROM workers on the DPU and using the UCX communication framework, which leverages the DPU's low-latency and high-bandwidth utilization of its network engine.

15.3.17 YARA Inspection

The DOCA YARA Inspection application describes how to build YARA rule inspection for processes and is based on the DOCA APSH library, which leverages DPU capabilities such as the regular expression (RXP) acceleration engine, hardware-based DMA, and more.

15.4 NVIDIA DOCA App Shield Agent Application Guide

This guide provides process introspection system implementation on top of NVIDIA® BlueField® DPU.

15.4.1 Introduction

App Shield Agent monitors a process in the host system using the DOCA App Shield library.

This security capability helps identify corruption of core processes in the system from an independent and trusted DPU. This is a major and innovate intrusion detection system (IDS) ability since it cannot be provided from inside the host.

The DOCA App Shield Library gives the capability to read, analyze, and authenticate the host (bare metal/VM) memory directly from the DPU.

Using the library, this application hashes the un-writeable memory pages (also unloaded pages) of a specific process and its libraries. Then, at regular intervals, the app authenticates the loaded pages.

The app reports pass/fail after every iteration until the first attestation failure. The reports are both printed to the console and exported to the DOCA Telemetry Service (DTS) using inter-process communication (IPC).

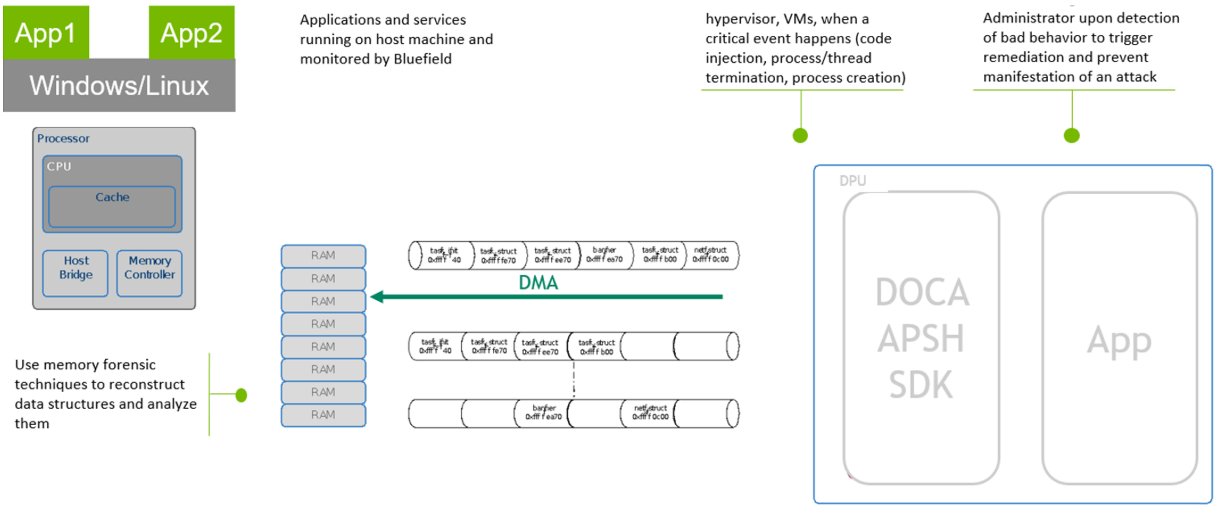
This guide describes how to build secure process monitoring using the DOCA App Shield library, which leverages the DPU's advantages such as hardware-based DMA, integrity, and more.

15.4.2 System Design

The App Shield agent is designed to run independently on the DPU's Arm without hindering the host.

The host's involvement is limited to configuring monitoring of a new process when there is a need to generate the needed ZIP and JSON files to pass to the DPU. This is done at inception ("time 0") which is when the host is still in a "safe" state.

Generating the needed files can be done by running DOCA App Shield's `doca_apsh_config.py` tool on the host. See DOCA App Shield for more info.



Applications and services running on host machine and monitored by Bluefield

hypervisor, VMs, when a critical event happens (code injection, process/thread termination, process creation)

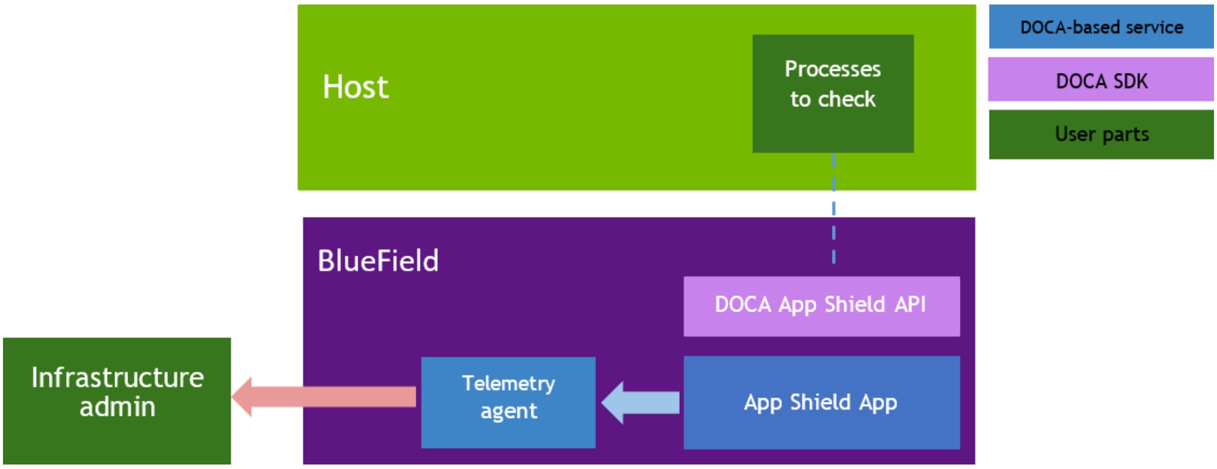
Administrator upon detection of bad behavior to trigger remediation and prevent manifestation of an attack

Use memory forensic techniques to reconstruct data structures and analyze them

15.4.3 Application Architecture

The user creates three mandatory files using the DOCA tool `doca_apsh_config.py` and copies them to the DPU. The application can report attestation results to the:

- File
- Terminal
- DTS



1. The files are generated by running `doca_apsh_config.py` on the host against the process at time zero.

   > The actions 2-5 recur at regular time intervals.

2. The App Shield agent requests new attestation from DOCA App Shield library.
3. The DOCA App Shield library creates a new attestation:
   a. Scans and hashes process memory pages (that are currently in use).
   b. Compares the hash to the original hash.
   c. Creates attestation for each lib/exe involved in the process. Each of attestation includes the number of valid pages and the number of pages.

4. The App Shield agent searches each attestation for inconsistency between number of used pages and number of valid pages.
5. The App Shield agent reports results with a timestamp and scan count to:
    a. Local telemetry files – a folder and files representing the data a real DTS would have received. These files are used for the purposes of this example only as normally this data is not exported into user-readable files.
    b. DOCA log (without scan count).
    c. DTS IPC interface (even if no DTS is active).
6. The App Shield agent exits on first attestation failure.

15.4.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA App Shield
- DOCA Telemetry Exporter

Refer to their respective programming guide for more information.

15.4.5 Compiling the Application

> Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

> For more information about the applications as well as development and compilation tips, refer to the DOCA Applications page.

The sources of the application can be found under the application's directory: `/opt/mellanox/doca/applications/app_shield_agent/` .

15.4.5.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> `doca_app_shield_agent` is created under `/tmp/build/app_shield_agent/` .

15.4.5.2 Compiling Only the Current Application

To build only the App Shield Agent application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_app_shield_agent=true
ninja -C /tmp/build
```

`doca_app_shield_agent` is created under `/tmp/build/app_shield_agent/`.

Alternatively, the user can set the desired flags in the `meson_options.txt` file instead of providing them in the compilation command line:

1. Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt`:
    - Set `enable_all_applications` to `false`
    - Set `enable_app_shield_agent` to `true`
2. Run the following compilation commands:

    ```
    cd /opt/mellanox/doca/applications/
    meson /tmp/build
    ninja -C /tmp/build
    ```

    `doca_app_shield_agent` is created under `/tmp/build/app_shield_agent/`.

15.4.5.3 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application.

15.4.6 Running the Application

15.4.6.1 Prerequisites

1. Configure the BlueField's firmware.
    a. On the BlueField system, configure the PF base address register and NVMe emulation. Run:

    ```
    dpu> mlxconfig -d /dev/mst/mt41686_pciconf0 s PF_BAR2_SIZE=2 PF_BAR2_ENABLE=1
     NVME_EMULATION_ENABLE=1
    ```

    b. Perform a BlueField system reboot for the `mlxconfig` settings to take effect.
    c. You may verify these configurations using the following command:

    ```
    dpu> mlxconfig -d /dev/mst/mt41686_pciconf0 q | grep -E "NVME|BAR"
    ```

2. Download target system (host/VM) symbols.
    - For Ubuntu:

```
host> sudo tee /etc/apt/sources.list.d/ddebs.list << EOF
deb http://ddebs.ubuntu.com/ $(lsb_release -cs) main restricted universe multiverse
deb http://ddebs.ubuntu.com/ $(lsb_release -cs)-updates main restricted universe multiverse
deb http://ddebs.ubuntu.com/ $(lsb_release -cs)-proposed main restricted universe multiverse
EOF
host> sudo apt install ubuntu-dbgsym-keyring
host> sudo apt-get update
host> sudo apt-get install linux-image-$(uname -r)-dbgsym
```

- For CentOS:

```
host> yum install --enablerepo=base-debuginfo kernel-devel-$(uname -r) kernel-debuginfo-$(uname -r)
kernel-debuginfo-common-$(uname -m)-$(uname -r)
```

- No action is needed for Windows

3. Perform IOMMU passthrough. This stage is only necessary if IOMMU is not enabled by default (e.g., when the host is using an AMD CPU).

> Skip this step if you are not sure whether it is needed. Return to it only if DMA fails with a message similar to the following in `dmesg`:
>
> ```
> host> dmesg
> [ 3839.822897] mlx5_core 0000:81:00.0: AMD-Vi: Event logged [IO_PAGE_FAULT domain=0x0047
> address=0x2a0aff8 flags=0x0000]
> ```

   a. Locate your OS's `grub` file (most likely `/boot/grub/grub.conf`, `/boot/grub2/grub.cfg`, or `/etc/default/grub`) and open it for editing. Run:

```
host> vim /etc/default/grub
```

   b. Search for the line defining `GRUB_CMDLINE_LINUX_DEFAULT` and add the argument `iommu=pt`. For example:

```
GRUB_CMDLINE_LINUX_DEFAULT="iommu=pt <intel/amd>_iommu=on"
```

   c. Run:

> Prior to performing a power cycle, make sure to do a graceful shutdown.

- For Ubuntu:

```
host> sudo update-grub
host> ipmitool power cycle
```

- For CentOS:

```
host> grub2-mkconfig -o /boot/grub2/grub.cfg
host> ipmitool power cycle
```

- For Windows targets, turn off Hyper-V capability.

4. Prepare target:

   a. Install DOCA on the target system.

   b. Create the ZIP and JSON files. Run:

```
target-system> cd /opt/mellanox/doca/tools/
```

```
target-system> python3 doca_apsh_config.py --pid <pid-of-process-to-monitor> --os <windows/linux>
--path <path to dwarf2json executable  or pdbparse-to-json.py>
target-system> cp /opt/mellanox/doca/tools/*.* <shared-folder-with-baremetal>
dpu> scp <shared-folder-with-baremetal>/* <path-to-app-shield-binary>
```

If the target system does not have DOCA installed, the script can be copied from the BlueField.
The required `dwaf2json` and `pdbparse-to-json.py` are not provided with DOCA.

> If the kernel and process `.exe` have not changed, there no need to redo this step.

15.4.6.2 Application Execution

1. The App Shield Agent application is provided in source form; hence a compilation is required before the application can be executed.
    a. Application usage instructions:

```
Usage: doca_app_shield_agent [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRI
TICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=
CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags from an input json file

Program Flags:
  -p, --pid                         Process ID of process to be attested
  -e, --ehm <path>                  Exec hash map path
  -m, --memr <path>                 System memory regions map
  -f, --vuid                        VUID of the System device
  -d, --dma                         DMA device name
  -o, --osym <path>                 System OS symbol map path
  -s, --osty <windows|linux>        System OS type - windows/linux
  -t, --time <seconds>              Scan time interval in seconds
```

> This usage printout can be printed to the command line using the `-h` (or `--help`) options:
>
> ```
> ./doca_app_shield_agent -h
> ```

> For additional information, please refer to section "Command Line Flags".

    b. CLI example for running the application on the BlueField:

```
./doca_app_shield_agent -p 13577 -e hash.zip -m mem_regions.json -o symbols.json -f
MT2125X03335MLNXS0D0F0VF1 -d mlx5_0 -t 3 -s linux
```

> All used identifiers (`-f` , `-p` and `-d` flags) should match the identifier of the desired devices and processes.

15.4.6.3 Command Line Flags

Flag Type	Short Flag	Long Flag	Description
General flags	`h`	`help`	Print a help synopsis
	`v`	`version`	Print program version information
	`l`	`log-level`	Set the log level for the application: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70 (requires compilation with `TRACE` log level support)
	N/A	`sdk-log-level`	Set the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70
	`j`	`json`	Parse all command flags from an input JSON file
Program flags	`p`	`pid`	PID of the process to be attested
	`e`	`ehm`	Path to the pre-generated `hash.zip` file transferred from the host
	`m`	`memr`	Path to the pre-generated `mem_regions.json` file transferred from the host

<table>
<thead>
<tr><th>Flag Type</th><th>Short Flag</th><th>Long Flag</th><th>Description</th></tr>
</thead>
<tbody>
<tr><td></td><td><code>f</code></td><td><code>pcif</code></td><td>System PCIe function vendor unique identifier (VUID) of the VF/PF exposed to the target system. Used for DMA operations.
To obtain this argument, run:
<pre>target-system> lspci -vv | grep
"\[VU\] Vendor specific:"</pre>
Example output:
<pre>[VU] Vendor specific:
MT2125X03335MLNXS0D0F0
[VU] Vendor specific:
MT2125X03335MLNXS0D0F1</pre>
Two VUIDs are printed for each DPU connected to the target system. The first is of the DPU on <code>pf0</code> and the second is of the DPU on port <code>pf1</code> .
<blockquote>⚠ Running this command on the DPU outputs VUIDs with an additional "EC" string in the middle. You must remove the "EC" to arrive at the correct VUID.</blockquote>
The VUID of a VF allocated on PF0/1 is the VUID of the PF with an additional suffix, <code>VF<vf-number></code> , where <code>vf-number</code> is the VF index +1.
For example, for the output in the example above:
<ul>
<li>PF0 VUID = MT2125X03335MLNXS0D0F0</li>
<li>PF1 VUID = MT2125X03335MLNXS0D0F1</li>
<li>VUID of VF0 on PF0 = MT2125X03335MLNXS0D0F0VF1</li>
</ul>
VUIDs are persistent even on reset.</td></tr>
<tr><td></td><td><code>d</code></td><td><code>dma</code></td><td>DMA device name to use</td></tr>
<tr><td></td><td><code>o</code></td><td><code>osym</code></td><td>Path to the pre-generated <code>symbols.json</code> file transferred from the host</td></tr>
<tr><td></td><td><code>s</code></td><td><code>osty</code></td><td>OS type (<code>windows</code> or <code>linux</code>) of the system where the process is running</td></tr>
<tr><td></td><td><code>t</code></td><td><code>time</code></td><td>Number of seconds to sleep between scans</td></tr>
</tbody>
</table>

ⓘ Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

15.4.6.4 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

15.4.7 Application Code Flow

1. Parse application argument.
    a. Initialize arg parser resources and register DOCA general parameters.

    ```
    doca_argp_init();
    ```

    b. Register application parameters.

    ```
    register_apsh_params();
    ```

    c. Parse the arguments.

    ```
    doca_argp_start();
    ```

2. Initialize DOCA App Shield lib context.
    a. Create lib context.

    ```
    doca_apsh_create();
    ```

    b. Set DMA device for lib.

    ```
    doca_devinfo_list_create();
    doca_dev_open();
    doca_devinfo_list_destroy();
    doca_apsh_dma_dev_set();
    ```

    c. Start the context

    ```
    doca_apsh_start();
    apsh_system_init();
    ```

3. Initialize DOCA App Shield lib system context handler.
    a. Get the representor of the remote PCIe function exposed to the system.

    ```
    doca_devinfo_remote_list_create();
    doca_dev_remote_open();
    doca_devinfo_remote_list_destroy();
    ```

    b. Create and start the system context handler.

    ```
    doca_apsh_system_create();
    doca_apsh_sys_os_symbol_map_set();
    doca_apsh_sys_mem_region_set();
    doca_apsh_sys_dev_set();
    doca_apsh_sys_os_type_set();
    doca_apsh_system_start();
    ```

4. Find target process by `pid`.

   ```
   doca_apsh_processes_get();
   ```

5. Telemetry initialization.

   ```
   telemetry_start();
   ```

   a. Initialize a new telemetry schema.
   b. Register attestation type event.
   c. Set up output to file (in addition to default IPC).
   d. Start the telemetry schema.
   e. Initialize and start a new DTS source with the `gethostname()` name as source ID.

6. Get initial attestation of the process.

   ```
   doca_apsh_attestation_get();
   ```

7. Loop until attestation validation fail.

   ```
   doca_apsh_attst_refresh();
   /* validation logic */
   doca_telemetry_exporter_source_report();
   DOCA_LOG_INFO();
   sleep();
   ```

8. DOCA App Shield Agent destroy.

   ```
   doca_apsh_attestation_free();
   doca_apsh_processes_free();
   doca_apsh_system_destroy();
   doca_apsh_destroy();
   doca_dev_close();
   doca_dev_remote_close();
   ```

9. Telemetry destroy.

   ```
   telemetry_destroy();
   ```

10. Arg parser destroy.

    ```
    doca_argp_destroy();
    ```

15.4.8 References

- `/opt/mellanox/doca/applications/app_shield_agent/`

15.5 NVIDIA DOCA DMA Copy Application Guide

This guide provides an example of a DMA Copy implementation on top of NVIDIA® BlueField® DPU.

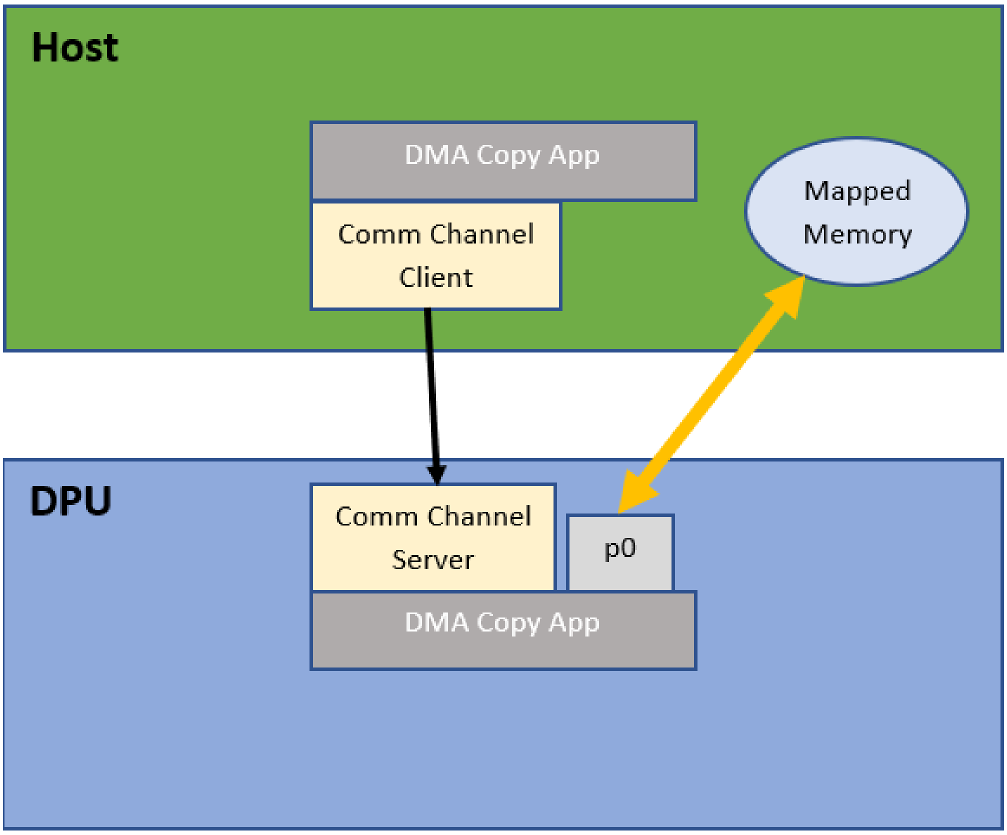
15.5.1 Introduction

DOCA DMA (direct memory access) Copy application transfers files (data path), up to the maximum supported size by the hardware, between the DPU and the x86 host using the DOCA DMA Library which provides an API to copy data between DOCA buffers using hardware acceleration, supporting both local and remote memory.

DOCA DMA allows complex memory copy operations to be easily executed in an optimized, hardware-accelerated manner.

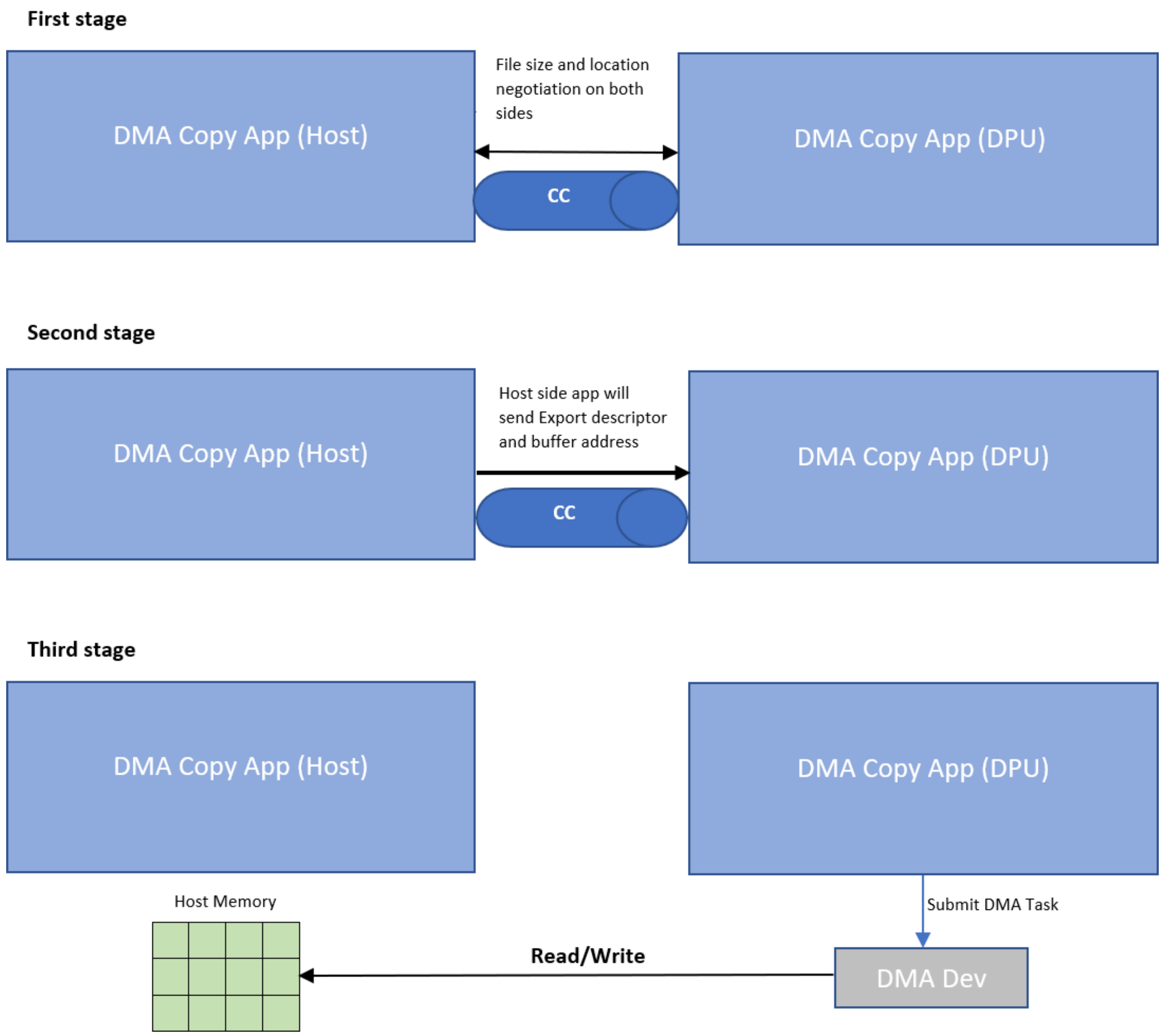
15.5.2 System Design

DOCA DMA Copy is designed to run on the instances of the BlueField DPU and x86 host. The DPU application must be the first to spawn as it opens the DOCA Comch server between the two sides on which all the necessary DOCA DMA library configuration files (control path) are transferred.



15.5.3 Application Architecture

DOCA DMA Copy runs on top of DOCA DMA to read/write directly from the host's memory without any user/kernel space context switches, allowing for a fast memory copy.



Flow:

1. The two sides initiate a short negotiation in which the file size and location are determined.
2. The host side creates the export descriptor with `doca_mmap_export_pci()` and sends it with the local buffer address and length on the Comch to the DPU side application.
3. The DPU side application uses the received export descriptor to create a remote memory map locally with `doca_mmap_create_from_export()` and the host buffer information to create a remote DOCA buffer.
4. From this point on, the DPU side application has all the necessary memory information and the DMA copy can take place.

15.5.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA DMA
- DOCA Comch

Refer to their respective programming guide for more information.

15.5.5 Compiling the Application

Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

For more information about the applications as well as development and compilation tips, refer to the DOCA Applications page.

The sources of the application can be found under the application's directory: `/opt/mellanox/doca/applications/dma_copy/` .

15.5.5.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

`doca_dma_copy` is created under `/tmp/build/dma_copy/` .

15.5.5.2 Compiling Only the Current Application

To directly build only the DMA Copy application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_dma_copy=true
ninja -C /tmp/build
```

`doca_dma_copy` is created under `/tmp/build/dma_copy/` .

Alternatively, one can set the desired flags in the `meson_options.txt` file instead of providing them in the compilation command line:

1. Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt` :
    - Set `enable_all_applications` to `false`
    - Set `enable_dma_copy` to `true`
2. Run the following compilation commands:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

`doca_dma_copy` is created under `/tmp/build/dma_copy/`.

15.5.5.3 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application.

15.5.6 Running the Application

15.5.6.1 Application Execution

The DMA Copy application is provided in source form. Therefore, a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_dma_copy [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                         Print a help synopsis
  -v, --version                      Print program version information
  -l, --log-level                    Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                    Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICA
L, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                  Parse all command flags from an input json file

Program Flags:
  -f, --file                         Full path to file to be copied/created after a successful DMA copy
  -p, --pci-addr                     DOCA Comm Channel device PCI address
  -r, --rep-pci                      DOCA Comm Channel device representor PCI address (needed only on DPU)
```

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
./doca_dma_copy -h
```

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the BlueField:

```
./doca_dma_copy -p 03:00.0 -r 3b:00.0 -f received.txt
```

Both the DOCA Comch device PCIe address (`03:00.0`) and the DOCA Comch device representor PCIe address (`3b:00.0`) should match the addresses of the desired PCIe devices.

3. CLI example for running the application on the host:

```
./doca_dma_copy -p 3b:00.0 -f send.txt
```

> The DOCA Comch device PCIe address, `3b:00.0`, should match the address of the desired PCIe device.

4. The application also supports a JSON-based deployment mode, in which all command-line arguments are provided through a JSON file:

```
./doca_dma_copy --json [json_file]
```

For example:

```
./doca_dma_copy --json ./dma_copy_params.json
```

> Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the PCIe addresses necessary for the deployment.

15.5.6.2 Command Line Flags

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
General flags	`h`	`help`	Print a help synopsis	N/A
	`v`	`version`	Print program version information	N/A
	`l`	`log-level`	Set the log level for the application: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70 (requires compilation with `TRACE` log level support)	`"log-level": 60`
	N/A	`sdk-log-level`	Set the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70	`"sdk-log-level": 40`
	`j`	`json`	Parse all command flags from an input JSON file	N/A

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
Program flags	`f`	`file`	Full path to file to be copied/created after a successful copy ⚠ This is a mandatory flag.	`"file": "/tmp/sample.txt"`
	`p`	`pci-addr`	DOCA Comch device PCIe address. ⚠ This is a mandatory flag.	`"pci-addr": "b1:00.0"`
	`r`	`rep-pci`	DOCA Comch device representor PCIe address. ⚠ This is a mandatory flag only on the DPU.	`"rep-pci": "b1:02.0"`

ⓘ Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

15.5.6.3 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

15.5.7 Application Code Flow

1. Parse application argument.
   a. Initialize arg parser resources and register DOCA general parameters.

      ```
      doca_argp_init();
      ```

   b. Register DMA Copy application parameters.

      ```
      register_dma_copy_params();
      ```

   c. Parse the arguments.

      ```
      doca_argp_start();
      ```

1. Initialize Comch endpoint.

```
    init_cc();
    ```

    a. Create Comch endpoint.
    b. Parse user PCIe address for Comch device.
    c. Open Comch DOCA device.
    d. Parse user PCIe address for Comch device representor (on DPU side).
    e. Open Comch DOCA device representor (on DPU side).
    f. Set Comch endpoint properties.

2. Open the DOCA hardware device from which the copy would be made.

    ```
    open_dma_device();
    ```

    a. Parse the PCIe address provided by the user.
    b. Create a list of all available DOCA devices.
    c. Find the appropriate DOCA device according to specific properties.
    d. Open the device.

3. Create all required DOCA core objects.

    ```
    create_core_objects();
    ```

4. Initiate DOCA core objects.

    ```
    init_core_objects();
    ```

5. Start host/DPU DMA Copy.
    a. Host side application:

        ```
        host_start_dma_copy();
        ```

        i. Start negotiation with the DPU side application for the location and size of the file.
        ii. Allocate memory for the DMA buffer.
        iii. Export the memory map and send the output (export descriptor) to the DPU side application.
        iv. Send the host local buffer memory address and length on the Comch to the DPU side application.
        v. Wait for the DPU to notify that DMA Copy ended.
        vi. Close all memory objects.
        vii. Clean resources.

    b. DPU side application:

        ```
        dpu_start_dma_copy();
        ```

        i. Start negotiation with the host side application for file location and size.
        ii. Allocate memory for the DMA buffer.
        iii. Receive the export descriptor on the Comch.
        iv. Create the DOCA memory map for the remote buffer on the host.
        v. Receive the host buffer information on the Comch.

   vi. Create two DOCA buffers, one for the remote (host) buffer and one for the local buffer.
   vii. Submit the DMA copy task.
   viii. Send a host message to notify that DMA copy ended.
   ix. Clean resources.
6. Destroy Comch.

   ```
   destroy_cc();
   ```

7. Destroy DOCA core objects.

   ```
   destroy_core_objects();
   ```

8. Arg parser destroy.

   ```
   doca_argp_destroy();
   ```

15.5.8 References

- `/opt/mellanox/doca/applications/dma_copy/`
- `/opt/mellanox/doca/applications/dma_copy/dma_copy_params.json`

15.6 NVIDIA DOCA DPA All-to-all Application Guide

This guide explains all-to-all collective operation example when accelerated using the DPA in NVIDIA® BlueField®-3 DPU.

15.6.1 Introduction

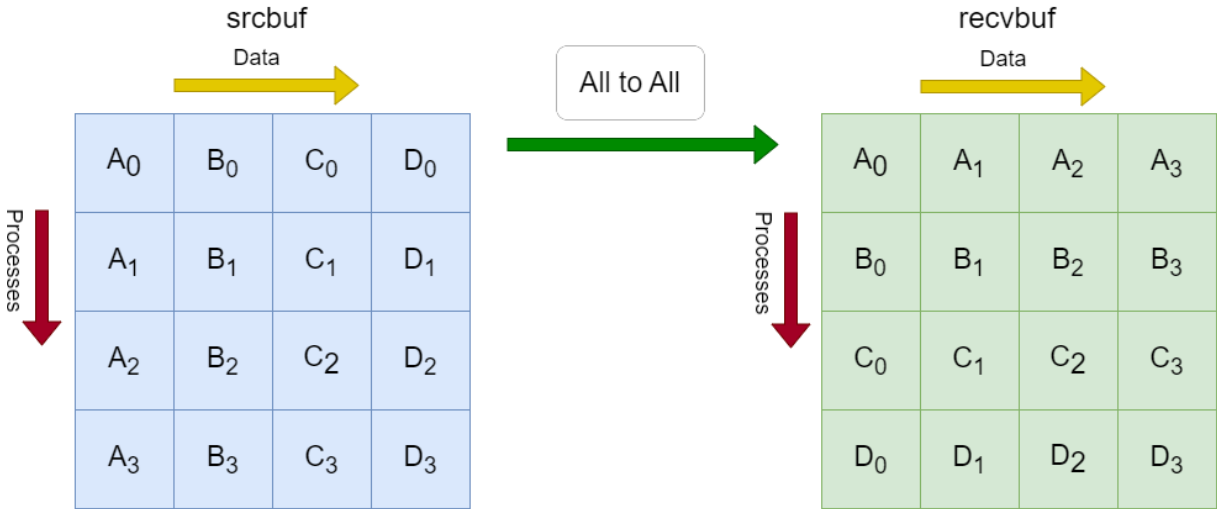
This reference application shows how the message passing interface (MPI) all-to-all collective can be accelerated on the Data Path Accelerator (DPA). In an MPI collective, all processes in the same job call the collective routine.

Given a communicator of *n* ranks, the application performs a collective operation in which all processes send and receive the same amount of data from all processes (hence all-to-all).

This document describes how to run the all-to-all example using the DOCA DPA API.

15.6.2 System Design

All-to-all is an MPI method. MPI is a standardized and portable message passing standard designed to function on parallel computing architectures. An MPI program is one where several processes run in parallel.

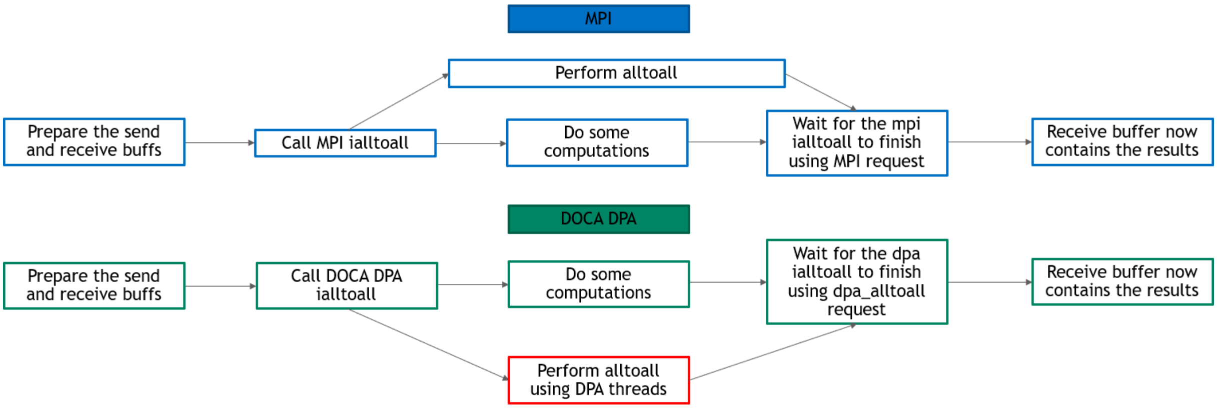


Each process in the diagram divides its local sendbuf into *n* blocks (4 in this example), each containing sendcount elements (4 in this example). Process *i* sends the *k*-th block of its local sendbuf to process *k* which places the data in the *i*-th block of its local recvbuf.

Implementing all-to-all method using DOCA DPA offloads the copying of the elements from the srcbuf to the recvbufs to the DPA, and leaves the CPU free to perform other computations.

15.6.3 Application Architecture

The following diagram describes the differences between host-based all-to-all and DPA all-to-all.


MPI
Perform alltoall
Prepare the send and receive buffs
Call MPI ialltoall
Do some computations
Wait for the mpi ialltoall to finish using MPI request
Receive buffer now contains the results
DOCA DPA
Prepare the send and receive buffs
Call DOCA DPA ialltoall
Do some computations
Wait for the dpa ialltoall to finish using dpa_alltoall request
Receive buffer now contains the results
Perform alltoall using DPA threads

- In DPA all-to-all, DPA threads perform all-to-all and the CPU is free to do other computations
- In host-based all-to-all, CPU must still perform all-to-all at some point and is not completely free for other computations

15.6.4 DOCA Libraries

This application leverages the following DOCA library:

- DOCA DPA

Refer to its programming guide for more information.

15.6.5 Dependencies

- NVIDIA BlueField-3 platform is required
- The application can be run on target BlueField or on host.
- Open MPI version 4.1.5rc2 or greater (included in DOCA's installation).

15.6.6 Compiling the Application

ⓘ Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

✓ For more information about the applications as well as development and compilation tips, refer to the DOCA Applications page.

The sources of the application can be found under the application's directory: `/opt/mellanox/doca/applications/dpa_all_to_all/` .

15.6.6.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

MPI is used for the compilation of this application. Make sure that MPI is installed on your setup (`openmpi` is provided as part of the installation of DOCA, as part of the `doca-all` and `doca-ofed` meta-packages).

⚠ Compiling the application requires updating the `LD_LIBRARY_PATH` and `PATH` environment variable to include MPI. For example, if `openmpi` is installed under `/usr/mpi/gcc/openmpi-4.1.7a1` , then updating the environment variables should be like the following

```
export PATH=/usr/mpi/gcc/openmpi-4.1.7a1/bin:${PATH}
export LD_LIBRARY_PATH=/usr/mpi/gcc/openmpi-4.1.7a1/lib:${LD_LIBRARY_PATH}
```

To build all applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

ⓘ `doca_dpa_all_to_all` is created under `/tmp/build/dpa_all_to_all/` .

15.6.6.2 Compiling DPA All-to-all Application Only

To directly build only all-to-all application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_dpa_all_to_all=true
ninja -C /tmp/build
```

> `doca_dpa_all_to_all` is created under `/tmp/build/dpa_all_to_all/`.

Alternatively, one can set the desired flags in `meson_options.txt` file instead of providing them in the compilation command line:

1. Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt`:
   - Set `enable_all_applications` to `false`
   - Set `enable_dpa_all_to_all` to `true`
2. Run the following compilation commands:

   ```
   cd /opt/mellanox/doca/applications/
   meson /tmp/build
   ninja -C /tmp/build
   ```

   > `doca_dpa_all_to_all` is created under `/tmp/build/dpa_all_to_all/`.

15.6.6.3 Troubleshooting

Please refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application.

15.6.7 Running the Application

15.6.7.1 Prerequisites

MPI is used to run this application. Make sure that MPI is installed on your setup (`openmpi` is provided as part of the installation of DOCA on the host, as part of the `doca-all` and `doca-ofed` meta-packages).

> Running the application requires updating the `LD_LIBRARY_PATH` and `PATH` environment variable to include MPI. For example, if `openmpi` is installed under `/usr/mpi/gcc/openmpi-4.1.7a1`, then updating the environment variables should be like the following:
>
> ```
> export PATH=/usr/mpi/gcc/openmpi-4.1.7a1/bin:${PATH}
> export LD_LIBRARY_PATH=/usr/mpi/gcc/openmpi-4.1.7a1/lib:${LD_LIBRARY_PATH}
> ```

15.6.7.2 Application Execution

DPA all-to-all application is provided in source form. Therefore, a compilation is required before application can be executed.

1. Application usage instructions:

   ```
   Usage: doca_dpa_all_to_all [DOCA Flags] [Program Flags]

   DOCA Flags:
     -h, --help                            Print a help synopsis
     -v, --version                         Print program version information
     -l, --log-level                       Set the (numeric) log level for the program <10=DISABLE, 20=CRITICA
   L, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
     --sdk-log-level                       Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRI
   TICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
     -j, --json <path>                     Parse all command flags from an input json file

   Program Flags:
     -m, --msgsize <Message size>          The message size - the size of the sendbuf and recvbuf (in bytes).
   Must be in multiples of integer size. Default is size of one integer times the number of processes.
     -d, --devices <IB device names>       IB devices names that supports DPA, separated by comma without
   spaces (max of two devices). If not provided then a random IB device will be chosen.
   ```

   > This usage printout can be printed to the command line using the `-h` (or `--help`) option:
   >
   > ```
   > ./doca_dpa_all_to_all -h
   > ```

   > For additional information, please refer to section "Command Line Flags".

2. CLI example for running the application on host:

   > This is an MPI program, so use `mpirun` to run the application (with the `-np` flag to specify the number of processes to run).

   - The following runs the DPA all-to-all application with 8 processes using the default message size (the number of processes, which is 8, times the size of 1 integer) with a random InfiniBand device:

     ```
     mpirun -np 8 ./doca_dpa_all_to_all
     ```

   - The following runs DPA all-to-all application with 8 processes, with 128 bytes as message size, and with `mlx5_0` and `mlx5_1` as the InfiniBand devices:

     ```
     mpirun-np 8 ./doca_dpa_all_to_all -m 128 -d "mlx5_0,mlx5_1"
     ```

     > The application supports running with a maximum of 16 processes. If you try to run with more processes, an error is printed and the application exits.

3. The application also supports a JSON-based deployment mode, in which all command-line arguments are provided through a JSON file:

```
./doca_dpa_all_to_all --json [json_file]
```

For example:

```
./doca_dpa_all_to_all --json ./dpa_all_to_all_params.json
```

Before execution, ensure that the used JSON file contains the correct configuration parameters, especially the InfiniBand device identifiers.

15.6.7.3 Command Line Flags

Flag Type	Short Flag	Long Flag/ JSON Key	Description	JSON Content
General flags	`h`	`help`	Prints a help synopsis	N/A
	`v`	`version`	Prints program version information	N/A
	`l`	`log-level`	Set the log level for the application: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70 (requires compilation with `TRACE` log level support)	`"log-level": 60`
	N/A	`sdk-log-level`	Sets the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70	`"sdk-log-level": 40`
	`j`	`json`	Parse all command flags from an input json file	N/A

Flag Type	Short Flag	Long Flag/ JSON Key	Description	JSON Content
Program flags	m	msgsize	The message size. The size of the sendbuf and recvbuf (in bytes). Must be in multiples of an integer. The default is size of 1 integer times the number of processes.	`"msgsize": -1` ⚠ The value -1 is a placeholder to use the default size, which is only known at run time (because it depends on the number of processes).
	d	devices	InfiniBand devices names that support DPA, separated by comma without spaces (max of two devices). If `NOT_SET` then a random InfiniBand device is chosen.	`"devices": "NOT_SET"`

ⓘ Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

15.6.7.4 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

15.6.8 Application Code Flow

1. Initialize MPI.

```
MPI_Init(&argc, &argv);
```

2. Parse application arguments.
   a. Initialize arg parser resources and register DOCA general parameters.

   ```
   doca_argp_init();
   ```

   b. Register the application's parameters.

   ```
   register_all_to_all_params();
   ```

   c. Parse the arguments.

   ```
   doca_argp_start();
   ```

        i. The `msgsize` parameter is the size of the sendbuf and recvbuf (in bytes). It must be in multiples of an integer and at least the number of processes times an integer size.
        ii. The `devices_param` parameter is the names of the InfiniBand devices to use (must support DPA). It can include up to two devices names.
    d. Only let the first process (of rank 0) parse the parameters to then broadcast them to the rest of the processes.

3. Check and prepare the needed resources for the `all_to_all` call:
    a. Check the number of processes (maximum is 16).
    b. Check the `msgsize`. It must be in multiples of integer size and at least the number of processes times integer size.
    c. Allocate the sendbuf and recvbuf according to `msgsize`.
4. Prepare the resources required to perform all-to-all method using DOCA DPA:

    > The application uses MPI without an additional security layer. The data exported (for sync event, RDMA, and MMAP), as described in this step, should be passed over a secure channel in a production deployment.

    a. Initialize DOCA DPA context:
        i. Open DOCA DPA device (DOCA device that supports DPA).

```
open_dpa_device(&doca_device);
```

        ii. Initialize DOCA DPA context using the opened device.

```
extern struct doca_dpa_app *dpa_all2all_app;

doca_dpa_create(doca_device, &doca_dpa);

doca_dpa_set_app(doca_dpa, dpa_all2all_app);

doca_dpa_start(doca_dpa);
```

    b. Initialize the required DOCA Sync Events for the all-to-all:
        i. One completion event for the kernel launch where the subscriber is CPU and the publisher is DPA.
        ii. Kernel events, published by remote peer and subscribed to by DPA, as the number of processes.

```
create_dpa_a2a_events() {
    // initialize completion event
    doca_sync_event_create(&comp_event);

    doca_sync_event_add_publisher_location_dpa(comp_event);

    doca_sync_event_add_subscriber_location_cpu(comp_event);

    doca_sync_event_start(comp_event);

    // initialize kernels events
    for (i = 0; i < resources->num_ranks; i++) {
        doca_sync_event_create(&(kernel_events[i]));

        doca_sync_event_add_publisher_location_remote_net(kernel_events[i]);

        doca_sync_event_add_subscriber_location_dpa(kernel_events[i]);

        doca_sync_event_start(kernel_events[i]);
    }
}
```

    c. Prepare DOCA RDMAs and set them to work on DPA:

i. Create DOCA RDMAs as the number of processes/ranks.

```
for (i = 0; i < resources->num_ranks; i++) {
    doca_rdma_create(&rdma);

    rdma_as_doca_ctx = doca_rdma_as_ctx(rdma);

    doca_rdma_set_permissions(rdma);

    doca_rdma_set_grh_enabled(rdma);

    doca_ctx_set_datapath_on_dpa(rdma_as_doca_ctx, doca_dpa);

    doca_ctx_start(rdma_as_doca_ctx);
}
```

ii. Connect local DOCA RDMAs to the remote DOCA RDMAs.

```
connect_dpa_a2a_rdmas();
```

iii. Get DPA handles for local DOCA RDMAs (so they can be used by DPA kernel) and copy them to DPA heap memory.

```
for (int i = 0; i < resources->num_ranks; i++) {
    doca_rdma_get_dpa_handle(rdmas[i], &(rdma_handles[i]));
}

doca_dpa_mem_alloc(&dev_ptr_rdma_handles);

doca_dpa_h2d_memcpy(dev_ptr_rdma_handles, rdma_handles);
```

d. Prepare the memory required to perform all-to-all method using DOCA Mmap. This includes creating DPA memory handles for sendbuf and recvbuf, getting other processes recvbufs handles, and copying these memory handles and their remote keys and events handlers to DPA heap memory.

```
prepare_dpa_a2a_memory();
```

5. Launch `alltoall_kernel` using DOCA DPA kernel launch with all required parameters:
   a. Every MPI rank launches a kernel of up to `MAX_NUM_THREADS`. This example defines `MAX_NUM_THREADS` as 16.
   b. Launch `alltoall_kernel` using `kernel_launch`.

```
doca_dpa_kernel_launch_update_set();
```

   c. Each process should perform `num_ranks` RDMA write operations, with local and remote buffers calculated based on the rank of the process that is performing the RDMA write operation and the rank of the remote process that is being written to. The application iterates over the rank of the remote process.`i`
   Each process runs `num_threads` threads on this kernel, therefore the number of RDMA write operations (which is the number of processes) is divided by the number of threads.
   Each thread should wait on its local events to make sure that the remote processes have finished RDMA write operations.
   Each thread should also synchronize its RDMA DPA handles to make sure that the local RDMA operation calls has finished.

```
for (i = thread_rank; i < num_ranks; i += num_threads) {
    doca_dpa_dev_rdma_post_write();
```

```
        doca_dpa_dev_rdma_signal_set();
    }

    for (i = thread_rank; i < num_ranks; i += num_threads) {
        doca_dpa_dev_sync_event_wait_gt();
        doca_dpa_dev_rdma_synchronize();
    }
```

> RDMA operations should be executed over a secure channel in a production deployment, given the sensitivity arising from the nature of the protocol.

d. Wait until `alltoall_kernel` has finished.

```
doca_sync_event_wait_gt();
```

> Add an MPI barrier after waiting for the event to make sure that all of the processes have finished executing `alltoall_kernel` .
>
> ```
> MPI_Barrier();
> ```

After `alltoall_kernel` is finished, the recvbuf of all processes contains the expected output of all-to-all method.

6. Destroy `a2a_resources` :
   a. Free all DOCA DPA memories.

```
doca_dpa_mem_free();
```

   b. Destroy all DOCA Mmaps

```
doca_mmap_destroy();
```

   c. Destroy all DOCA RDMAs.

```
doca_ctx_stop();
doca_rdma_destroy();
```

   d. Destroy all DOCA Sync Events.

```
doca_sync_event_destroy();
```

   e. Destroy DOCA DPA context.

```
doca_dpa_destroy();
```

   f. Close DOCA device.

```
doca_dev_close();
```

15.6.9 References

- `/opt/mellanox/doca/applications/dpa_all_to_all/`

- `/opt/mellanox/doca/applications/dpa_all_to_all/dpa_all_to_all_params.json`

15.7 NVIDIA DOCA DPA L2 Reflector Application Guide

This document provides a DPA L2 reflector implementation on top of the NVIDIA® BlueField®-3 DPU.

15.7.1 Introduction

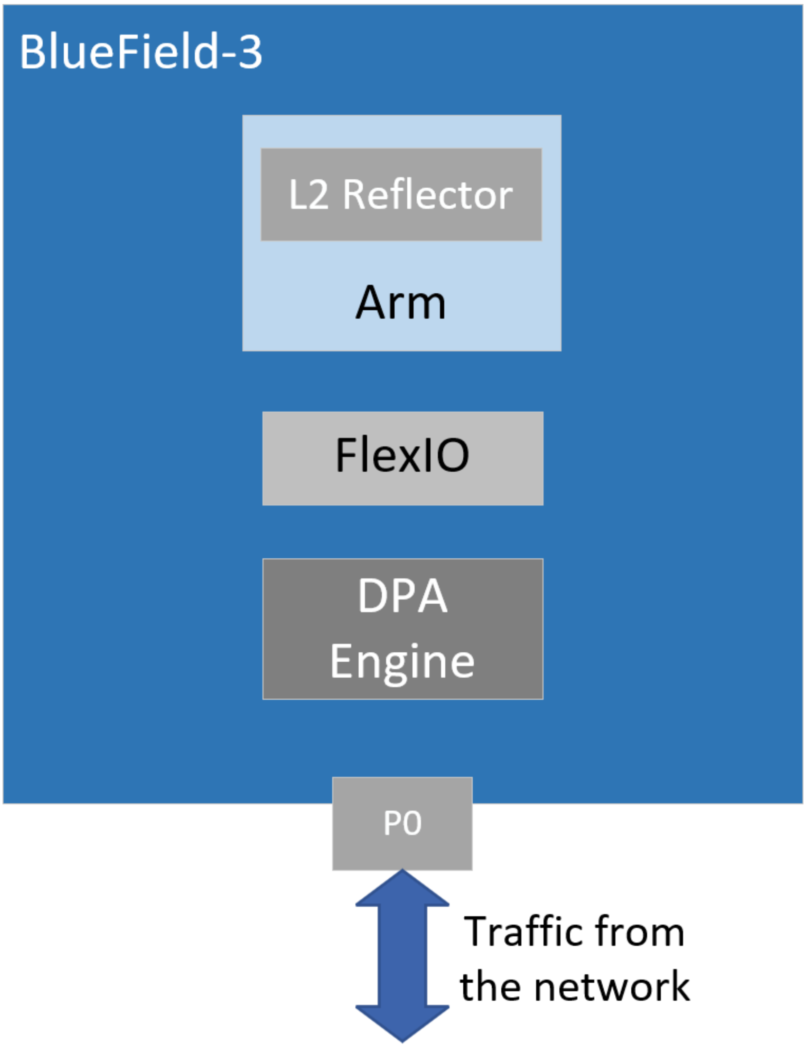
The BlueField-3 DPU supports high-speed Data Path Accelerator (DPA). Data path accelerator allows for accelerated packet processing and manipulation.

DOCA layer-2 reflector uses the DPA engine to intercept network traffic and swap the source and destination MAC addresses of each packet.

15.7.2 System Design

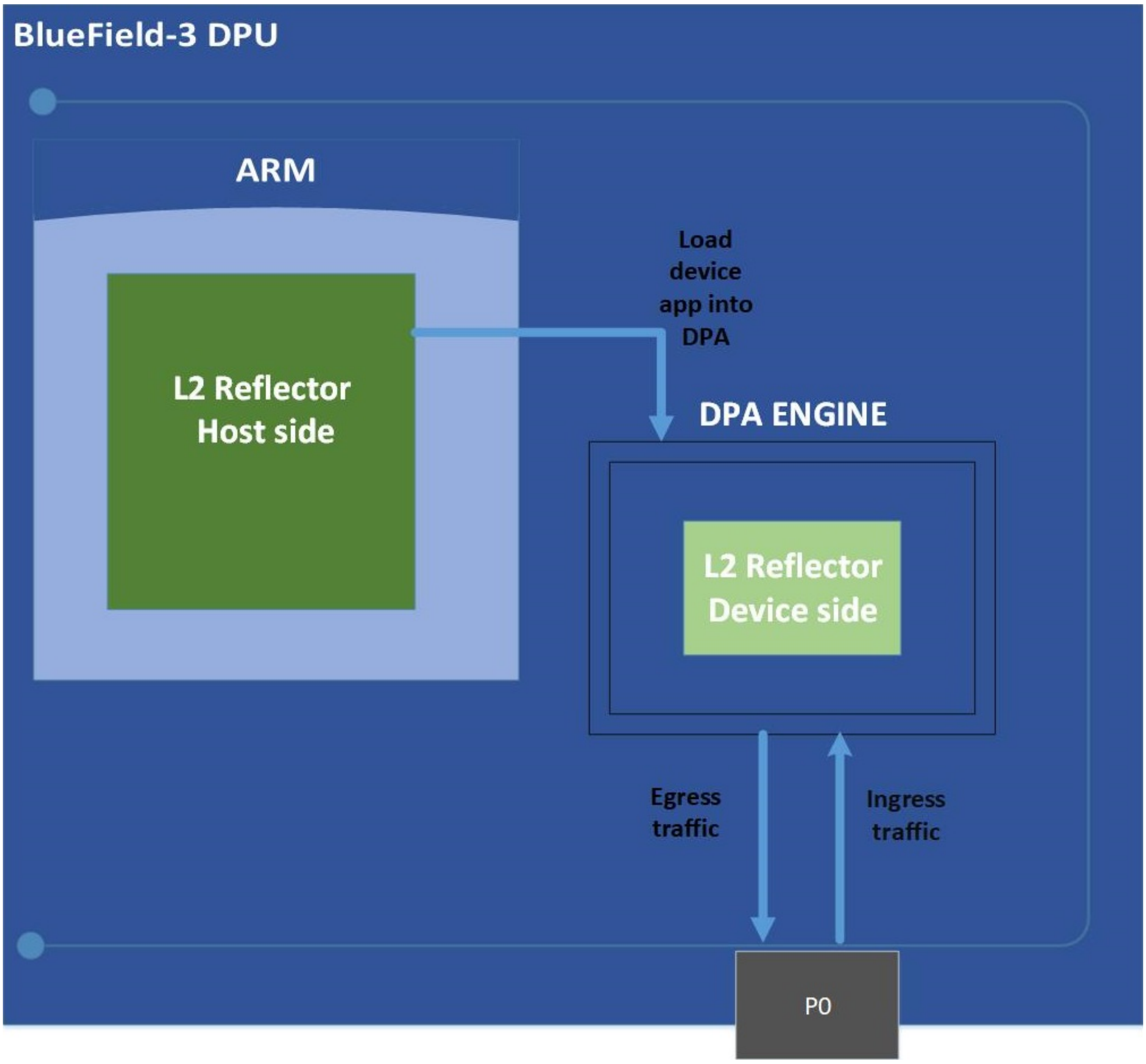
The application accepts traffic from a specific port given as an argument and leverages DPA capabilities for accelerated processing.

The following figure provides a high-level view of the components of the application:



15.7.3 Application Architecture

DOCA L2 reflector runs on top of FlexIO SDK to configure the DPA engine.



The FlexIO application consist of two parts:

- Host side - responsible for allocating resources and loading them to the DPA
- Device side – core processing logic of the application which swaps the MACs on the DPA

For more information, refer to "Programming FlexIO SDK".

15.7.4 DOCA Libraries and Drivers

This application leverages the following DOCA driver:

- FlexIO SDK

Refer to its programming guide for more information.

15.7.5 Dependencies

NVIDIA® BlueField®-3 DPU and above is required.

15.7.6 Compiling the Application

ⓘ Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

✓ For more information about the applications as well as development and compilation tips, refer to the DOCA Applications page.

The sources of the application can be found under the application's directory: `/opt/mellanox/doca/applications/l2_reflector/`.

15.7.6.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

ⓘ `l2_reflector` is created under `/tmp/build/l2_reflector/host`.

15.7.6.2 Compiling DPA L2 Reflector Application Only

To directly build only the L2 reflector application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_l2_reflector=true
ninja -C /tmp/build
```

ⓘ `l2_reflector` is created under `/tmp/build/l2_reflector/host`.

Alternatively, one can set the desired flags in the `meson_options.txt` file instead of providing them in the compilation command line:

1. Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt`:
    - Set `enable_all_applications` to `false`
    - Set `enable_l2_reflector` to `true`
2. Run the following compilation commands:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

`l2_reflector` is created under `/tmp/build/l2_reflector/host`.

15.7.6.3 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the compilation of the DOCA applications.

15.7.7 Running the Application

15.7.7.1 Application Execution

The L2 reflector application is provided in source form. Therefore, a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: l2_reflector [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICA
L, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags from an input json file

Program Flags:
  -d, --device <device name>        Device name
```

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
./l2_reflector -h
```

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on BlueField or host:

```
./l2_reflector -d mlx5_0
```

The used device name (`-d` flag) must match the identifier of the desired IB device.

To run the application on the second port, verify that it has a partition. Run:

```
dpaeumgmt partition info -d mlx5_1
```

If DPA EU partition creation is required, refer to NVIDIA DOCA DPA Execution Unit Management Tool.

3. The application also supports a JSON-based deployment mode, in which all command-line arguments are provided through a JSON file:

```
./l2_reflector --json [json_file]
```

For example:

```
./l2_reflector --json ./l2_reflector_params.json
```

Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the desired PCIe addresses required for the deployment.

15.7.7.2 Command Line Flags

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
General flags	`h`	`help`	Prints a help synopsis	N/A
	`v`	`version`	Prints program version information	N/A
	`l`	`log-level`	Set the log level for the application: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70 (requires compilation with `TRACE` log level support)	`"log-level": 60`
	N/A	`sdk-log-level`	Sets the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70	`"sdk-log-level": 40`
	`j`	`json`	Parse all command flags from an input JSON file	N/A

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
Program flags	`d`	`device`	Device name	`"device": mlx5_0`

ⓘ Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

15.7.7.3 Troubleshooting

DPA L2 reflector works with packets with a specific source MAC address. To check the supported MAC address, refer to `/opt/mellanox/doca/applications/l2_reflector/src/host/l2_reflector_core.h`.

Please refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

15.7.8 Application Code Flow

This section lists the application's configuration flow which includes different FlexIO functions and wrappers.

1. Parse application argument.
   a. Initialize arg parser resources and register DOCA general parameters.

      ```
      doca_argp_init();
      ```

   b. Register the application's parameters.

      ```
      register_l2_reflector_params();
      ```

   c. Parse the arguments.

      ```
      doca_argp_start();
      ```

2. Setup the InfiniBand device.

   ```
   l2_reflector_setup_ibv_device();
   ```

3. Setup the DPA device.

   ```
   l2_reflector_setup_device();
   ```

4. Allocate the device's resources.

```
l2_reflector_allocate_device_resources();
```

5. Run initialization function on the device.

```
flexio_process_call();
```

6. Create the steering rule.

```
l2_reflector_create_steering_rule();
```

7. Start the event handler on the device.

```
flexio_event_handler_run();
```

8. Main loop.

```
while (!force_quit)
    sleep(10);
```

9. Cleanup the resources.

```
l2_reflector_destroy();
```

15.7.9 References

- `/opt/mellanox/doca/applications/l2_reflector/`
- `/opt/mellanox/doca/applications/l2_reflector/l2_reflector_params.json`

15.8 NVIDIA DOCA East-West Overlay Encryption Application

This guide describes IPsec-based strongSwan solution on top of NVIDIA® BlueField® DPU.

> ⚠ **Important note for NVIDIA® BlueField®-2 DPUs**
>
> If your target application utilizes 100Gb/s or higher bandwidth, where a substantial part of the bandwidth is allocated for IPsec traffic, please refer to the *NVIDIA BlueField-2 DPUs Product Release Notes* to learn about a potential bandwidth limitation. To access the relevant product release notes, please contact your NVIDIA sales representative.

15.8.1 Introduction

IPsec is used to set up encrypted connections between different devices. It helps keep data sent over public networks secure. IPsec is often used to set up VPNs, and it works by encrypting IP packets as well as authenticating the packets' originator.

IPsec contains the following main modules:

- Key exchange – a key is a string of random bytes that can be used for encryption and decryption of messages. IPsec sets up keys with a key exchange between the connected devices, so that each device can decrypt the other device's messages.
- Authentication – IPsec provides authentication for each packet which ensures that they come from a trusted source.
- Encryption – IPsec encrypts the payloads within each packet and possibly, based on the transport mode, the packet's IP header.
- Decryption – at the other end of the communication, packets are decrypted by the IPsec supported node.

IPsec supports two types of headers:

- Authentication header (AH) – AH protocol ensures that packets are from a trusted source. AH does not provide any encryption.
- Encapsulating security protocol (ESP) – ESP encrypts the payload for each packet as well as the IP header depending on the transport mode. ESP adds its own header and a trailer to each data packet.

IPsec support two types of transport mode:

- IPsec tunnel mode – used between two network nodes, each acting as tunnel initiator/terminator on a public network. In this mode, the original IP header and payload are both encrypted. Since the IP header is encrypted, an IP tunnel is added for network forwarding. At each end of the tunnel, the routers decrypt the IP headers to route the packets to their destinations.
- Transport mode – the payload of each packet is encrypted, but the original IP header is not. Intermediary network nodes are therefore able to view the destination of each packet and route the packet, unless a separate tunneling protocol is used.

strongSwan is an open-source IPsec-based VPN solution. For more information, refer to strongSwan documentation.

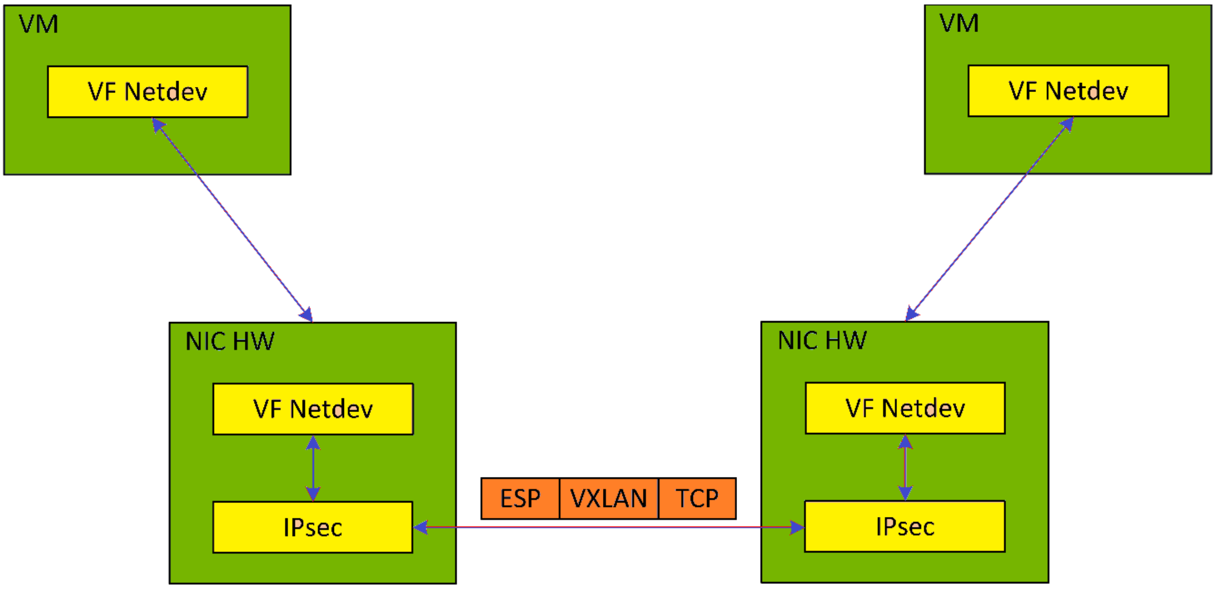
15.8.2 System Design

IPsec packet offload offloads both IPsec crypto (encrypt/decrypt) and IPsec encapsulation to the hardware.

The deployment model allows the IPsec offload to be transparent to the host with the benefits of securing legacy workloads (no dependency on host SW stack) and to zero CPU utilization on host.

IPsec packet offload configuration works with and is transparent to OVS offload. This means all packets from OVS offload are encrypted by IPsec rules.

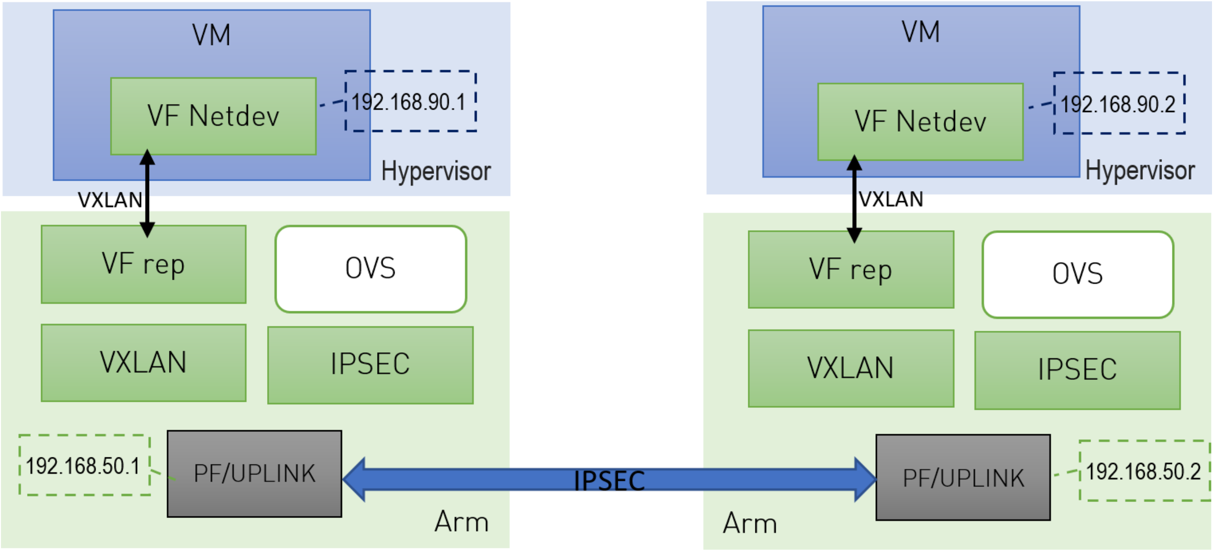
The following figure illustrates the interaction between IPsec packet offload and OVS VXLAN offload.



⚠ IPsec packet offload is only supported on Ubuntu Bluefield kernel 5.15

⚠ OVS offload and IPsec IPv6 do not work together.

15.8.3 Application Architecture



1. Configure strongSwan IPsec offload using `swanctl.conf` configuration file.
2. Traffic is sent from the host through BlueField.
3. Using OVS, the packets are encapsulated on ingress using tunnel protocols (VXLAN for example) to match IPsec configuration by strongSwan.
4. Set by strongSwan configuration file, traffic will be encrypted using the hardware offload.
5. Egress flow is decryption first, decapsulation of the tunnel header and forward to the relevant physical function.

15.8.4 DOCA Libraries

N/A

15.8.5 Configuration Flow

The following section provides information on manually configuring IPsec packet offload in general and on using OVS IPsec with strongSwan specifically.

> ⚠ There is a script, `east_west_overlay_encryption.sh` which performs the steps in this section automatically.

If you are working directly with the `ip xfrm` tool, use `/opt/mellanox/iproute2/sbin/ip` to benefit from IPsec packet offload support.

There are two parts in the configuration flow

1. Enabling IPsec packet offload mode.
2. Configuring the IPsec OVS bridge using one of three modes of authentication.

> ⚠ An alternative for step two is configuring `swanctl.conf` files (configuration files for strongSwan) manually and using strongSwan directly instead of using IPsec OVS (which automatically generates `swanctl.conf` files) as explained in section "Configuring OVS IPsec Using strongSwan Manually".

15.8.5.1 Enabling IPsec Packet Offload

This section explicitly enables IPsec packet offload on the Arm cores before setting up offload-aware IPsec tunnels.

> ⚠ If an OVS VXLAN tunnel configuration already exists, stop `openvswitch` service prior to performing the steps below and restart the service afterwards.

Explicitly enable IPsec full offload on the Arm cores.

1. Set `IPSEC_FULL_OFFLOAD="yes"` in `/etc/mellanox/mlnx-bf.conf`.

   > ⚠ If `IPSEC_FULL_OFFLOAD` does not appear in `/etc/mellanox/mlnx-bf.conf` then you are probably using an old version of the BlueField image. Check the way of enabling IPsec full offload in a previous DOCA versions in the NVIDIA DOCA Documentation Archives.

2. Restart IB driver (rebooting also works). Run:

   ```
   /etc/init.d/openibd restart
   ```

If `mlx-regex` is running:

a. Disable `mlx-regex` prior to running restarting the IB driver:

```
systemctl stop mlx-regex
```

b. Restart IB driver according to the command above.

c. Re-enable `mlx-regex` after the restart has finished:

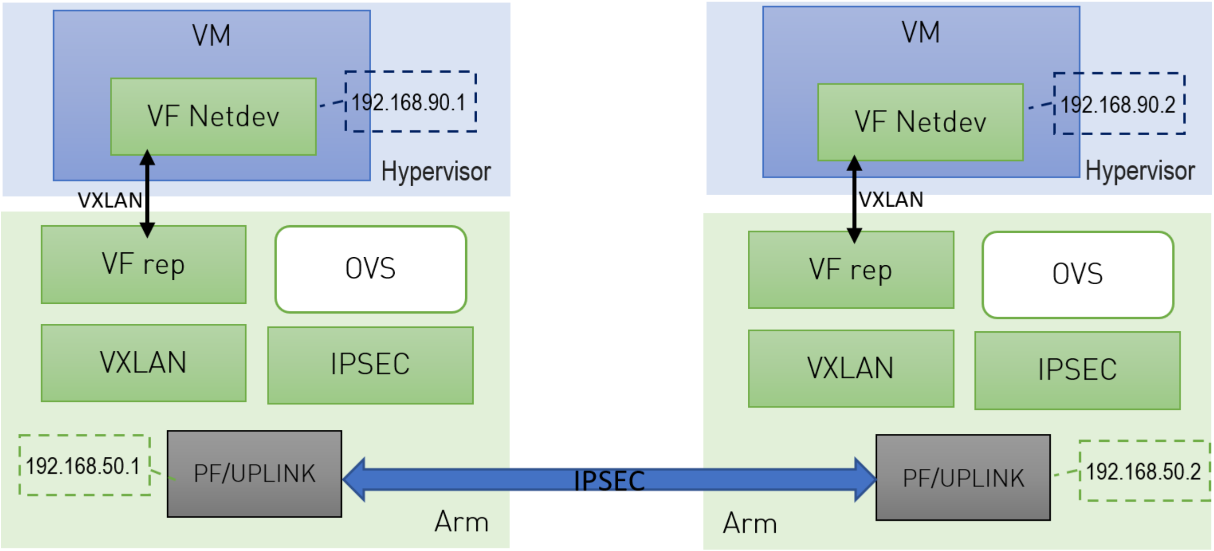
```
systemctl restart mlx-regex
```

To revert IPsec full offload mode, redo the procedure from step 1, only difference is to set `IPSEC_FULL_OFFLOAD="no"` in `/etc/mellanox/mlnx-bf.conf`.

15.8.5.2 Configuring OVS IPsec

Before proceeding with this section, make sure to follow the procedure in section "Enabling IPsec Packet Offload" for both DPUs.

This section configures OVS IPsec VXLAN tunnel which automatically generates the `swanctl.conf` files and runs strongSwan (the IPsec daemon). The following figure illustrates an example with two BlueField DPUs, Left and Right, operating with a secured VXLAN channel.



Two BlueField DPUs are required to build an OVS IPsec tunnel between the two hosts, Right and Left.

The OVS IPsec tunnel configures an unaware IPsec connection between the two hosts' InfiniBand devices. For the sake of this example, the host's InfiniBand network device is `HOST_PF`, and the DPU's host representor is `PF_REP` and the DPU's physical function `PF`.

This example sets up the following variables on both Arms:

```
# host_ip1=1.1.1.1
# host_ip2=1.1.1.2
# HOST_PF=ens7np0
# ip1=192.168.50.1
# ip2=192.168.50.2
# PF=p0
# PF_REP=pf0hpf
```

> The name of the `HOST_PF` could be different in your machine. You may verify this by running:
>
> ```
> host# ibdev2netdev
> mlx5_0 port 1 ==> ens7np0 (Down)
> mlx5_1 port 1 ==> ens8np1 (Down)
> ```
>
> This example uses the first InfiniBand's (`mlx5_0`) network device which is `ens7np0` .

1. Configure IP addresses for the `HOST_PF` s of both hosts (x86):
    a. On `host_1` :

    ```
    # ifconfig $HOST_PF $host_ip1/24 up
    ```

    b. On `host_2` :

    ```
    # ifconfig $HOST_PF $host_ip2/24 up
    ```

    > Step 1 is the only command that is performed on the host, the rest of the commands are performed on the Arm (DPU) side.

2. Configure IP addresses for the PFs of both Arms:
    a. On `Arm_1` :

    ```
    # ifconfig $PF $ip1/24 up
    ```

    b. On `Arm_2` :

    ```
    # ifconfig $PF $ip2/24 up
    ```

3. Start Open vSwitch. If your operating system is Ubuntu, run the following on both `Arm_1` and `Arm_2` :

    ```
    # service openvswitch-switch start
    ```

    If your operating system is CentOS, run the following on both `Arm_1` and `Arm_2` :

    ```
    # service openvswitch restart
    ```

4. Start OVS IPsec service. Run on both `Arm_1` and `Arm_2` :

    ```
    # systemctl start openvswitch-ipsec.service
    ```

5. Set up OVS bridges in both DPUs. Run on both `Arm_1` and `Arm_2`:

   ```
   # ovs-vsctl add-br vxlan-br
   # ovs-vsctl add-port ovs-br $PF_REP
   # ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
   ```

   > ⚠ Configuring `other_config:hw-offload=true` sets IPsec Packet offload. Setting it to `false` sets software IPsec.

   > ⚠ The MTU of the MTU of the tunnel interface (PF) should be at least 50 bytes larger than the MTU of the endpoints of the tunnels above (PF_REP) to account for the size of the VXLAN tunnel header. For example, if the MTU of PF_REP is 1500 then the MTU of PF should be at least 1550.
   >
   > To configure the MTU of the PF:
   >
   > ```
   > # ifconfig $PF mtu $PF_MTU up
   > ```

6. Set up IPsec tunnel on the OVS bridge. Three authentication methods are possible, choose your preferred authentication method and follow the steps relevant to it. Note that the last two authentication methods requires you to create certificates (self-signed certificates or certificate authority certificates).

> ⚠ After the IPsec tunnel is set up using one of the three methods of authentication, strongSwan configuration is done automatically and the `swanctl.conf` files will be generated and strongSwan will run automatically.

15.8.5.2.1 Authentication Methods

The following subsections detail the possible authentication methods for setting up the IPsec tunnel on the OVS bridge.

15.8.5.2.1.1 Pre-shared Key

This method configures OVS IPsec using a pre-shared key. You must select a pre-shared key, for example:

```
psk=swordfish
```

1. Set up the VXLAN tunnel:
   a. On `Arm_1`, run:

      ```
      # ovs-vsctl add-port vxlan-br tun -- \
                  set interface tun type=vxlan \
                                options:local_ip=$ip1 \
                                options:remote_ip=$ip2 \
                                options:key=100 \
                                options:dst_port=4789 \
                                options:psk=$psk
      ```

b. On `Arm_2`, run:

```
# ovs-vsctl add-port vxlan-br tun -- \
            set interface tun type=vxlan \
                          options:local_ip=$ip2 \
                          options:remote_ip=$ip1 \
                          options:key=100 \
                          options:dst_port=4789\
                          options:psk=$psk
```

15.8.5.2.1.2 Self-signed Certificate

This method configures OVS IPsec using self-signed certificates. You must generate self-signed certificates and keys. This example demonstrates how to generate self-signed certificates using `ovs-pki` but you may generate them in any other way while skipping step 1.

1. Generate self-signed certificates using `ovs-pki`:
   a. On `Arm_1`, run:

   ```
   # ovs-pki req -u host_1
   # ovs-pki self-sign host_1
   ```

   After running this code you should have `host_1-cert.pem` and `host_1-privkey.pem`.
   b. On `Arm_2`, run:

   ```
   # ovs-pki req -u host_2
   # ovs-pki self-sign host_2
   ```

   After running this code you should have `host_2-cert.pem` and `host_2-privkey.pem`.
2. Configure the certificates and private keys:
   a. Copy the certificate of `Arm_1` to `Arm_2`, and the certificate of `Arm_2` to `Arm_1`.
   b. On each machine, move both `host_1-privkey.pem` and `host_2-cert.pem` to `/etc/swanctl/x509/` if on Ubuntu, or `/etc/strongswan/swanctl/x509/` if on CentOS.
   c. On each machine, move the local private key (`host_1-privkey.pem` on `Arm_1` and `host_2-privkey.pem` on `Arm_2`) to `/etc/swanctl/private` if on Ubuntu, or `/etc/strongswan/swanctl/private` if on CentOS.
3. Set up OVS `other_config` on both sides.
   a. On `A rm_1`:

   ```
   # ovs-vsctl set Open_vSwitch . other_config:certificate=/etc/swanctl/x509/host_1-cert.pem \
     other_config:private_key=/etc/swanctl/private/host_1-privkey.pem
   ```

   b. On `Arm_2`:

   ```
   # ovs-vsctl set Open_vSwitch . other_config:certificate=/etc/swanctl/x509/host_2-cert.pem \
     other_config:private_key=/etc/swanctl/private/host_2-privkey.pem
   ```

4. Set up the VXLAN tunnel:
   a. On `Arm_1`:

   ```
   # ovs-vsctl add-port vxlan-br vxlanp0 -- set interface vxlanp0 type=vxlan options:local_ip=$ip1 \
     options:remote_ip=$ip2 options:key=100 options:dst_port=4789 \
     options:remote_cert=/etc/swanctl/x509/host_2-cert.pem
   # service openvswitch-switch restart
   ```

b. On `Arm_2`:

```
# ovs-vsctl add-port vxlan-br vxlanp0 -- set interface vxlanp0 type=vxlan options:local_ip=$ip2 \
  options:remote_ip=$ip1 options:key=100 options:dst_port=4789 \
  options:remote_cert=/etc/swanctl/x509/host_1-cert.pem
# service openvswitch-switch restart
```

> ⚠ In steps 3 and 4, if you are in CentOS you must change the path of the certificates to `/etc/strongswan/swanctl/x509/` and the path of the private keys to `/etc/strongswan/swanctl/private`.

15.8.5.2.1.3 CA-signed Certificate

This method configures OVS IPsec using certificate authority (CA)-signed certificates. You must generate CA-signed certificates and keys. The example demonstrates how to generate CA-signed certificates using `ovs-pki` but you may generate them in any other way while skipping step 1.

1. Generate CA-signed certificates using `ovs-pki`. For this method, all the certificates and the requests must be in the same directory during the certificate generating and signing. This example refers to this directory as `certsworkspace`.

   a. On `Arm_1`, run:

   ```
   # ovs-pki init --force
   # cp /var/lib/openvswitch/pki/controllerca/cacert.pem <path_to>/certsworkspace
   # cd <path_to>/certsworkspace
   # ovs-pki req -u host_1
   # ovs-pki sign host1 switch
   ```

   After running this code, you should have `host_1-cert.pem`, `host_1-privkey.pem`, and `cacert.pm` in the `certsworkspace` folder.

   b. On `Arm_2`, run:

   ```
   # ovs-pki init --force
   # cp /var/lib/openvswitch/pki/controllerca/cacert.pem <path_to>/certsworkspace
   # cd <path_to>/certsworkspace
   # ovs-pki req -u host_2
   # ovs-pki sign host_2 switch
   ```

   After running this code, you should have `host_2-cert.pem`, `host_2-privkey.pem`, and `cacert.pm` in the `certsworkspace` folder.

2. Configure the certificates and private keys:

   a. Copy the certificate of `Arm_1` to `Arm_2` and the certificate of `Arm_2` to `Arm_1`.

   b. On each machine, move both `host_1-privkey.pem` and `host_2-cert.pem` to `/etc/swanctl/x509/` if on Ubuntu, or `/etc/strongswan/swanctl/x509/` if on CentOS.

   c. On each machine, move the local private key (`host_1-privkey.pem` if on `Arm_1` and `host_2-privkey.pem` if on `Arm_2`) to `/etc/swanctl/private` if on Ubuntu, or `/etc/strongswan/swanctl/private` if on CentOS.

   d. On each machine, copy `cacert.pem` to the `x509ca` directory under `/etc/swanctl/x509ca/` if on Ubuntu, or `/etc/strongswan/swanctl/x509ca/` if on CentOS.

3. Set up OVS `other_config` on both sides.

   a. On `Arm_1`:

```
# ovs-vsctl set Open_vSwitch . \
        other_config:certificate=/etc/strongswan/swanctl/x509/host_1.pem \
        other_config:private_key=/etc/strongswan/swanctl/private/host_1-privkey.pem \
        other_config:ca_cert=/etc/strongswan/swanctl/x509ca/cacert.pem
```

   b. On `Arm_2`:

```
# ovs-vsctl set Open_vSwitch . \
        other_config:certificate=/etc/strongswan/swanctl/x509/host_2.pem \
        other_config:private_key=/etc/strongswan/swanctl/private/host_2-privkey.pem \
        other_config:ca_cert=/etc/strongswan/swanctl/x509ca/cacert.pem
```

4. Set up the tunnel:
   a. On `Arm_1`:

```
# ovs-vsctl add-port vxlan-br vxlanp0 -- set interface vxlanp0 type=vxlan options:local_ip=$ip1 \
options:remote_ip=$ip2 options:key=100 options:dst_port=4789 \ options:remote_name=host_2
# service openvswitch-switch restart
```

   b. On `Arm_2`:

```
# ovs-vsctl add-port vxlan-br vxlanp0 -- set interface vxlanp0 type=vxlan options:local_ip=$ip2 \
options:remote_ip=$ip1 options:key=100 options:dst_port=4789 \ options:remote_name=host_1
# service openvswitch-switch restart
```

In steps 3 and 4, if you are in CenOS you must change the path of the certificates to `/etc/strongswan/swanctl/x509/`, the path of the CA certificate to `/etc/strongswan/swanctl/x509ca/`, and the path of the private keys to `/etc/strongswan/swanctl/private/`.

15.8.5.3 Ensuring IPsec is Configured

Using `/opt/mellanox/iproute2/sbin/ip xfrm state show`, you should be able to see 4 IPsec states for the IPsec connection you configured with the keyword `in mode packet` meaning which means that you are in IPsec packet HW offload mode.

For example, after configuring IPsec using pre-shared key method, you would get something similar to the following on `Arm_1`:

```
# /opt/mellanox/iproute2/sbin/ip xfrm state show

src 192.168.50.1 dst 192.168.50.2
    proto esp spi 0xcc8bf8ad reqid 1 mode transport
    replay-window 0 flag esn
    aead rfc4106(gcm(aes)) 0x9f45cc4577e70c4e077bcc0c1473a782143e7ad199f58566519639d03b593b8996383f11 128
    anti-replay esn context:
     seq-hi 0x0, seq 0x0, oseq-hi 0x0, oseq 0x0
     replay_window 1, bitmap-length 1
     00000000
    crypto offload parameters: dev p0 dir out mode packet
    sel src 192.168.50.1/32 dst 192.168.50.2/32 proto udp sport 4789
src 192.168.50.2 dst 192.168.50.1
    proto esp spi 0xce8bf4b6 reqid 1 mode transport
    replay-window 0 flag esn
    aead rfc4106(gcm(aes)) 0xf2d0e335d9a64ef6e385a630a32b0e43bb52f581290cd34bbb8f7592d54f11657ed0258e 128
    anti-replay esn context:
     seq-hi 0x0, seq 0x0, oseq-hi 0x0, oseq 0x0
     replay_window 32, bitmap-length 1
     00000000
    crypto offload parameters: dev p0 dir in mode packet
    sel src 192.168.50.2/32 dst 192.168.50.1/32 proto udp dport 4789
src 192.168.50.1 dst 192.168.50.2
    proto esp spi 0xcb600a84 reqid 2 mode transport
    replay-window 0 flag esn
    aead rfc4106(gcm(aes)) 0x7fb26035299bcc9b973abea5d581acfbcf87cbf0bd053b745c4d95c62311f934010973f6 128
    anti-replay esn context:
     seq-hi 0x0, seq 0x0, oseq-hi 0x0, oseq 0x0
     replay_window 1, bitmap-length 1
     00000000
    crypto offload parameters: dev p0 dir out mode packet
    sel src 192.168.50.1/32 dst 192.168.50.2/32 proto udp dport 4789
src 192.168.50.2 dst 192.168.50.1
    proto esp spi 0xc137d5a0 reqid 2 mode transport
    replay-window 0 flag esn
    aead rfc4106(gcm(aes)) 0x28e3d12ad4e24aa9d9de9459de8ef8bb4379e8e12faac0054c5b629b6aa50fdeda8e4574 128
    anti-replay esn context:
     seq-hi 0x0, seq 0x0, oseq-hi 0x0, oseq 0x0
     replay_window 32, bitmap-length 1
     00000000
    crypto offload parameters: dev p0 dir in mode packet
    sel src 192.168.50.2/32 dst 192.168.50.1/32 proto udp sport 4789
```

After insuring that the IPsec connection is configured, you can send encrypted traffic between `host_1` and `host_2` using the `HOST_PF` s IP addresses.

15.8.5.4 Configuring OVS IPsec Using strongSwan Manually

This section configures an OVS VXLAN tunnel which then uses `swanctl.conf` files and runs strongSwan (the IPsec daemon) manually.

> Before proceeding with this section, make sure to follow the procedure in section "Enabling IPsec Packet Offload" for both DPUs.

1. Build a VXLAN tunnel over OVS and connect the PF representor to the same OVS bridge.
   a. On `Arm_1`:

   ```
   # ovs-vsctl add-br vxlan-br
   # ovs-vsctl add-port vxlan-br PF_REP
   # ovs-vsctl add-port vxlan-br vxlan11 -- set interface vxlan11 type=vxlan options:local_ip=$ip1 \
            options:remote_ip=$ip2 options:key=100 options:dst_port=4789 \
   # ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
   ```

   b. On `Arm_2`:

   ```
   # ovs-vsctl add-br vxlan-br
   # ovs-vsctl add-port vxlan-br PF_REP
   # ovs-vsctl add-port vxlan-br vxlan11 -- set interface vxlan11 type=vxlan options:local_ip=$ip2 \
            options:remote_ip=$ip1 options:key=100 options:dst_port=4789 \
   # ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
   ```

2. If your operating system is Ubuntu, run on both `Arm_1` and `Arm_2`:

   ```
   service openvswitch-switch start
   ```

   If your operating system is CentOS, run:

   ```
   service openvswitch restart
   ```

3. Enable TC offloading for the PF. Run on both `Arm_1` and `Arm_2`:

   ```
   # ethtool -K $PF hw-tc-offload on
   ```

4. Disable host PF as the port owner from Arm. Run on both `Arm_1` and `Arm_2`:

   ```
   # mlxprivhost -d /dev/mst/mt${pciconf} --disable_port_owner r
   ```

   > To get `${pciconf}`, run the following on the DPU:

```
# ls --color=never /dev/mst/ | grep --color=never '^m.*f0$' | cut -c 3-
```

For example:

```
# mlxprivhost -d /dev/mst/mt41686_pciconf0 --disable_port_owner r
```

5. Configure the `swanctl.conf` files for each machine. See section swanctl.conf Files.

   ⚠ Each machine should have exactly one `swanctl.conf` file in `/etc/swanctl/conf.d/`.

6. Load the `swanctl.conf` files and initialize strongSwan. Run:
   a. On the `Arm_2`, run:

   ```
   systemctl restart strongswan.service
   swanctl --load-all
   ```

   b. On the `Arm_1`, run:

   ```
   systemctl restart strongswan.service
   swanctl --load-all
   swanctl -i --child bf
   ```

Now the IPsec connection should be established.

15.8.5.5 swanctl.conf Files

strongSwan configures IPSec packet HW offload using a new value added to its configuration file `swanctl.conf`. The file should be placed under `sysconfdir` which by default can be found at `/etc/swanctl/swanctl.conf`.

The terms Left (BFL) and Right (BFR), in reference to the illustration under "Application Architecture", are used to identify the two nodes (or machines) that communicate.

⚠ Either side (BFL or BFR) can fulfill either role (initiator or receiver).

In this example, 192.168.50.1 is used for the left PF uplink and 192.168.50.2 for the right PF uplink.

```
connections {
   BFL-BFR {
      local_addrs  = 192.168.50.1
      remote_addrs = 192.168.50.2

      local {
        auth = psk
        id = host1
      }
      remote {
        auth = psk
        id = host2
      }
        children {
         bf-out {
            local_ts = 192.168.50.1/24 [udp]
            remote_ts = 192.168.50.2/24 [udp/4789]
            esp_proposals = aes128gcm128-x25519-esn
            mode = transport
            policies_fwd_out = yes
            hw_offload = packet
         }
         bf-in {
            local_ts = 192.168.50.1/24 [udp/4789]
            remote_ts = 192.168.50.2/24 [udp]
```

```
            esp_proposals = aes128gcm128-x25519-esn
            mode = transport
            policies_fwd_out = yes
            hw_offload = packet
         }
      }
      version = 2
      mobike = no
      reauth_time = 0
      proposals = aes128-sha256-x25519
   }
}

secrets {
   ike-BF {
      id-host1 = host1
      id-host2 = host2
      secret = 0sv+NkxY9LLZvwj4qCC2o/gGrWDF2d21jL
   }
}
```

The BFB installation will place two example `swanctl.conf` files for BFL and BFR (`BFL.swanctl.conf` and `BFR.swanctl.conf` respectively) in the strongSwan `conf.d` directory. Each node should have only one `swanctl.conf` file in its strongSwan `conf.d` directory.

Note that:

- " `hw_offload = packet"` is responsible for configuring IPsec packet offload
- Packet offload support has been added to the existing `hw_offload` field and preserves backward compatibility.
  For your reference:

Value	Description
no	Do not configure HW offload.
crypto	Configure crypto HW offload if supported by the kernel and hardware, fail if not supported.
yes	Same as crypto (considered legacy).
packet	Configure packet HW offload if supported by the kernel and hardware, fail if not supported.
auto	Configure packet HW offload if supported by the kernel and hardware, do not fail (perform fallback to crypto or no as necessary).

- Whenever the value of `hw_offload` is changed, strongSwan configuration must be reloaded.
- Switching to crypto HW offload requires setting up `devlink/ipsec_mode` to `none` beforehand.
- Switching to packet HW offload requires setting up
- `[udp/4789]` is crucial for instructing strongSwan to IPSec only VXLAN communication.
- Packet HW offload can only be done on what is streamed over VXLAN.

Mind the following limitations:

Fields	Limitation
reauth_time	Ignored if set
rekey_time	Do not use. Ignored if set.
rekey_bytes	Do not use. Not supported and will fail if it is set.
rekey_packets	Use for rekeying

15.8.6 Running the Application

15.8.6.1 Installation

Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

15.8.6.2 Application Execution

Notes:

- IPsec daemons are started by `systemd strongswan.service`
- Use `systemctl [start | stop | restart]` to control IPsec daemons through `strongswan.service`. For example, to restart, run:

  ```
  systemctl restart strongswan.service
  ```

  This command effectively does the same thing as `ipsec restart`.

  ⚠ Do not use the `ipsec` script (located under `/usr/sbin/ipsec`) to restart/stop/start the IPsec connection.

This subsection explains how to configure and set an IPsec connection using the script. To configure the IPsec connection, you need two DPUs, referred to as the initiator and receiver machines. There are no differences between the two machines except that the initiator is the one that initiates the connection between the two (and should run the script after the receiver).

The script is located under `/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh`.

15.8.6.2.1 Script Parameters

Parameter	Description	Valid Values	Use when	Notes
`side`	The side of the connection (receiver or initiator).	• `r|receiver` • `i|intitiator`	Always	This parameter must be always passed on the command line and cannot be passed in the JSON parameter file.

Parameter	Description	Valid Values	Use when	Notes
`j|json`	The JSON parameters file full path.	JSON file path, written according to the template demonstrated in the following file: `/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption_params.json`.	To pass the parameters as a JSON file	When running the script with JSON file, you cannot pass on the command line other parameters than the `side` and the JSON file.
`initiator_ip_addr`	The IP address of the initiator machine's port interface for the IPsec connection.	A valid IP address, ranging from 1.1.1.1 to 255.255.255.255.	Always	In the JSON file, it is set by default to 192.168.50.1.
`receiver_ip_addr`	The IP address of the receiver machine's port interface for the IPsec connection.	A valid IP address, ranging from 1.1.1.1 to 255.255.255.255.	Always	In the JSON file, it is set by default to 192.168.50.2.
`port_num`	The number of the port interface (p0/p1) for the IPsec connection.	0 or 1.	Always	In the JSON file, it is set by default to 0.
`auth_method`	the authentication method of IPsec. can be `psk` (pre-shared key), `ssc` (self-signed certificate) or `ca` (CA-signed certificate). Set by default to `psk`.	Can be `psk` (pre-shared key), `ssc` (self-signed certificate) or `ca` (CA-signed certificate).	Always	In the JSON file, it is set by default to `psk.`
`preshared_key`	The pre-shared key.	A sequence of characters (string).	The `auth_method` parameter is `psk`	In the JSON file it is set by default to `swordfish`. Both the initiator and receiver must configure the same `preshared_key`.
`initiator_cert_path`	The initiator's certificate.	Any valid self-signed or CA signed certificate. Must provide full path of certificate.	The `auth_method` parameter is `ssc` or `ca`	Both the initiator and receiver must configure the same `initiator_cert_path`.
`receiver_cert_path`	The receiver's certificate.	Any valid self-signed or CA signed certificate. Must provide full path of certificate.	The `auth_method` parameter is `ssc` or `ca`	Both the initiator and receiver must configure the same `receiver _cert_path`.

Parameter	Description	Valid Values	Use when	Notes
`initiator_key_path`	the initiator's private-key.	Any valid private key that is generated with the certificate. Must provide full path of private key.	The `side` parameter is set to `initiator` and the `auth_method` is set to `ssc` or `ca`	N/A
`receiver_key_path`	the receiver's private-key.	Any valid private key that is generated with the certificate. Must provide full path of private key.	The `side` parameter is set to `receiver` and the `auth_method` is set to `ssc` or `ca`	N/A
`initiator_cacert_path`	The initiator's CA certificate.	Any valid CA certificate. Must provide full path of certificate.	The `side` and `auth_method` parameters are set to `initiator` and `ca` respectively	N/A
`receiver_cacert_path`	The receiver's CA certificate.	Any valid CA certificate. Must provide full path of certificate.	The `side` and `auth_method` parameters are set to `receiver` and `ca` respectively	N/A
`initiator_cn`	The common name (CN) of the initiator's certificate.	Must be the same as the CN described in the initiator's certificate.	The `side` and `auth_method` parameters are set to `receiver` and `ca` respectively	N/A
`receiver_cn`	The CN of the receiver's certificate.	Must be the same as the CN described in the receiver's certificate.	The `side` and `auth_method` parameters are set to `initiator` and `ca` respectively	N/A

There are two ways of passing the parameters, either using the JSON parameters file or by passing the parameters on the command line.

15.8.6.2.2 Using JSON Parameters File

In this method, you must configure the parameters file and the then run the script:

1. Configure the JSON parameters file located under `/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption_params.json` or create a JSON file according to the template of `east_west_overlay_encryption_params.json` for the script according to the explanation under section "Script Parameters".
2. Run the script on the receiver's DPU with the JSON file:

   ```
   /opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh --side=r --
   json=/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption_params.json
   ```
3. Run the script on the initiator's DPU:

   ```
   /opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh --side=i --
   json=/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption_params.json
   ```
   You may now send encrypted data over the PF interface (192.168.50.[1|2]) configured for VXLAN.

15.8.6.2.3 Passing Parameters on Command Line

In this method, you do not need to configure the parameters file and can run the script with the appropriate parameters.

15.8.6.2.3.1 Passing Parameters for Pre-shared Key Authentication Method

1. Run the script on the receiver's DPU:

   ```
   /opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh --side=r --
   initiator_ip_addr=INITIATOR_IP_ADDRESS --receiver_ip_addr=RECEIVER_IP_ADDRESS --port_num=PORT_NUM \
   --auth_method=psk --preshared_key=PRESHARED_KEY
   ```
2. Run the script on the initiator's DPU:

   ```
   /opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh --side=i --
   initiator_ip_addr=INITIATOR_IP_ADDRESS --receiver_ip_addr=RECEIVER_IP_ADDRESS --port_num=PORT_NUM \
   --auth_method=psk --preshared_key=PRESHARED_KEY
   ```

15.8.6.2.3.2 Passing Parameters for Self-signed Certificates Authentication Method

1. Run the script on the receiver's DPU:

   ```
   /opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh --side=r --
   initiator_ip_addr=INITIATOR_IP_ADDRESS --receiver_ip_addr=RECEIVER_IP_ADDRESS --port_num=PORT_NUM \
   --auth_method=ssc --initiator_cert_path=INITIATOR_CERT_PATH --receiver_cert_path=RECEIVER_CERT_PATH --
   receiver_key_path=RECEIVER_KEY_PATH
   ```
2. Run the script on the initiator's DPU:

   ```
   /opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh --side=i --
   initiator_ip_addr=INITIATOR_IP_ADDRESS --receiver_ip_addr=RECEIVER_IP_ADDRESS --port_num=PORT_NUM \
   --auth_method=ssc --initiator_cert_path=INITIATOR_CERT_PATH --receiver_cert_path=RECEIVER_CERT_PATH --
   initiator_key_path=INITIATOR_KEY_PATH
   ```

15.8.6.2.3.3 Passing Parameters for CA Certificates Authentication Method

1. Run the script on the receiver's DPU:

```
/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh --side=r --
initiator_ip_addr=INITIATOR_IP_ADDRESS --receiver_ip_addr=RECEIVER_IP_ADDRESS --port_num=PORT_NUM \
--auth_method=ca --initiator_cert_path=INITIATOR_CERT_PATH --receiver_cert_path=RECEIVER_CERT_PATH --
receiver_key_path=RECEIVER_KEY_PATH --receiver_cacert_path=RECEIVER_CACERT_PATH --initiator_cn=INITIATOR_CN
```

2. Run the script on the initiator's DPU:

```
/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh --side=i --
initiator_ip_addr=INITIATOR_IP_ADDRESS --receiver_ip_addr=RECEIVER_IP_ADDRESS --port_num=PORT_NUM \
--auth_method=ssc --initiator_cert_path=INITIATOR_CERT_PATH --receiver_cert_path=RECEIVER_CERT_PATH --
initiator_key_path=INITIATOR_KEY_PATH --initiator_cacert_path=INITIATOR_CACERT_PATH --
receiver_cn=RECEIVER_CN
```

For help and usage, run the script with `--help` / `-h` flag:

```
/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh -h
```

15.8.6.3 Troubleshooting

Please refer to the NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation or execution of the DOCA applications.

15.8.6.4 Building strongSwan

> Perform the following only if you want to build your own BFB and would like to rebuild strongSwan.

1. Install dependencies mentioned here. libgmp-dev is missing from that list, so make sure to install that as well.
2. Git clone https://github.com/Mellanox/strongswan.git.
3. Git checkout BF-5.9.10. This branch is based on the official strongSwan 5.9.10 branch (https://github.com/strongswan/strongswan/tree/5.9.10) with added packaging and support for DOCA IPsec plugin (check NVIDIA DOCA IPsec Security Gateway Application Guide for more information regarding strongSwan DOCA plugin).
4. Run autogen.sh within the strongSwan repo.
5. Run the following:

```
configure --enable-openssl --disable-random --prefix=/usr/local --sysconfdir=/etc --enable-systemd
make
make install
```

Notes:

- `--enable-systemd` enables the systemd service for strongSwan present inside the GitHub repo (see step 3) at `init/systemd-starter/strongswan.service.in`.
- When building strongSwan on your own, the `openssl.cnf.mlnx` file, required for PK and RNG HW offload via OpenSSL plugin, is not installed. It must be copied over manually from GitHub repo inside the `openssl-conf` directory. See section "Running Strongswan Example" for important notes.
- The `openssl.cnf.mlnx` file references PKA engine shared objects. libpka (version 1.3 or later) and openssl (version 1.1.1) must be installed for this to work.

15.8.6.5 Reverting IPsec Configuration

To destroy IPsec configuration, run the following on both machines:

```
/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh -d
```

⚠ Make sure to run this command only after at least two minutes have passed from running the application on either machines. Otherwise, this may lead to errors.

⚠ If you run this command without initializing the connection first (in Running strongSwan Example), you may receive errors. These errors have no functional impact and may be safely ignored.

15.8.7 References

- `/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh`
- `/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption_params.json`

15.9 NVIDIA DOCA Eth L2 Forwarding Application Guide

This document provides an Ethernet L2 Forwarding implementation on top of the NVIDIA® BlueField® DPU.

15.9.1 Introduction

The Ethernet L2 Forwarding application is a DOCA Ethernet based application that forwards traffic from a single RX port to a single TX port and vice versa, leveraging DOCA's task/event batching feature for enhanced performance.

The application can run both on the host and the BlueField, and has two main modes:

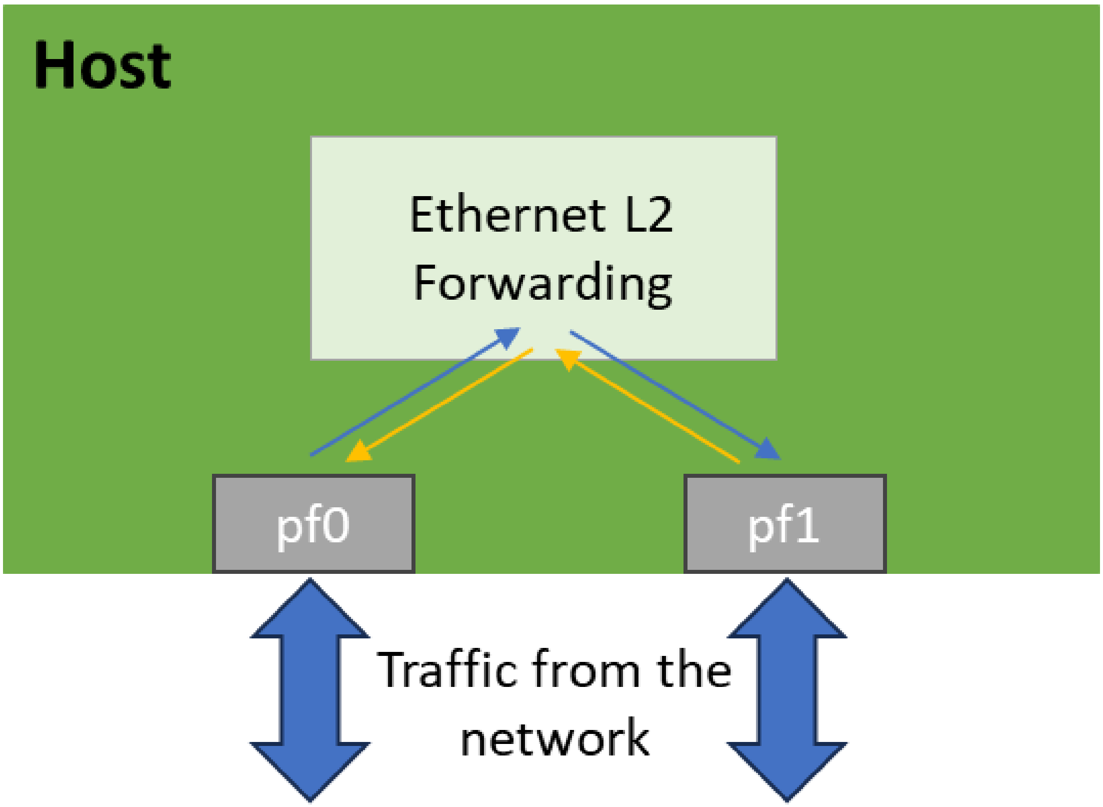
- Two-sided forwarding - device 1 → device 2 and device 2 → device 1
- One-sided forwarding - device 1 → device 2 or device 2 → device 1

The one-sided mode offers better performance, enlarging the packets forwarding rate.

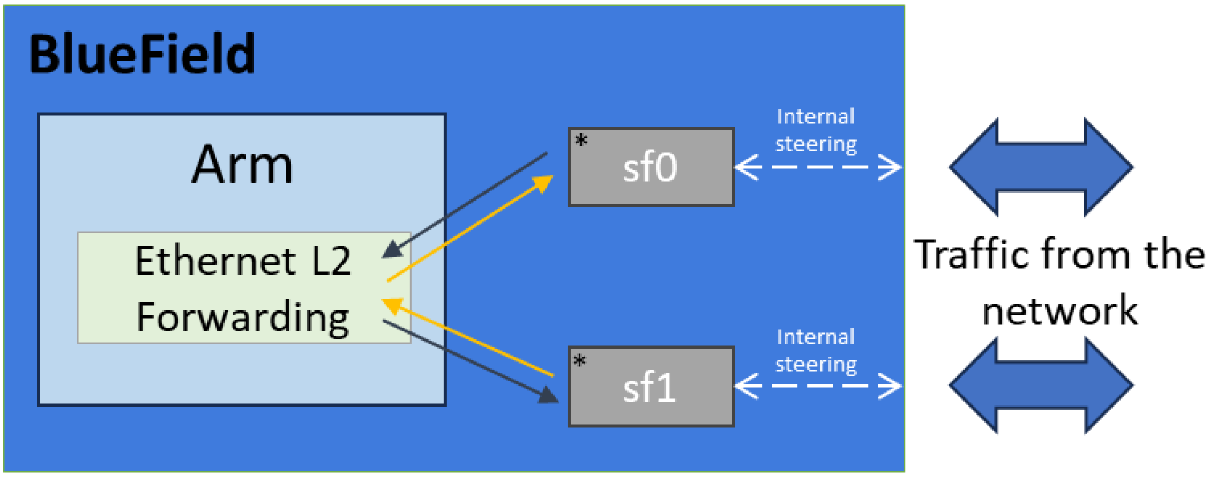
15.9.2 System Design

The following diagram shows the application running on the host:

The Ethernet L2 Forwarding application runs on the host or the BlueField.



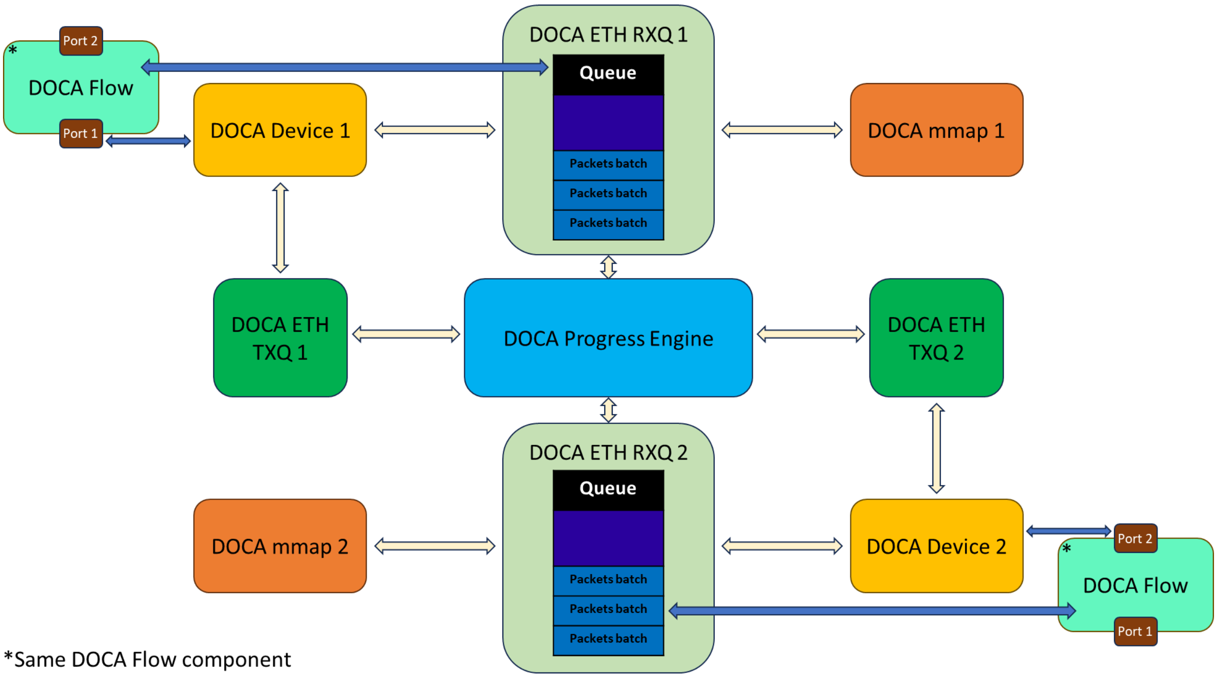
The following diagram shows the application running on the BlueField:



*DOCA Ethernet library only handles SFs

15.9.3 Application Architecture

The Ethernet L2 Forwarding application runs on top of the DOCA Ethernet API to form an (two/one-sided) L2 forwarding between two ports.



*Same DOCA Flow component

1. Two DOCA devices are opened.
2. Two DOCA mmaps are created.
3. Two DOCA Flow ports are configured and started, each with a different opened DOCA device.
4. Two DOCA Ethernet TXQ and RXQ contexts are initialized, each TXQ-RXQ pair with a different opened DOCA device such that traffic is steered from the device to the corresponding RXQ, and from the corresponding TXQ to the device.
5. Forwarding - Packets received by device x are steered to RXQ x, then allocated to TXQ y and sent by device y (and vice versa).

15.9.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Ethernet - Programming Guide
- DOCA Flow - Programming Guide

For additional information about the used DOCA libraries, please refer to the respective programming guides.

15.9.5 Compiling the Application

15.9.5.1 Installation

Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

15.9.5.2 Overview

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for both compilation of the applications "as-is", as well as provides the ability to modify the sources and then compile the new version of the application. For more information about the applications, as well as development and compilation tips, please refer to the DOCA Applications main guide.

The sources of the application can be found under the application's directory: `/opt/mellanox/doca/applications/eth_l2_fwd/`.

15.9.5.3 Compiling All Applications

The applications are all defined under a single meson project, meaning that the default compilation will compile all the DOCA applications.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

⚠ `doca_eth_l2_fwd` will be created under `/tmp/build/eth_l2_fwd/`.

15.9.5.4 Compiling Only the Current Application

1. To directly build only the Ethernet L2 Forwarding application:

   ```
   cd /opt/mellanox/doca/applications/
   meson /tmp/build -Denable_all_applications=false -Denable_eth_l2_fwd=true
   ninja -C /tmp/build
   ```

   ⚠ `doca_eth_l2_fwd` will be created under `/tmp/build/eth_l2_fwd/`.

2. Alternatively, one can set the desired flags in the `meson_options.txt` file instead of providing them in the compilation command line:
   a. Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt`:
      - Set `enable_all_applications` to `false`
      - Set `enable_eth_l2_fwd` to `true`
   b. The same compilation commands should be used, as were shown in the previous section:

      ```
      cd /opt/mellanox/doca/applications/
      meson /tmp/build
      ninja -C /tmp/build
      ```

      ⚠ `doca_eth_l2_fwd` will be created under `/tmp/build/eth_l2_fwd/`.

15.9.5.5 Troubleshooting

Please refer to the NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the compilation of the DOCA applications.

15.9.6 Running the Application

15.9.6.1 Application Execution

The Ethernet L2 Forwarding application is provided in source form, hence a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_eth_l2_fwd [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                          Print a help synopsis
  -v, --version                       Print program version information
  -l, --log-level                     Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                     Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICA
L, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                   Parse all command flags from an input json file

Program Flags:
  -d, --devs-names <name1,name2>    Set two IB devices names separated by a comma, without spaces.
  -r, --rate <rate>                 Set packets receive rate (in [MB/s]), default is 12500.
  -ps, --pkt-size <size>            Set max packet size (in [B]), default is 1600.
  -t, --time <time>                 Set packet max process time (in [µs]), default is 1.
  -nt, --num-tasks <num>            Set number of tasks per batch, default is 128.
  -nb, --num-batches <num>          Set number of task batches, default is 32.
  -o, --one-sided-forwarding <num>  Set one-sided forwarding: 0 - two-sided forwarding, 1 - device 1 ->
device 2, 2 - device 2 -> device 1. default is 0.
  -f, --max-forwardings <num>       Set max forwardings after which the application run will end, default
 is 0, meaning no limit.
```

   For additional information, please refer to the Command Line Flags section below.

   The above usage printout can be printed to the command line using the `-h` (or `--help`) options:

   ```
   ./doca_eth_l2_fwd -h
   ```

2. CLI example for running the application either on the BlueField or on the host:

   ```
   ./doca_eth_l2_fwd -d mlx5_0,mlx5_1
   ```

   Both IB devices identifiers (`mlx5_0, mlx5_1`) should match the identifiers of the desired IB devices.

3. The application also supports a JSON-based deployment mode, in which all command-line arguments are provided through a JSON file:

   ```
   ./doca_eth_l2_fwd --json [json_file]
   ```

   For example:

```
./doca_eth_l2_fwd --json ./eth_l2_fwd_params.json
```

⚠ Before execution, please ensure that the used JSON file contains the correct configuration parameters, and especially the desired IB devices names needed for the deployment.

15.9.6.2 Command Line Flags

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
General flags	`h`	`help`	Prints a help synopsis	N/A
	`v`	`version`	Prints program version information	N/A
	`l`	`log-level`	Set the log level for the application: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70 (Requires compilation with Trace level support)	`"log-level": 60`
	N/A	`sdk-log-level`	Sets the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70	N/A
	`j`	`json`	Parse all command flags from an input json file	N/A
Program flags	`d`	`devs-names`	**Two** IB devices names, separated by a comma, without spaces. ⚠ This is a mandatory flag.	`"devs-names": "mlx5_0,mlx5_1"`
	`r`	`rate`	The rate (in [MB/s]) in which the RX port is expected to receive traffic.	`"rate": 12500`
	`ps`	`pkt-size`	The maximum size (in [B]) of a received packet.	`"pkt-size": 1600`

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
	`t`	`time`	The maximum time taking to process a single packet.	`"time": 1`
	`nt`	`num-tasks`	The number of tasks to set per a single task batch.	`"num-tasks": 128`
	`nb`	`num-batches`	The number of task batches to set for the TX side.	`"num-batches": 32`
	`o`	`one-sided-forwarding`	Flag to set one of 3 options: 0 - Two-sided forwarding. 1 - One-sided forwarding from device 1 to device 2. 2 - One-sided forwarding from device 2 to device 1.	`"one-sided-forwarding": 0`
	`f`	`max-forwardings`	The maximum number of forwarding	`"max-forwardings": 32`

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

15.9.6.3 Troubleshooting

Please refer to the NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation or execution of the DOCA applications.

15.9.7 Application Code Flow

1. Parse application argument.
   a. Initialize Arg parser resources and register DOCA general parameters.

      ```
      doca_argp_init();
      ```

   b. Register Ethernet L2 Forwarding application parameters.

      ```
      register_eth_l2_forwarding_params();
      ```

   c. Parse the arguments.

      ```
      doca_argp_start();
      ```

      i. Parse DOCA flags.
      ii. Parse application parameters.
2. Execute Ethernet L2 Forwarding application main logic.

   ```
   eth_l2_fwd_execute();
   ```

    a. Open the two chosen DOCA devices.
    b. Initialize necessary DOCA Core objects.
    c. Initialize ETH RXQ/TXQ contexts for the devices.
    d. Forward packets.

3. Clean up application resources.

    ```
    eth_l2_fwd_cleanup();
    ```

    a. Stop all contexts and drain tasks.
    b. Free all application resources.

4. Arg parser destroy.

    ```
    doca_argp_destroy()
    ```

15.9.8 References

- `/opt/mellanox/doca/applications/eth_l2_fwd/`
- `/opt/mellanox/doca/applications/eth_l2_fwd/eth_l2_fwd_params.json`

15.10 NVIDIA DOCA File Compression Application Guide

This document provides a file compression implementation on top of the NVIDIA® BlueField® DPU.

15.10.1 Introduction

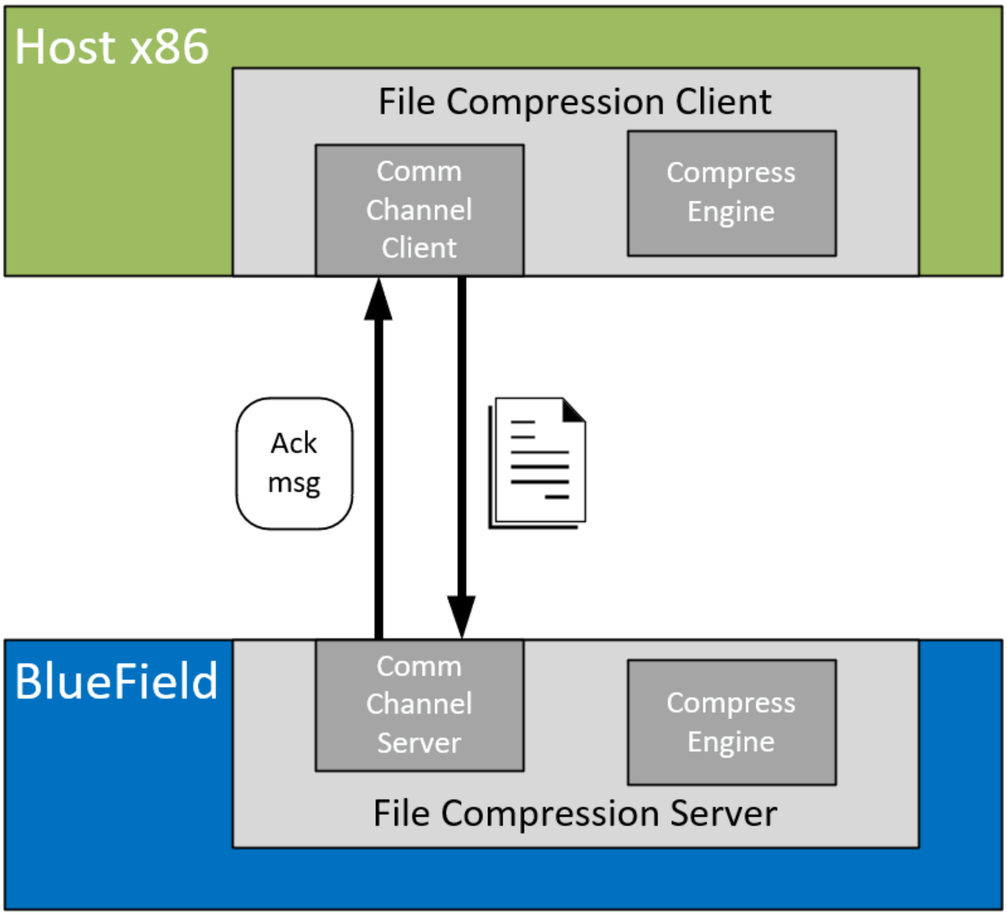
The file compression application exhibits how to use the DOCA Compress API to compress and decompress data using hardware acceleration as well as sending and receiving it using the DOCA Comch API.

The application's logic includes both a client and a server:

- Client side – the application opens a file, compresses it, and sends the checksum of the source file with the compressed data to the server
- Server side – the application saves the received file in a buffer, decompresses it, and compares the received checksum with the calculated one

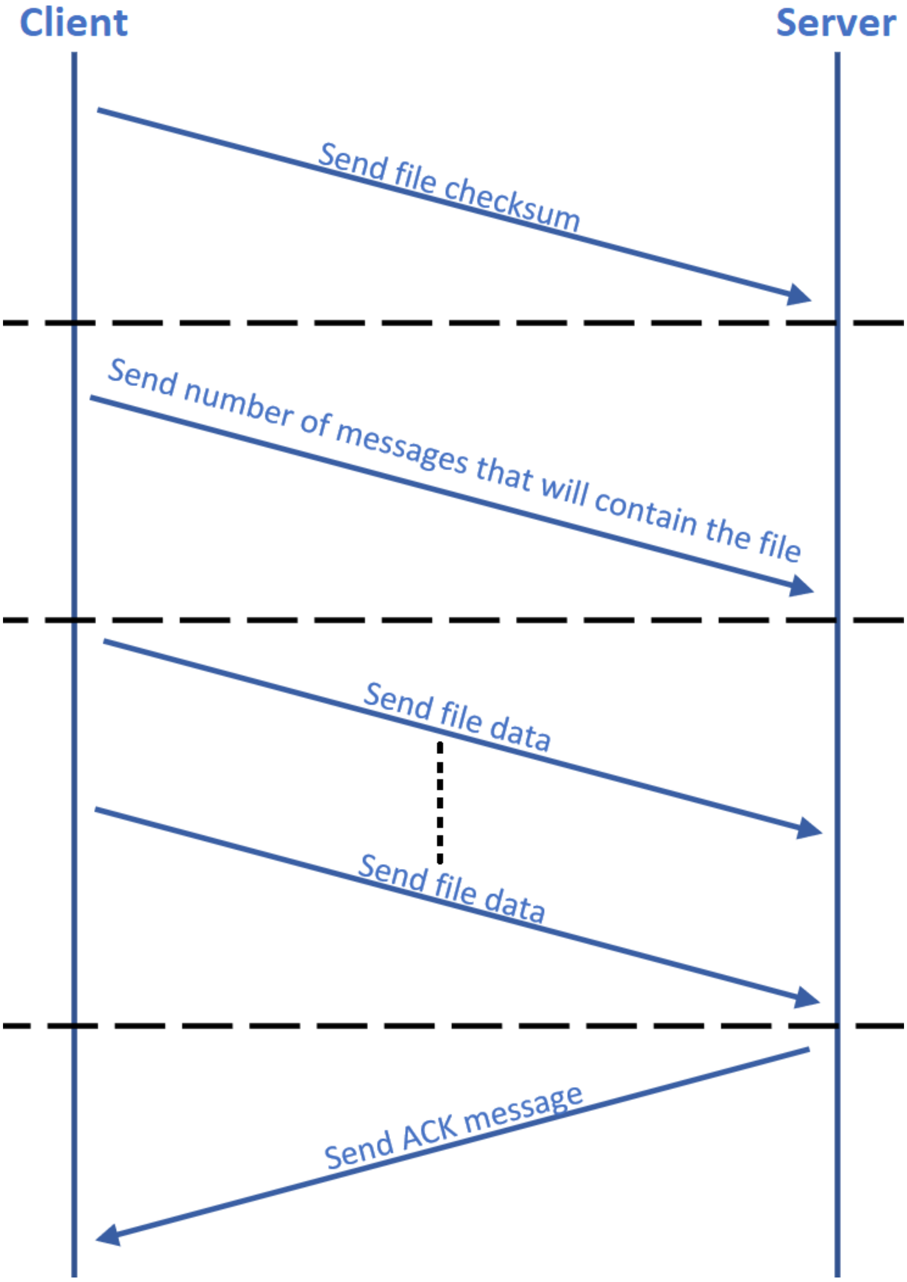
15.10.2 System Design

The file compression application client runs on the host and the server runs on the DPU.



15.10.3 Application Architecture

The file compression application runs on top of the DOCA Comm Channel API to send and receive the file from the host and to the DPU.



1. Connection is established on both sides by DOCA Comm Channel API.
2. Client compresses the data:
    - When compress engine is available – submits compress job with DOCA Compress API and sends the result to the server
    - When compress engine is unavailable – compresses the data in software
3. Client sends the number of messages needed to send the compressed content of the file.
4. Client sends data segments in size of up to 4080 bytes.
5. Server saves the received data in a buffer and submits a decompress job.
6. Server sends an ACK message to the client when all parts of the file are received successfully.
7. Server compares the received checksum to the calculated checksum.
8. Server writes the decompressed data to an output file.

15.10.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Compress
- DOCA Comch

Refer to their respective programming guide for more information.

15.10.5 Compiling the Application

Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

For more information about the applications as well as development and compilation tips, refer to the DOCA Applications page.

The sources of the application can be found under the application's directory: `/opt/mellanox/doca/applications/file_compression/` .

15.10.5.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

`doca_file_compression` is created under `/tmp/build/file_compression/` .

15.10.5.2 Compiling File Compression Application Only

To directly build only the file compression application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_file_compression=true
ninja -C /tmp/build
```

`doca_file_compression` is created under `/tmp/build/file_compression/` .

Alternatively, the user may set the desired flags in the `meson_options.txt` file instead of providing them in the compilation command line:

1. Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt`:
    - Set `enable_all_applications` to `false`
    - Set `enable_file_compression` to `true`
2. Run the following compilation commands:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> `doca_file_compression` is created under `/tmp/build/file_compression/`.

15.10.5.3 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application.

15.10.6 Running the Application

15.10.6.1 Application Execution

The file compression application is provided in source form. Therefore, a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_file_compression [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICA
L, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags from an input json file

Program Flags:
  -p, --pci-addr                    DOCA Comm Channel device PCI address
  -r, --rep-pci                     DOCA Comm Channel device representor PCI address
  -f, --file                        File to send by the client / File to write by the server
  -t, --timeout                     Application timeout for receiving file content messages, default is 5
 sec
```

> This usage printout can be printed to the command line using the `-h` (or `--help`) options:
>
> ```
> ./doca_file_compression -h
> ```

> For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on BlueField:

```
./doca_file_compression -p 03:00.0 -r 3b:00.0 -f received.txt
```

> ⚠ Both the DOCA Comm Channel device PCIe address (`03:00.0`) and the DOCA Comm Channel device representor PCIe address (`3b:00.0`) should match the addresses of the desired PCIe devices.

3. CLI example for running the application on the host:

   ```
   ./doca_file_compression -p 3b:00.0 -f send.txt
   ```

   > ⚠ The DOCA Comm Channel device PCIe address (`3b:00.0`) should match the address of the desired PCIe device.

4. The application also supports a JSON-based deployment mode, in which all command-line arguments are provided through a JSON file:

   ```
   ./doca_file_compression --json [json_file]
   ```

   For example:

   ```
   ./doca_file_compression --json ./file_compression_params.json
   ```

   > ⚠ Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the PCIe addresses necessary for the deployment.

15.10.6.2 Command Line Flags

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
General flags	`h`	`help`	Prints a help synopsis	N/A
	`v`	`version`	Prints program version information	N/A
	`l`	`log-level`	Set the log level for the application: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70 (requires compilation with `TRACE` log level support)	`"log-level": 60`

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
	N/A	`sdk-log-level`	Sets the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70	`"sdk-log-level": 40`
	`j`	`json`	Parse all command flags from an input JSON file	N/A
Program flags	`f`	`file`	For client - path to the file to be sent For server - path to write the file into ⚠ This is a mandatory flag.	`"file": "/tmp/data.txt"`
	`p`	`pci-addr`	Comm Channel DOCA device PCIe address ⚠ This is a mandatory flag.	`"pci-addr": 03:00.1`
	`r`	`rep-pci`	Comm Channel DOCA device representor PCIe address ⚠ This flag is mandatory only on the DPU.	`"rep-pci": b1:00.1`

ⓘ Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

15.10.6.3 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

15.10.7 Application Code Flow

1. Parse application argument.
    a. Initialize arg parser resources and register DOCA general parameters.

```
    doca_argp_init();
    ```

    b. Register file compression application parameters.

    ```
    register_file_compression_params();
    ```

    c. Parse the arguments.

    ```
    doca_argp_start();
    ```

        i. Parse app parameters.

2. Set endpoint attributes.

    ```
    set_endpoint_properties();
    ```

    a. Set maximum message size of 4080 bytes.
    b. Set maximum number of messages allowed.

3. Create comm channel endpoint.

    ```
    doca_comm_channel_ep_create();
    ```

    a. Create endpoint for client/server.

4. Run client/server main logic.

    ```
    file_compression_client/server();
    ```

5. Clean up the file compression application.

    ```
    file_compression_cleanup();
    ```

    a. Free all application resources.

6. Arg parser destroy.

    ```
    doca_argp_destroy()
    ```

15.10.8 References

- `/opt/mellanox/doca/applications/file_compression/`
- `/opt/mellanox/doca/applications/file_compression/file_compression_params.json`

15.11 NVIDIA DOCA File Integrity Application Guide

This guide provides a file integrity implementation on top of NVIDIA® BlueField® DPU.

15.11.1 Introduction

The file integrity application exhibits how to use the DOCA Comch and DOCA SHA libraries to send and receive a file securely.

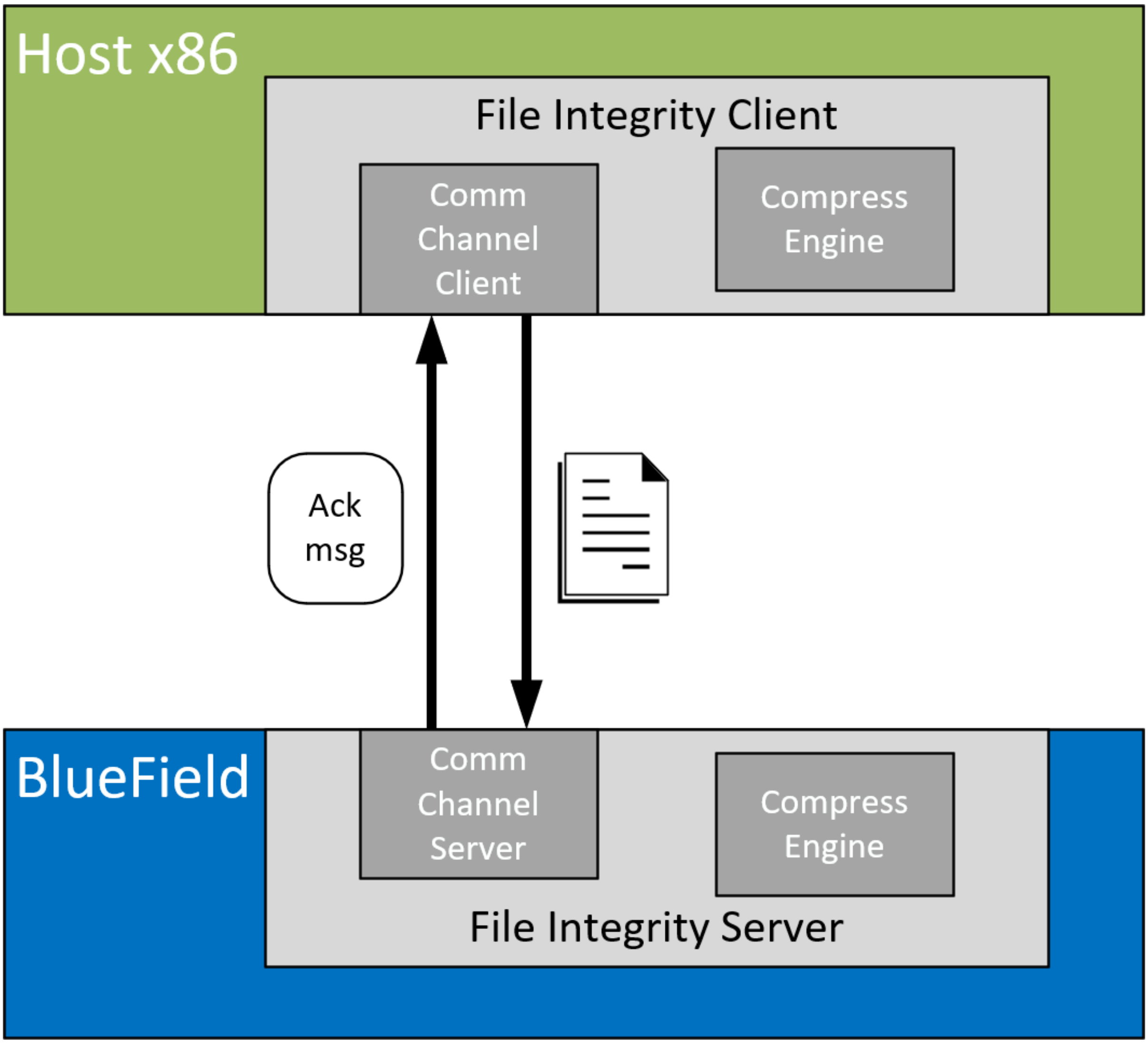
The application's logic includes both a client and a server:

- Client side – the application opens a file, calculates the SHA (secure hash algorithm) digest on it, and sends the digest of the source file alongside the file itself to the server
- Server side – the application calculates the SHA on the received file and compares the received digest to the calculated one to check if the file has been compromised

> ⚠ SHA hardware acceleration is only available on the BlueField-2 DPU. This application is not supported on BlueField-3.

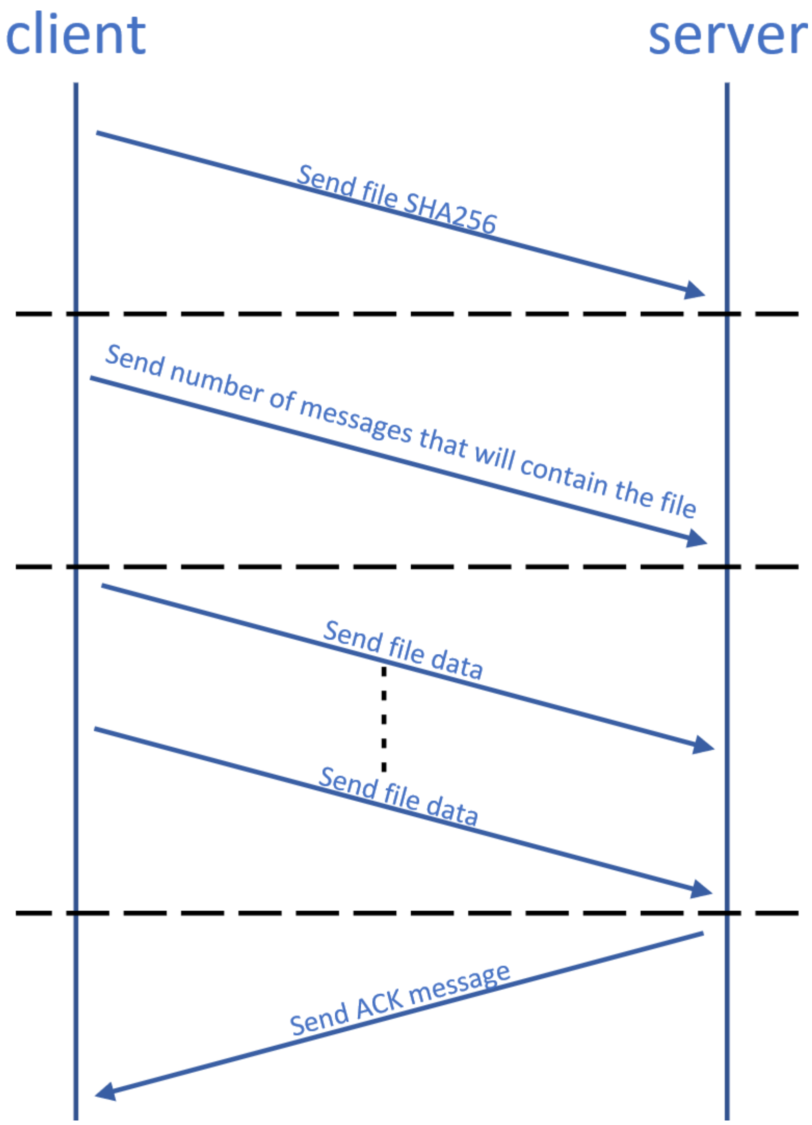
15.11.2 System Design

The file integrity application runs in client mode (host) and server mode (DPU).



15.11.3 Application Architecture

The file integrity application runs on top of the DOCA Comm Channel API to send and receive files from the host and DPU.



1. Connection is established on both sides by the Comm Channel API.
2. Client submits SHA job with the DOCA SHA library and sends the result to the server.
3. Client sends the number of messages required to send the content of the file.
4. Client sends data segments in size of up to 4032 bytes.
5. Server submits a partial SHA job on each received segment.
6. Server sends an ACK message to the client when all parts of the file are received successfully.
7. Server compares the received SHA to the calculated SHA.

15.11.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA SHA
- DOCA Comch

Refer to their respective programming guide for more information.

15.11.5 Compiling the Application

Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

For more information about the applications as well as development and compilation tips, refer to the DOCA Applications page.

The sources of the application can be found under the application's directory: `/opt/mellanox/doca/applications/file_integrity/` directory.

15.11.5.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

`doca_file_integrity` is created under `/tmp/build/file_integrity/`.

15.11.5.2 Compiling Only the Current Application

To directly build only the file integrity application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_file_integrity=true
ninja -C /tmp/build
```

`doca_file_integrity` is created under `/tmp/build/file_integrity/`.

Alternatively, one can set the desired flags in the `meson_options.txt` file instead of providing them in the compilation command line:

1. Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt`:
   - Set `enable_all_applications` to `false`
   - Set `enable_file_integrity` to `true`
2. Run the following compilation commands:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

`doca_file_integrity` is created under `/tmp/build/file_integrity/`.

15.11.5.3 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application.

15.11.6 Running the Application

15.11.6.1 Application Execution

The file integrity application is provided in source form. Therefore, a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_file_integrity [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                            Print a help synopsis
  -v, --version                         Print program version information
  -l, --log-level                       Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                       Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICA
L, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                     Parse all command flags from an input json file

Program Flags:
  -p, --pci-addr                        DOCA Comm Channel device PCI address
  -r, --rep-pci                         DOCA Comm Channel device representor PCI address
  -f, --file                            File to send by the client / File to write by the server
  -t, --timeout                         Application timeout for receiving file content messages, default is 5
 sec
```

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
./doca_file_integrity -h
```

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on BlueField:

```
./doca_file_integrity -p 03:00.0 -r 3b:00.0 -f received.txt
```

> Both the DOCA Comm Channel device PCIe address (`03:00.0`) and the DOCA Comm Channel device representor PCIe address (`3b:00.0`) should match the addresses of the desired PCIe devices.

3. CLI example for running the application on the host:

```
./doca_file_integrity -p 3b:00.0 -f send.txt
```

> The DOCA Comm Channel device PCIe address (`3b:00.0`) should match the address of the desired PCIe device.

4. The application also supports a JSON-based deployment mode, in which all command-line arguments are provided through a JSON file:

```
./doca_file_integrity --json [json_file]
```

For example:

```
./doca_file_integrity --json ./file_integrity_params.json
```

> Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the PCIe addresses necessary for the deployment.

15.11.6.2 Command Line Flags

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
General flags	`h`	`help`	Prints a help synopsis	N/A
	`v`	`version`	Prints program version information	N/A
	`l`	`log-level`	Set the log level for the application: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70 (requires compilation with `TRACE` log level support)	`"log-level": 60`

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
	N/A	`sdk-log-level`	Sets the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70	`"sdk-log-level": 40`
	`j`	`json`	Parse all command flags from an input JSON file	N/A
Program flags	`f`	`file`	For client - path to the file to be sent For server - path to write the file into ⚠ This is a mandatory flag.	`"file": "/tmp/data.txt"`
	`p`	`pci-addr`	Comm Channel DOCA device PCIe address ⚠ This is a mandatory flag.	`"pci-addr": 03:00.1`
	`r`	`rep-pci`	Comm Channel DOCA device representor PCIe address ⚠ This flag is mandatory only on the DPU.	`"rep-pci": b1:00.1`

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

15.11.6.3 Troubleshooting

Please refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

15.11.7 Application Code Flow

1. Parse application argument.
    a. Initialize the arg parser resources and register DOCA general parameters.

```
    doca_arg_init();
    ```

   b. Register file integrity application parameters.

    ```
    register_file_integrity_params();
    ```

   c. Parse application parameters.

    ```
    doca_argp_start();
    ```

2. Set endpoint attributes.

   ```
   set_endpoint_properties();
   ```

   a. Set maximum message size of 4032 bytes.
   b. Set number of maximum messages allowed per connection.
3. Create Comm Channel endpoint.

   ```
   doca_comm_channel_ep_create();
   ```

   a. Create endpoint for client/server.
4. Create SHA context.

   ```
   doca_sha_create();
   ```

   a. Create SHA context for submitting SHA jobs for client/server.
5. Run client/server main logic.

   ```
   file_integrity_client/server();
   ```

6. Clean up the File Integrity app.

   ```
   file_integrity_cleanup();
   ```

   a. Free all application resources.

15.11.8 References

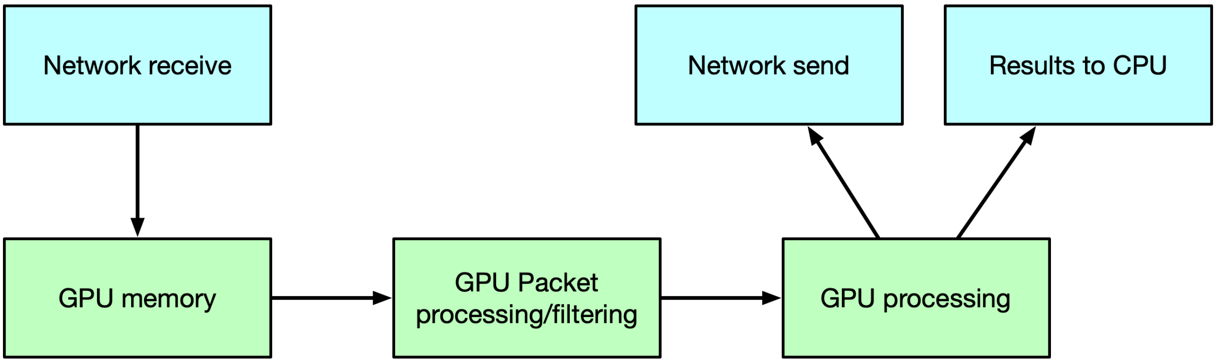
- `/opt/mellanox/doca/applications/file_integrity/`
- `/opt/mellanox/doca/applications/file_integrity/file_integrity_params.json`

15.12 NVIDIA DOCA GPU Packet Processing Application Guide

This guide provides a description of the GPU packet processing application to demonstrate the use of DOCA GPUNetIO, DOCA Ethernet, and DOCA Flow libraries to implement a GPU traffic analyzer.

15.12.1 Introduction

Real-time GPU processing of network packets is a useful technique to several different application domains, including signal processing, network security, information gathering, and input reconstruction. The goal of these applications is to realize an inline packet processing pipeline to receive packets in GPU memory (without staging copies through CPU memory), process them in parallel with one or more CUDA kernels, and then run inference, evaluate, or send the result of the calculation over the network.

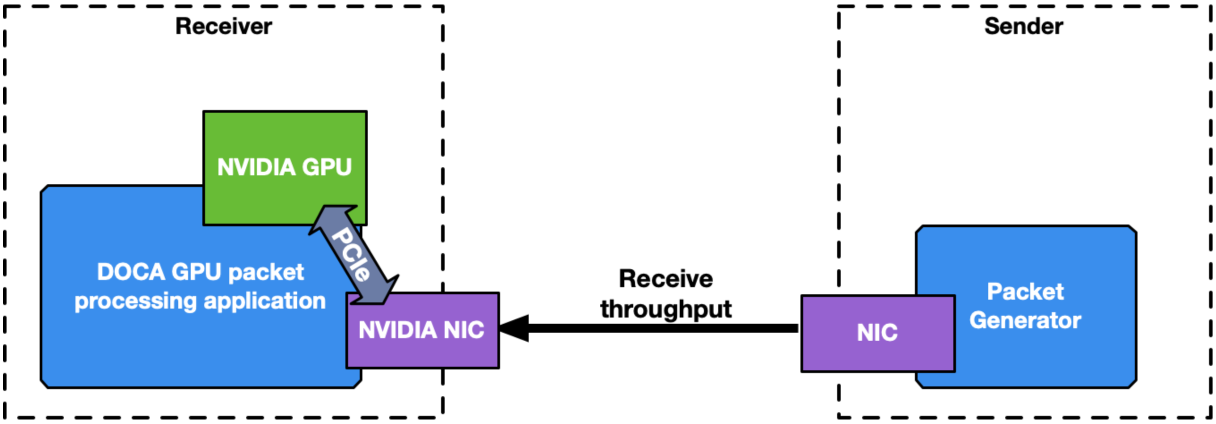


The type of data processing heavily depends on the use case. The goal of this application is to provide a basic layout to reuse in the most common use cases of being able to receive, differentiate and manage the following types of network traffic in multiple queues: UDP, TCP and ICMP.

This application is an enhancement of the use cases presented in this NVIDIA blog post about DOCA GPUNetIO.

15.12.2 System Design

This is a receive-and-process DOCA application, so a packet generator sending packets is required to test the application.



To launch the application, the PCIe address of the GPU and NIC are required.

15.12.3 Application Architecture

The application manages different types of traffic differently, dedicating up to 4 receive queues to each one using DOCA Flow with RSS mode to assign each packet to the right queue. The more queues the application uses, the higher is the degree of parallelism in how receive data is processed and how long it takes.

> ✓ It is highly recommended to use more than one receive queue for 100Gb/s or higher network traffic throughput.

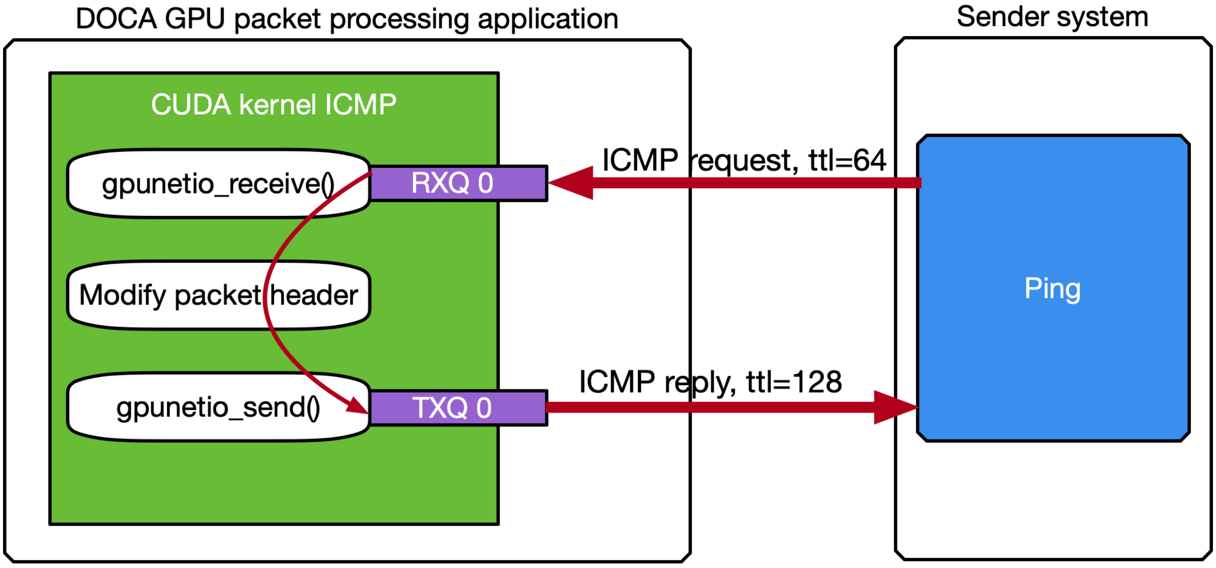
15.12.3.1 ICMP Network Traffic

If the network interface used for the application has an IP address, it is possible to ping that interface. ICMP packets are received by a dedicated CUDA kernel (file `gpu_kernels/receive_icmp.cu`) which:

1. Receives packets using the DOCA GPUNetIO CUDA warp-level function `doca_gpu_dev_eth_rxq_receive_warp`.
2. Checks if the packet is an ICMP echo request.
3. Forwards the same packet, modifying some header info (e.g., swapping MAC and IP addresses, changing ICMP packet type).
4. Pushes the modified packet into the send queue using the DOCA GPUNetIO thread-level function `doca_gpu_dev_eth_txq_send_enqueue_strong`.
5. Sends the packet using the DOCA GPUNetIO thread-level functions `doca_gpu_dev_eth_txq_commit_strong` and `doca_gpu_dev_eth_txq_push`.

> ⓘ This is not a compute intensive use case, so a single CUDA warp with only one receive queue and one send queue is enough to keep up with a decent latency.

By default, the OS CPU ping TTL is set to 64. Therefore, to be sure the GPU is actually replying to ICMP ping requests, TTL is set to 128 in this application.



The following are motivations for this use case:

- Providing an easy tool to check connectivity between packet the generator machine and the DOCA application machine
- Having a sense of network latency between the two machines using a well-known tool like ping
- Showing an easy way to receive and forward modified packets
- Providing a warp-level implementation of a CUDA kernel receiving and forwarding traffic

Assuming the IP address of the network interface to ping is `192.168.1.1`, this is the expected output:

```
$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=0.324 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=64 time=0.332 ms
64 bytes from 192.168.1.1: icmp_seq=3 ttl=64 time=0.299 ms
64 bytes from 192.168.1.1: icmp_seq=4 ttl=64 time=0.309 ms
64 bytes from 192.168.1.1: icmp_seq=5 ttl=64 time=0.323 ms
64 bytes from 192.168.1.1: icmp_seq=6 ttl=64 time=0.300 ms
64 bytes from 192.168.1.1: icmp_seq=7 ttl=64 time=0.274 ms
64 bytes from 192.168.1.1: icmp_seq=8 ttl=64 time=0.314 ms
64 bytes from 192.168.1.1: icmp_seq=9 ttl=64 time=0.327 ms
64 bytes from 192.168.1.1: icmp_seq=10 ttl=64 time=0.384 ms
# At this point, the DOCA application has been started on the 192.168.1.1 interface
# TTL becomes 128 as it's the GPU replying to ICMP requests now instead of the OS
64 bytes from 192.168.1.1: icmp_seq=11 ttl=128 time=0.346 ms
64 bytes from 192.168.1.1: icmp_seq=12 ttl=128 time=0.274 ms
64 bytes from 192.168.1.1: icmp_seq=13 ttl=128 time=0.294 ms
64 bytes from 192.168.1.1: icmp_seq=14 ttl=128 time=0.240 ms
64 bytes from 192.168.1.1: icmp_seq=15 ttl=128 time=0.273 ms
64 bytes from 192.168.1.1: icmp_seq=16 ttl=128 time=0.238 ms
64 bytes from 192.168.1.1: icmp_seq=17 ttl=128 time=0.252 ms
64 bytes from 192.168.1.1: icmp_seq=18 ttl=128 time=0.232 ms
64 bytes from 192.168.1.1: icmp_seq=19 ttl=128 time=0.278 ms
......
```

A DOCA Progress Engine is attached to the DOCA Ethernet Txq context used to forward ICMP packets. Those packets are sent from the GPU with the `DOCA_GPU_SEND_FLAG_NOTIFY` flag, which result in creating a notification after every packet is sent by the NIC.

All the notifications are then analyzed by the CPU through the `doca_pe_progress` function. The final effect is the output of the application which returns the distance, in seconds, between two pings. The following is an example with a ping every 0.5 seconds:

```
$ ping -i 0.5 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=128 time=0.202 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=128 time=0.179 ms
64 bytes from 192.168.1.1: icmp_seq=3 ttl=128 time=0.199 ms
64 bytes from 192.168.1.1: icmp_seq=4 ttl=128 time=0.180 ms
64 bytes from 192.168.1.1: icmp_seq=5 ttl=128 time=0.200 ms
64 bytes from 192.168.1.1: icmp_seq=6 ttl=128 time=0.189 ms
......
```

On the DOCA side, the application should print a log for all the ICMP packets received and retransmitted:

```
Seconds 5
[UDP] QUEUE: 0 DNS: 0 OTHER: 0 TOTAL: 0
[TCP] QUEUE: 0 HTTP: 0 HTTP HEAD: 0 HTTP GET: 0 HTTP POST: 0 TCP [SYN: 0 FIN: 0 ACK: 0] OTHER: 0 TOTAL: 0
[13:54:19:202061][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 3 sent at 1702302859201997120 time from last ICMP is 0.512025 sec
[13:54:19:713960][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 4 sent at 1702302859713896620 time from last ICMP is 0.511899 sec
[13:54:20:225891][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 5 sent at 1702302860225868072 time from last ICMP is 0.511971 sec
[13:54:20:737823][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 6 sent at 1702302860737781760 time from last ICMP is 0.511914 sec
[13:54:21:249763][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 7 sent at 1702302861249723044 time from last ICMP is 0.511941 sec
[13:54:21:761614][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 8 sent at 1702302861761588848 time from last ICMP is 0.511866 sec
[13:54:22:273689][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 9 sent at 1702302862273643536 time from last ICMP is 0.512055 sec
[13:54:22:785543][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 10 sent at 1702302862785527576 time from last ICMP is 0.511884 sec
```

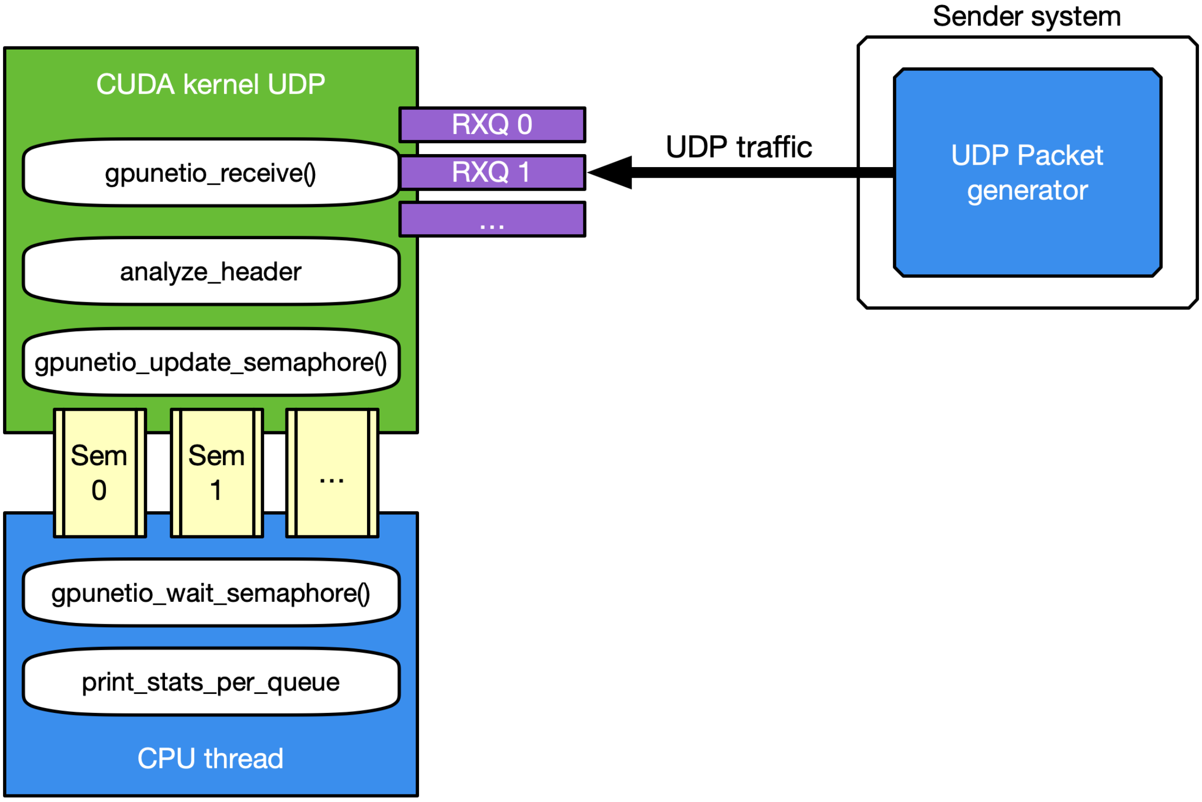
```
[13:54:23:297545][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 11 sent at 1702302863297501448 time from last ICMP is 0.511974 sec
[13:54:23:809406][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 12 sent at 1702302863809350664 time from last ICMP is 0.511849 sec

Seconds 10
[UDP] QUEUE: 0 DNS: 0 OTHER: 0 TOTAL: 0
[TCP] QUEUE: 0 HTTP: 0 HTTP HEAD: 0 HTTP GET: 0 HTTP POST: 0 TCP [SYN: 0 FIN: 0 ACK: 0] OTHER: 0 TOTAL: 0
[13:54:24:321405][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 13 sent at 1702302864321391148 time from last ICMP is 0.512040 sec
[13:54:24:833338][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 14 sent at 1702302864833270356 time from last ICMP is 0.511879 sec
[13:54:25:345302][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 15 sent at 1702302865345282728 time from last ICMP is 0.512012 sec
[13:54:25:857199][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 16 sent at 1702302865857133664 time from last ICMP is 0.511851 sec
[13:54:26:369131][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 17 sent at 1702302866369128728 time from last ICMP is 0.511995 sec......
```

15.12.3.2 UDP Network Traffic

This is the most generic use case of receive-and-analyze packet headers. Designed to keep up with 100Gb/s of incoming network traffic, the CUDA kernel responsible for the UDP traffic dedicates one CUDA block of 512 CUDA threads (file `gpu_kernels/receive_udp.cu`) to a different Ethernet UDP receive queue.

The data path loop is:

1. Receive packets using the DOCA GPUNetIO CUDA block-level function `doca_gpu_dev_eth_rxq_receive_block`.
2. Each CUDA thread works on a subset of received packets.
3. DOCA buffer containing the packet is retrieved.
4. Packet payload is analyzed to differentiate between DNS packets from other UDP generic packets.
5. Packet payload is wiped-out to ensure that old stale packets are not analyzed again.
6. Each CUDA block reports to the CPU thread statistics about types of received packets through a DOCA GPUNetIO semaphore.
7. CPU thread polls on semaphores to retrieve and print the statistics to the console.



The motivation for this use case is mostly to provide an application template to:

- Receive and analyze packet headers to differentiate across different UDP protocols
- Report statistics to the CPU through the DOCA GPUNetIO semaphore

Several well-known packet generators can be used to test this mode like T-Rex or DPDK testpmd.

15.12.3.3 TCP Network Traffic and HTTP Echo Server

By default, the TCP flow management is the same as UDP: Receive TCP packets and analyze their headers to report to the CPU statistics about the types of received packets. This is good for passive traffic analyzers or sniffers but sometimes a packet processing application requires receiving packets directly from TCP peers which implies the establishment of a TCP-reliable connection through the 3-way handshake method. Therefore, it is possible to enable TCP "server" mode through the `-s` command-line flag which enables an "HTTP echo server" mode where the CPU and GPU cooperate to establish a TCP connection and process TCP data packets.

Specifically, in this case there are two different sets of receive queues:

- CPU DPDK receive queues which receive TCP "control" packets (e.g. SYN, FIN or RST)
- DOCA GPUNetIO receive queues to receive TCP "data" packets

This distinction is possible thanks to DOCA Flow capabilities.

The application's flow requires CPU and GPU collaboration as described in the following subsections.

15.12.3.3.1 Step 1: TCP Connection Establishment

A CPU thread through DPDK queues receives a TCP SYN packet from a remote TCP peer. The CPU thread establishes a TCP reliable connection (replies with a TCP SYN-ACK packet) with the peer and uses DOCA Flow to create a new steering rule to redirect TCP data packets to one of the DOCA GPUNetIO receive queues. The new steering rule excludes control packets (e.g., SYN, FIN or RST).

15.12.3.3.2 Step 2: TCP Data Processing

The CUDA kernel responsible for TCP processing receives TCP data packets and performs TCP packet header analysis. If it receives an HTTP GET request, it stores the relevant packet's info in the next item of a DOCA GPUNetIO semaphore, setting it to `READY`.

15.12.3.3.3 Step 3: HTTP Echo Server

A second CUDA kernel responsible for HTTP processing polls the DOCA GPUNetIO semaphore. Once it detects the update of the next item to `READY`, it reads the HTTP GET packet info and crafts an HTTP response packet with an HTML page.

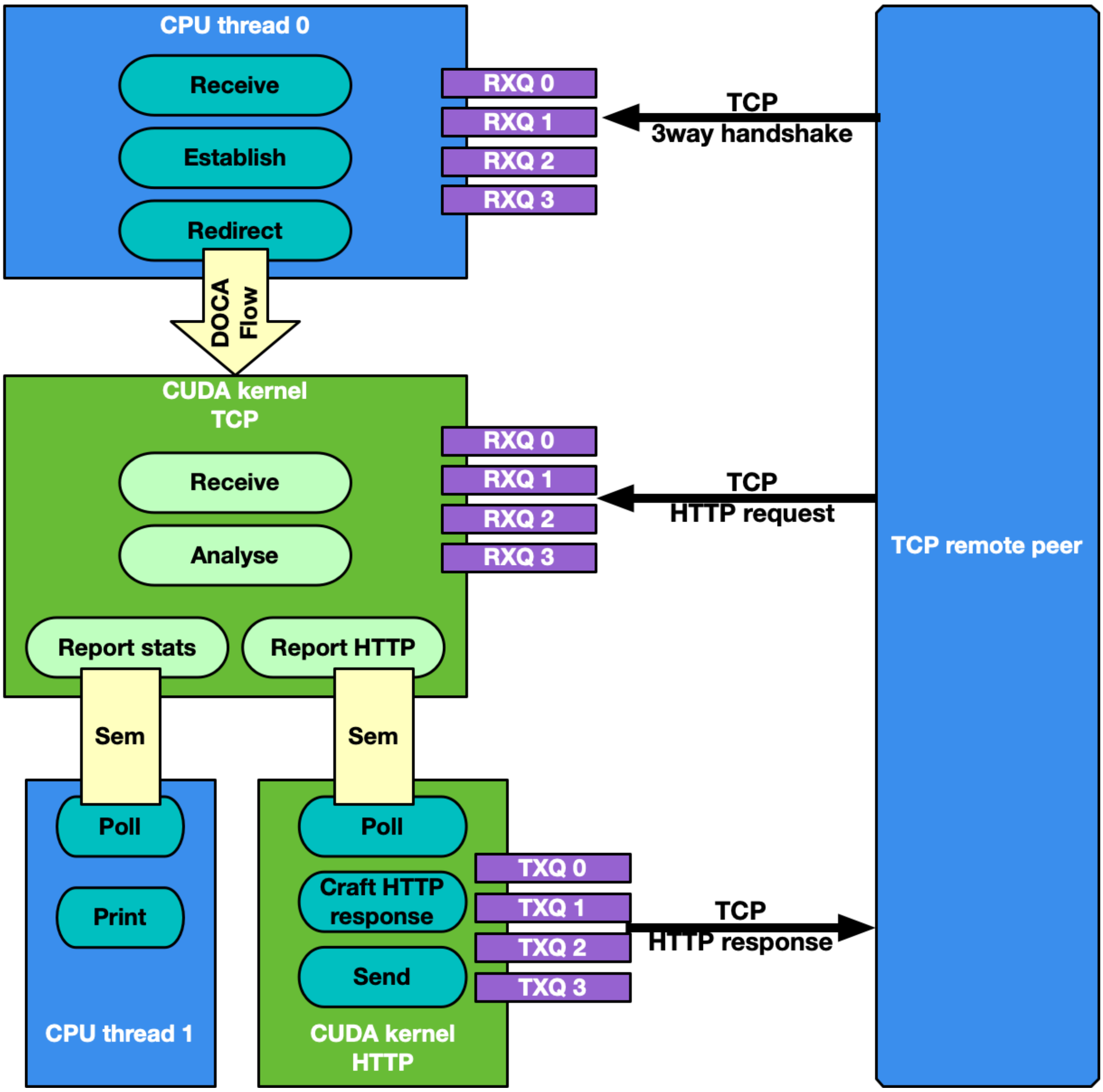
If the request is about `index.html` or `contacts.html`, the CUDA kernel replies with the appropriate HTML page using a `200 OK` code. For all other requests, the it returns a "Page not found" and `404 Error` code.

HTTP response packets are sent by this second HTTP CUDA kernel using DOCA GPUNetIO.

⚠ Care must be taken to maintain TCP sequence/ack numbers in the packet headers.

15.12.3.3.4 Step 4: TCP Connection Closure

If the CPU receives a TCP FIN packet through the DPDK queues, it closes the connection with the remote TCP peer and removes the DOCA Flow rule from the DOCA GPUNetIO queues so the CUDA kernel cannot receive anymore packets from that TCP peer.



Motivations for this use case:

- Receiving and analyzing packet headers to differentiate across different TCP protocols
- Processing TCP packets on GPU in passive mode (sniffing) and active mode (reliable connection)
- Having a DOCA-DPDK application able to establish a TCP reliable connection without using any OS socket and bypassing kernel routines
- Having CUDA-kernel-to-CUDA-kernel communication through a DOCA GPUNetIO semaphore
- Showing how to create and send a packet from scratch with DOCA GPUNetIO

Assuming the network interface used to run the application has the IP address `192.168.1.1` , it is possible to test this HTTP echo server mode using simple tools like `curl` or `wget` .

Example with `curl` :

```
$ curl http://192.168.1.1/index.html -ivvv
```

```
*   Trying 192.168.1.1:80...
* Connected to 192.168.1.1 (192.168.1.1) port 80 (#0)
> GET /index.html HTTP/1.1
> Host: 192.168.1.1
> User-Agent: curl/7.81.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
HTTP/1.1 200 OK
< Date: Sun, 30 Apr 2023 20:30:40 GMT
Date: Sun, 30 Apr 2023 20:30:40 GMT
< Content-Type: text/html; charset=UTF-8
Content-Type: text/html; charset=UTF-8
< Content-Length: 158
Content-Length: 158
< Last-Modified: Sun, 30 Apr 2023 22:38:34 GMT
Last-Modified: Sun, 30 Apr 2023 22:38:34 GMT
< Server: GPUNetIO
Server: GPUNetIO
< Accept-Ranges: bytes
Accept-Ranges: bytes
< Connection: keep-alive
Connection: keep-alive
< Keep-Alive: timeout=5
Keep-Alive: timeout=5

<
<html>
  <head>
    <title>GPUNetIO index page</title>
  </head>
  <body>
    <p>Hello World, the GPUNetIO server Index page!</p>
  </body>
</html>

* Connection #0 to host 192.168.1.1 left intact
```

15.12.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA GPUNetIO
- DOCA Ethernet
- DOCA Flow

Refer to their respective programming guide for more information on system configuration and requirements.

Refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install DOCA package software.

15.12.5 Dependencies

Before running the application you need to be sure you have the following:

- `gdrdrv` kernel module - active and running on the system
- `nvidia-peermem` kernel module - active and running on the system
- Network card interface you want to use is up

15.12.6 Compiling the Application

Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

> For more information about the applications as well as development and compilation tips, refer to the DOCA Applications page.

The sources of the application can be found under the application's directory: `/opt/mellanox/doca/applications/gpu_packet_processing/`.

15.12.6.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> `doca_gpu_packet_processing` is created under `/tmp/build/gpu_packet_processing/`.

15.12.6.2 Compiling Only the Current Application

To directly build only the GPU packet processing application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_gpu_packet_processing=true
ninja -C /tmp/build
```

> `doca_gpu_packet_processing` is created under `/tmp/build/gpu_packet_processing/`.

Alternatively, users can set the desired flags in the `meson_options.txt` file instead of providing them in the compilation command line:

1. Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt`:
    - Set `enable_all_applications` to `false`
    - Set `enable_gpu_packet_processing` to `true`
2. Run the following compilation commands:

    ```
    cd /opt/mellanox/doca/applications/
    meson /tmp/build
    ninja -C /tmp/build
    ```

    > `doca_gpu_packet_processing` is created under `/tmp/build/gpu_packet_processing/`.

15.12.6.3 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application.

15.12.7 Running the Application

The GPU packet processing application is provided in source form. Therefore, a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_gpu_packet_processing [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICA
L, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags from an input json file

Program Flags:
  -g, --gpu <GPU PCIe address>      GPU PCIe address to be used by the app
  -n, --nic <NIC PCIe address>      DOCA device PCIe address used by the app
  -q, --queue <GPU receive queues>  DOCA GPUNetIO receive queue per flow
  -s, --httpserver <Enable GPU HTTP server> Enable GPU HTTP server mode
```

   This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
./doca_gpu_packet_processing -h
```

   For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the host:
   a. Assuming a GPU PCIe address `ca:00.0` and NIC PCIe address `17:00.0` with 2 GPUNetIO receive queues:

```
./doca_gpu_packet_processing -n 17:00.0 -g ca:00.0 -q 2
```

   Refer to section "Running DOCA Application on Host" in the NVIDIA DOCA Virtual Functions User Guide.

15.12.7.1 Command Line Flags

Flag Type	Short Flag	Long Flag	Description
General flags	`h`	`help`	Prints a help synopsis
	`v`	`version`	Prints program version information

Flag Type	Short Flag	Long Flag	Description
	`l`	`log-level`	Set the log level for the application: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70 (requires compilation with `TRACE` log level support)
	N/A	`sdk-log-level`	Sets the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70
	`j`	`json`	Parse all command flags from an input JSON file
Program flags	`g`	`gpu`	GPU PCIe address in `<bus>:<device>.<function>` format. This can be obtained using the `nvidia-smi` or `lspci` commands.
	`n`	`nic`	Network card port PCIe address in `<bus>:<device>.<function>` format. This can be obtained using the `lspci` command.
	`q`	`queue`	Number of receive queues to use in the example. Default is 1, maximum allowed is 4.
	`s`	`httpserver`	Enable the TCP HTTP server mode. With this flag, TCP packets are not received by GPUNetIO as regular sniffer as it requires a TCP 3-way handshake to establish a reliable connection first.

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

15.12.7.2 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

15.12.8 Application Code Flow

The following explains the application's flow, highlighting main code blocks and functions:

1. Parse application argument.

   ```
   doca_argp_init();
   register_application_params();
   doca_argp_start();
   ```

2. Initialize network device as DOCA device, initialize DPDK, and get device DPDK port ID.

   ```
   init_doca_device();
   ```

   Calls `rte_eal_init()` with empty flags to initialize EAL resources.

3. Initialize a GPU device, creating a DOCA GPUNetIO handle for it.

   ```
   doca_gpu_create();
   ```

4. Initialize DOCA Flow, starting the DPDK port.

   ```
   init_doca_flow();
   ```

   Flags to initialize DOCA Flow are VNF, HW steering, and isolated mode (to prevent the default RSS flows from interfering with the GPUNetIO queues).

5. Create RX and TX queue related objects (i.e., Ethernet handlers, GPUNetIO handlers, flow rules, semaphores) to manage UDP, TCP and ICMP flows.

   ```
   create_udp_queues();
   create_tcp_queues();
   create_icmp_queues();
   /* Depending on TCP mode (HTTP server or not) properly connect different DOCA Flow pipes */
   create_root_pipe();
   ```

6. Allocate generic exit flag. All CUDA kernels periodically poll on this flag. If the CPU set it to 1, CUDA kernels exit from their main loop and return.

   ```
   doca_gpu_mem_alloc(gpu_dev, sizeof(uint32_t), alignment, DOCA_GPU_MEM_GPU_CPU, (void
    **)&gpu_exit_condition, (void **)&cpu_exit_condition);
   ```

7. Launch CUDA kernels, each on a different stream.

   ```
   kernel_receive_udp(rx_udp_stream, gpu_exit_condition, &udp_queues);
   kernel_receive_tcp(rx_tcp_stream, gpu_exit_condition, &tcp_queues, app_cfg.http_server);
   kernel_receive_icmp(rx_icmp_stream, gpu_exit_condition, &icmp_queues);
   if (app_cfg.http_server)
       kernel_http_server(tx_http_server, gpu_exit_condition, &tcp_queues, &http_queues);
   ```

8. Launch the CPU thread responsible to poll on DOCA GPUNetIO semaphores and print UDP and TCP stats on the console.

   ```
   rte_eal_remote_launch((void *)stats_core, NULL, current_lcore);
   ```

9. Launch CPU thread responsible for managing TCP 3-way handshake connections.

   ```
   if (app_cfg.http_server) {
       ...
       rte_eal_remote_launch(tcp_cpu_rss_func, &tcp_queues, current_lcore);
   ```

```
}
```

10. Wait for the user to send a signal to quit the application. When this happens, the signal handler function sets the `force_quit` flag to true which causes the main thread to move forward and set the exit condition to 1.

    ```
    while (DOCA_GPUNETIO_VOLATILE(force_quit) == false);
    DOCA_GPUNETIO_VOLATILE(*cpu_exit_condition) = 1;
    ```

11. Wait for CUDA kernels to exit and finalize all DOCA Flow and GPUNetIO resources.

    ```
    cudaStreamSynchronize(rx_udp_stream);
    cudaStreamSynchronize(rx_tcp_stream);
    cudaStreamSynchronize(rx_icmp_stream);
    if (app_cfg.http_server)
        cudaStreamSynchronize(tx_http_server);
    destroy_flow_queue();
    doca_gpu_destroy();
    ```

15.12.9 References

- `/opt/mellanox/doca/applications/gpu_packet_processing/`

15.13 NVIDIA DOCA IPsec Security Gateway Application Guide

This document provides an IPsec security gateway implementation on top of NVIDIA® BlueField® DPU.

> **Important note for NVIDIA® BlueField®-2 DPUs**
>
> If your target application utilizes 100Gb/s or higher bandwidth, where a substantial part of the bandwidth is allocated for IPsec traffic, please refer to the *NVIDIA BlueField-2 DPUs Product Release Notes* to learn about a potential bandwidth limitation. To access the relevant product release notes, please contact your NVIDIA sales representative.

15.13.1 Introduction

> DOCA IPsec Security Gateway is supported at alpha level.

DOCA IPsec Security Gateway leverages the DPU's hardware capability for secure network communication. The application demonstrates how to insert rules related to IPsec encryption and decryption based on the DOCA Flow library.

The application demonstrates how to insert rules to create an IPsec tunnel.

> An example for configuring the Internet Key Exchange (IKE) can be found under section "Keying Daemon Integration (StrongSwan)" but is not considered part of the application.

The application can be configured to receive IPsec rules in one of the following ways:

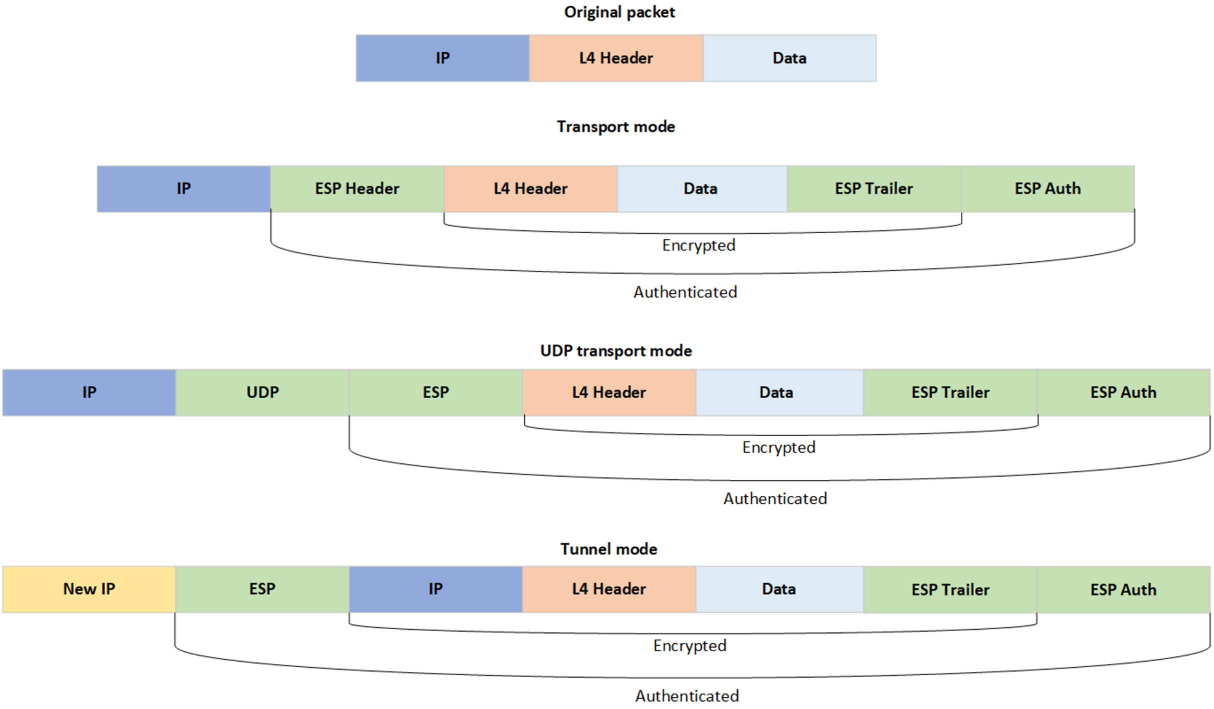
- Static configuration – (default) receives a fixed list of rules for IPsec encryption and decryption

> ⚠ When creating the security association (SA) object, the application gets the key, salt, and other SA attributes from the JSON input file.

- Dynamic configuration – receives IPsec encryption and decryption rules during runtime through a Unix domain socket (UDS) which is enabled when providing a socket path to the application

> ⚠ You may find an example of integrating a rules generator with the application under strongSwan project (DOCA plugin).

The application supports the following IPsec modes: Tunnel, transport, UDP transport.

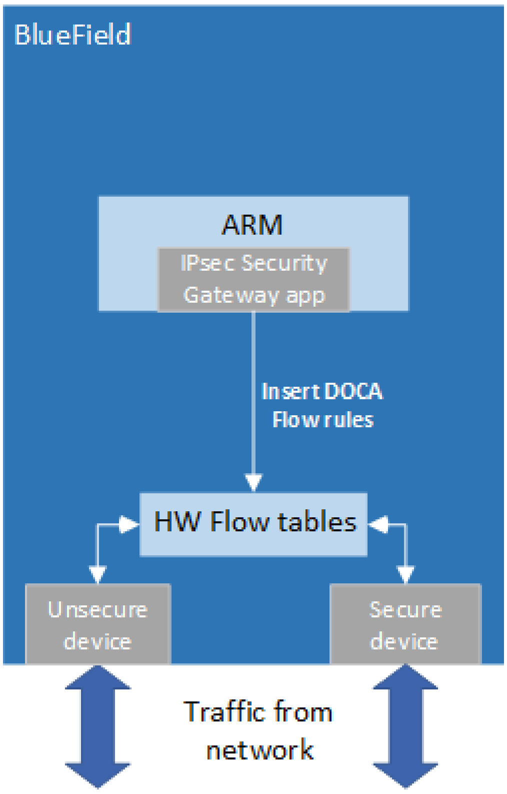


15.13.2 System Design

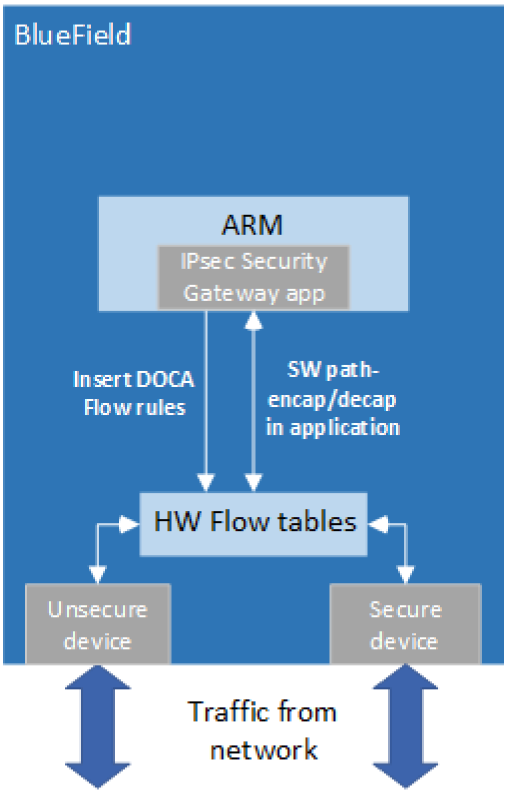
DOCA IPsec Security Gateway is designed to run with 2 ports, secured and unsecured:

- Secured port – BlueField receives IPsec encrypted packets and, after decryption, they are sent through the unsecured port
- Unsecured port – BlueField receives regular (plain text) packets and, after encryption, they are sent through the secured port

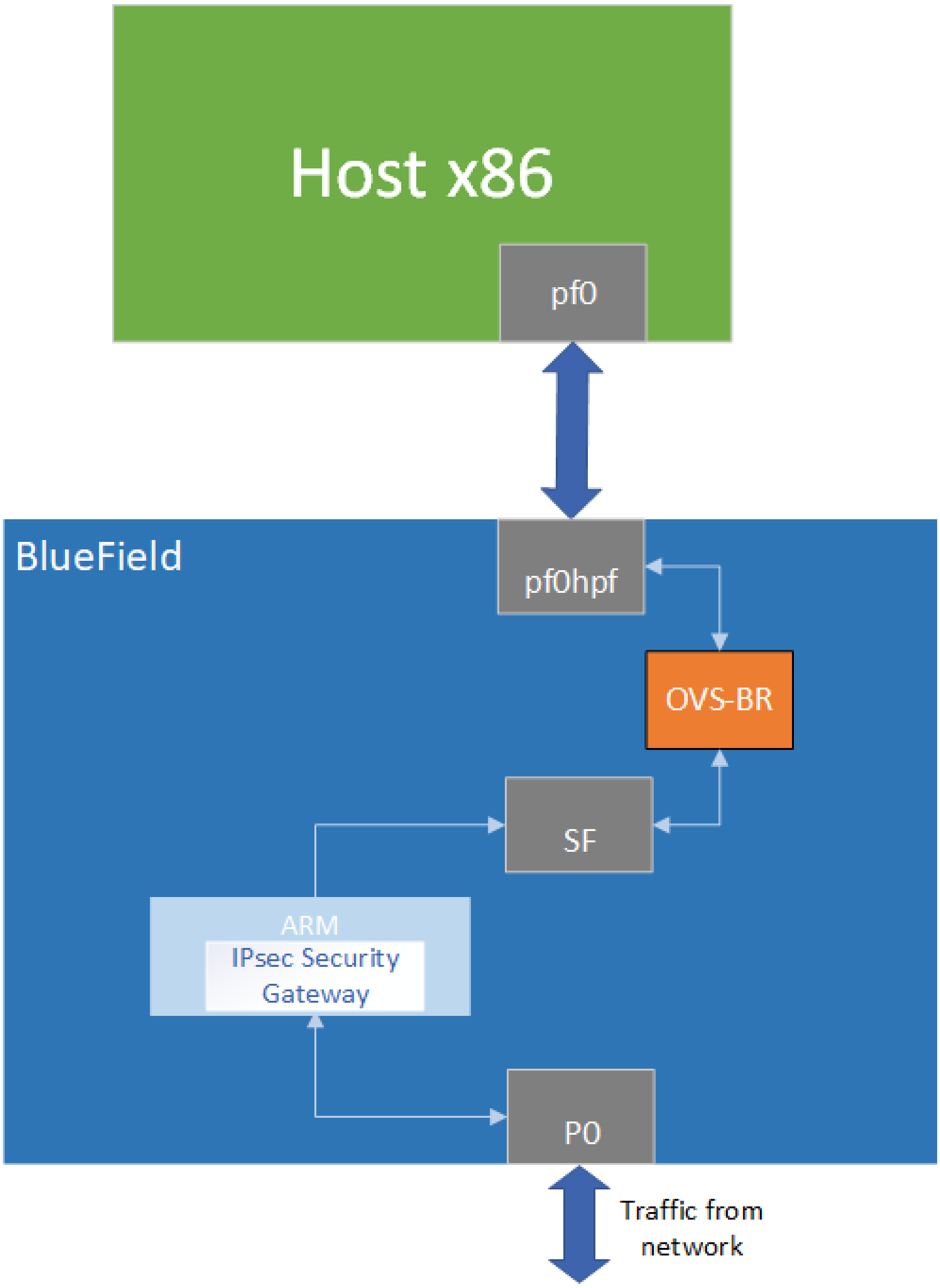
Example packet path for hardware (HW) offloading:



Example packet path for partial software processing (handling encap/decap in software):



Using the application with SF:



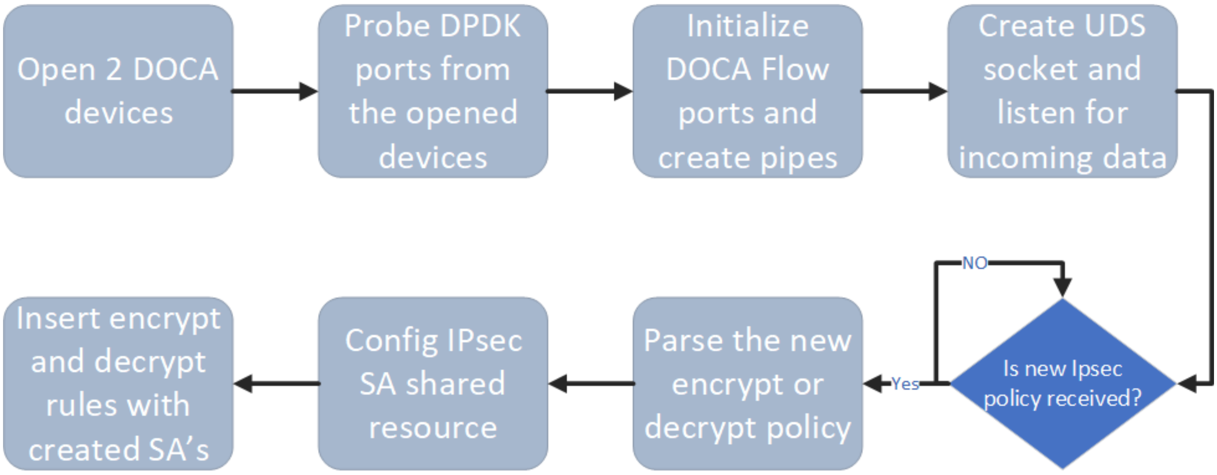
15.13.3 Application Architecture

15.13.3.1 Static Configuration

1. Open two DOCA devices, one for the secured port and another for the unsecured port.
2. With the open DOCA devices, the application probes DPDK ports and initializes DOCA Flow and DOCA Flow ports accordingly.
3. On the created ports, build DOCA Flow pipes.
4. In a loop according to the JSON rules:

a. Create IPSec SA shared resource for the new rule.
b. Insert encrypt or decrypt rule to DOCA Flow pipes.

15.13.3.2 Dynamic Configuration


Open 2 DOCA devices
Probe DPDK ports from the opened devices
Initialize DOCA Flow ports and create pipes
Create UDS socket and listen for incoming data
Is new Ipsec policy received?
NO
Yes
Parse the new encrypt or decrypt policy
Config IPsec SA shared resource
Insert encrypt and decrypt rules with created SA's

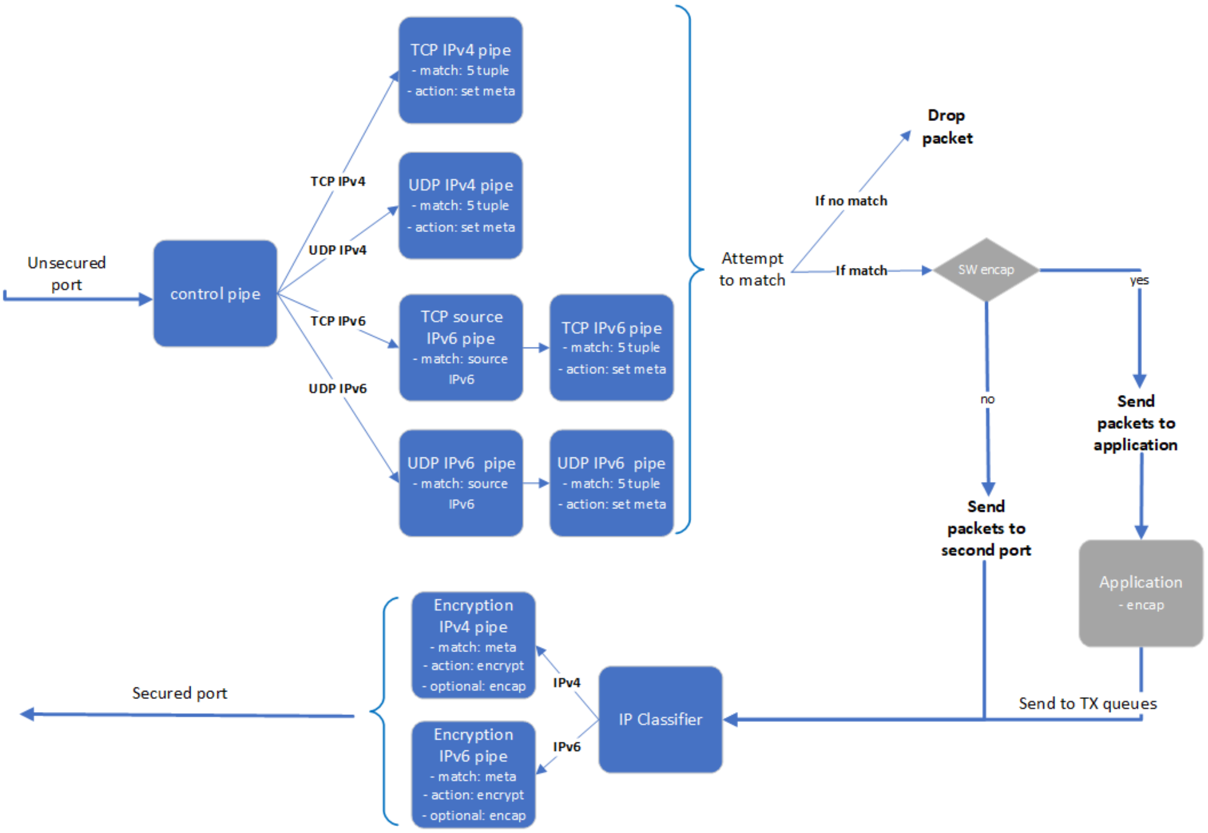
1. Open two DOCA devices, one for the secured port and another for the unsecured port.
2. With the open DOCA devices, the application probes DPDK ports and initializes DOCA Flow and DOCA Flow ports accordingly.
3. On the created ports, build DOCA Flow pipes.
4. Create UDS socket and listen for incoming data.
5. While waiting for new IPsec policies to be received in a loop, if a new IPsec policy is received:
   a. Parse the policy whether it is an encryption or decryption rule.
   b. Create IPSec SA shared resource for the new rule.
   c. Insert encrypt or decrypt rule to DOCA Flow pipes.

15.13.3.3 DOCA Flow Modes

The application can run in two modes, `vnf` and `switch`. For more information about the modes, please refer to "Pipe Mode" in the DOCA Flow.

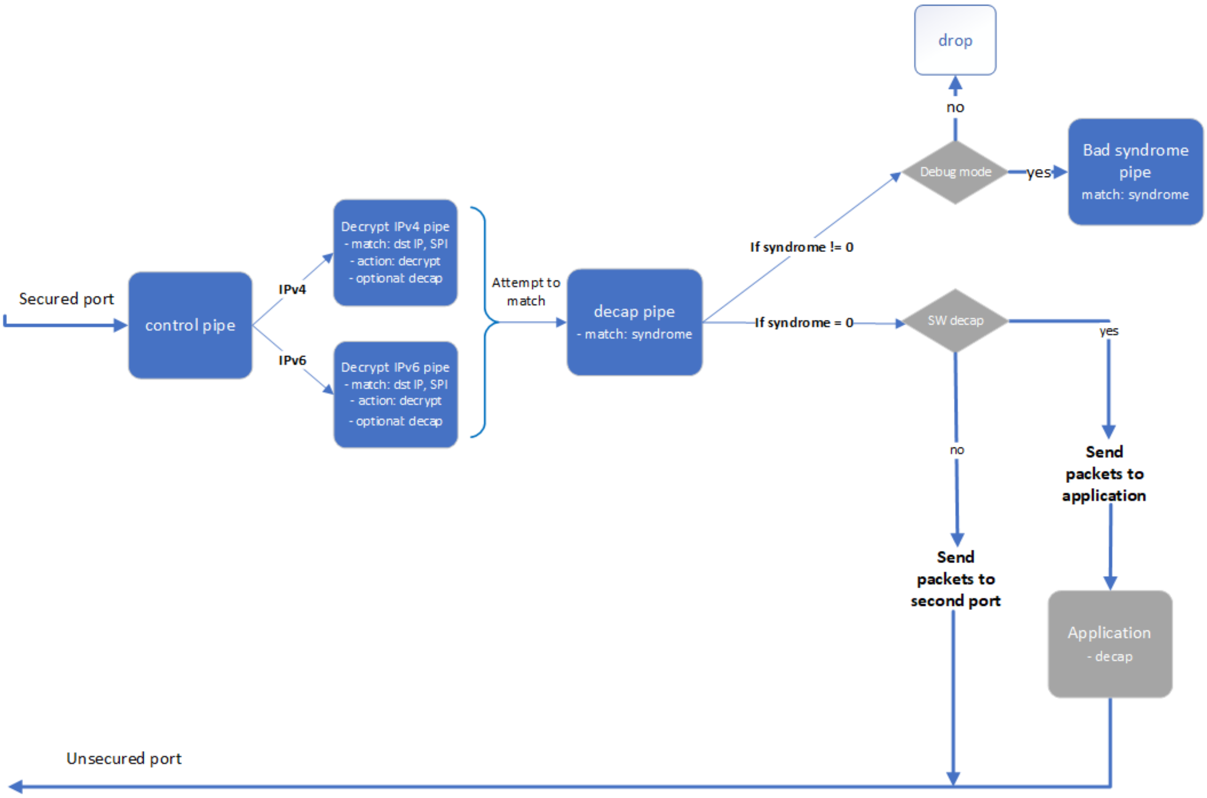
15.13.3.3.1 VNF Mode

15.13.3.3.1.1 Encryption



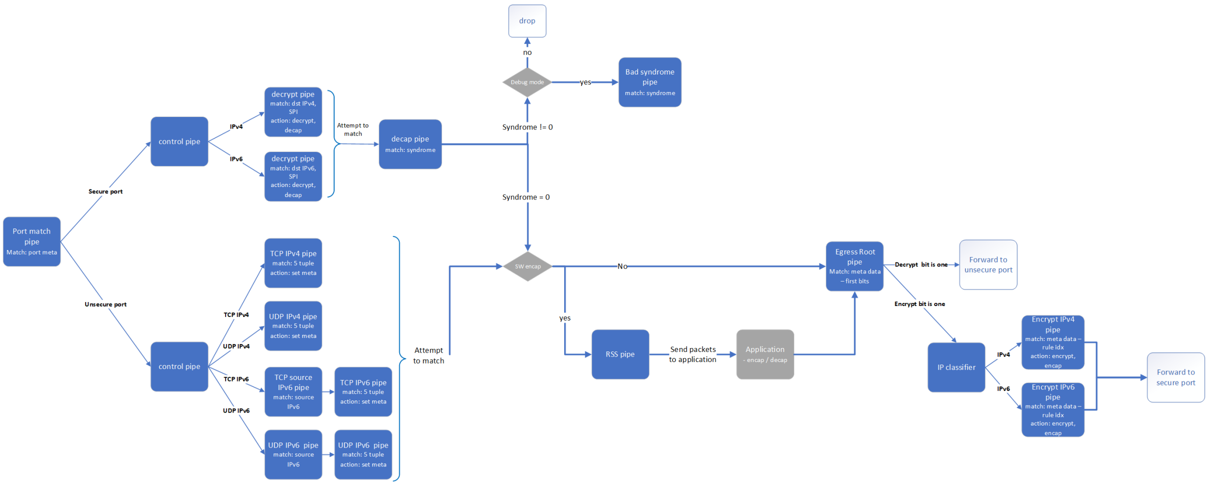
1. The application builds pipes for encryption. Control pipe as root with four entries that match L3 and L4 types and forward the traffic to the relevant pipes.
   a. IPv6 pipes – match the source IP address and forward the traffic to a pipe that matches 5-tuple excluding the source IP.
   b. In the 5-tuple match pipes set action of "set meta data", the metadata would be the rule's index in the JSON file.
   c. The matched packet is forwarded to the second port.
2. In the secured egress domain, the IP classifier pipe sends the packets to the correct encryption pipe (IPv4 or IPv6) which has a shared IPsec encrypt action. According to the metadata match, the packet is encrypted with the encap destination IP and SPI as defined in the user's rules.

15.13.3.3.1.2 Decryption



1. The application builds pipes for decryption. Control pipe as root with two entries that match L3 type and forward the traffic to the relevant decrypt pipe.
2. The decrypt pipe matches the destination IP and SPI according to the rule files and has a shared IPsec action for decryption.
3. After decryption, the matched packets are forwarded to the decap pipe and, if the syndrome is non-zero, the packets are dropped. Otherwise, the packets decap the ESP header and forward to the second port.
    a. In debug mode, if syndrome is non-zero, then it sends to bad syndrome pipe to match on the syndrome, count and drop/send to application.

15.13.3.3.2 Switch Mode



In switch mode, an ingress root pipe matches the source port to decide what the next pipe is:

- Based on the port, the packet passes through almost the same path as VNF mode and the metadata is set. Afterwards, the packet moves to egress root pipe.

In egress root pipe, the match is on encrypt and decrypt bits that were set in the packet meta:

- Decrypt bit is 1 – packet finishes the decrypt path and must be sent to the unsecure port
- Encrypt bit is 1 – packet almost finishes the encrypt path and must be sent to the encrypt pipe on the secure egress domain and to the secure port from there

15.13.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Flow

Refer to their respective programming guide for more information.

15.13.5 Compiling the Application

Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

For more information about the applications as well as development and compilation tips, refer to the DOCA Applications page.

The sources of the application can be found under the application's directory: `/opt/mellanox/doca/applications/ipsec_security_gw/`.

15.13.5.1 Prerequisites

The application relies on the `json-c` open source, hence requires the following installation:

- Ubuntu/Debian:

  ```
  $ sudo apt install libjson-c-dev
  ```

- CentOS/RHEL:

  ```
  $ sudo yum install json-c-devel
  ```

15.13.5.2 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> `doca_ipsec_security_gw` is created under `/tmp/build/ipsec_security_gw/`.

15.13.5.3 Compiling Only the Current Application

To directly build only the IPsec Security Gateway application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_ipsec_security_gw=true
ninja -C /tmp/build
```

> `doca_ipsec_security_gw` is created under `/tmp/build/ipsec_security_gw/`.

Alternatively, users can set the desired flags in the `meson_options.txt` file instead of providing them in the compilation command line:

1. Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt`:
   - Set `enable_all_applications` to `false`
   - Set `enable_ipsec_security_gw` to `true`
2. Run the following compilation commands:

   ```
   cd /opt/mellanox/doca/applications/
   meson /tmp/build
   ninja -C /tmp/build
   ```

> ⓘ `doca_ipsec_security_gw` is created under `/tmp/build/ipsec_security_gw/` .

15.13.5.4 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application.

15.13.6 Running the Application

15.13.6.1 Prerequisites

1. The IPsec security gateway application is based on DOCA Flow. Therefore, the user is required to allocate huge pages.

```
echo '2048' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

> ⚠ On some operating systems (RockyLinux, OpenEuler, CentOS 8.2) the default huge page size on the DPU (and Arm hosts) is larger than 2MB, and is often 512MB instead. Once can find out the sige of the huge pages using the following command:
>
> ```
> $ grep -i huge /proc/meminfo
>
> AnonHugePages:          0 kB
> ShmemHugePages:         0 kB
> FileHugePages:          0 kB
> HugePages_Total:        4
> HugePages_Free:         4
> HugePages_Rsvd:         0
> HugePages_Surp:         0
> Hugepagesize:      524288 kB
> Hugetlb:          6291456 kB
> ```
>
> Given that the guiding principal is to allocate 4GB of RAM, in such cases instead of allocating 2048 pages, one should allocate the matching amount (8 pages):
>
> ```
> echo '8' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-524288kB/nr_hugepages
> ```

2. VNF mode - the IPsec security gateway application requires disabling some of the hardware tables:

```
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode legacy
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode legacy

echo none > /sys/class/net/p0/compat/devlink/encap
echo none > /sys/class/net/p1/compat/devlink/encap

/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode switchdev
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode switchdev
```

To restore the old configuration:

```
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode legacy
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode legacy

echo basic > /sys/class/net/p0/compat/devlink/encap
echo basic > /sys/class/net/p1/compat/devlink/encap
```

```
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode switchdev
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode switchdev
```

3. Switch mode - the IPsec security gateway application requires configuring the ports to run in switch mode:

```
sudo mlxconfig -d /dev/mst/mt41686(mt41692)_pciconf0 s LAG_RESOURCE_ALLOCATION=1
# power cycle the host to apply this setting

/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode legacy
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode legacy

sudo devlink dev param set pci/0000:03:00.0 name esw_pet_insert value false cmode runtime
sudo devlink dev param set pci/0000:03:00.1 name esw_pet_insert value false cmode runtime

/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode switchdev
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode switchdev

sudo devlink dev param set pci/0000:03:00.0 name esw_multiport value true cmode runtime
sudo devlink dev param set pci/0000:03:00.1 name esw_multiport value true cmode runtime
```

Make sure to perform graceful shutdown prior to power cycling the host.

To restore the old configuration:

```
sudo devlink dev param set pci/0000:03:00.0 name esw_multiport value false cmode runtime
sudo devlink dev param set pci/0000:03:00.1 name esw_multiport value false cmode runtime
```

15.13.6.2 Application Execution

The IPsec Security Gateway application is provided in source form. Therefore, a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_ipsec_security_gw [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                          Print a help synopsis
  -v, --version                       Print program version information
  -l, --log-level                     Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                     Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICA
L, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                   Parse all command flags from an input json file

Program Flags:
  -s, --secured                       secured port pci-address
  -u, --unsecured                     unsecured port pci-address
  -c, --config                        Path to the JSON file with application configuration
  -m, --mode                          ipsec mode - {tunnel/transport/udp_transport}
  -i, --ipc                           IPC socket file path
  -sn, --secured-name                 secured port interface name
  -un, --unsecured-name               unsecured port interface name
  -n, --nb-cores                      number of cores
  --debug                             Enable debug counters
```

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
./doca_ipsec_security_gw -h
```

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the BlueField or host:
    - Static Configuration:

    ```
    ./doca_ipsec_security_gw -s 03:00.0 -u 03:00.1 -c ./ipsec_security_gw_config.json -m transport
    ```

    > ⚠ Both the PCIe address identifiers (`-s` and `-u` flags) should match the addresses of the desired PCIe devices.

    - Dynamic Configuration:

    ```
    ./doca_ipsec_security_gw -s 03:00.0 -u 03:00.1 -c ./ipsec_security_gw_config.json -m transport -i /
    tmp/rules_socket
    ```

    > ⚠ Both the PCIe address identifiers (`-s` and `-u` flags) should match the addresses of the desired PCIe devices.

3. The application also supports a JSON-based deployment mode, in which all command-line arguments are provided through a JSON file:

    ```
    ./doca_ipsec_security_gw --json [json_file]
    ```

    For example

    ```
    ./doca_ipsec_security_gw --json ipsec_security_gw_params.json
    ```

    > ⚠ Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the PCIe addresses necessary for the deployment.

15.13.6.3 Command Line Flags

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
General flags	`h`	`help`	Prints a help synopsis	N/A
	`v`	`version`	Prints program version information	N/A
	`l`	`log-level`	Set the log level for the application: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70 (requires compilation with `TRACE` log level support)	`"log-level": 60`

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
	N/A	`sdk-log-level`	Sets the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70	`"sdk-log-level": 40`
	`j`	`json`	Parse all command flags from an input json file	N/A
Program flags	`c`	`config`	Path to JSON file with configurations	`"config": "security_gateway_config.json"`
	`u`	`unsecured`	PCIe address for the unsecured port	`"unsecured": "03:00.1"`
	`s`	`secured`	PCIe address for the secured port	`"secured": "03:00.0"`
	`m`	`mode`	IPsec mode. Possible values: `tunnel`, `transport`, `udp_transport`	`"mode": "tunnel"`
	`un`	`unsecured-name`	Interface name of the unsecured port	`"unsecured-name": "p1"`
	`sn`	`secured-name`	Interface name of the secured port	`"secured-name": "p0"`
	`i`	`ipc`	IPC socket file path for receiving IPsec rules during runtime	`"ipc": "/tmp/rules_socket"`
	`n`	`nb-cores`	Number of cores	`"nb-cores": 8`
	N/A	`debug`	Add counters to all the entries	`"debug": true`

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

15.13.6.4 Static Configuration IPsec Rules

IPsec rules and other configuration can be added with a JSON config file which is passed using the `--config` parameter.

Section	Field	Type	Description	Example
`config`	`switch`	bool	Configures whether DOCA Flow runs in VNF (`false`) or switch (`true`) mode	`"switch": true`
	`esp-header-offload`	string	Decap and encap offloading: `both` , `encap` , `decap` , or `none` . Default is `both` (offloading both encap and decap).	`"esp-header-offload": "none"`
	`sw-sn-inc-enable`	bool	Increments sequence number of ESP in software if set to `true` . Default is false. ⚠ Available only if `esp_header_offload` is `decap` or `none` .	`"sw-sn-inc-enable": true`
	`sw-antireplay-enable`	bool	Enables anti-replay mechanism in software if set to `true` . Default is false. ⚠ Available only if `esp_header_offload` is `encap` or `none` . ⚠ Window size is 64. Not ESN. Supports non-zero `sn_initial` .	`"sw-antireplay-enable": true`
	`sn-initial`	uint	Initial sequence number for ESP header. Used also when `sw_antireplay_enable` is true. Default is 0.	`"sn-initial": 0`
	`debug`	bool	Set debug counter for all entries when `true` . Default is `false` . This parameter is also used from CLI, will be taken as true if was sent in one of them.	`"debug": false`

Section	Field	Type	Description	Example
	`fwd-bad-syndrome`	string	Forward packets that has bad syndrome: `drop` , `RSS` . Default is `drop` . ⚠ Only available in debug mode.	`"fwd-bad-syndrome": "drop"`
	`perf-measurements`	string	Possible values: `none` , `insertion-rate` , `bandwidth` , `both` . Default is `none` . • `insertion-rate` - print the total time it took to add the entries • `bandwidth` - optimize the pipe to improve pps for IPv6	`"perf-measurements": "both"`
	`vxlan-encap`	bool	When `true` , preform `vxlan-encap` after encryption and decap before decryption. Default is `false` .	`"vxlan-encap": false`
	`vni`	uint	When `vxlan-encap` is true, use this `vni` value in the VXLAN tunnel.	`"vni": 5`
	`marker-encap`	bool	When `true` , add an extra non-ESP marker of 8 bytes. Default is `false` .	`"marker-encap": false`
`encrypt_rules`	`ip-version`	int	Source and destination IP version. Possible values: `4` , `6` . Optional; default is `4` .	`"ip-version": 6`
	`src-ip`	string	Source IP to match	`"src-ip": "1.2.3.4"`
	`dst-ip`	string	Destination IP to match	`"dst-ip": "101:101:101:101:101:101:101:101"`
	`protocol`	string	L4 protocol: TCP or UDP	`"protocol"`
	`src-port`	int	Source port to match	
	`dst-port`	int	Destination port to match	`"dst-port": 55`
	`encap-ip-version`	int	Encap IP version: `4` or `6` . Optional; default is `4` .	`"ip-version": 4`

Section	Field	Type	Description	Example
	`encap-dst-ip`	string	Encap destination IP ⚠ Mandatory for tunnel mode only.	`"encap-dst-ip": "1.1.1.1"`
	`spi`	int	SPI integer to set in the ESP header	`"spi": 5`
	`key`	string	Key for creating the SA (in hex format)	`"key": "112233445566778899aabbccdd"`
	`key-type`	int	Key size: `128` or `256`. Optional; default is `256`.	`"key-type": 128`
	`salt`	int	Salt value for creating the SA. Default is `6`.	`"salt": 1212`
	`icv-length`	int	ICV length value: `8`, `12`, or `16`. Default is `16`.	`"icv-length": 12`
	`lifetime-threshold`	int	Set IPsec lifetime threshold. Ignored if `sw-sn-inc-enable` is true. Default is 0.	`"lifetime-threshold": 1000000`
	`esn_en`	bool	Enables extended sequence number. Default is `false`.	`"esn_en" : true`
`decrypt_rules`	`ip-version`	int	Destination IP version: `4` or `6`. Optional; default is `4`.	`"ip-version": 6`
	`dst-ip`	string	Destination IP to match	`"dst-ip": "1122:3344:5566:7788:99aa:bbcc:ddee:ff00"`
	`inner-ip-version`	int	Inner IP version: `4` or `6`. Optional; default is `4`. ⚠ Mandatory for tunnel mode only.	`"inner-ip-version": 4`
	`spi`	int	SPI to match in the ESP header	`"spi": 5`

Section	Field	Type	Description	Example
	`key`	string	Key for creating the SA (in hex format)	`"key": "112233445566778899aabbccdd"`
	`key-type`	int	Key size: `128` or `256`. Optional; default is `256`.	`"key-type": 128`
	`salt`	int	Salt value for creating the SA. Default is `6`.	`"salt": 1212`
	`icv-length`	int	ICV length value: `8`, `12`, or `16`. Default is `16`.	`"icv-length": 12`
	`lifetime-threshold`	int	Set IPsec lifetime threshold. Ignored if `sw-antireplay-enable` is true. Default is 0.	`"lifetime-threshold": 1000000`
	`esn_en`	bool	Enables extended sequence number. Default is `false`.	`"esn_en" : true`

15.13.6.5 Dynamic Configuration IPsec Rules

The application listens on the UDS socket for receiving a predefined structure for the IPsec policy defined in the `policy.h` file.

The client program or keying daemon should connect to the socket with the same socket file path provided to the application by the `--ipc` / `-i` flags, and send the policy structure as packed to the application through the same socket.

> In the dynamic configuration, the application uses the `config` section from the JSON config file and ignores the `encrypt_rules` and `decrypt_rules` sections.

The IPsec policy structure:

```
struct ipsec_security_gw_ipsec_policy {
    /* Protocols attributes */
    uint16_t src_port;                   /* Policy inner source port */
    uint16_t dst_port;                   /* Policy inner destination port */
    uint8_t l3_protocol;                 /* Policy L3 proto {POLICY_L3_TYPE_IPV4, POLICY_L3_TYPE_IPV6} */
    uint8_t l4_protocol;                 /* Policy L4 proto {POLICY_L4_TYPE_UDP, POLICY_L4_TYPE_TCP} */
    uint8_t outer_l3_protocol;           /* Policy outer L3 type {POLICY_L3_TYPE_IPV4, POLICY_L3_TYPE_IPV6} */

    /* Policy attributes */
    uint8_t policy_direction;            /* Policy direction {POLICY_DIR_IN, POLICY_DIR_OUT} */
    uint8_t policy_mode;                 /* Policy IPSEC mode {POLICY_MODE_TRANSPORT, POLICY_MODE_TUNNEL} */

    /* Security Association attributes */
    uint8_t esn;                             /* Is ESN enabled? */
    uint8_t icv_length;                      /* ICV length in bytes {8, 12, 16} */
    uint8_t key_type;                        /* AES key type {POLICY_KEY_TYPE_128, POLICY_KEY_TYPE_256} */
    uint32_t spi;                            /* Security Parameter Index */
    uint32_t salt;                           /* Cryptographic salt */
    uint8_t enc_key_data[MAX_KEY_LEN];       /* Encryption key (binary) */

    /* Policy inner and outer addresses */
    char src_ip_addr[MAX_IP_ADDR_LEN + 1];       /* Policy inner IP source address in string format */
    char dst_ip_addr[MAX_IP_ADDR_LEN + 1];       /* Policy inner IP destination address in string format */
    char outer_src_ip[MAX_IP_ADDR_LEN + 1];        /* Policy outer IP source address in string format */
    char outer_dst_ip[MAX_IP_ADDR_LEN + 1];        /* Policy outer IP destination address in string format */
};
```

> ⚠ The policy type, whether it is encrypted or decrypted, is classified according to the `policy_direction` attribute:
>
> - `POLICY_DIR_IN` - decryption policy
> - `POLICY_DIR_OUT` - encryption policy

15.13.6.6 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

15.13.7 Application Code Flow

1. Parse application argument.
   a. Initialize arg parser resources and register DOCA general parameters.

      ```
      doca_argp_init();
      ```

   b. Register the application's parameters.

      ```
      register_ipsec_security_gw_params();
      ```

   c. Parse the arguments.

      ```
      doca_argp_start();
      ```

      i. Parse app parameters.
2. DPDK initialization.

   ```
   rte_eal_init();
   ```

   Call `rte_eal_init()` to initialize EAL resources with the provided EAL flags for not probing the ports.
3. Parse config file.

   ```
   ipsec_security_gw_parse_config();
   ```

4. Initialize devices and ports.

   ```
   ipsec_security_gw_init_devices();
   ```

   a. Open DOCA devices with input PCIe addresses / interface names.
   b. Probe DPDK port from each opened device.
5. Initialize and start DPDK ports.

   ```
   dpdk_queues_and_ports_init();
   ```

    a. Initialize DPDK ports, including mempool allocation.
    b. Initialize hairpin queues if needed.
    c. Binds hairpin queues of each port to its peer port.

6. Initialize DOCA Flow.

   ```
   ipsec_security_gw_init_doca_flow();
   ```

    a. Initialize DOCA Flow library.
    b. Find the indices of the DPDK-probed ports and start DOCA Flow ports with them.

7. Insert rules.
    a. Insert encryption rules.

       ```
       ipsec_security_gw_insert_encrypt_rules();
       ```

    b. Insert decryption rules.

       ```
       ipsec_security_gw_insert_decrypt_rules();
       ```

8. Wait for traffic.

   ```
   ipsec_security_gw_wait_for_traffic();
   ```

    a. wait in a loop until the user terminates the program.

9. IPsec security gateway cleanup:
    a. DOCA Flow cleanup; destroy initialized ports.

       ```
       doca_flow_cleanup();
       ```

    b. SA destruction.

       ```
       ipsec_security_gw_destroy_sas();
       ```

    c. IPsec objects destruction.

       ```
       ipsec_security_gw_ipsec_ctx_destroy();
       ```

    d. Destroy DPDK ports and queues.

       ```
       dpdk_queues_and_ports_fini();
       ```

    e. DPDK finish.

       ```
       dpdk_fini();
       ```

       Calls `rte_eal_destroy()` to destroy initialized EAL resources.

    f. Arg parser destroy.

       ```
       doca_argp_destroy()
       ```

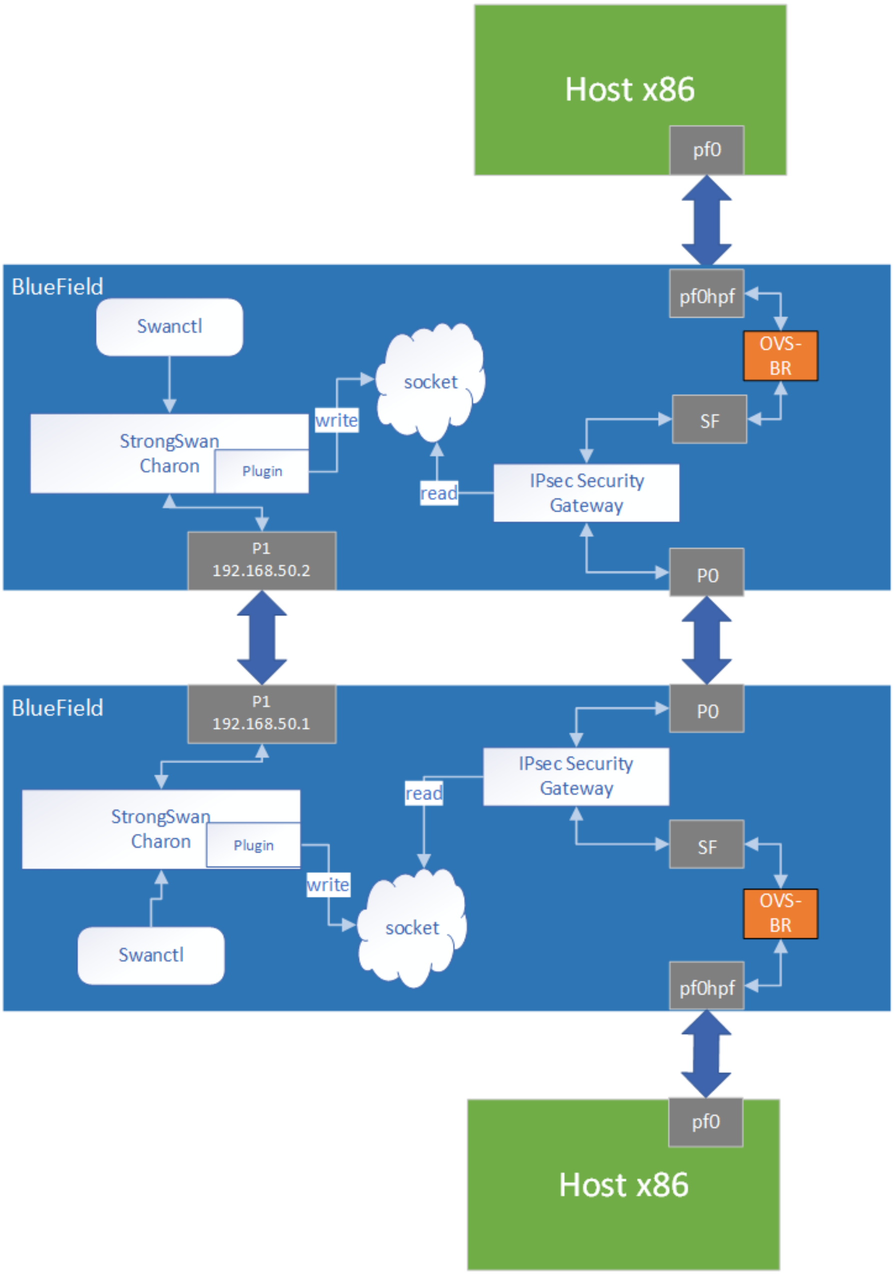
15.13.8 Keying Daemon Integration (StrongSwan)

strongSwan is a keying daemon that uses the Internet Key Exchange Version 2 (IKEv2) protocol to establish SAs between two peers. strongSwan includes a DOCA plugin that is part of the strongSwan package in BFB. The plugin is loaded only if the DOCA IPsec Security Gateway is triggered. The plugin connects to UDS socket and sends IPsec policies to the application after the key exchange completes.

For more information about the key daemon, refer to strongSwan documentation.

15.13.8.1 End-to-end Architecture

The following diagram presents an architecture where two BlueField DPUs are connected to each other with DOCA IPsec Security Gateway running on each.



`swanctl` is a command line tool used for strongSwan IPsec configuration:

1. Run DOCA IPsec Security Gateway on both sides in a dynamic configuration.
2. Start strongSwan service.
3. Configure strongSwan IPsec using the `swanctl.conf` configuration file on both sides.
4. Start key exchange between the two peers. At the end of the flow, the result arrives to the DOCA plugin, populates the policy-defined structure, and sends it to the socket.
5. DOCA IPsec Security Gateway on both sides reads new policies from the socket, performs the parsing, creates a DOCA SA object, and adds flow decrypt/encrypt entry.

This architecture uses P1 uplink on both BlueField DPUs to run the strongSwan key daemon. To configure the uplink:

1. Configure an IP addresses for the PFs of both DPUs:

a. On BF1:

```
ip addr add 192.168.50.1/24 dev p1
```

b. On BF2:

```
ip addr add 192.168.50.2/24 dev p1
```

> ⚠ It is possible to configure multiple IP addresses to uplinks to run key exchanges with different policy attributes.

2. Verify the connection between two BlueField DPUs.

```
BF1> ping 192.168.50.2
```

> ⚠ Make sure that the uplink is not in OVS bridges.

3. Configure the `swanctl.conf` files for each machine. The file should be located under `/etc/swanctl/conf.d/`.

   Adding `swanctl.conf` file examples:

   - Transport mode
     - `swanctl.conf` example for BF1:

```
connections {
   BF1-BF2 {
      local_addrs  = 192.168.50.1
      remote_addrs = 192.168.50.2
      rekey_time = 0

      local {
        auth = psk
        id = host1
      }
      remote {
        auth = psk
        id = host2
      }

      children {
         bf {
            local_ts = 192.168.50.1/32 [udp/60]
            remote_ts = 192.168.50.2/32 [udp/90]
            esp_proposals = aes128gcm128-x25519-esn
            mode = transport
            policies_fwd_out = yes
            life_time = 0
         }
      }
      version = 2
      mobike = no
      reauth_time = 0
      proposals = aes128-sha256-x25519
   }
}

secrets {
   ike-BF {
      id-host1 = host1
      id-host2 = host2
      secret = 0sv+NkxY9LLZvwj4qCC2o/gGrWDF2d21jL
   }
}
```

     - `swanctl.conf` example for BF2:

```
connections {
   BF2-BF1 {
      local_addrs  = 192.168.50.2
      remote_addrs = 192.168.50.1
      rekey_time = 0

      local {
        auth = psk
        id = host2
      }
      remote {
        auth = psk
        id = host1
      }

      children {
         bf {
            local_ts = 192.168.50.2/32 [udp/90]
            remote_ts = 192.168.50.1/32 [udp/60]
            esp_proposals = aes128gcm128-x25519-esn
            mode = transport
            life_time = 0
         }
      }
      version = 2
      mobike = no
      reauth_time = 0
      proposals = aes128-sha256-x25519
   }
}

secrets {
   ike-BF {
      id-host1 = host1
      id-host2 = host2
      secret = 0sv+NkxY9LLZvwj4qCC2o/gGrWDF2d21jL
   }
}
```

- Tunnel mode

```
connections {
   BF1-BF2 {
      local_addrs  = 192.168.50.2
      remote_addrs = 192.168.50.1
      rekey_time = 0

      local {
        auth = psk
        id = host2
      }
      remote {
        auth = psk
        id = host1
      }

      children {
         bf {
            local_ts = 2001:db8:85a3::8a2e:370:7334/128 [udp/3030]
            remote_ts = 2001:db8:85a3::8a2e:370:7335/128 [udp/55]
            esp_proposals = aes128gcm128-x25519-esn
            life_time = 0
         }
      }
      version = 2
      mobike = no
      proposals = aes128-sha256-x25519
   }
}

secrets {
   ike-BF {
      id-host1 = host1
      id-host2 = host2
      secret = 0sv+NkxY9LLZvwj4qCC2o/gGrWDF2d21jL
   }
}
```

`local_ts` and `remote_ts` must have a netmask of /32 for IPv4 addresses and /128 for IPv6 addresses.

SA rekey is not supported in DOCA plugin. `connection.rekey_time` must be set to 0 and `connection.child.life_time` must be set to 0.

DOCA IPsec only supports ESP headers, AES-GCM encryption algorithm, and key sizes 128 or 256. Therefore, when setting ESP proposals in the `swanctl.conf`, please adhere to the values provided in the following table:

ESP Proposal	Algorithm Type Including ICV Length	Key Size
aes128gcm8	ENCR_AES_GCM_ICV8	128
aes128gcm64	ENCR_AES_GCM_ICV8	128
aes128gcm12	ENCR_AES_GCM_ICV12	128
aes128gcm96	ENCR_AES_GCM_ICV12	128
aes128gcm16	ENCR_AES_GCM_ICV16	128
aes128gcm128	ENCR_AES_GCM_ICV16	128
aes128gcm	ENCR_AES_GCM_ICV16	128
aes256gcm8	ENCR_AES_GCM_ICV8	256
aes256gcm64	ENCR_AES_GCM_ICV8	256
aes256gcm12	ENCR_AES_GCM_ICV12	256
aes256gcm96	ENCR_AES_GCM_ICV12	256
aes256gcm16	ENCR_AES_GCM_ICV16	256
aes256gcm128	ENCR_AES_GCM_ICV16	256
aes256gcm	ENCR_AES_GCM_ICV16	256

15.13.8.2 Running the Solution

Run the following commands on both BlueField peers.

1. Run DOCA IPsec Security Gateway in dynamic configuration, assuming the socket location is `/tmp/rules_socket`.

```
doca_ipsec_security_gw -s 03:00.0 -un <sf_net_dev> -c ./ipsec_security_gw_config.json -m transport -i /tmp/
rules_socket
```

> ⚠ DOCA IPsec Security Gateway application should be run first.

2. Edit the `/etc/strongswan.d/charon/doca.conf` file and add the UDS socket path. If the `socket_path` is not set, the plugin uses the default path `/tmp/strongswan_doca_socket`.

```
doca {

# Whether to load the plugin
load = yes

# Path to DOCA socket
socket_path = /tmp/rules_socket
}
```

> ⚠ You must provide the application with this path as well.

3. Restart the strongSwan server:

```
systemctl restart strongswan.service
```

> If the application has been run with log level debug, you can see that the connection has been done successfully and the application is waiting for new IPsec policies.

4. Verify that the `swanctl.conf` file exists in `/etc/swanctl/conf.d/` . directory.

> It is recommended to remove any unused conf files under `/etc/swanctl/conf.d/` .

5. Load IPsec configuration:

```
swanctl --load-all
```

6. Start IKE protocol on either the initiator or the target side:

```
swanctl -i --child <child_name>
```

> In the example above, the child's name is `bf` .

15.13.8.3 Building strongSwan

To perform some changes in the DOCA plugin in strongSwan zone:

1. Verify that the dependencies listed here are installed in your environment. `libgmp-dev` is missing from that list so make sure to install that as well.
2. Git clone https://github.com/Mellanox/strongswan.git.
3. Git checkout BF-5.9.10 branch.
4. Add your changes in the plugin located under `src/libcharon/plugins/doca` .
5. Run `autogen.sh` within the strongSwan repo.
6. Run the following:

```
./configure --enable-openssl --disable-random --prefix=/usr/local --sysconfdir=/etc --enable-systemd --
enable-doca
make
make install
systemctl daemon-reload
systemctl restart strongswan.service
```

15.13.9 References

- `/opt/mellanox/doca/applications/ipsec_security_gw/`
- `/opt/mellanox/doca/applications/ipsec_security_gw/ipsec_security_gw_params.json`

15.14 NVIDIA DOCA PCC Application Guide

This document provides a DOCA PCC implementation on top of NVIDIA® BlueField® networking platform.

15.14.1 Introduction

Programmable Congestion Control (PCC) allows users to design and implement their own congestion control (CC) algorithm, giving them the flexibility to work out an optimal solution to handle congestion in their clusters. On BlueField-3 networking platforms, PCC is provided as a component of DOCA.

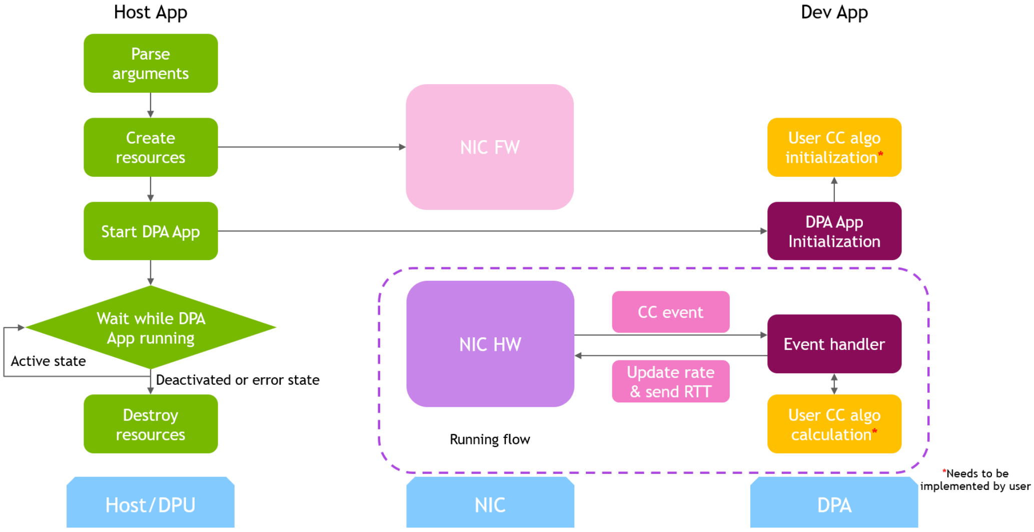
The application leverages the DOCA PCC API to provide users the flexibility to manage allocation of DPA resources according to their requirements.

Typical DOCA application includes App running on host/Arm and App running on DPA. Developers are advised to use the host/Arm application with minimal changes and focus on developing their algorithm and integrating it into the DPA application.

15.14.2 System Design

DOCA PCC application consists of two parts:

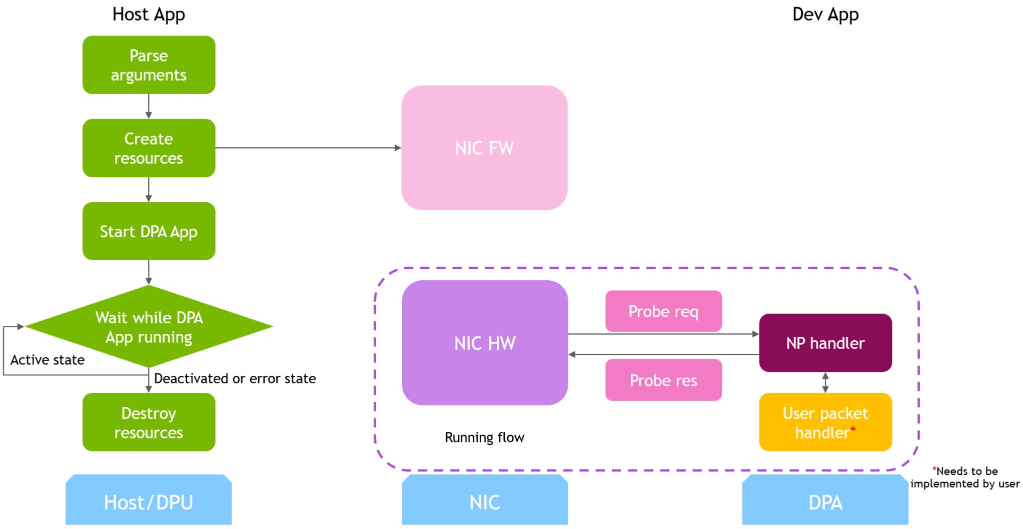
- Host/Arm app is the control plane. It is responsible for allocating all resources and handover to the DPA app initially, then destroying everything when the DPA app finishes its operation. The host app must always be alive to stay in control while the device app is working.
- Device/DPA app is the data plane.
  - The default mode of the data plane is running as a reaction point (RP). When the first thread is activated, DPA App initialization is done in the DOCA PCC library by calling the algorithm initialization function implemented by the user in the app. Moreover, the user algorithm execution function is called when a CC event arrives. The user algorithm takes event data as input and performs a calculation, using per-flow context, and replies with the updated rate value and a flag to send an RTT request. The following is an illustration of the general RP application flow:



The host/Arm application sends a command to the BlueField platform firmware when allocating or destroying resources. CC events are generated by the BlueField platform hardware automatically when sending data or receiving ACK/NACK/CNP/RTT packets, then the device application handles these events by calling the user algorithm. After the DPA application replies to hardware, handling of current event is done, and the next event can arrive.

> ⓘ The device/DPA app can also run different algorithms for the RP program, which users can configure as a runtime option.

- The device/DPA app can function as a notification point (NP). When a new probe request packet arrives, the user handler can read and analyze the data and send a probe response back. The following is an illustration of the general NP application flow:



> ⓘ The device/DPA app is as well capable of functioning as a telemetry program for a NP NIC or switch operations, which users can configure as a runtime option.

15.14.3 Application Architecture

```
/opt/mellanox/doca/applications/pcc/
    host
         pcc.c
         pcc_core.c
         pcc_core.h
    device
         pcc_common_dev.h
         rp
              rtt_template
                        algo
                             rtt_template.h
                             rtt_template_algo_params.h
                             rtt_template_ctxt.h
                             rtt_template.c
                        rp_rtt_template_dev_main.c
              switch_telemetry
                        algo
                             telem_template.h
                             telem_template_algo_params.h
                             telem_template_ctxt.h
                             telem_template.c
                        rp_switch_telemetry_dev_main.c
         np
             np_nic_telemetry_dev_main.c
             np_switch_telemetry_dev_main.c
```

The main content of the reference DOCA PCC application files are the following:

- `host/pcc.c` - entry point to entire application
- `host/pcc_core.c` - host functions to initialize and destroy the PCC application resources, parsers for PCC command line parameters
- `device/pcc_common_dev.h` - common util calls and definitions for device programs
- `device/rp/rtt_template/rp_rtt_template_dev_main.c` - callbacks for user CC algorithm initialization, user CC algorithm calculation and algorithm parameter change notification of the RTT template algorithm reference
- `device/rp/rtt_template/algo/*` - user CC algorithm reference for RTT template. Put user algorithm code here
- `device/rp/switch_telemetry/rp_switch_telemetry_dev_main.c` - callbacks for user CC algorithm initialization, user CC algorithm calculation, and algorithm parameter change notification of the switch telemetry algorithm reference
- `device/rp/switch_telemetry/algo/*` - user CC algorithm reference for switch telemetry. Put user algorithm code here.
- `device/np/np_nic_telemetry_dev_main.c` - callback for user NP handling, implemented as a NIC telemetry program to observe RX counters
- `device/np/np_switch_telemetry_dev_main.c` - callback for user NP handling, implemented as a switch telemetry program to observe last hop switch metadata

15.14.4 DOCA Libraries

This application leverages the following DOCA library:

- DOCA PCC

Refer to its respective programming guide for more information.

15.14.5 Dependencies

- NVIDIA BlueField-3 Platform is required
- Firmware 32.38.1000 and higher
- MFT 4.25 and higher

15.14.6 Compiling the Application

ⓘ Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

✓ For more information about the applications as well as development and compilation tips, refer to the DOCA Applications page.

The sources of the application can be found under the application's directory: `/opt/mellanox/doca/applications/pcc/` .

15.14.6.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

`doca_pcc` is created under `/tmp/build/pcc/` .

15.14.6.2 Compiling Only the Current Application

To directly build only the PCC application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_pcc=true
ninja -C /tmp/build
```

`doca_pcc` is created under `/tmp/build/pcc/` .

Alternatively, one can set the desired flags in the `meson_options.txt` file instead of providing them in the compilation command line:

1. Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt` :
    - Set `enable_all_applications` to `false`
    - Set `enable_pcc` to `true`
2. Run the following compilation commands:

    ```
    cd /opt/mellanox/doca/applications/
    meson /tmp/build
    ninja -C /tmp/build
    ```

    `doca_pcc` is created under `/tmp/build/pcc/` .

15.14.6.3 Compilation Options

The application offers specific compilation flags which one can set for a desired behavior in the device/DPA program.

In the `meson_options.txt` file, one can find the following options:

- `enable_pcc_application_tx_counter_sampling` : set to `true` to use TX counters sampled at runtime in the RP CC handling algorithm.
- `enable_pcc_application_np_rx_rate` : set to `true` to use RX counters received from NP in the RP CC handling algorithm.

15.14.6.4 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application.

15.14.7 Running the Application

15.14.7.1 Prerequisites

Enable `USER_PROGRAMMABLE_CC` in `mlxconfig` :

```
mlxconfig -y -d /dev/mst/mt41692_pciconf0 set USER_PROGRAMMABLE_CC=1
```

Perform a BlueField system reboot for the `mlxconfig` settings to take effect.

15.14.7.2 Application Execution

The PCC application is provided in source form. Therefore, a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_pcc [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                    Print a help synopsis
  -v, --version                 Print program version information
  -l, --log-level               Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR
, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level               Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=E
RROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>             Parse all command flags from an input json file

Program Flags:
  -d, --device <IB device names>       IB device name that supports PCC (mandatory).
  -np-nt, --np-nic-telemetry <PCC Notification Point NIC Telemetry> Flag to indicate running as a
Notification Point NIC Telemetry (optional). The application will generate CCMAD probe packets. By default
 the flag is set to false.
  -rp-st, --rp-switch-telemetry <PCC Reaction Point Switch Telemetry> Flag to indicate running as a
Reaction Point Switch Telemetry (optional). The application will generate IFA2 probe packets. By default
 the flag is set to false.
  -np-st, --np-switch-telemetry <PCC Notification Point Switch Telemetry> Flag to indicate running as a
Notification Point Switch Telemetry (optional). The application will generate IFA2 probe packets. By
default the flag is set to false.
  -t, --threads <PCC threads list>  A list of the PCC threads numbers to be chosen for the DOCA PCC context
to run on (optional). Must be provided as a string, such that the number are separated by a space.
  -w, --wait-time <PCC wait time>      The duration of the DOCA PCC wait (optional), can provide negative
values which means infinity. If not provided then -1 will be chosen.
  -r-handler, --remote-sw-handler <CCMAD remote SW handler> CCMAD remote SW handler flag (optional). If not
provided then false will be chosen.
  -hl, --hop-limit <IFA2 hop limit> The IFA2 probe packet hop limit (optional). If not provided then 0XFE
 will be chosen.
  -gns, --global-namespace <IFA2 global namespace> The IFA2 probe packet global namespace (optional). If
not provided then 0XF will be chosen.
  -gns-ignore_mask, --global-namespace-ignore-mask <IFA2 global namespace ignore mask> The IFA2 probe
packet global namespace ignore mask (optional). If not provided then 0 will be chosen.
  -gns-ignore_val, --global-namespace-ignore-value <IFA2 global namespace ignore value> The IFA2 probe
packet global namespace ignore value (optional). If not provided then 0 will be chosen.
  -f, --coredump-file <PCC coredump file> A pathname to the file to write coredump data in case of
unrecoverable error on the device (optional). Must be provided as a string.
  -i, --port-id <Physical port ID>  The physical port ID of the device running the application (optional).
If not provided then ID 0 will be chosen.
```

ⓘ This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
./doca_pcc -h
```

ⓘ For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the BlueField Platform or the host:

```
./doca_pcc -d mlx5_0
```

⚠ The IB device identifier (`mlx5_0`) should match the identifier of the desired IB device.

3. The application also supports a JSON-based deployment mode, in which all command-line arguments are provided through a JSON file:

```
./doca_pcc --json [json_file]
```

For example:

```
./doca_pcc --json ./pcc_params.json
```

⚠ Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the PCIe addresses necessary for the deployment.

15.14.7.3 Command Line Flags

Flag Type	Short Flag	Long Flag/ JSON Key	Description	JSON Content
General flags	`h`	`help`	Prints a help synopsis	N/A
	`v`	`version`	Prints program version information	N/A

Flag Type	Short Flag	Long Flag/ JSON Key	Description	JSON Content
	`l`	`log-level`	Sets the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70 The application uses a unique logging implementation that makes use of DOCA's logging levels.	N/A
	N/A	`sdk-log-level`	Sets the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70	N/A
	`j`	`json`	Parse all command flags from an input JSON file	N/A
Program flags	`d`	`device`	IB device name that supports PCC	`"device": ""`
	`np-nt`	`np-nic-telemetry`	(Optional) Flag to indicate running as a NP NIC telemetry. The DOCA PCC application can run as a NP NIC telemetry program. If this flag is used, the application loads a program to run on the DPA to sample RX NIC counters and send them in a response packet.	`"np-nic-telemetry": false`
	`rp-st`	`rp-switch-telemetry`	(Optional) Flag to indicate running as a RP switch telemetry. The DOCA PCC application can run as a RP switch telemetry program. If this flag is used, the application loads a program to run on the DPA of a switch telemetry algorithm which receives metadata from the last hop switch congestion point from the NP node.	`"rp-switch-telemetry": false`

Flag Type	Short Flag	Long Flag/ JSON Key	Description	JSON Content
	`np-st`	`np-switch-telemetry`	(Optional) Flag to indicate running as a NP switch telemetry. The DOCA PCC application can run as a NP switch telemetry program. If this flag is used, the application loads a program to run on the DPA to sample metadata from the last hop switch congestion point and send them in response packet.	`"np-switch-telemetry": false`

	`t`	`threads`	(Optional) A list of the PCC EU indexes to be chosen for the DOCA PCC event handler threads to run on. Must be provided as a string, such that the numbers are separated by a space.
			The placement of the PCC threads per core can be controlled using the EU indexes. Utilizing a large number of EUs, while limiting the number of threads per core, gives the best event handling rate and lowest event latency.
			The last EU is used for communication with the BlueField Platform while all others are for data path CC event handling.
			⚠ If `np-nic-telemetry` option is chosen by the user, a different set of threads will be chosen as default list.
				`"pcc-threads": "176 177 178 179 180 181 182 183`
				`184 185 186 187 192 193 194 195 196 197 198 199`
				`200 201 202 203 208 209 210 211 212 213 214 215`
				`216 217 218 219 224 225 226 227 228 229 230 231`
				`232 233 234 235 240"`

	`w`	`wait-time`	(Optional) In seconds, the duration of the DOCA PCC wait. Negative values mean infinity.	`"wait-time": -1`
	`r-handler`	`remote-sw-handler`	(Optional) CCMAD remote SW handler flag. Relevant for RP contexts. This flag indicates whether the expected CCMAD probe packet responses are generated by a remote DOCA NP process or not. ⚠ If using other probe types than CCMAD, probe packet responses are always expected to be generated from a remote DOCA NP process.	`"remote-sw-handler": false`

Flag Type	Short Flag	Long Flag/ JSON Key	Description	JSON Content
	`hl`	`hop-limit`	(Optional) The IFA2 probe packet hop limit Relevant for RP contexts.	`"hop-limit": 0xFE`
	`gns`	`global-namespace`	(Optional) The IFA2 probe packet global namespace Relevant for RP contexts.	`"global-namespace": 0xF`
	`gns-ignore-mask`	`global-namespace-ignore-mask`	(Optional) The IFA2 probe packet global namespace ignore mask Relevant for NP contexts.	`"global-namespace-ignore-mask": 0`
	`gns-ignore-val`	`global-namespace-ignore-value`	(Optional) The IFA2 probe packet global namespace ignore value Relevant for NP contexts.	`"global-namespace-ignore-value": 0`
	`f`	`coredump-file`	(Optional) A pathname to the file to write core dump data if an unrecoverable error occurs on the device	`"coredump-file": "/tmp/doca_pcc_coredump.txt"`
	`i`	`port-id`	(Optional) The physical port ID of the device running the application	`"port-id": 0`

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

15.14.7.4 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

15.14.8 Application Code Flow

This section lists the application's configuration flow, explaining the different DOCA function calls and wrappers.

1. Parse application argument.
   a. Initialize arg parser resources and register DOCA general parameters.

```
doca_argp_init();
```

b. Register PCC application parameters.

```
register_pcc_params();
```

c. Parse the arguments.

```
doca_argp_start();
```

i. Parse DOCA flags.
ii. Parse DOCA PCC parameters.

2. PCC initialization.

```
pcc_init();
```

a. Open DOCA device that supports PCC.
b. Create DOCA PCC context.
c. Configure affinity of threads handling CC events.

3. Start DOCA PCC.

```
doca_pcc_start();
```

a. Create PCC process and other resources.
b. Trigger initialization of PCC on device.
c. Register the PCC in the BlueField Platform hardware so CC events can be generated and an event handler can be triggered.

4. Process state monitor loop.

```
doca_pcc_get_process_state();
doca_pcc_wait();
```

a. Get the state of the process:

State	Description
`DOCA_PCC_PS_ACTIVE = 0`	The process handles CC events (only one process is active at a given time)
`DOCA_PCC_PS_STANDBY = 1`	The process is in standby mode (another process is already `ACTIVE`)
`DOCA_PCC_PS_DEACTIVATED = 2`	The process has been deactivated by the BlueField Platform firmware and should be destroyed
`DOCA_PCC_PS_ERROR = 3`	The process is in error state and should be destroyed

b. Wait on process events from the device.

5. PCC destroy.

```
doca_pcc_destroy();
```

a. Destroy PCC resources. The process stops handling PCC events.
b. Close DOCA device.

6. Arg parser destroy.

```
doca_argp_destroy()
```

15.14.9 Port Programmable Congestion Control Register

The Port Programmable Congestion Control (PPCC) register allows the user to configure and read PCC algorithms and their parameters/counters.

It supports the following functionalities:

- Enabling different algorithms on different ports
- Querying information of both algorithms and tunable parameters/counters
- Changing algorithm parameters without compiling and reburning user image
- Querying or clearing programmable counters

15.14.9.1 Usage

The PPCC register can be accessed using a string similar to the following:

```
sudo mlxreg -d /dev/mst/mt41692_pciconf0 -y --get --op "cmd_type=0" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=0,algo_param_index=0"
sudo mlxreg -d /dev/mst/mt41692_pciconf0 -y --set "cmd_type=1" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=0,algo_param_index=0"
```

Where you must:

- Set the `cmd_type` and the indexes
- Give values for `algo_slot` , `algo_param_index`
- Keep `local_port=1` , `pnat=0` , `lp_msb=0`
- Keep `doca_pcc` application running

cmd_type	Description	Method	Index	Input (in -- set)	Output
`0x0`	Get algorithm info	Get	`algo_slot`	N/A	• Value - 32-bit `algo_num` or 0 if no algo is available at this index • Text - algorithm description • `sl_bitmask_support` - indicates whether the device supports `sl_bitmask` logic

`0x1`	Enable algorithm	Set		`sl_bitmask`
				`trace_en`
				`counter_en`
					N/A

`0x2`	Disable algorithm	Set		N/A	N/A

cmd_type	Description	Method	Index	Input (in -- set)	Output
`0x3`	Get algorithm enabling status	Get		N/A	• Value: • 0 - disabled • 1 - enabled • `sl_bitmask` - this field allows to apply to specific SLs based on the bitmask • `sl_bitmask_support` - indicates whether the device supports `sl_bitmask` logic
`0x4`	Get number of parameters	Get		N/A	• Value - num of params of algo
`0x5`	Get parameter information	Get	`algo_slot` `algo_param_index`	N/A	• `param_value1` - default value of param • `param_value2` - min value of param • `param_value3` - max value of param • prm - • 0: read-only • 1: read-write • 2: read-only but may be cleared using the "get and clear" command
`0x6`	Get parameter value	Get		N/A	• Value - param value
`0x7`	Get and clear parameter	Get		N/A	• Value - param value
`0x8`	Set parameter value	Set		Parameter value	N/A
`0xA`	Bulk get parameters	Get	`algo_slot`	N/A	• `text_length` - param num x 4 bytes • `text[0]…text[n]` - param values
`0xB`	Bulk set parameters	Set		`text_length` - param num x 4 `text[0]…text[n]` - param values	N/A
`0xC`	Bulk get counters	Get		N/A	• `text_length` - counter num x 4 bytes • `text[0]…text[n]` - counter values

cmd_type	Description	Method	Index	Input (in --set)	Output
`0xD`	Bulk get and clear counters	Get		N/A	• `text_length` - counter num x 4 bytes • `text[0]…text[n]` - counter values
`0xE`	Get number of counters	Get		N/A	• Value - num of counters of algo
`0xF`	Get counter information	Get	`algo_slot` `algo_param_index`	N/A	• `param_value3` - max value of parameter • prm - • 0: read-only • 1: read-write • 2: read-only but may be cleared via "get & clear" command
`0x10`	Get algorithm info array	Get	N/A	N/A	• `text_length` - algo slot initialized x 4 bytes • `text[0]…text[n]` - 32-bit `algo_num` or 0 if no algorithm is available at this slot index

15.14.9.2 Internal Default Algorithm

The internal default algorithm is used when enhanced connection establishment (ECE) negotiation fails. It is mainly used for backward compatibility and can be disabled using "force mode". Otherwise, users may change `doca_pcc_dev_user_algo()` in the device app to run a specific algorithm without considering the algorithm negotiation.

The force mode command is per port:

```
sudo mlxreg -d /dev/mst/mt41692_pciconf0 -y --get --op "cmd_type=2" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=15,algo_param_index=0"
sudo mlxreg -d /dev/mst/mt41692_pciconf0.1 -y --get --op "cmd_type=2" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=15,algo_param_index=0"
```

15.14.9.3 Counters

Counters are shared on the port and are only enabled on one `algo_slot` per port. The following command enables the counters while enabling the algorithm according to the `algo_slot`:

```
sudo mlxreg -d /dev/mst/mt41692_pciconf0 -y --set "cmd_type=1,counter_en=1" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=0,algo_param_index=0"
```

After counters are enabled on the `algo_slot`, they can be queried using `cmd_type` 0xC or 0xD.

```
sudo mlxreg -d /dev/mst/mt41692_pciconf0 -y --get --op "cmd_type=12" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=0,algo_param_index=0"
sudo mlxreg -d /dev/mst/mt41692_pciconf0 -y --get --op "cmd_type=13" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=0,algo_param_index=0"
```

15.14.10 References

- `/opt/mellanox/doca/applications/pcc/`
- `/opt/mellanox/doca/applications/pcc/pcc_params.json`

15.15 NVIDIA DOCA PSP Gateway Application Guide

This document describes the usage of the NVIDIA DOCA PSP Gateway sample application on top of an NVIDIA® BlueField® networking platform or NVIDIA® ConnectX® SmartNIC.

15.15.1 Introduction

DOCA PSP Gateway is supported at alpha level.

DOCA PSP Gateway is supported only on BlueField-3 or ConnectX-7 and later.

The DOCA PSP Gateway application leverages the BlueField or ConnectX hardware capability for fully offloaded secure network communication using the PSP security protocol. The application demonstrates how to exchange keys between application instances and insert rules controlling PSP encryption and decryption using the DOCA Flow library.

The application exchanges keys using an unencrypted gRPC channel. If your environment requires the protection of encryption keys, you must modify the application to create the gRPC channel using the applicable certificates.

The PSP Gateway application supports only the PSP tunnel protocol. The PSP transport protocol is not supported by the application in this release, although it is supported by the underlying DOCA Flow library.

The PSP Gateway application supports only IPv4 inner and IPv6 outer headers. Other combinations are not supported by the application in the current release, although they are supported by the underlying DOCA Flow library.

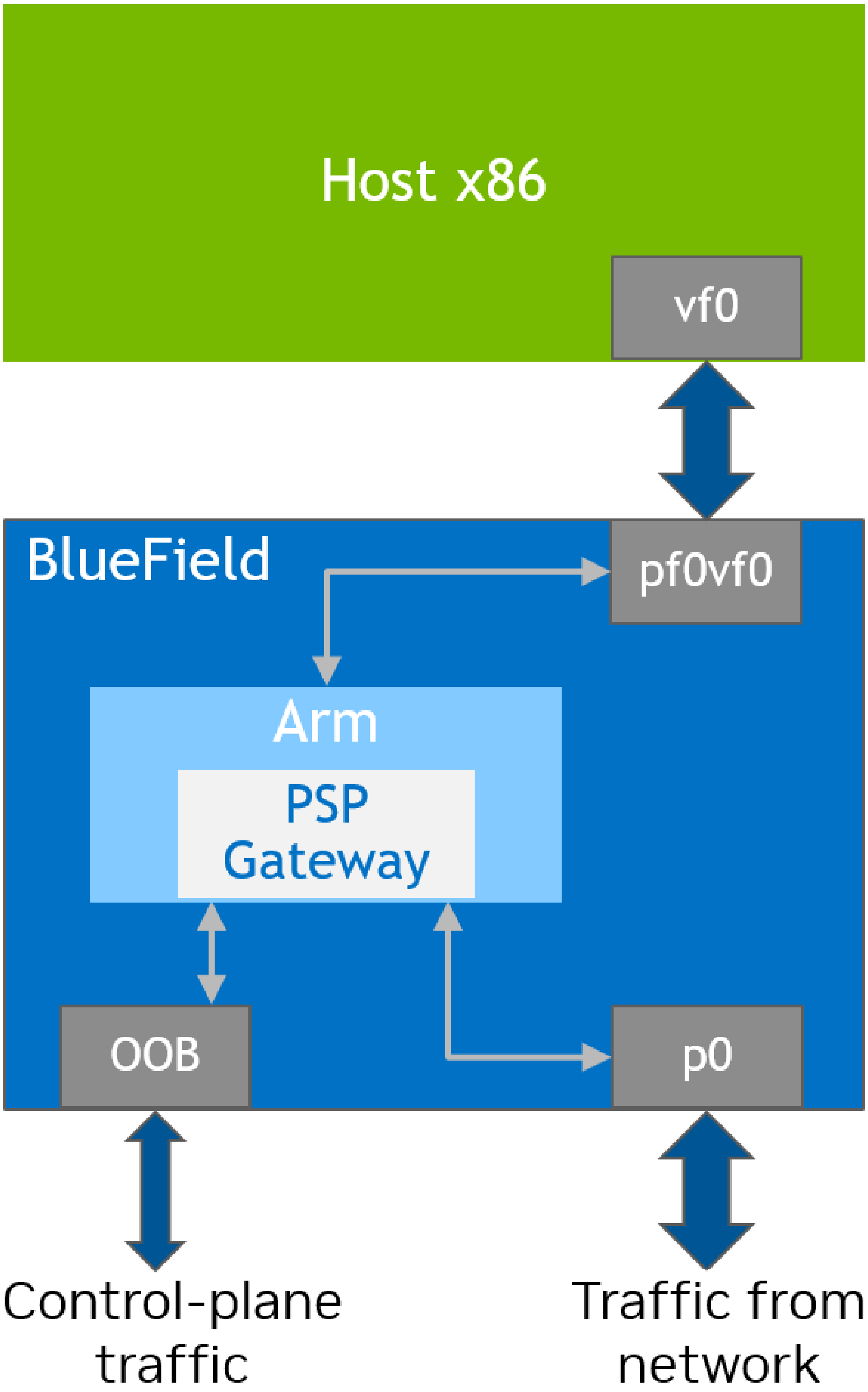
The application can be configured to establish out-bound PSP tunnel connections via individual command-line arguments, or via a text file configured via a command-line argument. The connections are established on-demand by default, but can also be configured to connect at startup.

15.15.2 System Design

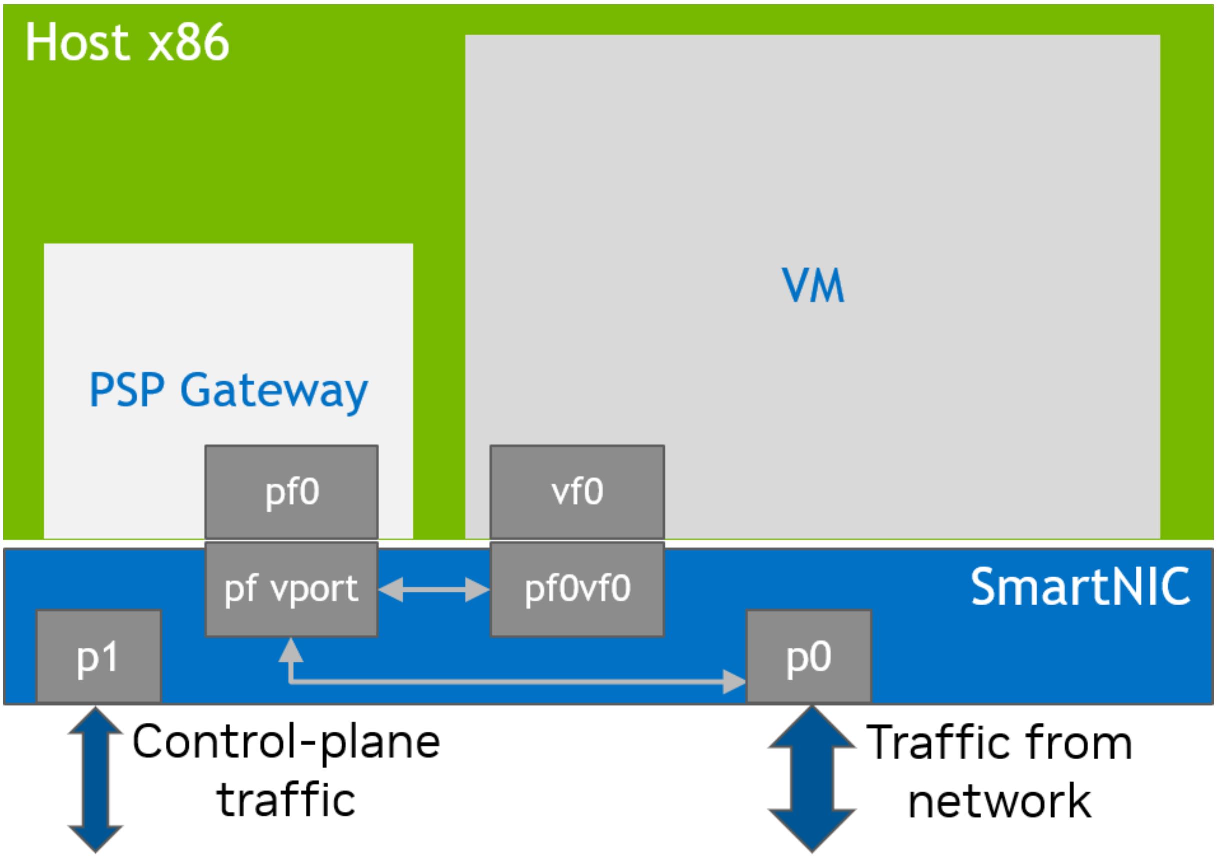
The DOCA PSP Gateway is designed to run with three ports:

- A secure (encrypted) uplink netdev (i.e., `p0`)
- An unsecured (plaintext) netdev representor (VF or SF)
- An out-of-bound (OOB) management port, used to communicate with peer instances using standard sockets

Whether the DOCA PSP Gateway is deployed to a BlueField or a ConnectX device, the functionality is the same. The Out of Bounds (OOB) network device carries PSP parameters between peers, the Uplink port carries secure (encrypted) traffic, and the VF carries the unencrypted traffic.



When the application is deployed to a DPU, the operation of the PSP encryption protocol is entirely transparent to the Host. All the resources required to manage the PSP connections are physically located on the DPU.



When the application is deployed to the host, the operation of the PSP encryption protocol is the responsibility of the host, and resources are allocated from the host. However, the operation of the PSP encryption protocol is entirely transparent to any virtual machines and containers attached to the VF network devices.

15.15.3 Application Architecture

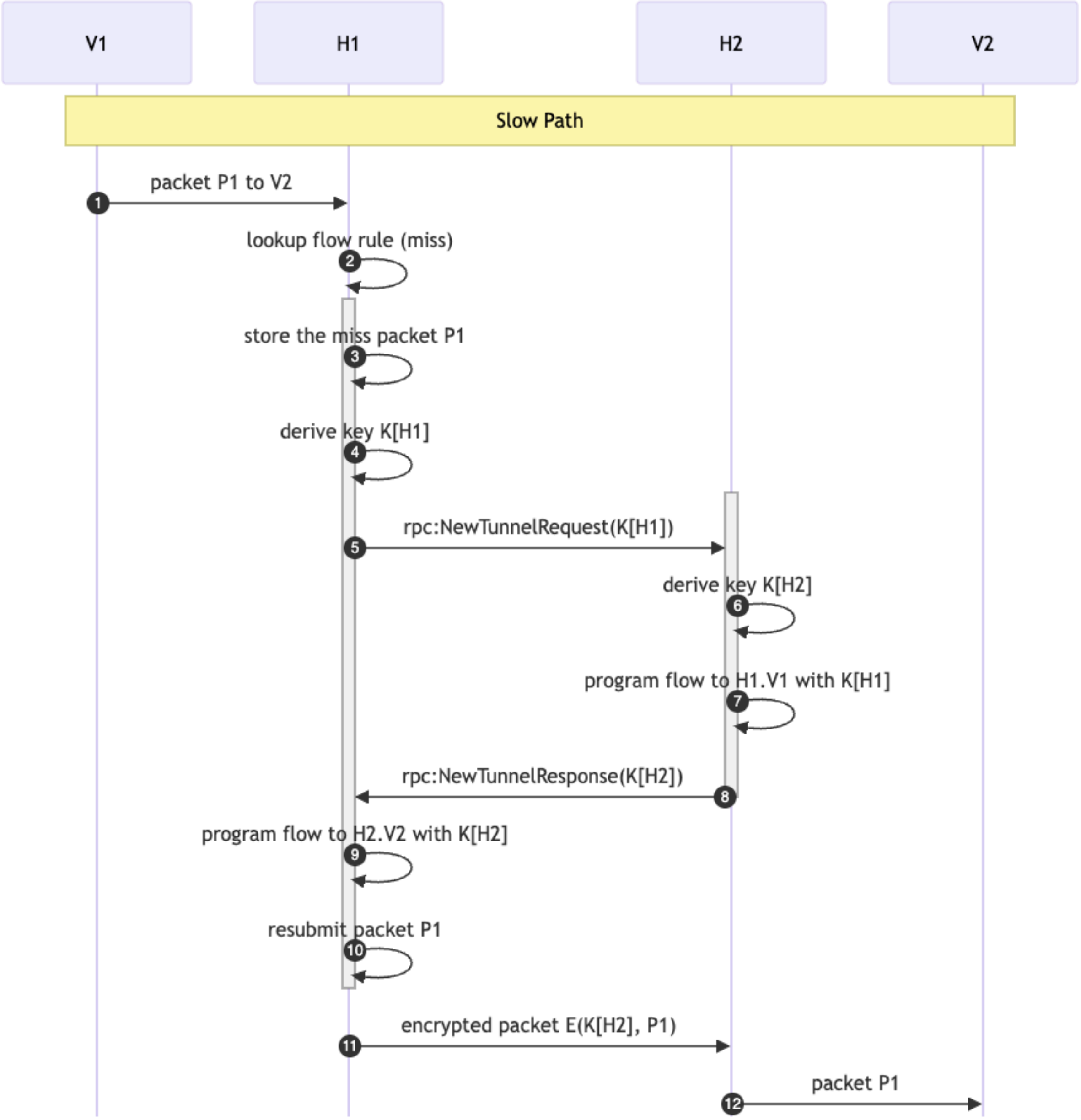
The creation of PSP tunnel connections requires two-way communication between peers. Each "sender" must request a unique security parameters index (SPI) and encryption key from the intended "receiver". The receiver derives sequential SPIs and encryption keys using the hardware resources inside the BlueField or ConnectX device, which manages a secret pair of master keys to produce the SPIs and encryption keys.

One key architectural benefit of PSP over similar protocols (e.g., IPsec) is that the receiver does not incur any additional resource utilization whenever it creates a new SPI and encryption key. This is because the decryption key associated with the SPI is computed on the fly, based on the SPI and master key, for each received packet. This lack of requirement for additional context memory for each additional decryption rule is partly responsible for the ability of the PSP protocol to scale to many thousands of peers.

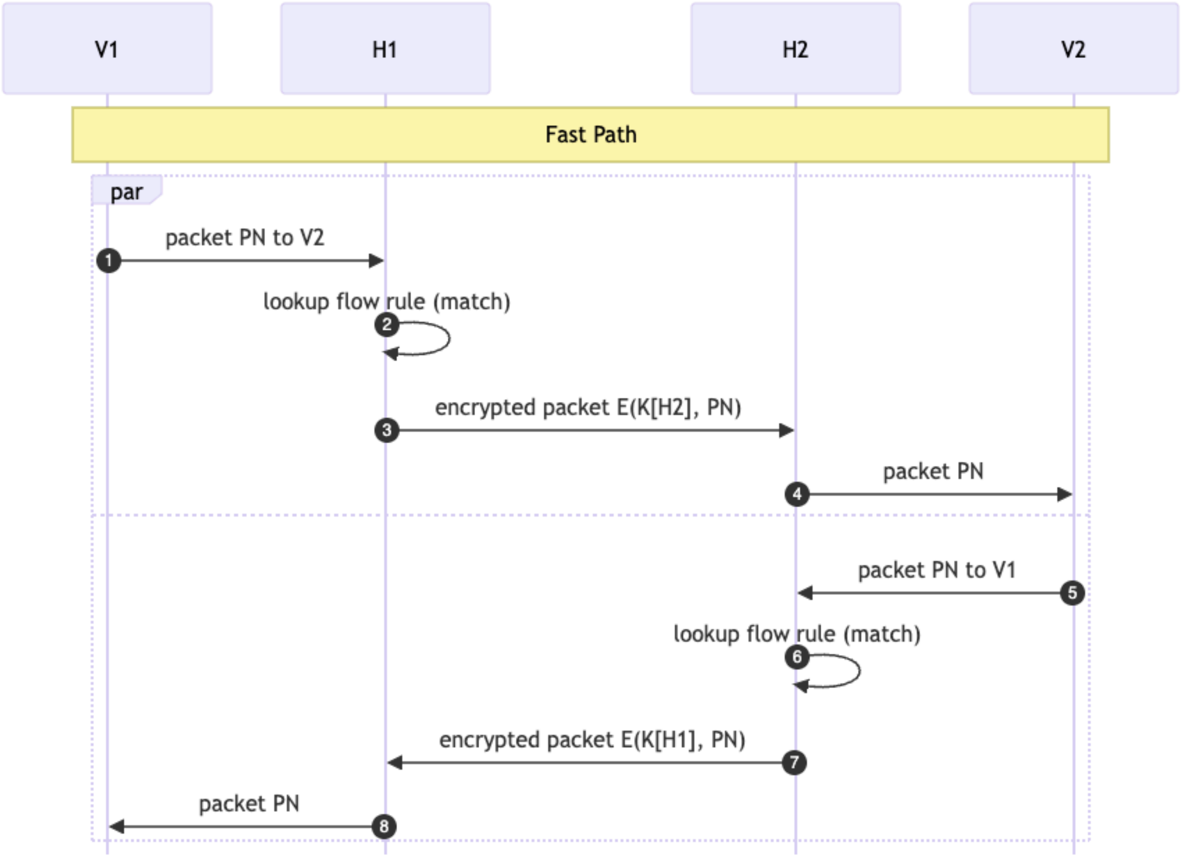
15.15.3.1 Startup vs. On-Demand Tunnel Creation

The default mode of operation is on-demand tunnel creation. That is, when a packet is received from the unsecured port for which the flow pipeline does not have an encryption rule, the packet misses to RSS, where the CPU must decide how to handle the packet. If the destination IP address in the packet belongs to a known peer’s virtual network, the CPU uses gRPC on the OOB network connection to attempt a key exchange with the peer. If the key exchange is successful and a new encryption flow is created successfully, then the packet is then resubmitted to the pipeline, where it is encrypted and sent just as any of the following packets having the same destination IP address.

The following diagram illustrates this sequence (the "Slow Path"), for Virtual Machine V1 which intends to send a packet to Virtual Machine V2. In this case, V1 is hosted on physical host H1 and V2 on physical host H2. The first packet sent (1) results in a miss (2), so the packet is retained (3) while the keys are exchanged in both directions (4-8). Then the pipeline is updated (9) and the original packet is resubmitted (10). From there, the packet follows the same logic as the fast path, below.


V1
H1
H2
V2
Slow Path
1 packet P1 to V2
2 lookup flow rule (miss)
3 store the miss packet P1
4 derive key K[H1]
5 rpc:NewTunnelRequest(K[H1])
6 derive key K[H2]
7 program flow to H1.V1 with K[H1]
8 rpc:NewTunnelResponse(K[H2])
9 program flow to H2.V2 with K[H2]
10 resubmit packet P1
11 encrypted packet E(K[H2], P1)
12 packet P1

Once the tunnel is established, and packets received from the VF (1) match a rule (2) and are encrypted and sent (3-4) without any intervention from the CPU ("Fast Path").


V1
H1
H2
V2
Fast Path
par
1 packet PN to V2
2 lookup flow rule (match)
3 encrypted packet E(K[H2], PN)
4 packet PN
5 packet PN to V1
6 lookup flow rule (match)
7 encrypted packet E(K[H1], PN)
8 packet PN

In the case of on-startup tunnel creation, the application's main thread repeatedly attempts to perform the key exchange for each of the peers specified on the command line until the list is completed. Each peer is connected only once and, if a connection to one peer fails, the loop continues onto the next peer and retries the failed connection after all the others have been attempted.

15.15.3.2 Sampling

The PSP gateway application supports the sample-at-receiver (S) bit in the PSP header. If sampling is enabled, then packets marked with the S bit are mirrored to the RSS queues and logged to the console. In addition, on transmit, the S bit of a random subset of packets (1 out of 2^N for command-line parameter N) is set to 1, and those packets are mirrored to RSS.

To avoid out-of-sequence indication in ROCE traffic, the `--maintain-order` flag can be used. When used, the application creates two mirror resources for egress sampling:

- One for when the sample bit is on – mirror is to RSS
- One for when the sample bit is off – mirror with fwd type drop

> Sampling packets on transmit is currently supported only following encryption. Sampling of egress packets before encryption will be supported in a future release.

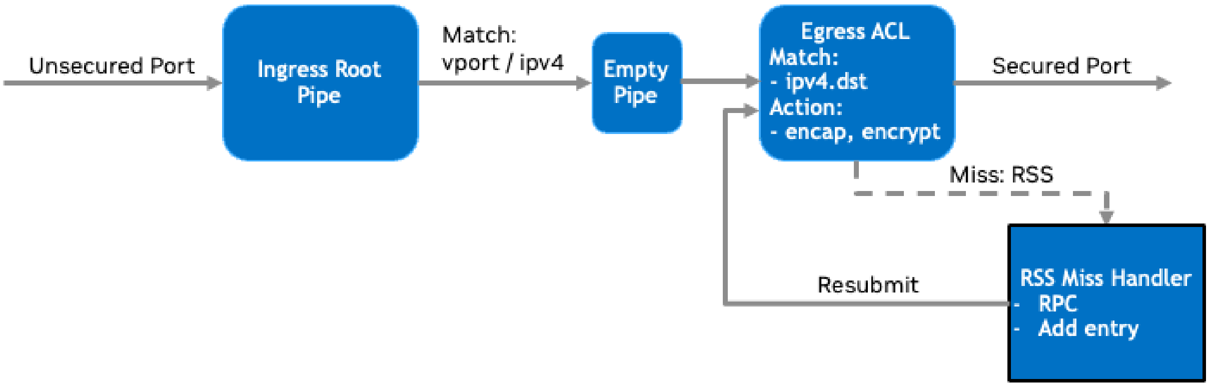
15.15.3.3 Pipelines

15.15.3.3.1 Host-to-Network Flows

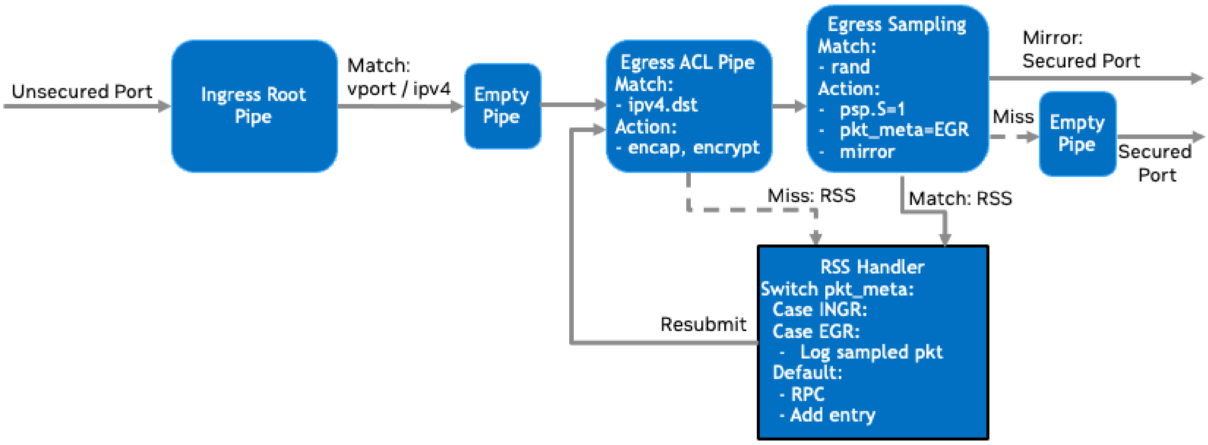
Traffic sent from the local, unsecured port (host-to-net) without sampling enabled travels through the pipeline as shown in the diagrams that follow. Note that the Ingress Root Pipe is the first destination for packets arriving from either the VF or the secured uplink port. However, the Egress ACL pipe is the first destination for packets sent via `tx_burst` on the PF (in the switch model's expert mode).

The Empty Pipe is a vestigial transition from the Default Domain, in which the Ingress Root Pipe is created, to the Secure Egress Domain, where the Egress ACL pipe performs encryption.

> ⚠ This pipe may be removed in a future release.



If sampling is enabled, the host-to-net pipeline is modified as shown in the following:



Here, an Egress Sampling Pipe is added between the Egress ACL Pipe and the Secured Port. It performs a match of the `random` metadata, masked according to command-line parameters, and then:

- On match, the following actions occur:
    a. Packet modifications:
        i. The S bit in the PSP header is set to `true`.

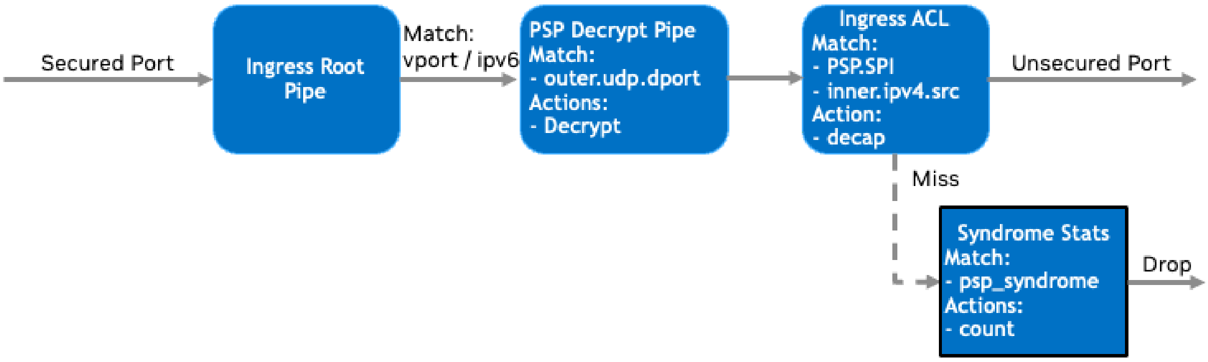
        ii. The `pkt_meta` field is set to a sentinel value to indicate to CPU software why the packet was sent to RSS.
    b. The original packet is forwarded to RSS.
    c. The mirror action forwards the packet to the secured port.
- On miss, the following actions occur:
    a. No packet modifications are made.
    b. The packet is forwarded to a vestigial pipe which can then forward the packet to the wire.

> A `fwd_miss` cannot target a port.

> This pipe may be removed in a future release.

15.15.3.3.2 Network-to-Host Flows

When a packet arrives from the secured port, the following flows are executed.

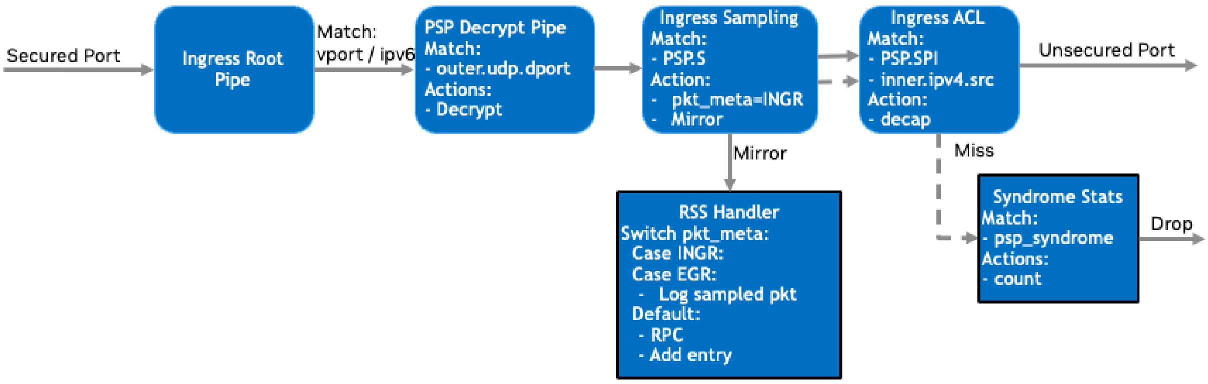


As before, the Ingress Root Pipe is the first destination and, here, the secured port ID as well as IPv6 outer L3 type are matched for. Matching packets flow to the decryption pipe, which matches the outer UDP port number against 1000, the constant specified in the PSP specification. On match, the packet is decrypted, but not yet de-capped. Then the Ingress ACL pipe checks the following:

- PSP_Syndrome - did the packet decrypt correctly and pass its ICV check?
- PSP SPI and inner IP src address - was this packet encrypted with the key associated with the given source?

If the packet passes the syndrome and ACL check, it is forwarded to the VF. Otherwise, the Syndrome Stats pipe counts the occurrences of the different bits in the PSP Syndrome word.

When sampling is enabled, the Ingress Sampling Pipe is inserted before the ACL. Unlike the Egress Sampling Pipe, no randomness is involved; the match criteria is the sample-on-receive flag in the PSP header. On a match, the incoming packet are mirrored to RSS with `pkt_meta` indicating the reason for forwarding the packet to RSS. On match or miss, the next pipe is the Ingress ACL Pipe.



15.15.3.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Flow

Refer to their respective programming guide for more information.

15.15.4 Compiling the Application

Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

For more information about the applications as well as development and compilation tips, refer to the DOCA Applications page.

The sources of the application can be found under the application's directory: `/opt/mellanox/doca/applications/psp_gateway/`.

15.15.4.1 Prerequisites

- The application relies on the `json-c` open source, requiring the following to be installed:
  - Ubuntu/Debian:

    ```
    $ sudo apt install libjson-c-dev
    ```

  - CentOS/RHEL:

    ```
    $ sudo yum install json-c-devel
    ```

- Installing the `gRPC` open source

15.15.4.2 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> `doca_psp_gateway` is created under `/tmp/build/psp_gateway/`.

15.15.4.3 Compiling Only the Current Application

To directly build only the PSP Gateway application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_psp_gateway=true
ninja -C /tmp/build
```

> `doca_psp_gateway` is created under `/tmp/build/psp_gateway/`.

Alternatively, users can set the desired flags in the `meson_options.txt` file instead of providing them in the compilation command line:

1. Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt`:
   - Set `enable_all_applications` to `false`
   - Set `enable_psp_gateway` to `true`
2. Run the following compilation commands:

   ```
   cd /opt/mellanox/doca/applications/
   meson /tmp/build
   ninja -C /tmp/build
   ```

   > `doca_psp_gateway` is created under `/tmp/build/psp_gateway/`.

15.15.4.4 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application.

15.15.5 Running the Application

15.15.5.1 Prerequisites

The PSP gateway application is based on DOCA Flow. Therefore, the user is required to allocate huge pages:

```
echo '2048' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

On some OSs (e.g., RockyLinux, OpenEuler, CentOS 8.2), the default huge page size on the BlueField (and Arm hosts) is larger than 2MB, and is often 512MB instead. The user can find out the size of the huge pages using the following command:

```
$ grep -i huge /proc/meminfo

AnonHugePages:         0 kB
ShmemHugePages:        0 kB
FileHugePages:         0 kB
HugePages_Total:       4
HugePages_Free:        4
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:     524288 kB
Hugetlb:         6291456 kB
```

Given that the guiding principle is to allocate 4GB of RAM, in such cases instead of allocating 2048 pages, the user should allocate the matching amount (8 pages):

```
echo '8' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-524288kB/nr_hugepages
```

15.15.5.2 Application Execution

The PSP Gateway application is provided in source form. Therefore, a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_psp_gateway [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags from an input json file

Program Flags:
  -p, --pci-addr                    PCI BDF of the device in BB:DD.F format (required)
  -r, --repr                        Device representor list in vf[x-y]pf[x-y] format (required)
  -m, --core-mask                   EAL Core Mask
  --decap-dmac                      mac_dst addr of the decapped packets (cannot use w/ vf-name)
  --local-virt-ip                   Local IP addr of VF (cannot use w/ vf-name)
  -d, --vf-name                     Name of the virtual function device / unsecured port
                                    (Host only. Automatically detects MAC/VIP. Requires IPv4 addr bound to
VF.)
  -n, --nexthop-dmac                next-hop mac_dst addr of the encapped packets
  -s, --svc-addr                    Service address of locally running gRPC server; port number optional
  -t, --tunnel                      Remote host tunnel(s), formatted 'svc-ip:virt-ip'
  -f, --tunnels-file                Specifies the location of the tunnels-file. Format: rpc-addr:virt-
addr,virt-addr,...
  -c, --cookie                      Enable use of PSP virtualization cookies
  -a, --disable-ingress-acl         Allows any ingress packet that successfully decrypts
  --sample-rate                     Sets the log2 sample rate: 0: disabled, 1: 50%, ... 16: 1.5e-3%
  -x, --max-tunnels                 Specify the max number of PSP tunnels
```

```
    -o, --crypt-offset                Specify the PSP crypt offset
    --psp-version                     Specify the PSP version for outgoing connections (0 or 1)
    -z, --static-tunnels              Create tunnels at startup
    -k, --debug-keys                  Enable debug keys
    --stat-print                      Enable printing statistics
    --perf-print                      Enable printing performance metrics (key-gen, insertion, all)
    --show-rss-rx-packets             Show RSS rx packets
    --outer-ip-type                   outer IP type
```

2. This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
./doca_psp_gateway -h
```

> For additional information, refer to section "Command Line Flags".

3. CLI example for running the application on the BlueField or host:

```
./doca_psp_gateway -p 03:00.0 -r vf0pf0 -d ens2f0v0 -t 10.1.1.55:192.168.1.55
```

   - The PCIe address identifier (`-p` flag) should match the addresses of the desired PCIe device
   - The `-d` flag indicates the MAC address that should be applied to incoming packets upon decap. It should match the MAC address of the virtual function specified by the `-r` argument.
   - The `-t` flag indicates the mapping of the virtual IP address `192.168.x.y` to an out-of-bounds network address `10.1.1.55`

4. The application also supports a JSON-based deployment mode, in which all command-line arguments are provided through a JSON file:

```
./doca_psp_gateway --json [json_file]
```

For example:

```
./doca_psp_gateway --json psp_gateway_params.json
```

> Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the PCIe addresses necessary for the deployment.

15.15.5.3 Command Line Flags

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
General flags	`h`	`help`	Prints a help synopsis	N/A
	`v`	`version`	Prints program version information	N/A

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
	`l`	`log-level`	Set the log level for the application: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70 (requires compilation with `TRACE` log level support)	`"log-level": 60`
	N/A	`sdk-log-level`	Sets the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70	`"sdk-log-level": 40`
	`j`	`json`	Parse all command flags from an input JSON file	N/A
Program flags	`p`	`pci-addr`	PCIe BDF of the device in `BB:DD.F` format	`"p": "03:00.0"`
	`r`	`repr`	Device representor list in `vf[x-y]pf[x-y]` format	`"r": "vf0pf0"`
	`m`	`core-mask`	EAL core mask	`"m": "0xf"`
	N/A	`decap-dmac`	`mac_dst` address of the decapped packets (cannot be used with `vf-name`)	`"dcap-dmac": "11:22:33:44:55::66"`
	N/A	`local-virt-ip`	Local IP address of VF (cannot be used with `vf-name`)	`"local-virt-ip": "192,168.1.100"`
	N/A	`vf-name`	Name of the VF device/unsecured port (Host only. Automatically detects MAC/VIP. Requires IPv4 address bound to VF.)	`"vf-name": "ens2f0v0"`
	`n`	`nexthop-dmac`	Next-hop `mac_dst` address of the encapped packets	`"nexthop-dmac": "77:88:99:aa:bb:cc"`

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
	`s`	`svc-addr`	Service address of locally running gRPC server; port number optional	`"svc-addr": "10.1.1.50"`
	`t`	`tunnel`	Remote host tunnel(s), formatted `rpc-addr:virt-addr`	`"tunnel": "10.1.1.55:192.168.1.100"`
	`f`	`tunnels-file`	Specifies the location of the tunnels-file. Format: `rpc-addr:virt-addr,virt-addr,...`	`"tunnels-file": "tunnels.txt"`
	`c`	`cookie`	Enable use of PSP virtualization cookies	`"cookie": true`
	`a`	`disable-ingress-acl`	Allows any ingress packet that successfully decrypts	`"disable-ingress-acl": true`
	N/A	`sample-rate`	Sets the log2 sample rate: - 0 - disabled, - 1 - 50%, ... - 16 - 1.5e-3%	`"sample-rate": 16`
	`x`	`max-tunnels`	Specify the max number of PSP tunnels	`"max-tunnels": 4096`
	`o`	`crypt-offset`	Specify the PSP crypt offset	`"crypt-offset": 7`
	N/A	`psp-version`	Specify the PSP version for outgoing connections (`0` or `1`)	`"psp-version": 0`
	`z`	`static-tunnels`	Create tunnels at startup using the given local IP address	`"static-tunnels": "192.168.1.99"`
	`k`	`debug-keys`	Enable debug keys	`"debug-keys": true`
	N/A	`outer-ip-type`	Outer IP tunnel type (`ipv4` or `ipv6`). When not specified, default is `ipv6` .	`"outer-ip-type": ipv4`

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

15.15.5.4 Tunnel Mappings File

A text file which maps an OOB network address to a list of virtual IP addresses behind that physical address can be specified on the command line. The format is as follows:

```
# (Comments are allowed)
# Format:
# svc-oob-ip-addr:virt-addr,virt-addr,...
# Specify a service address of 10.1.1.55 which hosts virtual addresses 192.168.1.101 and others.
10.1.1.55:192.168.1.101,192.168.1.102,192.168.1.103,192.168.1.104
# Specify a service address of 10.1.1.56 which hosts virtual addresses 192.168.1.201 and others.
10.1.1.56:192.168.1.201,192.168.1.202,192.168.1.203,192.168.1.204
```

When a packet from the VF does not match any existing flows, this table defines the physical host which should provide the tunnel to the given (virtual) destination.

15.15.5.5 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

15.15.6 Application Code Flow

1. Main loop code flow
    a. Initialize the logger facility.
        i. The standard logger and the SDK logger are created, and the SDK logger default log level is selected.

        ```
        doca_log_backend_create_standard();
        doca_log_backend_create_with_file_sdk(stdout, &sdk_log);
        doca_log_backend_set_sdk_level(sdk_log, DOCA_LOG_LEVEL_WARNING);
        ```

        ii. The signal handler is connected to enable a clean shutdown.

        ```
        signal(SIGINT, signal_handler);
        signal(SIGTERM, signal_handler);
        ```

    b. Parse application arguments. The main function invokes `psp_gw_argp_exec()` , which initializes the arg parser resources and registers DOCA general parameters, and then registers the PSP application-specific parameters. Then the parser is invoked.

    ```
    doca_argp_init();
    psp_gw_register_params();
    doca_argp_start();
    ```

    c. DPDK initialization. Call `rte_eal_init()` to initialize EAL resources with the provided EAL flags for not probing the ports (`-a00:0.0`).

    ```
    rte_eal_init(n_eal_args, (char **)eal_args);
    ```

    d. Initialize devices and ports.
        i. Open DOCA devices with input PCIe addresses/interface names.
        ii. Probe DPDK port from each opened device.

```
open_doca_device_with_pci(...); // not part of doca_flow; see doca/samples/common.c
doca_dpdk_port_probe(...);
```

iii. The MAC and IP addresses of the PF are queried and logged.

```
rte_eth_macaddr_get(...);
doca_devinfo_get_ipv6_addr(...);
DOCA_LOG_INFO("Port %d: Detected PF mac addr: %s, IPv6 addr: %s, total ports: %d", ...);
```

e. Initialize and start DPDK ports. Initialize DPDK ports, including mempool allocation. No hairpin queues are created.

```
dpdk_queues_and_ports_init(); // not part of doca_flow; see doca/applications/common/dpdk_utils.c
```

f. Initialize DOCA Flow objects used by the PSP Gateway application. The DOCA Flow library is initialized with the string `"switch,hws,isolated,expert"`, because it is desirable for the application to act as an intermediary between the uplink physical port and some number of VF representors (switch mode), and `hws` (hardware steering mode) and `isolated` mode are mandatory for switch mode. The optional `expert` flag prevents DOCA Flow from automating certain packet operations and gives more control to the application, as described in the DOCA Flow page.

```
PSP_GatewayFlows psp_flows(&pf_dev, vf_port_id, &app_config);
psp_flows.init();
```

i. Initialize DOCA Flow library.
ii. Start the ports.
iii. Allocate shared resources (PSP crypto objects and Mirror actions).
iv. Create the ingress and egress pipes.

g. Create the gRPC service.

```
PSP_GatewayImpl psp_svc(&app_config, &psp_flows);
```

h. Launch the L-Core threads to handle RSS packets.

```
rte_eal_remote_launch(lcore_pkt_proc_func, &lcore_params, lcore_id);
```

i. Launch the gRPC service.

i. This implementation uses `InsecureServerCredentials`. Update as needed.

```
grpc::ServerBuilder builder;
builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());
builder.RegisterService(&psp_svc);
auto server_instance = builder.BuildAndStart();
```

j. Wait for traffic. If configured to connect at startup, process the list of remaining connections. Then display the flow pipe counters.

```
while (!force_quit) {
    psp_svc.try_connect(remotes_to_connect, local_vf_addr);
...
    psp_flows.show_static_flow_counts();
    psp_svc.show_flow_counts();
}
```

        - Wait in a loop until the user terminates the program.
    k. PSP Gateway cleanup:
        i. Destroy DPDK ports and queues.

        ```
        dpdk_queues_and_ports_fini();
        ```

        ii. DPDK finish.

        ```
        dpdk_fini();
        ```

        Calls `rte_eal_destroy()` to destroy initialized EAL resources.
        iii. Arg parser destroy.

        ```
        doca_argp_destroy()
        ```

2. Miss-packet code flow.
    a. The L-Core launch routine from the main loop pointed to the `lcore_pkt_proc_func` routine.
    b. The `force_quit` flag is polled to respond to the signal handler.

    ```
    while (!*params->force_quit) { ... }
    ```

    c. The `rte_eth_rx_burst` function polls the PF queue for received packets.

    ```
    nb_rx_packets = rte_eth_rx_burst(port_id, queue_id, rx_packets, MAX_RX_BURST_SIZE);
    ```

    d. Inside `handle_packet()` , the packet metadata is inspected to detect whether this packet is sampled on ingress, sampled on egress, or a miss packet.

    ```
    uint32_t pkt_meta = rte_flow_dynf_metadata_get(packet);
    ```

        i. Sampled packets are simply logged using the `rte_pktmbuf_dump` function.
    e. Miss packets are passed to the `handle_miss_packet` method of the gRPC service. This method handles cases where an application attached to the VF wishes to send a packet to another virtual address, but a PSP tunnel must first be established by exchanging SPI and key information between hosts.
    f. The service acts as a gRPC client, and the appropriate server is looked up from the `config->net_config.hosts` vector, which is comprised of hosts passed via the `-t` tunnels arguments or the `-f` tunnels file argument.
    g. Once the client connection exists, the `request_tunnel_to_host` method takes care of invoking the the `RequestTunnelParams` operation defined in the schema.
        - Optionally, this function generates a corresponding set of tunnel parameters appropriate for the server host to send traffic back via `generate_tunnel_params()` .

        ```
        doca_flow_crypto_psp_spi_key_bulk_generate(bulk_key_gen);
        doca_flow_crypto_psp_spi_key_bulk_get(bulk_key_gen, 0, &spi, key);
        doca_flow_crypto_psp_spi_key_wipe(bulk_key_gen, 0);
        ```

h. The RPC operation is invoked, and if successful, `create_tunnel_flow` is called to create the egress flow:

```
status = stub->RequestTunnelParams(&context, request, &response);
```

i. The `create_tunnel_flow` method translates the resulting Protobuf objects to application-specific data structures and passes them to the `add_encrypt_entry` method of the flows object. Here, the PSP SPI and key are programmed into an available `crypto_id` index as follows.

> ⚠ SPI and `crypto_id` are two independent concepts:
>
> - The SPI value in the PSP packet header indicates to the receiver which key was used by the sender to encrypt the data. Each receiver computes an SPI and key to provide to a sender. Since each receiver is responsible for tracking its next SPI, multiple receivers may provide the same SPI to a sender, so one sender may send the same SPI to multiple different peers. This is allowed, as each of the receiving peers has its own decryption key to handle that SPI.
> - The `crypto_id` acts as an index into the bucket of PSP keys allocated by DOCA Flow. The `doca_flow_shared_resource_cfg()` function writes a given PSP encryption key to a given slot in the bucket of keys in NIC memory. These slots can be overwritten as needed by the application.
> - There is no explicit association between `crypto_id` and SPI. The `doca_flow_shared_resource_cfg()` function writes a key at the slot provided by the `crypto_id` argument, then the flow pipe entry `actions.crypto.crypto_id` references this key, and `actions.crypto_encap.encap_data` includes a PSP header with the desired SPI.

```
struct doca_flow_shared_resource_cfg res_cfg = {};
res_cfg.domain = DOCA_FLOW_PIPE_DOMAIN_SECURE_EGRESS;
res_cfg.psp_cfg.key_cfg.key_type = DOCA_FLOW_CRYPTO_KEY_256;
res_cfg.psp_cfg.key_cfg.key = (uint32_t *)encrypt_key;
doca_flow_shared_resource_cfg(DOCA_FLOW_SHARED_RESOURCE_PSP, session->crypto_id, &res_cfg);
```

j. A flow pipe entry which references the newly programmed PSP encryption key (via its index `crypto.crypto_id`) must be inserted. Additionally, this pipe entry must specify all the outer Ethernet, IP, UDP, and PSP header fields to insert.

```
format_encap_data(session, actions.crypto_encap.encap_data);
actions.crypto.action_type = DOCA_FLOW_CRYPTO_ACTION_ENCRYPT;
actions.crypto.resource_type = DOCA_FLOW_CRYPTO_RESOURCE_PSP;
actions.crypto.crypto_id = session->crypto_id;
...

doca_flow_pipe_add_entry(pipe_queue, pipe, match, actions, mon, fwd, flags, &status, entry);
...

doca_flow_entries_process(port, 0, DEFAULT_TIMEOUT_US, num_of_entries);
```

k. The original packet received via `rte_ethdev_rx_burst` is sent back through the newly updated pipelines via `rte_ethdev_tx_burst`. Since the `port_id` argument is that of the PF, and since DOCA Flow has been initialized in `expert` mode, the packet is transferred to the root of the egress domain (the "empty pipe" before `egress_acl_pipe`).

```
nsent = rte_eth_tx_burst(port_id, queue_id, &packet, 1);
```

3. Tunnel parameter request handling
    a. The gRPC service provided by the PSP Gateway implements the `RequestTunnelParams` operation referenced above. A client uses this operation to request an SPI and key to encrypt traffic to send to the server's NIC device. The request indicates the virtual remote address for which the tunnel will be created.
    b. This operation begins by generating a new SPI and key inside `generate_tunnel_params()` as described previously.
    c. The operation creates an ACL entry permitting the new SPI and the remote virtual address using the `add_ingress_acl_entry` method of the Flows object.

```
doca_flow_match match = {};
match.parser_meta.psp_syndrome = 0;
match.tun.type = DOCA_FLOW_TUN_PSP;
match.tun.psp.spi = RTE_BE32(session->spi_ingress);
match.inner.l3_type = DOCA_FLOW_L3_TYPE_IP4;
match.inner.ip4.src_ip = session->src_vip;
...

doca_flow_pipe_add_entry(pipe_queue, pipe, match, actions, mon, fwd, flags, &status, entry);
...

doca_flow_entries_process(port, 0, DEFAULT_TIMEOUT_US, num_of_entries);
```

    d. If the request included parameters for traffic in the reverse direction (traffic to encrypt and send to the client), these parameters are translated and passed to the Flows object by calling `create_tunnel_flow` described above.

15.15.6.1 References

- PSP Security Protocol Specification
- Google's Open-Source PSP tools
- Google Remote Procedure Calls library

15.16 NVIDIA DOCA Secure Channel Application Guide

This guide provides a secure channel implementation on top of NVIDIA® BlueField® DPU.

15.16.1 Introduction

The DOCA Secure Channel reference application leverages the DOCA Comch API which creates a secure, network independent communication channel between the host and the NVIDIA BlueField DPU.

Comch allows the host to control services on the DPU, activate certain offloads, or exchange messages using client-server/producer-consumer framework.

The client (host) side can communicate only with one server at a time while the server side is able to communicate with multiple clients.

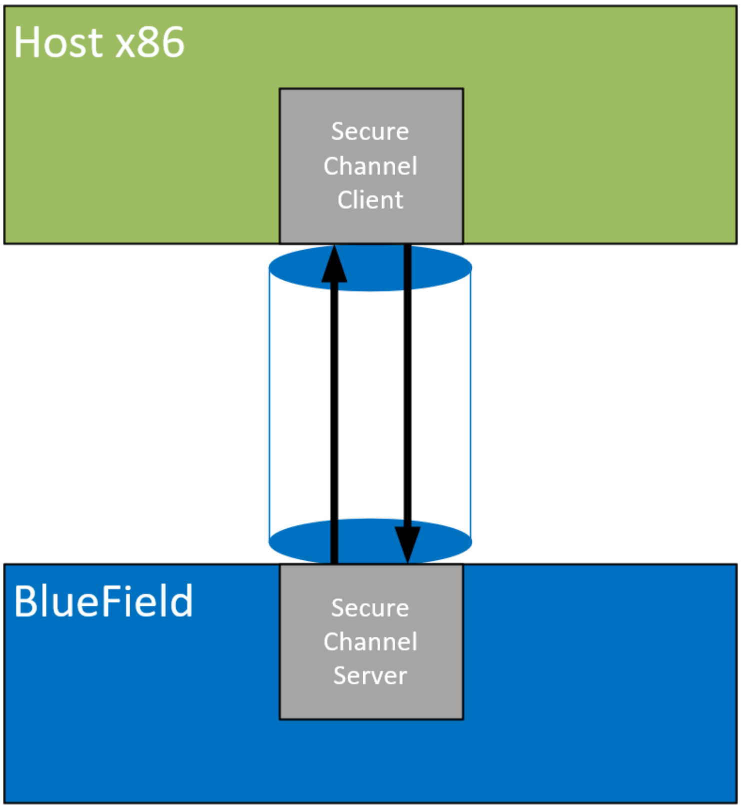
The API allows communication between any PF/VF/SF on the host to the server located on the DPU.

Once a client-server connection is established, multiple producer-consumer instances can be spawned to transfer large amount of data in a first-in-first-out (FIFO) style queue.

Secure channel allows the user to select the message size and amount to be exchanged between the client and the server to simulate heavy load on the channel.

15.16.2 System Design

A secure channel application runs on client mode (host) and server mode (DPU). Once a channel is open, messages can flow from both sides.



15.16.3 Application Architecture

The secure channel application runs on top of the DOCA Comch API.

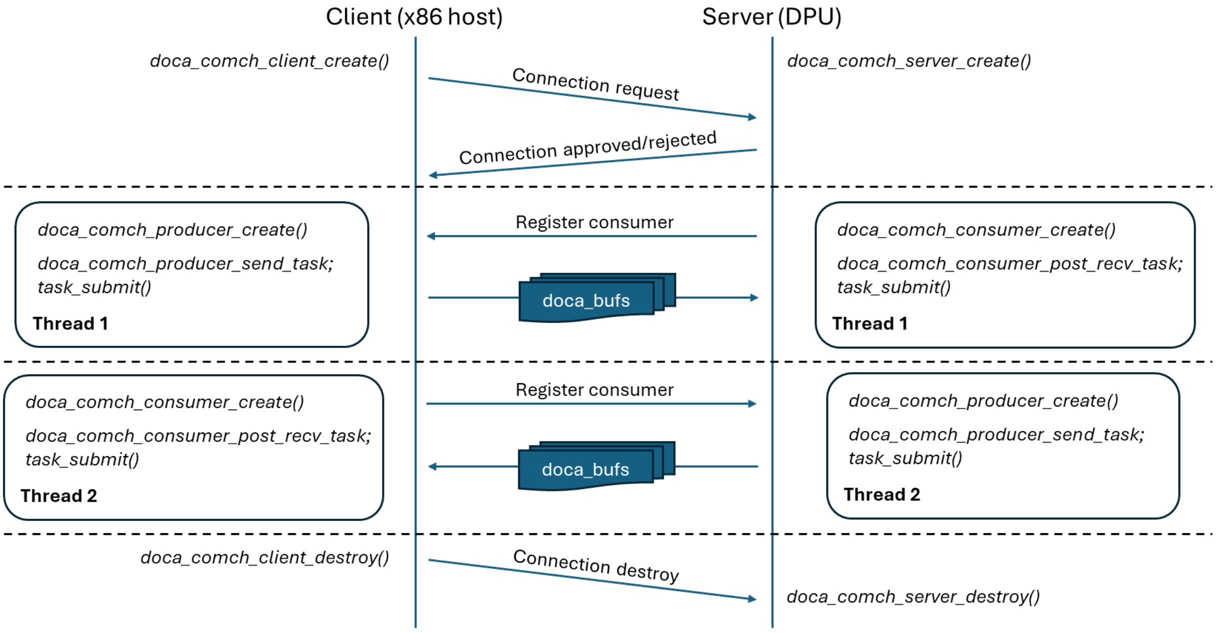
The application begins by establishing a client-server connection between host (client) and DPU (server). Once this connection is established, the application on both sides spawns two new threads:

- Consumer – The consumer thread registers a number of `post_recv` buffers with the opposite end of the connection
- Producer – The producer thread, after it detects the consumer, populates these `post_recv` buffers with data that can then be read by the consumer

Once the consumer has processed a received buffer, the buffer is reposted (via task submit) to again be populated by the opposite producer.

When the producer has sent the requested number of messages, and the consumer has received the same amount, both threads exit and the main client-server connection is destroyed.

The flow is described in the following diagram:


Client (x86 host)
Server (DPU)
doca_comch_client_create()
doca_comch_server_create()
Connection request
Connection approved/rejected
doca_comch_producer_create()
doca_comch_producer_send_task; task_submit()
Thread 1
Register consumer
doca_bufs
doca_comch_consumer_create()
doca_comch_consumer_post_recv_task; task_submit()
Thread 1
doca_comch_consumer_create()
doca_comch_consumer_post_recv_task; task_submit()
Thread 2
Register consumer
doca_bufs
doca_comch_producer_create()
doca_comch_producer_send_task; task_submit()
Thread 2
doca_comch_client_destroy()
Connection destroy
doca_comch_server_destroy()

15.16.4 DOCA Libraries

This application leverages the following DOCA library:

- DOCA Comch

Refer to its respective programming guide for more information.

15.16.5 Compiling the Application

Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

For more information about the applications as well as development and compilation tips, refer to the DOCA Applications page.

The sources of the application can be found under the application's directory: `/opt/mellanox/doca/applications/secure_channel/`.

15.16.5.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

`doca_secure_channel` is created under `/tmp/build/secure_channel/`.

15.16.5.2 Compiling Only the Current Application

To directly build only the secure channel application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_secure_channel=true
ninja -C /tmp/build
```

`doca_secure_channel` is created under `/tmp/build/secure_channel/`.

Alternatively, users can set the desired flags in the `meson_options.txt` file instead of providing them in the compilation command line:

1. Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt`:
   - Set `enable_all_applications` to `false`
   - Set `enable_secure_channel` to `true`
2. Run the following compilation commands:

   ```
   cd /opt/mellanox/doca/applications/
   meson /tmp/build
   ninja -C /tmp/build
   ```

   `doca_secure_channel` is created under `/tmp/build/secure_channel/`.

15.16.5.3 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application.

15.16.6 Running the Application

15.16.6.1 Application Execution

The secure channel application is provided in source form. Therefore, a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_secure_channel [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICA
L, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags from an input json file

Program Flags:
  -s, --msg-size                    Message size to be sent
  -n, --num-msgs                    Number of messages to be sent
  -p, --pci-addr                    DOCA Comm Channel device PCI address
  -r, --rep-pci                     DOCA Comm Channel device representor PCI address (needed only on DPU)
```

ⓘ This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
./doca_secure_channel -h
```

ⓘ For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the BlueField:

```
./doca_secure_channel -s 256 -n 10 -p 03:00.0 -r 3b:00.0
```

⚠ Both the DOCA Comch device PCIe address (`03:00.0`) and the DOCA Comch device representor PCIe address (`3b:00.0`) should match the addresses of the desired PCIe devices.

3. CLI example for running the application on the host:

```
./doca_secure_channel -s 256 -n 10 -p 3b:00.0
```

⚠ The DOCA Comch device PCIe address (`3b:00.0`) should match the address of the desired PCIe device.

4. The application also supports a JSON-based deployment mode, in which all command-line arguments are provided through a JSON file:

```
./doca_secure_channel --json [json_file]
```

For example:

```
./doca_secure_channel --json ./sc_params.json
```

⚠ Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the PCIe addresses necessary for the deployment.

15.16.6.2 Command Line Flags

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
General flags	`h`	`help`	Prints a help synopsis	N/A
	`v`	`version`	Prints program version information	N/A
	`l`	`log-level`	Set the log level for the application: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70 (requires compilation with `TRACE` log level support)	`"log-level": 60`
	N/A	`sdk-log-level`	Sets the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70	`"sdk-log-level": 40`
	`j`	`json`	Parse all command flags from an input JSON file	N/A
Program flags	`s`	`msg-size`	Message size in bytes ⚠ This is a mandatory flag.	`"msg-size": 128`
	`n`	`num-msgs`	Number of messages to send on both sides ⚠ This is a mandatory flag.	`"num-msgs": 256`
	`p`	`pci-addr`	DOCA Comch device PCIe address ⚠ This is a mandatory flag.	`"pci-addr": 03:00.1`

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
	`r`	`rep-pci`	DOCA Comch device representor PCIe address ⚠ This is a mandatory flag only on the DPU.	`"rep-pci": b1:00.1`

ⓘ Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

15.16.6.3 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

15.16.7 Application Code Flow

1. Parse application argument.
   a. Initialize the arg parser resources and register DOCA general parameters.

      ```
      doca_argp_init();
      ```

   b. Register secure channel application parameters.

      ```
      register_secure_channel_params();
      ```

   c. Parse application parameters:

      ```
      doca_argp_start();
      ```

   d. Establish the client-server control path connection:

      ```
      comch_utils_fast_path_init();
      ```

      i. Create a `doca_comch_client` or `doca_comch_server` depending on where the application is run (x86 host or DPU Arm cores).
      ii. Register a callback which is triggered when new consumers are created on the opposite end.
      iii. Progress the connection until it becomes established.
2. Run main logic.

   ```
   sc_start();
   ```

a. Send a message on the control channel telling the opposite end how many fastpath messages it intends to send and their length.
b. Wait to receive a similar message from the opposite side.
c. Start a producer thread:
    i. Create and starts a `doca_comch_producer`.
    ii. Wait until a consumer has been registered from the opposite end of the control channel.
    iii. Create a `doca_buf` of input message length to send repeatedly.
    iv. Submit the max number of `doca_comch_producer_task_send` tasks each containing the `doca_buf`.
    v. Resubmit each task from its completion callback until the requested number of tasks are sent.
    vi. Destroy producer and ends thread.
d. Start a consumer thread
    i. Create and starts a `doca_comch_consumer`.
    ii. Progress until consumer has been fully registered with the control channel connection.
    iii. Create `doca_buf`s of negotiated size to receive data on.
    iv. Submit the max number of `doca_comch_consumer_task_post_recv` tasks with each of the allocated `doca_buf`s.
    v. Resubmit each `post_recv` buffer from its callback when it has been populated by a producer.
    vi. Destroy consumer and thread when it has received the communicated number of fastpath messages.
e. Sends (DPU) or waits (host) on a message to indicate that all fastpath messages have completed and the Comch control connection can be destroyed.

15.16.8 References

- `/opt/mellanox/doca/applications/secure_channel/`
- `/opt/mellanox/doca/applications/secure_channel/sc_params.json`

15.17 NVIDIA DOCA Simple Forward VNF Application Guide

This guide provides a Simple Forward implementation on top of NVIDIA® BlueField® DPU.

15.17.1 Introduction

Simple forward is a forwarding application that leverages the DOCA Flow API to take either VXLAN, GRE, or GTP traffic from a single RX port and transmits it on a single TX port.

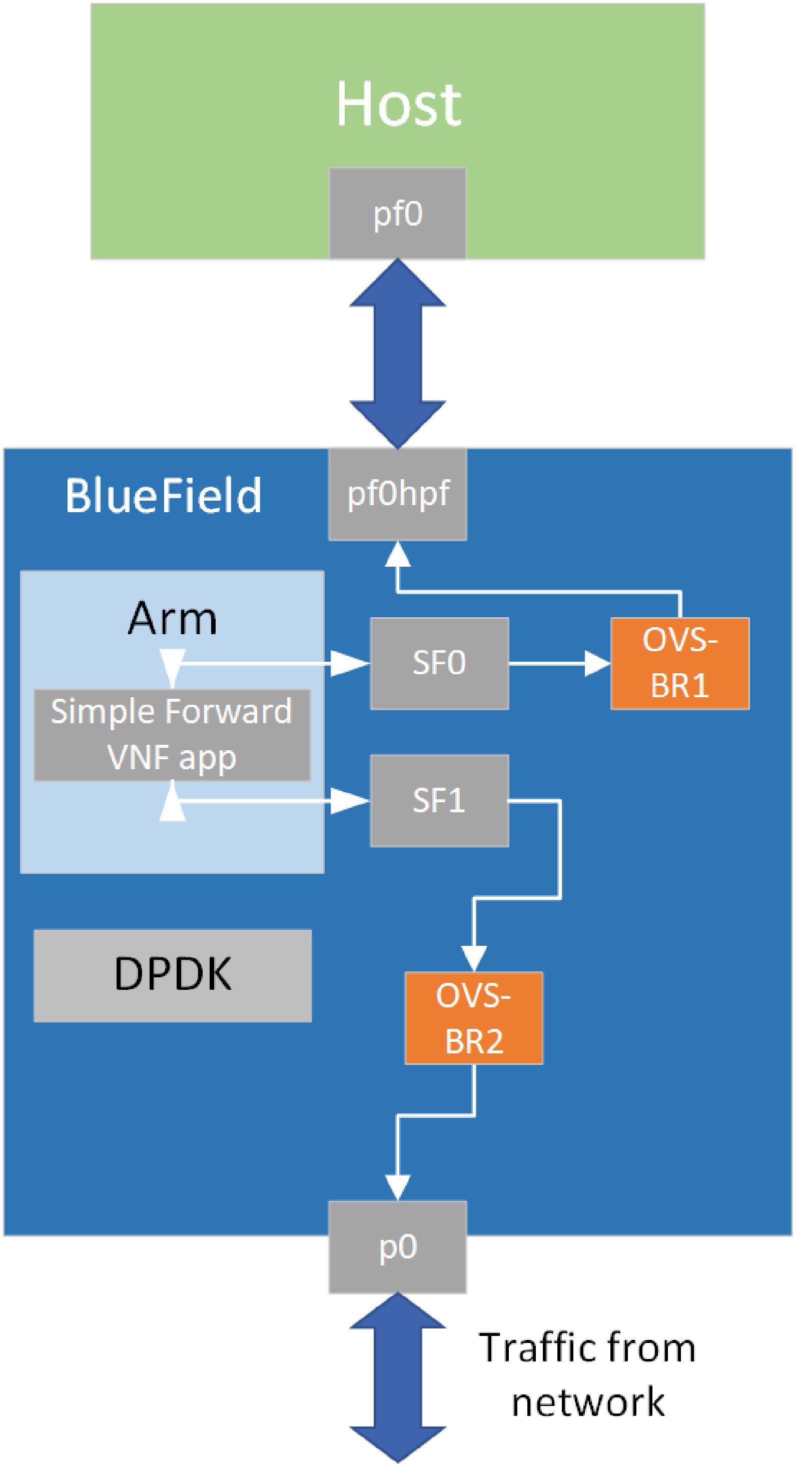
For every packet received on an RX queue on a given port, DOCA Simple Forward checks the packet's key, which consists of a 5-tuple. If it finds that the packet matches an existing flow, then it does not create a new one. Otherwise, a new flow is created with a FORWARDING component. Finally, the packet is forwarded to the TX queue of the egress port if the "rx-only" mode is not set.

The FORWARDING component type depends on the flags delivered when running the application. For example, if the `hairpinq` flag is provided, then the FORWARDING component would be hairpin. Otherwise, it would be RSS'd to software, and hence every VXLAN, GTP, or GRE packet would be received on RX queues.

Simple forward should be run with dual ports. By using a traffic generator, the RX port receives the VXLAN, GRE, or GTP packets and forwarding forwards them back to the traffic generator.

15.17.2 System Design

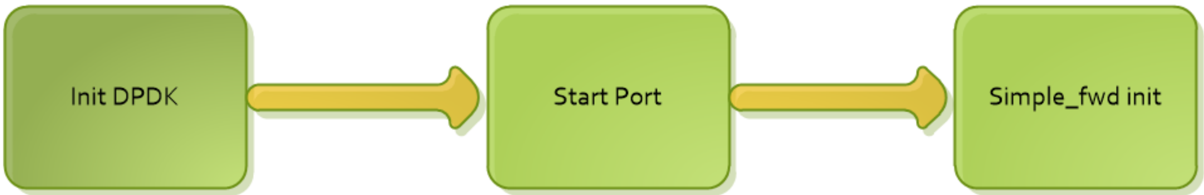
The following diagram illustrates simple forward's packet flows. It receives traffic coming from the wire and passes it to the other port.



15.17.3 Application Architecture

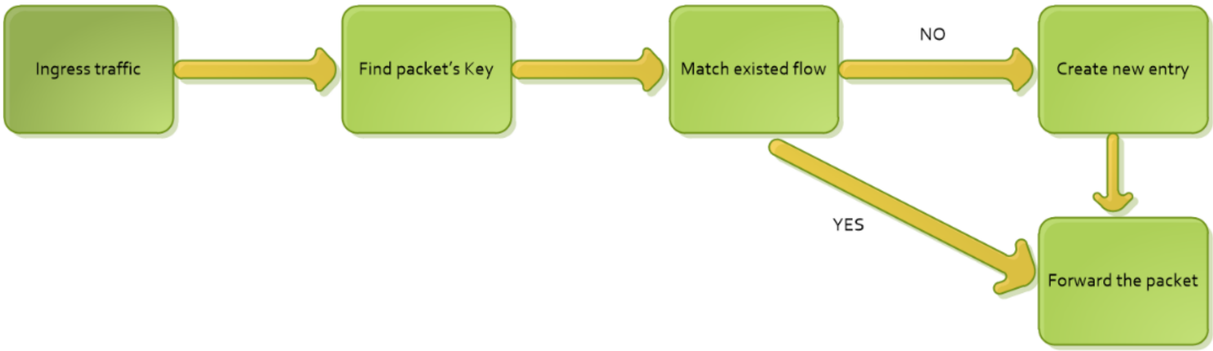
Simple forward first initializes DPDK, after which the application handles the incoming packets.

The following diagram illustrates the initialization process.



1. `Init_DPDK` - EAL init, parse argument from command line and register signal.
2. Start port - `mbuf_create` , `dev_configure` , rx/tx/hairpin queue setup and start the port.
3. `Simple_fwd INIT` - create flow tables, build default forward pipes.

The following diagram illustrates how to process the packet.



1. Based on the packet's info, find the key values (e.g. src/dst IP, src/dst port, etc).
2. Traverse the inner flow tables, check if the keys exist or not.
    - If yes, update inner counter
    - If no, a new flow table is added to the DPU
3. Forward the packet to the other port.

15.17.4 DOCA Libraries

This application leverages the following DOCA library:

- DOCA Flow

Refer to its respective programming guide for more information.

15.17.5 Compiling the Application

ⓘ Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

✓ For more information about the applications as well as development and compilation tips, refer to the DOCA Applications page.

The sources of the application can be found under the application's directory: `/opt/mellanox/doca/applications/simple_fwd_vnf/` .

15.17.5.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> `doca_simple_fwd_vnf` is created under `/tmp/build/simple_fwd_vnf/` .

15.17.5.2 Compiling Simple Forward Application Only

To directly build only the simple forward application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_simple_fwd_vnf=true
ninja -C /tmp/build
```

> `doca_simple_fwd_vnf` is created under `/tmp/build/simple_fwd_vnf/` .

Alternatively, users can set the desired flags in the `meson_options.txt` file instead of providing them in the compilation command line:

1. Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt` :
   - Set `enable_all_applications` to `false`
   - Set `enable_simple_fwd_vnf` to `true`
2. Run the following compilation commands:

   ```
   cd /opt/mellanox/doca/applications/
   meson /tmp/build
   ninja -C /tmp/build
   ```

   > `doca_simple_fwd_vnf` is created under `/tmp/build/simple_fwd_vnf/` .

15.17.5.3 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application.

15.17.6 Running the Application

15.17.6.1 Prerequisites

1. A FLEX profile number should be manually set to 3 on the system for the application to build the GRE, standard VXLAN and GRE pipes.
   a. Set FLEX profile number to 3 from the DPU.

   ```
   sudo mlxconfig -d <pcie_address> s FLEX_PARSER_PROFILE_ENABLE=3
   ```

   b. Perform a BlueField system reboot for the `mlxconfig` settings to take effect.

   > Resetting the firmware can be done from the BlueField as well. For more information, refer to step 3.b of the "Upgrading Firmware" section of the NVIDIA DOCA Installation Guide for Linux.

2. The Simple Forward application is based on DOCA Flow. Therefore, the user is required to allocate huge pages.

   ```
   echo '2048' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
   ```

   On some operating systems (RockyLinux, OpenEuler, CentOS 8.2) the default huge page size on the DPU (and Arm hosts) is larger than 2MB, and is often 512MB instead. Once can find out the sige of the huge pages using the following command:

   ```
   $ grep -i huge /proc/meminfo

   AnonHugePages:         0 kB
   ShmemHugePages:        0 kB
   FileHugePages:         0 kB
   HugePages_Total:       4
   HugePages_Free:        4
   HugePages_Rsvd:        0
   HugePages_Surp:        0
   Hugepagesize:     524288 kB
   Hugetlb:         6291456 kB
   ```

   Given that the guiding principal is to allocate 4GB of RAM, in such cases instead of allocating 2048 pages, one should allocate the matching amount (8 pages):

   ```
   echo '8' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-524288kB/nr_hugepages
   ```

15.17.6.2 Application Execution

The simple forward application is provided in source form. Therefore, a compilation is required before the application can be executed.

1. Application usage instructions:

   ```
   Usage: doca_simple_forward_vnf [DPDK Flags] -- [DOCA Flags] [Program Flags]

   DOCA Flags:
     -h, --help                          Print a help synopsis
     -v, --version                       Print program version information
     -l, --log-level                     Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
   30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
   ```

```
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICA
L, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags from an input json file

Program Flags:
  -t, --stats-timer <time>          Set interval to dump stats information
  -q, --nr-queues <num>             Set queues number
  -r, --rx-only                     Set rx only
  -o, --hw-offload                  Set PCI address of the RXP engine to use
  -hq, --hairpinq                   Set forwarding to hairpin queue
  -a, --age-thread                  Start thread do aging
```

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
./doca_simple_fwd_vnf -- -h
```

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the BlueField:

```
./doca_simple_fwd_vnf -a auxiliary:mlx5_core.sf.4 -a auxiliary:mlx5_core.sf.5 -- -l 60
```

SFs must be enabled according to the NVIDIA BlueField DPU Scalable Function User Guide.

Before creating SFs on a specific physical port, it is important to verify the encap mode on the respective PF FDB. The default mode is `basic`. To check the encap mode, run:

```
cat /sys/class/net/p0/compat/devlink/encap
```

In this case, disable encap on the PF FDB before creating the SFs by running:

```
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode legacy
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode legacy
echo none > /sys/class/net/p0/compat/devlink/encap
echo none > /sys/class/net/p1/compat/devlink/encap
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode switchdev
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode switchdev
```

If the encap mode is set to `basic` then the application fails upon initialization.

The flag `-a auxiliary:mlx5_core.sf.4 -a auxiliary:mlx5_core.sf.5` is mandatory for proper usage of the application.

a. Modifying this flag results unexpected behavior as only 2 ports are supported.
b. The SF number is arbitrary and configurable.

The SF numbers must match the desired SF devices.

3. CLI example for running the application on the host:

```
./doca_simple_fwd_vnf -a 04:00.3 -a 04:00.4 -- -l 60
```

The device identifiers must match the desired network devices.

For more information, refer to section "Running DOCA Application on Host" in NVIDIA DOCA Virtual Functions User Guide.

4. The application also supports a JSON-based deployment mode, in which all command-line arguments are provided through a JSON file:

```
./doca_simple_fwd_vnf --json [json_file]
```

For example:

```
./doca_simple_fwd_vnf --json ./simple_fwd_params.json
```

Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the PCIe addresses necessary for the deployment.

15.17.6.3 Command Line Flags

Flag Type	Short Flag	Long Flag/ JSON Key	Description	JSON Content

DPDK Flags	`a`	`devices`	Add a PCIe device into the list of devices to probe.	`"devices":`
				`[`
				`    {"device": "sf", "id": "4","sft": true},`
				`    {"device": "sf", "id": "5","sft": true},`
				`]`

General flags	`h`	`help`	Prints a help synopsis	N/A
	`v`	`version`	Prints program version information	N/A
	`l`	`log-level`	Set the log level for the application: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70 (requires compilation with `TRACE` log level support)	`"log-level": 60`

Flag Type	Short Flag	Long Flag/ JSON Key	Description	JSON Content
	N/A	`sdk-log-level`	Sets the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70	`"sdk-log-level": 40`
	`j`	`json`	Parse all command flags from an input JSON file	N/A
Program flags	`t`	`stats-timer`	Set interval to dump stats information.	`"stats-timer": 2`
	`q`	`nr-queues`	Set queues number.	`"nr-queues": 4`
	`r`	`rx-only`	Set RX only. When set, the packets will not be sent to the TX queues.	`"rx-only": false`
	`o`	`hw-offload`	Set HW offload of the RXP engine to use.	`"hw-offload": false`
	`hq`	`hairpinq`	Set forwarding to hairpin queue.	`"hairpinq": false`
	`a`	`age-thread`	Start a dedicated thread that handles the aged flows.	`"age-thread": false`

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

15.17.6.4 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

15.17.7 Application Code Flow

1. Parse application argument.
   a. Initialize arg parser resources and register DOCA general parameters.

      ```
      doca_argp_init();
      ```

   b. Register application parameters.

```
    register_simple_fwd_params();
    ```

    c. Parse the arguments.

    ```
    doca_argp_start();
    ```

    i. Parse DPDK flags and invoke handler for calling the `rte_eal_init()` function.
    ii. Parse app parameters.

2. DPDK initialization.

   ```
   dpdk_init();
   ```

   Calls `rte_eal_init()` to initialize EAL resources with the provided EAL flags.

3. DPDK port initialization and start.

   ```
   dpdk_queues_and_ports_init();
   ```

   a. Initialize DPDK ports.
   b. Create mbuf pool using `rte_pktmbuf_pool_create`.
   c. Driver initialization - use `rte_eth_dev_configure` to configure the number of queues.
   d. Rx/Tx queue initialization - use `rte_eth_rx_queue_setup` and `rte_eth_tx_queue_setup` to initialize the queues.
   e. Rx hairpin queue initialization - use `rte_eth_rx_hairpin_queue_setup` to initialize the queues.
   f. Start the port using `rte_eth_dev_start`.

4. Simple forward initialization.

   ```
   simple_fwd_init();
   ```

   a. `simple_fwd_create_ins` - create flow tables using `simple_fwd_ft_create`.
   b. `simple_fwd_init_ports_and_pipes` - initialize DOCA port using `simple_fwd_init_doca_port` and build default pipes for each port.

5. Main loop.

   ```
   simple_fwd_process_pkts();
   ```

   a. Receive packets using `rte_eth_rx_burst` in a loop.
   b. Process packets using `simple_fwd_process_offload`.
   c. Transmit the packets on the other port by calling `rte_eth_tx_burst`. Or free the packet mbuf if `rx_only` is set to `true`.

6. Process packets.

```
simple_fwd_process_offload();
```

   a. Parse the packet's `rte_mbuf` using `simple_fwd_pkt_info`.
   b. Handle the packet using `simple_fwd_handle_packet`. If the packet's key does not match the existed the flow entry, create a new flow entry and PIPE using `simple_fwd_handle_new_flow`. Otherwise, increase the total packet's counter.

7. Simple forward destroy.

   ```
   simple_fwd_destroy();
   ```

   Simple forward close port and clean the flow resources.

8. DPDK ports and queues destruction.

   ```
   dpdk_queues_and_ports_fini();
   ```

9. DPDK finish.

   ```
   dpdk_fini();
   ```

   Calls `rte_eal_destroy()` to destroy initialized EAL resources.

10. Arg parser destroy.

    ```
    doca_argp_destroy();
    ```

    - Free DPDK resources by call `rte_eal_cleanup()` function.

15.17.8 References

- `/opt/mellanox/doca/applications/simple_fwd_vnf/`
- `/opt/mellanox/doca/applications/simple_fwd_vnf/simple_fwd_params.json`

15.18 NVIDIA DOCA Switch Application Guide

This guide provides an example of switch implementation on top of NVIDIA® BlueField® DPU.

15.18.1 Introduction

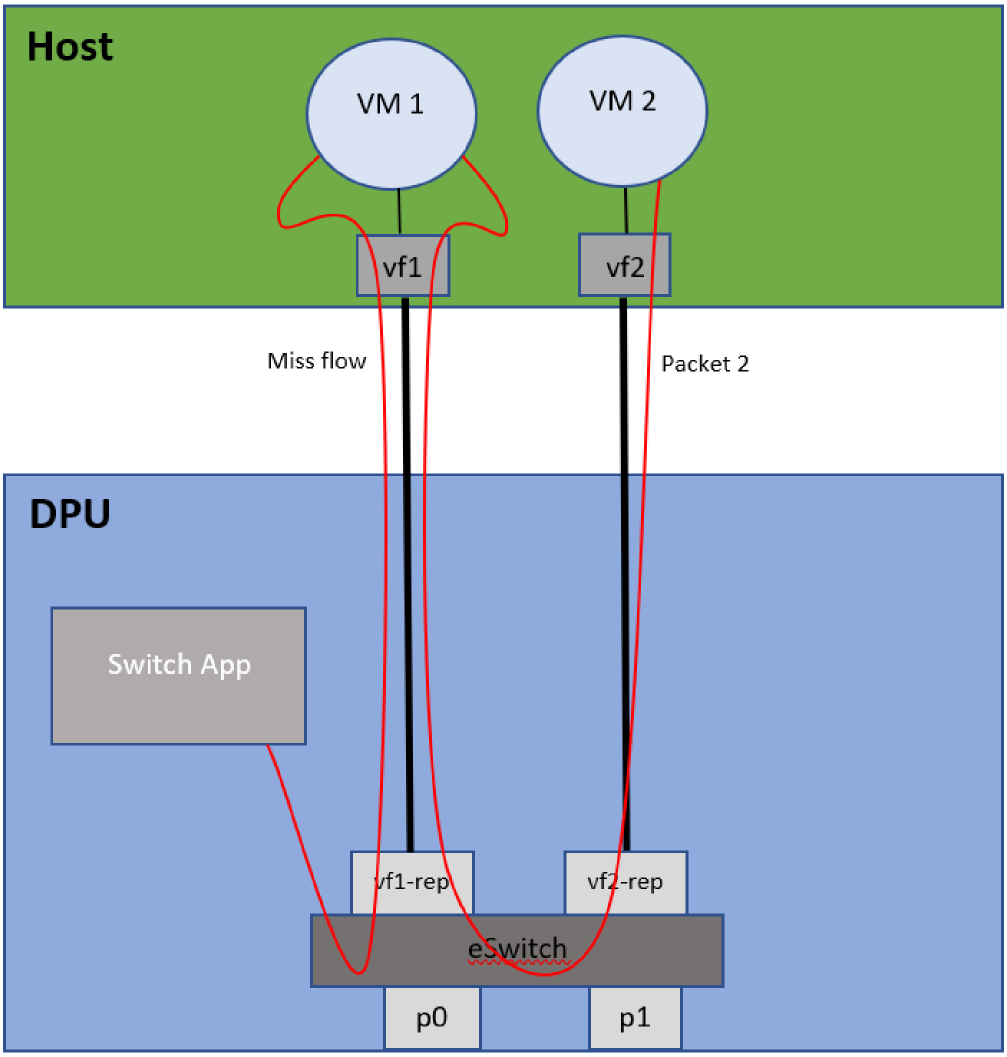
DOCA Switch is a network application that leverages the DPU's hardware capability for internal switching between representor ports on the DPU.

DOCA Switch is based on the DOCA Flow library. As such, it exposes a command line interface which receives DOCA Flow like commands to allow adding rules in real time.

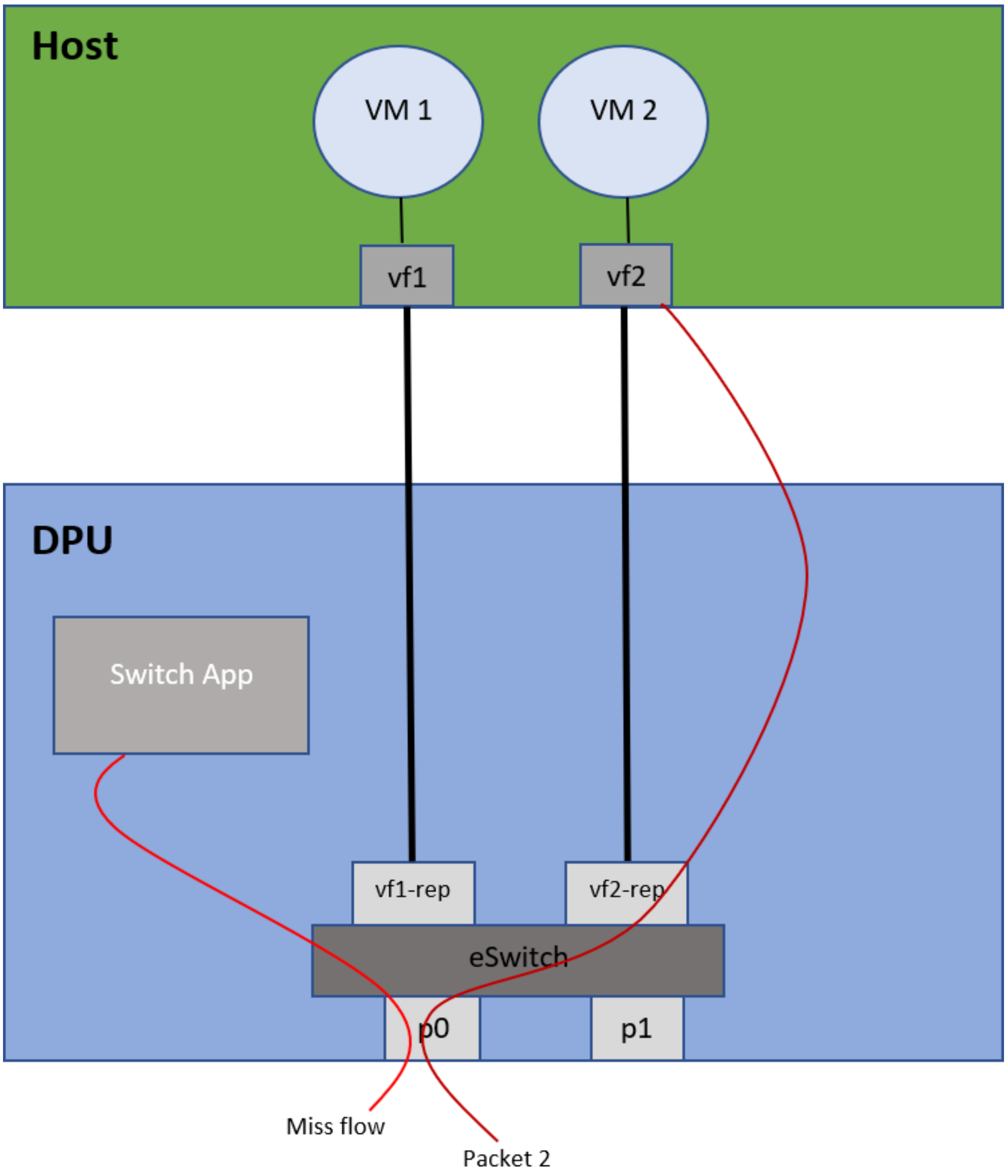
15.18.2 System Design

DOCA Switch is designed to run on the DPU as a standalone application (all network traffic goes directly through it).

Traffic flows between two VMs on the host:



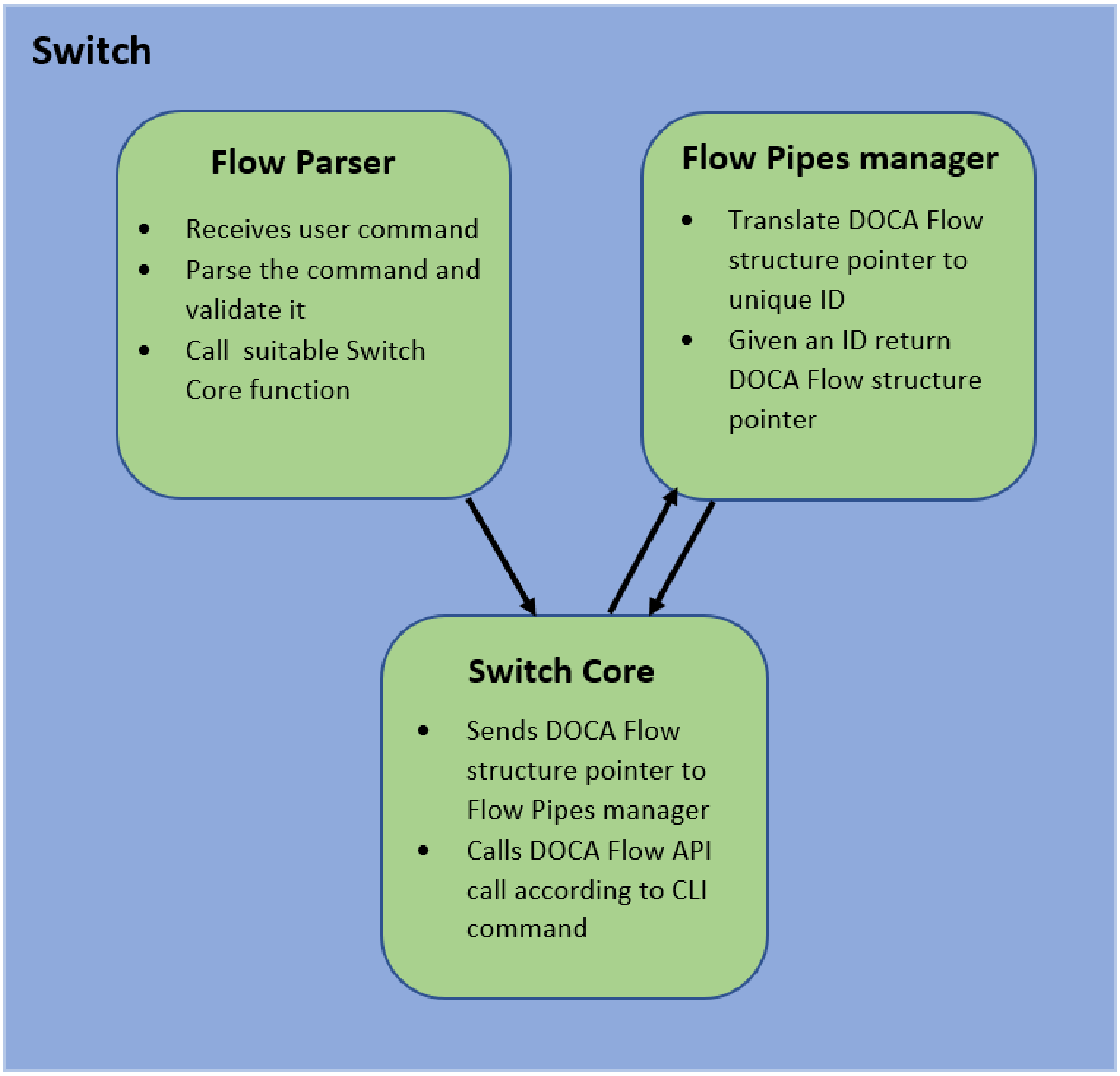
Traffic flow from a physical port to a VM on the host:



15.18.3 Application Architecture

DOCA Switch is based on 3 modules:

- Command line interface – receives pre-defined DOCA Flow-like commands and parses them
- Flow pipes manger – generates a unique identification number for each DOCA Flow structure created
- Switch core – combines all modules together and calls necessary DOCA Flow API


Switch
Flow Parser
Receives user command
Parse the command and validate it
Call suitable Switch Core function
Flow Pipes manager
Translate DOCA Flow structure pointer to unique ID
Given an ID return DOCA Flow structure pointer
Switch Core
Sends DOCA Flow structure pointer to Flow Pipes manager
Calls DOCA Flow API call according to CLI command

Port initialization cannot be made dynamically. All ports must be defined when running the application with standard DPDK flags.

- When adding a pipe or an entry, the user must run commands to create the relevant structs beforehand
- Optional parameters must be specified by the user in the command line; otherwise, `NULL` is used
- After a pipe or an entry is created successfully, the relevant ID is printed for future use

15.18.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Flow

Refer to its respective programming guide for more information.

15.18.5 Compiling the Application

> Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

> For more information about the applications as well as development and compilation tips, refer to the DOCA Applications page.

The sources of the application can be found under the application's directory: `/opt/mellanox/doca/applications/switch/` .

15.18.5.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> `doca_switch` is created under `/tmp/build/switch/` .

15.18.5.2 Recompiling Only the Current Application

To directly build only the switch application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_switch=true
ninja -C /tmp/build
```

> `doca_switch` is created under `/tmp/build/switch/` .

Alternatively, one can set the desired flags in the `meson_options.txt` file instead of providing them in the compilation command line:

1. Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt` :
   - Set `enable_all_applications` to `false`
   - Set `enable_switch` to `true`
2. Run the following compilation commands:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

`doca_switch` is created under `/tmp/build/switch/`.

15.18.5.3 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application.

15.18.6 Running the Application

15.18.6.1 Prerequisites

The switch application is based on DOCA Flow. Therefore, the user is required to allocate huge pages.

```
echo '2048' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

On some operating systems (RockyLinux, OpenEuler, CentOS 8.2) the default huge page size on the DPU (and Arm hosts) is larger than 2MB, and is often 512MB instead. Once can find out the sige of the huge pages using the following command:

```
$ grep -i huge /proc/meminfo

AnonHugePages:         0 kB
ShmemHugePages:        0 kB
FileHugePages:         0 kB
HugePages_Total:       4
HugePages_Free:        4
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:     524288 kB
Hugetlb:         6291456 kB
```

Given that the guiding principal is to allocate 4GB of RAM, in such cases instead of allocating 2048 pages, one should allocate the matching amount (8 pages):

```
echo '8' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-524288kB/nr_hugepages
```

15.18.6.2 Application Execution

The switch application is provided in source form. Therefore, hence a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_switch [DPDK Flags] -- [DOCA Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags from an input json file
```

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
./doca_switch -- -h
```

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the BlueField:

```
./doca_switch -a 03:00.0,representor=[0-2],dv_flow_en=2 -- -l 60
```

`dv_flow_en=2` is necessary to run the application with hardware steering.

The PCIe address (`03:00.0`) should match the address of the desired PCIe device.

15.18.6.3 Command Line Flags

Flag Type	Short Flag	Long Flag	Description	JSON Content
General flags	`h`	`help`	Prints a help synopsis	N/A
	`v`	`version`	Prints program version information	N/A
	`l`	`log-level`	Set the log level for the application: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70 (requires compilation with `TRACE` log level support)	`"log-level": 60`

Flag Type	Short Flag	Long Flag	Description	JSON Content
	N/A	`sdk-log-level`	Sets the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70	`"sdk-log-level": 40`
	`j`	`json`	Parse all command flags from an input JSON file	N/A

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

15.18.6.4 Supported Commands

- `create pipe port_id=[port_id][,<optional_parameters>]`
  Available optional parameters:
  - `name=<pipe-name>`
  - `root_enable=[1|0]`
  - `monitor=[1|0]`
  - `match_mask=[1|0]`
  - `fwd=[1|0]`
  - `fwd_miss=[1|0]`
  - `type=[basic|control]`
- `add entry pipe_id=<pipe_id>,pipe_queue=<pipe_queue>[,<optional_parameters>]`
  Available optional parameters:
  - `monitor=[1|0]`
  - `fwd=[1|0]`
- `add control_pipe entry priority=<priority>,pipe_id=<pipe_id>,pipe_queue=<pipe_queue>[,<optional_parameters>]`
  Available optional parameters:
  - `match_mask=[1|0]`
  - `fwd=[1|0]`
- `destroy pipe pipe_id=<pipe_id>`
- `rm entry pipe_queue=<pipe_queue>,entry_id=[entry_id]`
- `port pipes flush port_id=[port_id]`
- `port pipes dump port_id=[port_id],file=[file_name]`
- `query entry_id=[entry_id]`

- `create [struct] [field=value,…]`
  Struct options: `pipe_match` , `entry_match` , `match_mask` , `actions` , `monitor` , `fwd` , `fwd_miss`
  - Match struct fields:

Fields	Field Options
`flags`	
`port_meta`	
`outer.eth.src_mac`	
`outer.eth.dst_mac`	
`outer.eth.type`	
`outer.vlan_tci`	
`outer.l3_type`	`ipv4` , `ipv6`
`outer.src_ip_addr`	
`outer.dst_ip_addr`	
`outer.l4_type_ext`	`tcp` , `udp` , `gre`
`outer.tcp.flags`	`FIN` , `SYN` , `RST` , `PSH` , `ACK` , `URG` , `ECE` , `CWR`
`outer.tcp_src_port`	
`outer.tcp_dst_port`	
`outer.udp_src_port`	
`outer.udp_dst_port`	
`tun_type`	
`vxlan_tun_id`	
`gre_key`	
`gtp_teid`	
`inner.eth.src_mac`	
`inner.eth.dst_mac`	
`inner.eth.type`	
`inner.vlan_tci`	
`inner.l3_type`	`ipv4` , `ipv6`
`inner.src_ip_addr`	
`inner.dst_ip_addr`	

Fields	Field Options
`inner.l4_type_ext`	`tcp` , `udp`
`inner.tcp.flags`	`FIN` , `SYN` , `RST` , `PSH` , `ACK` , `URG` , `ECE` , `CWR`
`inner.tcp_src_port`	
`inner.tcp_dst_port`	
`inner.udp_src_port`	
`inner.udp_dst_port`	

- Actions struct fields:

Fields	Field Options
`decap`	`true` , `false`
`mod_src_mac`	
`mod_dst_mac`	
`mod_src_ip_type`	`ipv4` , `ipv6`
`mod_src_ip_addr`	
`mod_dst_ip_type`	`ipv4` , `ipv6`
`mod_dst_ip_addr`	
`mod_src_port`	
`mod_dst_port`	
`ttl`	
`has_encap`	`true` , `false`
`encap_src_mac`	
`encap_dst_mac`	
`encap_src_ip_type`	`ipv4` , `ipv6`
`encap_src_ip_addr`	
`encap_dst_ip_type`	`ipv4` , `ipv6`
`encap_dst_ip_addr`	
`encap_tup_type`	`vxlan` , `gtpu` , `gre`
`encap_vxlan-tun_id`	
`encap_gre_key`	
`encap_gtp_teid`	

- FWD struct fields:

Fields	Field Options
`type`	`rss` , `port` , `pipe` , `drop`
`rss_flags`	
`rss_queues`	
`num_of_queues`	
`port_id`	
`next_pipe_id`	

- Monitor struct fields:
  - `flags`
  - `cir`
  - `cbs`
  - `aging`

The physical port number (only one physical port is supported) will always be 0 and all representor ports are numbered from 1 to N where N is the number of representors being used. For example:

- Physical port ID: 0
- VF0 representor port ID: 1
- VF1 representor port ID: 2
- VF2 representor port ID: 3

The following is an example of creating a pipe and adding one entry into it:

```
create fwd type=port,port_id=0xffff
create pipe port_id=0,name=p0_to_vf1,root_enable=1,fwd=1
create fwd type=port,port_id=1
add entry pipe_queue=0,fwd=1,pipe_id=1012
          ....
rm entry pipe_queue=0,entry_id=447
```

1. Pipe is configured on port ID 0 (physical port).
2. Entry is configured to forward all traffic directly into port ID 1 (VF0).
3. When the forwarding rule is no longer needed, the entry is deleted.
4. Ultimately, both entries are deleted, each according to the unique random ID it was given:

15.18.6.5 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

15.18.7 Application Code Flow

1. Parse application argument.
   a. Initialize the arg parser resources and register DOCA general parameters.

```
    doca_argp_init();
    ```

   b. Register application parameters.

    ```
    register_switch_params();
    ```

   c. Parse app parameters.

    ```
    doca_argp_start();
    ```

2. Count total number of ports.

   ```
   switch_ports_count();
   ```

   a. Check how many ports are entered when running the application.

3. Initialize DPDK ports and queues.

   ```
   dpdk_queues_and_ports_init();
   ```

4. Initialize DOCA Switch.

   ```
   switch_init();
   ```

   a. Initialize DOCA Flow.
   b. Create port pairs.
   c. Create Flow Pipes Manger module.
   d. Register an action for each relevant CLI command.

5. Initialize Flow Parser.

   ```
   flow_parser_init();
   ```

   a. Reset all internal Flow Parser structures.
   b. Start the command line interface.
   c. Receive user commands, parse them, and call the required DOCA Flow API command.
   d. Close the interactive shell once a "quit" command is entered.

6. Clean Flow Parser resources.

   ```
   flow_parser_cleanup();
   ```

7. Destroy Switch resources.

   ```
   switch_destroy();
   ```

   a. Destroy Flow Pipes Manager resources.

8. Destroy DOCA Flow.

   ```
   switch_destroy();
   ```

9. Destroy DPDK ports and queues.

   ```
   dpdk_queues_and_ports_fini();
   ```

10. DPDK finish.

    ```
    dpdk_fini();
    ```

    a. Call `rte_eal_destroy()` to destroy initialized EAL resources.

11. Arg parser destroy.

    ```
    doca_argp_destroy();
    ```

15.18.8 References

- `/opt/mellanox/doca/applications/switch/`

15.19 NVIDIA DOCA UROM RDMO Application Guide

This guide provides a DOCA Remote Direct Memory Operation implementation on top of NVIDIA® BlueField® DPU using Unified Communication X (UCX).

15.19.1 Introduction

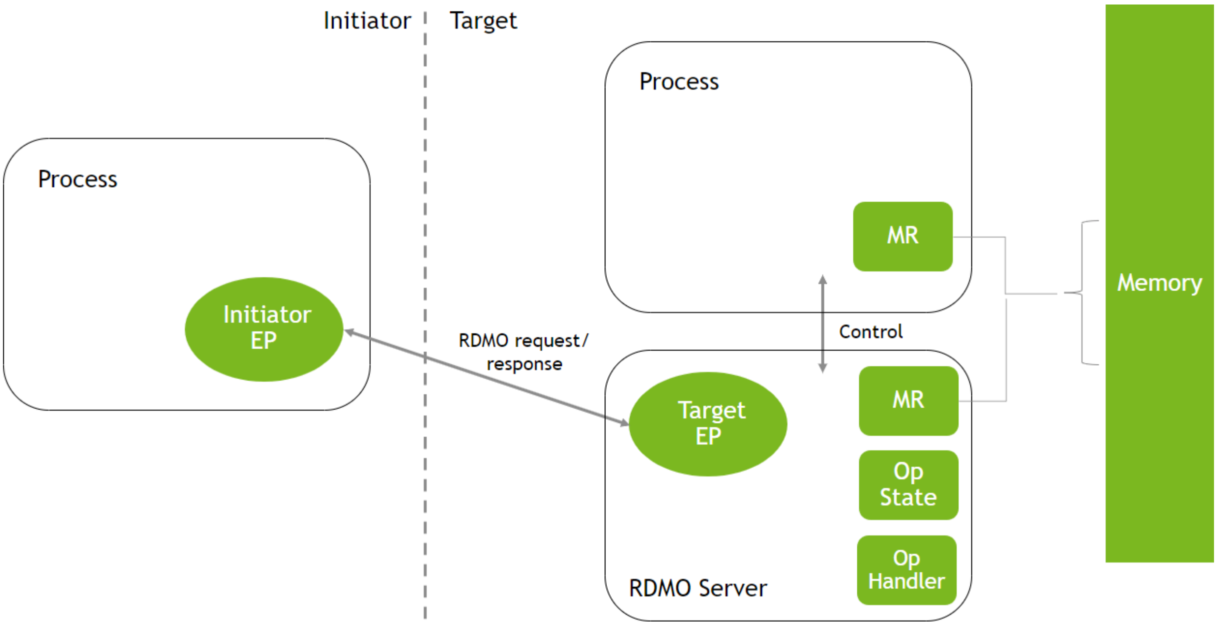
A remote direct memory operation (RDMO) is conceptionally an active message which is executed outside the context of the target process.

An RDMO involves the following entities:

- Target – establishes a connection to the server to use as the control path. The target interacts with the server to define target endpoints and memory regions. The target exchanges endpoint and memory region information with an initiator to facilitate its connection.
- Initiator – establishes a connection to the server to use as the data path. An RDMO is initiated by sending an RDMO command with an optional payload to the server. The server parses the commands and runs an associated RDMO handler. An RDMO handler interacts with the target process by performing one-sided memory accesses to target-defined memory regions.
- Server – responsible for executing RDMOs asynchronously from the target process. The server implements an RDMO handler for each supported operation. RDMO handlers may maintain a state within the server for optimization.

The DOCA UROM RDMO application includes the above three entities, split into the following parts:

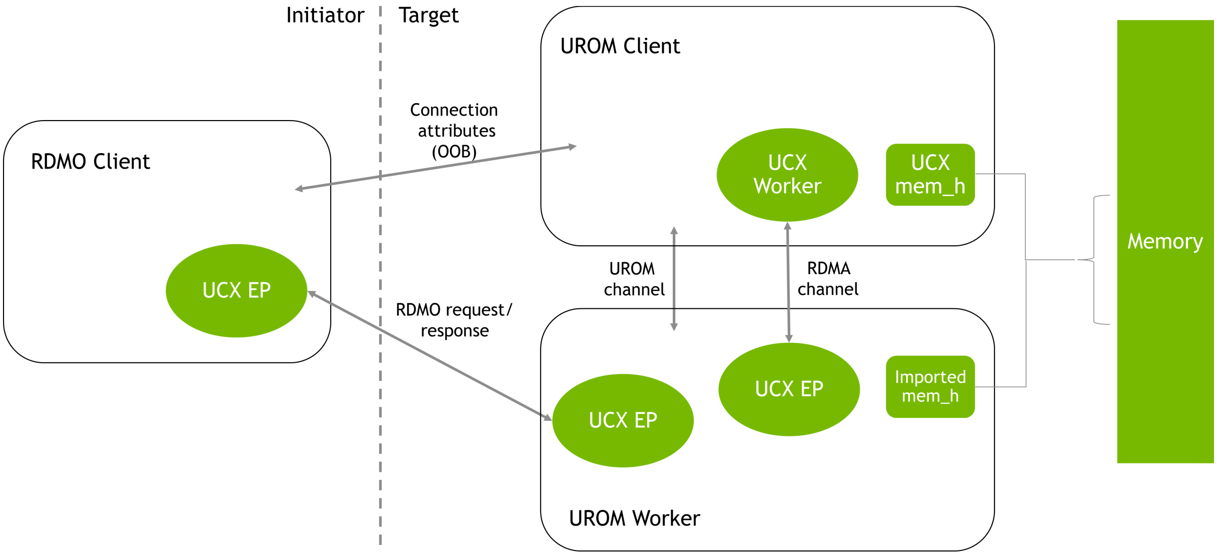
- BlueField side – the implementation of RDMO plugin component to be loaded by the DOCA UROM worker (which is the RDMO server)
- Host side – host application that runs using two modes: target and initiator



RDMOs are designed to take advantage of extra computing resources on a platform. While application processes run on the primary compute resources, an RDMO server can run on idle resources on the same host or be offloaded to run on a separate device (i.e., BlueField).

15.19.2 System Design

The application demonstrates the implementation of RDMO operations as a DOCA UROM worker plugin component. A target process would use the DOCA UROM API to create a worker with RDMO capabilities. An initiator process establishes an RDMO connection to the UROM worker. The plugin uses UCX as its transport.



15.19.2.1 Bootstrap Procedure

To connect the RDMO initiator and target, on the target side, UROM is used to retrieve an address for each created RDMO worker. This address would need to be delivered to the RDMO initiator side for connection establishment. The initiator address is obtained from the UCX worker created explicitly by the RDMO application. Both addresses are exchanged over the out-of-band (OOB) network and used to establish the connection:

- On the RDMO initiator side, a UCX endpoint is created using UCX API
- On the RDMO target side, the initiator's address is communicated to the RDMO worker using the UROM command channel

15.19.2.2 Memory Management

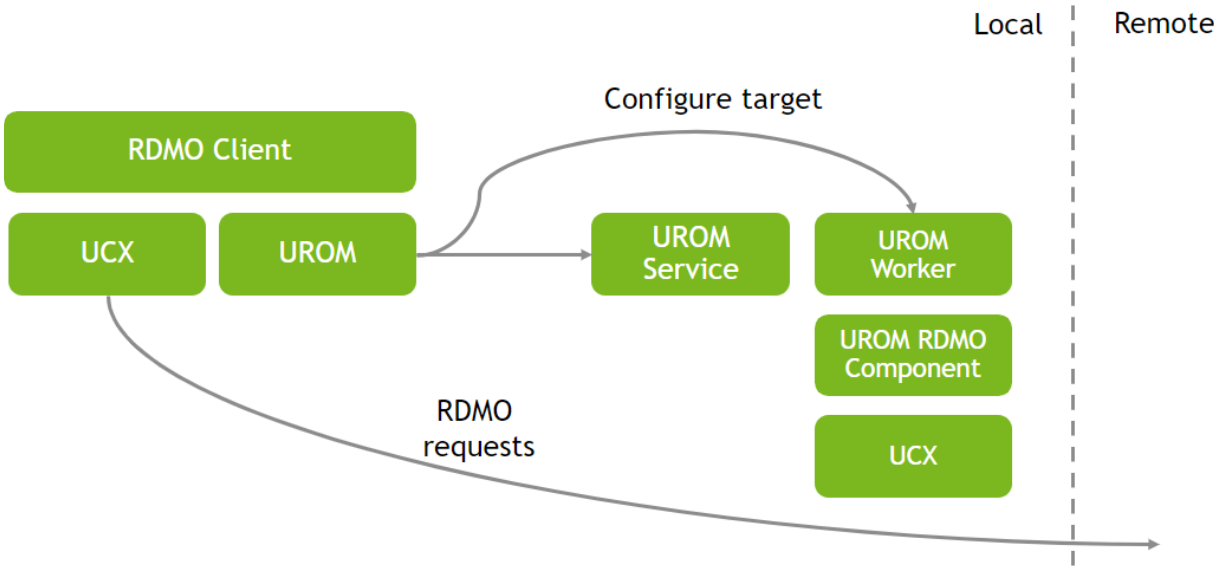
UROM returns an identifier (ID) for each memory region imported to the RDMO plugin component. This ID is used to refer to a target memory region in RDMO requests. It must be exchanged with the initiator process OOB.

15.19.2.3 RDMO UROM Worker Operation

Communication between the RDMO initiator and worker is implemented on top of UCX active messages. The worker's active message handler is the entry point that identifies the type of the RDMO operation based on the RDMO request header. The request is then forwarded to the corresponding RDMO operation handler which determines the operation parameters by inspecting the operation-specific sub-header in the request.

UCX active messages support eager and rendezvous protocols. When using a rendezvous protocol, the worker can choose whether to pull data to the server or move it directly to a target memory using a UCX-imported memory handle.

An RDMO operation handler may perform any combination of computation, initiator and target memory accesses, server state updates, or responses.



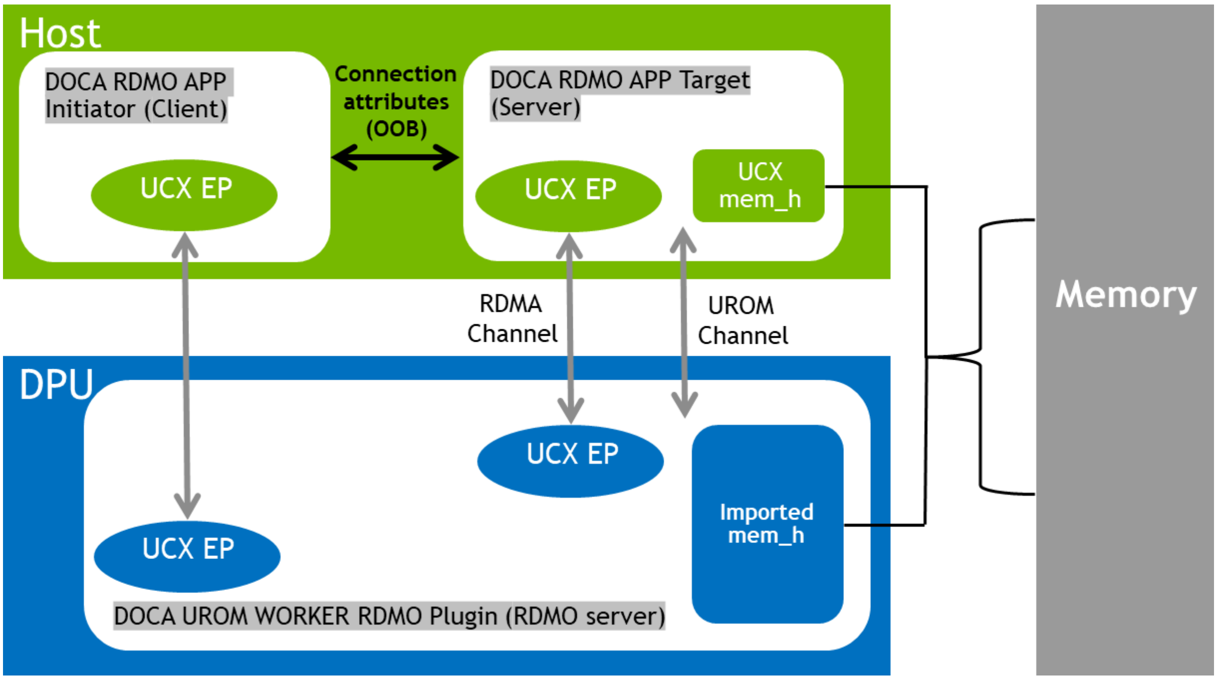
The RDMO client uses UROM to instantiate an RDMO worker and to configure target endpoints and memory regions. The client uses UCX directly to connect endpoints to the RDMO server. The client uses UCX to send formatted RDMO messages.

15.19.3 Application Architecture

DOCA's UROM RDMO application implementation uses UCX to support data exchange between endpoints. It utilizes UCX's sockaddr-based connection establishment and the UCX active messages (AM) API for communications, and UCX is responsible for all RDMO communications (control and data path).

The RDMO server application initiates a DOCA UROM worker RDMO component via the DOCA UROM service and shares the UROM worker UCX EP with the DOCA UROM RDMO client application. The RDMO server application imports memory regions into the UROM worker to facilitate RDMA operations from the BlueField on host memory.

The RDMO client application performs RDMO operations via the DOCA UROM worker. Upon receiving the UCX EP address from the server, the client application initially establishes a connection with the worker. It then proceeds to request the worker to execute the operation without the server application's awareness.


Host
DOCA RDMO APP Initiator (Client)
UCX EP
Connection attributes (OOB)
DOCA RDMO APP Target (Server)
UCX EP
UCX mem_h
RDMA Channel
UROM Channel
Memory
DPU
UCX EP
UCX EP
Imported mem_h
DOCA UROM WORKER RDMO Plugin (RDMO server)

15.19.3.1 UROM RDMO Worker Component

The UROM RDMO worker plugin component defines a small set of commands to enable the target to:

- Establish a UCX communication channel between the client and the worker
- Create a UCX endpoint capable of receiving RDMO request
- Import memory regions that can be used as a source or target for RDMA initiated by the worker

The set of commands are:

```
enum urom_worker_rdmo_cmd_type {
    UROM_WORKER_CMD_RDMO_CLIENT_INIT,
    UROM_WORKER_CMD_RDMO_RQ_CREATE,
    UROM_WORKER_CMD_RDMO_RQ_DESTROY,
    UROM_WORKER_CMD_RDMO_MR_REG,
    UROM_WORKER_CMD_RDMO_MR_DEREG,
};
```

The associated notification types are:

```
enum urom_worker_rdmo_notify_type {
    UROM_WORKER_NOTIFY_RDMO_CLIENT_INIT,
    UROM_WORKER_NOTIFY_RDMO_RQ_CREATE,
    UROM_WORKER_NOTIFY_RDMO_RQ_DESTROY,
    UROM_WORKER_NOTIFY_RDMO_MR_REG,
    UROM_WORKER_NOTIFY_RDMO_MR_DEREG,
};
```

15.19.3.1.1 Init

The Client Init command initializes the client to receive RDMOs. This includes establishing a connection between worker and host to allow the RDMO worker to access client memory.

The command is of type `UROM_WORKER_CMD_RDMO_CLIENT_INIT`. Command format:

```
struct urom_worker_rdmo_cmd_client_init {
    uint64_t id;
    void *addr;
    uint64_t addr_len;
};
```

- `id` - client ID used to identify the target process in RDMO commands
- `addr` - pointer to the client's UCP worker address to use for a worker-to-host connection
- `addr_len` - length of the address

This command returns a notification of type `UROM_WORKER_NOTIFY_RDMO_CLIENT_INIT`. Notification format:

```
struct urom_worker_rdmo_notify_client_init {
    void *addr;
    uint64_t addr_len;
```

- `addr` - pointer to the component's UCP worker address to use for initiator-to-server connections
- `addr_len` - length of the address

15.19.3.1.2 RQ Create

This Receive Queue (RQ) Create command creates and connects a new endpoint on the server. The endpoint may be targeted by formatted RDMO messages.

This command is of type `UROM_WORKER_CMD_RDMO_RQ_CREATE`. Command format:

```
struct urom_worker_rdmo_cmd_rq_create {
    void *addr;
    uint64_t addr_len;
};
```

- `addr` - the UCP worker address to use to connect the new endpoint
- `addr_len` - the length of address

The command returns a notification of type `UROM_WORKER_NOTIFY_RDMO_RQ_CREATE`. Notification format:

```
struct urom_worker_rdmo_notify_rq_create {
    uint64_t rq_id;
};
```

- `rq_id` - the RQ ID to use to destroy the RQ

15.19.3.1.3 RQ Destroy

The RQ Destroy command destroys an RQ.

The RQ Destroy command is of type `UROM_WORKER_CMD_RDMO_RQ_DESTROY`. Command format:

```
struct urom_worker_rdmo_cmd_rq_destroy {
    uint64_t rq_id;
};
```

- `rq_id` - the ID of a previously created RQ

The RQ destroy command returns a notification of type `UROM_WORKER_NOTIFY_RDMO_RQ_DESTROY`. Notification format:

```
struct urom_worker_rdmo_notify_rq_destroy {
    uint64_t rq_id;
};
```

- `rq_id` - the destroyed receive queue id

15.19.3.1.4 MR Register

The Memory Region (MR) Register command registers a UCP memory handle with the RDMO component. An MR must be registered with the RDMO component before use in RDMOs.

The command is of type `UROM_WORKER_CMD_RDMO_MR_REG`. Command format:

```
struct urom_worker_rdmo_cmd_mr_reg {
    uint64_t va;
    uint64_t len;
    void *packed_rkey;
    uint64_t packed_rkey_len;
    void *packed_memh;
    uint64_t packed_memh_len;
};
```

- `va` - the virtual address of the MR
- `len` - the length of the MR
- `packed_rkey` - pointer to the UCP packed R-key for the MR
- `packed_rkey_len` - the length of `packed_rkey`
- `packed_mem_h` - pointer to the UCP-packed memory handle for the MR. The memory handle must be packed with flag `UCP_MEMH_PACK_FLAG_EXPORT`.
- `packed_memh_len` - the length of `packed_memh`

The command returns a notification of type `UROM_WORKER_NOTIFY_RDMO_MR_REG`. Notification format:

```
struct urom_worker_rdmo_notify_mr_reg {
    uint64_t rkey;
};
```

- `rkey` - the ID used in RDMOs to refer to the MR

15.19.3.1.5 MR Deregister

The MR deregister command deregisters an MR from the RDMO component.

The command is of type `UROM_WORKER_CMD_RDMO_MR_DEREG`. Command format:

```
struct urom_worker_rdmo_cmd_mr_dereg {
    uint64_t rkey;
};
```

- `rkey` - the ID of a previously registered MR

The command returns a notification of type `UROM_WORKER_NOTIFY_RDMO_MR_DEREG`. Notification format:

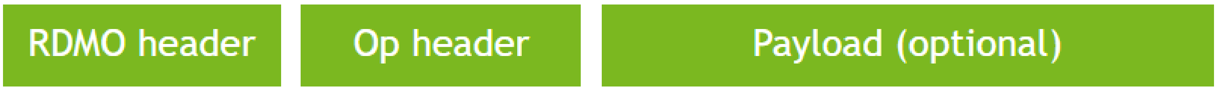
```
struct urom_worker_rdmo_notify_mr_dereg {
    uint64_t rkey;
};
```

- `rkey` - the deregistered memory region remote key

15.19.3.2 Command Format

An RDMO is initiated by sending an RDMO request via UCP active message to a UROM RDMO worker server.

The RDMO request format is:



The RDMO header identifies the operation type and flags, modifying how the RDMO is processed. The operation (op) header includes arguments specific to the operation type. Optionally, the operation type may include an arbitrary-sized payload.

RDMO header format:

```
struct urom_rdmo_hdr {
    uint32_t id;
    uint32_t op_id;
    uint32_t flags;
};
```

- `id` - the client ID
- `op_id` - the RDMO operation type ID
- `flags` - flags modifying how the RDMO is processed by the server

Valid flag values:

```
enum urom_rdmo_req_flags {
    UROM_RDMO_REQ_FLAG_FENCE,
};
```

- `UROM_RDMO_REQ_FLAG_FENCE` - Complete all outstanding RDMO requests on the connection before executing this request. This flag is required to implement a flush operation that guarantees remote completion.

Optionally, an operation may return a response to the initiator.

Response header format:

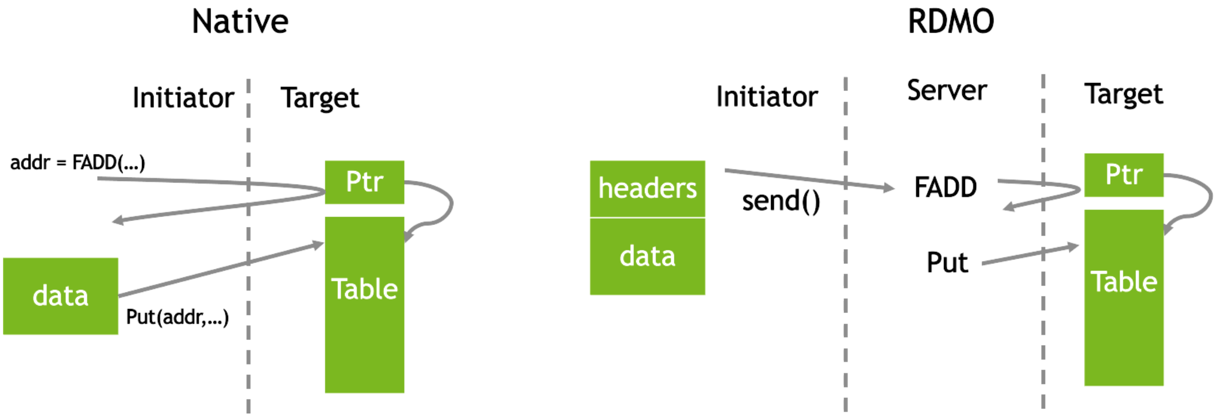
```
struct urom_rdmo_rsp_hdr {
    uint16_t op_id;
};
```

- `op_id` - the RDMO response type ID

15.19.3.2.1 Append

RDMO Append atomically appends data to a queue in remote memory. This can be achieved in a one-sided programming model with a Fetching-Add operation to the location of a pointer in remote memory, followed by a Put to the fetched address. RDMO Append allows these dependent operations to be offloaded to the target.

The following diagram provides a comparison of native and RDMO approaches to the Append operation:



Combining two dependent operations into a single RDMO allows the non-blocking implementation of Append, as the initiator does not need to wait between the Fetching Atomic and the data write operations. Using RDMO, the initiator can create a pipeline of operations and achieve a higher message rate.

The rate at which the RDMO server can perform operations on the target memory is expected to be a bottleneck. To improve the rate, the following optimizations can be looked at:

- The result of the Fetch-and-ADD (FADD) after the initial Append is performed can be cached in the server. Subsequent Appends can re-use the cached value, eliminating the atomic FADD operation. The modified pointer value is required to be synchronized during the flush command.
- For small Append sizes, the Append data can be cached in the RDMO server and coalesced into a single Put. As a result, the server requires, on average, a single Put access to target memory to execute several RDMOs.

- To avoid extra memory usage and lost bandwidth for large Append operations, the RDMO server may initiate direct transfers from the initiator to the target memory bypassing the acceleration device memory.

The Append operation uses an operation of type `UROM_RDMO_OP_APPEND`. Append header format:

```
struct urom_rdmo_append_hdr {
    uint64_t ptr_addr;
    uint16_t ptr_rkey;
    uint16_t data_rkey;
};
```

- `ptr_addr` - the address of the queue pointer in target memory
- `ptr_rkey` - the R-key used to access `ptr_addr`
- `data_rkey` - the R-key used to access the queue data

The RDMO payload is the local data buffer.

15.19.3.2.2 Flush

RDMO Flush is used to implement synchronization between the initiator and server. On execution, Flush sends a response message back to the initiator. Flush can be used to guarantee remote completion of a previously issued RDMO.

To achieve this, the initiator sends an in-order Flush command including the RDMO flag `UROM_RDMO_REQ_FLAG_FENCE`. This flag causes the server to complete all previously received RDMOs before executing the Flush. To complete previous operations, the server must write any cached data and make it visible in the target memory. Once complete, the server executes the Flush. Flush sends a response to the initiator. When the initiator receives the flush message, the result of all previously sent RDMOs is guaranteed to be visible in the target memory.

The Flush operation uses operation type `UROM_RDMO_OP_FLUSH`. Flush header format:

```
struct urom_rdmo_flush_hdr {
    uint64_t flush_id;
};
```

- `flush_id` - local ID used to track completion

Flush returns a response with the following header format:

```
struct urom_rdmo_flush_rsp_hdr {
    uint64_t    flush_id;
};
```

- `flush_id` - the ID of the completed Flush

Flush requests and responses do not include a payload.

15.19.3.2.3 Scatter

RDMO Scatter is used to support aggregating non-contiguous memory Puts. An RDMO may be defined to map non-contiguous virtual addresses into a single memory region using a network interface at the target platform, and then return a memory key for this region. The initiator may then perform Puts to this memory region, which are scattered by target hardware. Alternatively, an RDMO may be defined to post an IOV Receive. The initiator could then post a matching Send to scatter data at the target.

The Scatter operation uses operation type `UROM_RDMO_OP_SCATTER`. Scatter header format:

```
struct urom_rdmo_scatter_hdr {
    uint64_t count; /* Number of IOVs in the payload */
};
```

- `count` - Number of IOVs in the RDMO payload

IOVs are packed into the Scatter request payload, descriptor followed by data:

```
struct urom_rdmo_scatter_iov {
    uint64_t addr; /* Scattered data address */
    uint64_t rkey; /* Data remote key */
    uint16_t len;  /* Data length */
};
```

- `addr` - scattered data address
- `rkey` - data remote key
- `len` - data length

15.19.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA UROM
- UCX framework DOCA driver

Refer to their respective programming guide for more information.

15.19.5 Compiling the Application

Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

For more information about the applications as well as development and compilation tips, refer to the DOCA Applications page.

The sources of the application can be found under the application's directory: `/opt/mellanox/doca/applications/urom_rdmo/`.

15.19.5.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

On the host, `doca_urom_rdmo` is created under `/tmp/build/urom_rdmo/host/`. On the BlueField side, the RDMO worker plugin `worker_rdmo.so` is created under `/tmp/build/urom_rdmo/dpu/`.

15.19.5.2 Compiling Only the Current Application

To directly build only the UROM RDMO application (host) or plugin (DPU):

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_urom_rdmo=true
ninja -C /tmp/build
```

On the host, `doca_urom_rdmo` is created under `/tmp/build/urom_rdmo/host/`. On the BlueField side, the RDMO worker plugin `worker_rdmo.so` is created under `/tmp/build/urom_rdmo/dpu/`.

Alternatively, one can set the desired flags in the `meson_options.txt` file instead of providing them in the compilation command line:

1. Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt`:
    - Set `enable_all_applications` to `false`
    - Set `enable_urom_rdmo` to `true`
2. Run the following compilation commands:

    ```
    cd /opt/mellanox/doca/applications/
    meson /tmp/build
    ninja -C /tmp/build
    ```

    On the host, `doca_urom_rdmo` is created under `/tmp/build/urom_rdmo/host/`. On the BlueField side, the RDMO worker plugin `worker_rdmo.so` is created under `/tmp/build/urom_rdmo/dpu/`.

15.19.5.3 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application.

15.19.6 Running the Application

15.19.6.1 Host Application Execution

The UROM RDMO application is provided in source form; therefore, a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_urom_rdmo [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags from an input json file

Program Flags:
  -d, --device <IB device name>     IB device name.
  -s, --server-name <server name>   server name.
  -m, --mode {server, client}       Set mode type {server, client}
```

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
./doca_urom_rdmo -h
```

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application with server mode:

```
./doca_urom_rdmo -d mlx5_0 -m server
```

3. CLI example for running the application with client mode:

```
./doca_urom_rdmo -m clinet -s <server_host_name>
```

4. The application also supports a JSON-based deployment mode, in which all command-line arguments are provided through a JSON file:

```
./doca_urom_rdmo --json [json_file]
```

For example:

```
./doca_urom_rdmo --json ./urom_rdmo_params.json
```

15.19.6.2 RDMO DPU Plugin Component

The UROM RDMO plugin component is provided in source form, hence a compilation is required before the application can be executed in order when spawning UROM worker could load the plugin in runtime and it is compiled as `.so` file.

The plugin exposes the following symbols:

- Get DOCA worker plugin interface for RDMO plugin:

```
doca_error_t urom_plugin_get_iface(struct urom_plugin_iface *iface);
```

- Get the RDMO plugin version which will be used to verify that the host and DPU plugin versions are compatible:

```
doca_error_t urom_plugin_get_version(uint64_t *version);
```

15.19.6.3 Command Line Flags

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
General flags	`h`	`help`	Print a help synopsis	N/A
	`v`	`version`	Print program version information	N/A
	`l`	`log-level`	Set the log level for the application: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70 (requires compilation with `TRACE` log level support)	`"log-level": 60`
	N/A	`sdk-log-level`	Set the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70	`"sdk-log-level": 40`
	`j`	`json`	Parse all command flags from an input JSON file	N/A
Program flags	`d`	`device`	DOCA UROM IB device name	`"device": "mlx5_0"`
	`s`	`server-name`	RDMO server name	`"server-name": "<host-name>-oob"`
	`m`	`mode`	RDMO application mode [server, client]	`"mode": "client"`

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

15.19.6.4 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

15.19.7 Application Code Flow

1. Parse application argument.
    a. Initialize arg parser resources and register DOCA general parameters.

    ```
    doca_argp_init();
    ```

    b. Register UROM RDMO application parameters.

    ```
    register_urom_rdmo_params();
    ```

    c. Parse the arguments.

    ```
    doca_argp_start();
    ```

2. Run main logic:
    - If the application mode is server:
        i. Create UROM objects and spawn UROM worker on the BlueField.
        ii. Initialize UCP with features: `UCP_FEATURE_AM` , `UCP_FEATURE_EXPORTED_MEMH` .
        iii. Create a UCP worker and query the worker address
        iv. Initialize the RDMO worker client with the command `UROM_WORKER_CMD_RDMO_CLIENT_INIT` .
        v. Send UROM RDMO worker address to the initiator via OOB channel and receive the intiator's UCP worker address
        vi. Create a UCP memory handle and register it with the RDMO server using the command `UROM_WORKER_CMD_RDMO_MR_REG` . Receive an R-key in return.
        vii. Send the RDMO key to the initiator
        viii. Create an RDMO RQ by passing the initiator's UCP worker address to the UROM command `UROM_WORKER_CMD_RDMO_RQ_CREATE` .
        ix. Wait till the RDMO append operation is done and next validate the memory data.
        x. Wait till the RDMO scatter operation is done and next validate the memory data.
        xi. Destroy the UCP resources.
        xii. Destroy UROM RDMO worker and UROM objects.
    - If the application mode is client:
        i. Create UCP worker using UCX API directly.
        ii. Receive the UROM RDMO worker address via OOB channel and send the initiator's UCP worker address.
        iii. Create a UCP endpoint using the RDMO worker address.

    iv. Install an Active Message handler on the endpoint to receive RDMO responses.
    v. Send an RDMO requests via UCP Active Message protocol with the header pointing to the serialized RDMO and Op headers, and data pointing to the payload. The request parameter flag: `UCP_AM_SEND_FLAG_REPLY` will be set to allow the RDMO server to identify the sender.
    vi. Once the RDMO operations are done, Destroy UCP resources.

3. Arg parser destroy.

```
doca_argp_destroy();
```

15.19.8 References

- `/opt/mellanox/doca/applications/urom_rdmo/`
- `/opt/mellanox/doca/applications/urom_rdmo/urom_rdmo_params.json`

15.20 NVIDIA DOCA YARA Inspection Application Guide

This guide provides YARA inspection implementation on top of NVIDIA® BlueField® DPU.

15.20.1 Introduction

YARA inspection monitors all processes in the host system for specific YARA rules using the DOCA App Shield library.

This security capability helps identify malware detection patterns in host processes from an independent and trusted DPU. This is an innovative Intrusion Detection System (IDS) as it is designed to run independently on the DPU's Arm cores without hindering the host.

This DOCA App Shield based application provides the capability to read, analyze, and authenticate the host (bare metal/VM) memory directly from the DPU.

Using the library, this application scans host processes and looks for pre-defined YARA rules. After every scan iteration, the application indicates if any of the rules matched. Once there is a match, the application reports which rules were detected in which process. The reports are both printed to the console and exported to the DOCA Telemetry Service (DTS) using inter-process communication (IPC).

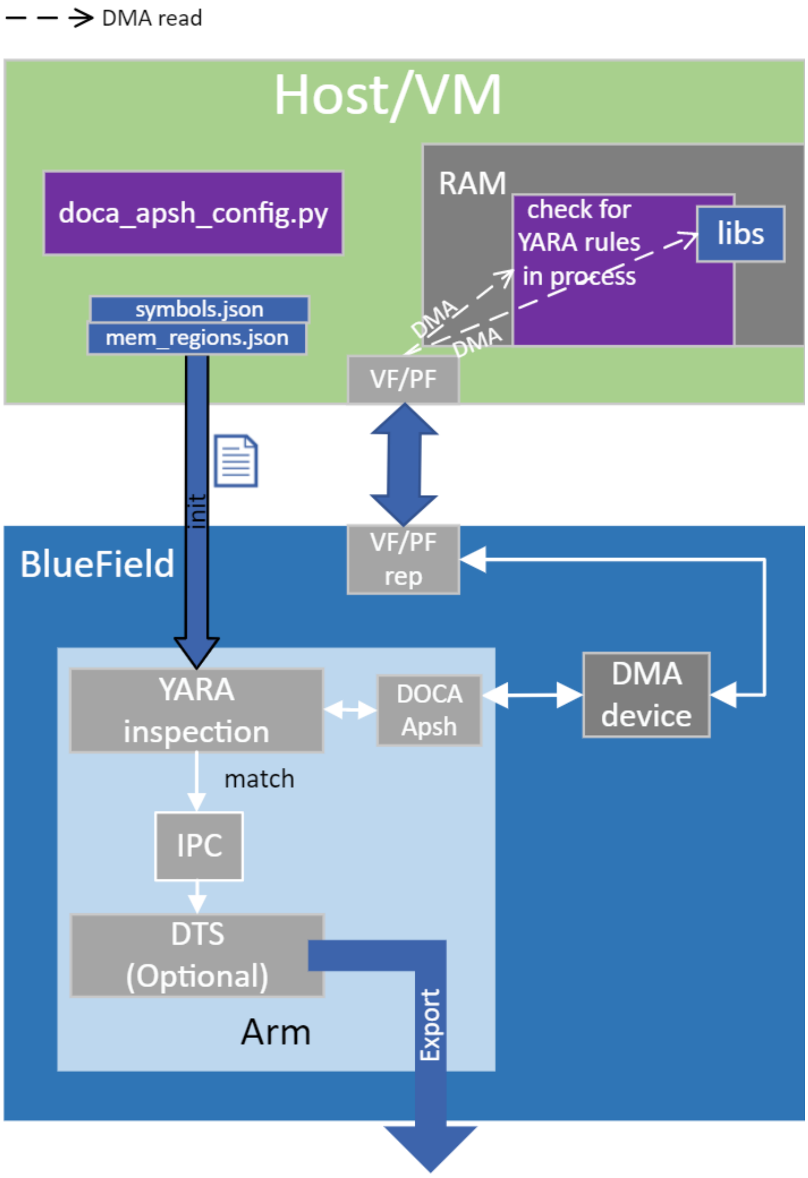
This guide describes how to build YARA inspection using the DOCA App Shield library which leverages DPU abilities such as hardware-based DMA, integrity, and more.

> ⚠ As the DOCA App Shield library only supports the YARA API for Windows hosts, this application can only be used to inspect Windows hosts.

15.20.2 System Design

The host's involvement is limited to generating the required ZIP and JSON files to pass to the DPU. This is done before the app is triggered, when the host is still in a "safe" state.

Generating the needed files can be done by running DOCA App Shield's `doca_apsh_config.py` tool on the host. See DOCA App Shield for more info.

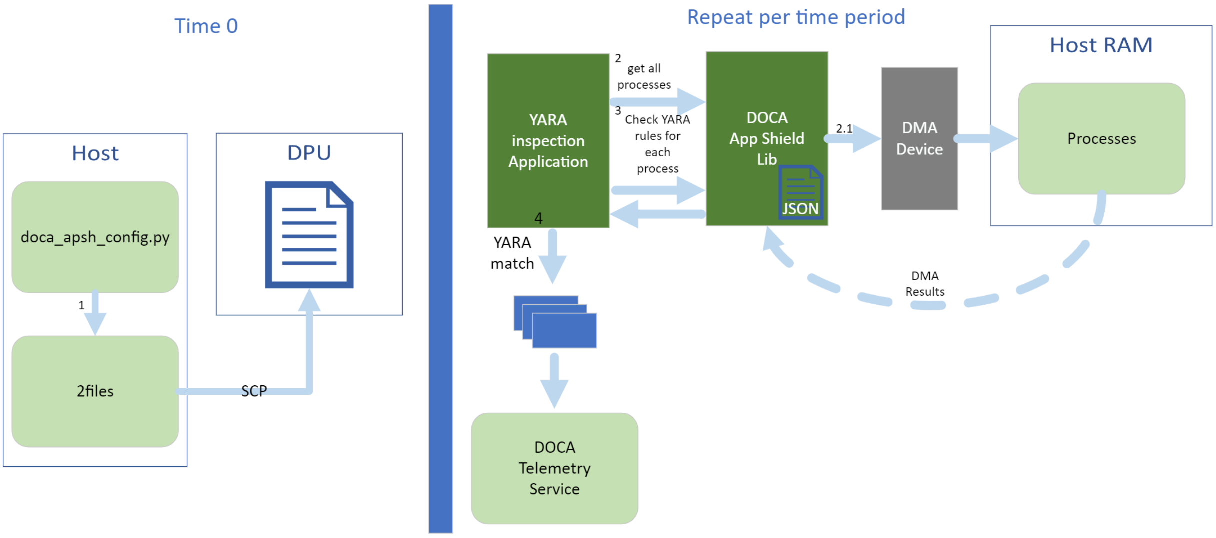


15.20.3 Application Architecture

The user creates the ZIP and JSON files using the DOCA tool `doca_apsh_config.py` and copies them to the DPU.

The application can report YARA rules detection to the:

- File
- Terminal
- DTS



1. The files are generated by running `doca_apsh_config.py` on the host against the process at time zero.
2. The following steps recur at regular time intervals:
   a. The YARA inspection app requests a list of all apps from the DOCA App Shield library.
   b. The app loops over all processes and checks for YARA rules match using the DOCA App Shield library.
   c. If YARA rules are found (1 or more), the YARA attestation app reports results with a timestamp and details about the process and rules to:
      - Local telemetry files – a folder and files representing the data a real DTS would have received

        > ⚠ These files are used for the purpose of this example only as normally this data is not exported into user-readable files.

      - DOCA log
      - DTS IPC interface (even if no DTS is active)
3. The App Shield agent exits on first YARA rule detection.

15.20.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA App Shield
- DOCA Telemetry Exporter

Refer to their respective programming guide for more information.

15.20.5 Limitations

- The application is only available on Ubuntu 22.04 environments
- The application only supports the inspection of Windows hosts

15.20.6 Compiling the Application

Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

For more information about the applications as well as development and compilation tips, refer to the DOCA Applications page.

The sources of the application can be found under the application's directory: `/opt/mellanox/doca/applications/yara_inspection/`.

15.20.6.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

`doca_yara_inspection` is created under `/tmp/build/yara_inspection/`.

15.20.6.2 Compiling Only the Current Application

To directly build only the YARA inspection application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_yara_inspection=true
ninja -C /tmp/build
```

`doca_yara_inspection` is created under `/tmp/build/yara_inspection/`.

Alternatively, one can set the desired flags in the `meson_options.txt` file instead of providing them in the compilation command line:

1. Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt`:
    - Set `enable_all_applications` to `false`
    - Set `enable_yara_inspection` to `true`
2. Run the following compilation commands:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

`doca_yara_inspection` is created under `/tmp/build/yara_inspection/` .

15.20.6.3 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application.

15.20.7 Running the Application

15.20.7.1 Prerequisites

1. Configure the BlueField's firmware
   a. On the BlueField system, configure the PF base address register and NVME emulation. Run:

   ```
   dpu> mlxconfig -d /dev/mst/mt41686_pciconf0 s PF_BAR2_SIZE=2 PF_BAR2_ENABLE=1
    NVME_EMULATION_ENABLE=1
   ```

   b. Perform a BlueField system reboot for the `mlxconfig` settings to take effect.
   c. This configuration can be verified using the following command:

   ```
   dpu> mlxconfig -d /dev/mst/mt41686_pciconf0 q | grep -E "NVME|BAR"
   ```

2. Download target system (host/VM) symbols.
   - For Ubuntu:

   ```
   host> sudo tee /etc/apt/sources.list.d/ddebs.list << EOF
   deb http://ddebs.ubuntu.com/ $(lsb_release -cs) main restricted universe multiverse
   deb http://ddebs.ubuntu.com/ $(lsb_release -cs)-updates main restricted universe multiverse
   deb http://ddebs.ubuntu.com/ $(lsb_release -cs)-proposed main restricted universe multiverse
   EOF
   host> sudo apt install ubuntu-dbgsym-keyring
   host> sudo apt-get update
   host> sudo apt-get install linux-image-$(uname -r)-dbgsym
   ```

   - For CentOS:

   ```
   host> yum install --enablerepo=base-debuginfo kernel-devel-$(uname -r) kernel-debuginfo-$(uname -r)
   kernel-debuginfo-common-$(uname -m)-$(uname -r)
   ```

   - No action is needed for Windows
3. Perform IOMMU passthrough. This stage is only needed on some of the cases where IOMMU is not enabled by default (e.g., when the host is using an AMD CPU).

   Skip this step if you are not sure whether you need it. Return to it only if DMA fails with a message in `dmesg` similar to the following:

```
host> dmesg
[ 3839.822897] mlx5_core 0000:81:00.0: AMD-Vi: Event logged [IO_PAGE_FAULT domain=0x0047 address=0
x2a0aff8 flags=0x0000]
```

- Locate your OS's `grub` file (most likely `/boot/grub/grub.conf`, `/boot/grub2/grub.cfg`, or `/etc/default/grub`) and open it for editing. Run:

  ```
  host> vim /etc/default/grub
  ```

- Search for the line defining `GRUB_CMDLINE_LINUX_DEFAULT` and add the argument `iommu=pt`. For example:

  ```
  GRUB_CMDLINE_LINUX_DEFAULT="iommu=pt <intel/amd>_iommu=on"
  ```

- Run:

  > ⚠ Prior to performing a power cycle, make sure to do a graceful shutdown.

  - For Ubuntu:

    ```
    host> sudo update-grub
    host> ipmitool power cycle
    ```

  - For CentOS:

    ```
    host> grub2-mkconfig -o /boot/grub2/grub.cfg
    host> ipmitool power cycle
    ```

  - For Windows targets: Turn off Hyper-V capability.

4. The DOCA App Shield library uses hugepages for DMA buffers. Therefore, the user must allocate 42 huge pages.
   a. Run:

   ```
   dpu> nr_huge=$(cat /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages)
        nr_huge=$((42+$nr_huge))
        echo $nr_huge | sudo tee -a /sys/devices/system/node/node0/hugepages/hugepages-2048kB/
   nr_hugepages
   ```

   b. Create the ZIP and JSON files. Run:

   ```
   target-system> cd /opt/mellanox/doca/tools/
   target-system> python3 doca_apsh_config.py <pid-of-process-to-monitor> --os <windows/linux> --path
   <path to dwarf2json executable  or pdbparse-to-json.py>
   target-system> cp /opt/mellanox/doca/tools/*.* <shared-folder-with-baremetal>
   dpu> scp <shared-folder-with-baremetal>/* <path-to-app-shield-binary>
   ```

   If the target system does not have DOCA installed, the script can be copied from the BlueField.
   The required `dwaf2json` and `pdbparse-to-json.py` are not provided with DOCA.

   > ⚠ If the kernel and process `.exe` have not changed, there no need to redo this step.

15.20.7.2 Application Execution

The YARA inspection application is provided in source form. Therefore, a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_yara_inspection [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICA
L, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags from an input json file

Program Flags:
  -m, --memr <path>                 System memory regions map
  -f, --vuid                        VUID of the System device
  -d, --dma                         DMA device name
  -o, --osym <path>                 System OS symbol map path
  -t, --time <seconds>              Scan time interval in seconds
```

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
./doca_yara_inspection -h
```

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the BlueField:

```
./doca_yara_inspection -m mem_regions.json -o symbols.json -f MT2125X03335MLNXS0D0F0VF1 -d mlx5_0 -t 3
```

All used identifiers (`-f` and `-d` flags) should match the identifier of the desired devices.

15.20.7.3 Command Line Flags

Flag Type	Short Flag	Long Flag	Description
General flags	`h`	`help`	Prints a help synopsis
	`v`	`version`	Prints program version information

Flag Type	Short Flag	Long Flag	Description
	`l`	`log-level`	Set the log level for the application: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70 (requires compilation with `TRACE` log level support)
	N/A	`sdk-log-level`	Sets the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70
	`j`	`json`	Parse all command flags from an input JSON file
Program flags	`m`	`memr`	Path to the pre-generated `mem_regions.json` file transferred from the host

Flag Type	Short Flag	Long Flag	Description

	`f`	`pcif`	System PCIe function vendor unique identifier (VUID) of the VF/PF exposed to the target system. Used for DMA operations. To obtain this argument, run:
			`target-system> lspci -vv | grep "\[VU\] Vendor specific:"`
			Example output:
			`[VU] Vendor specific: MT2125X03335MLNXS0D0F0`
			`[VU] Vendor specific: MT2125X03335MLNXS0D0F1`
			Two VUIDs are printed for each DPU connected to the target system. The first is of the DPU on `pf0` and the second is of the DPU on port `pf1` .
			Running this command on the DPU outputs VUIDs with an additional "EC" string in the middle. You must remove the "EC" to arrive at the correct VUID.
			The VUID of a VF allocated on PF0/1 is the VUID of the PF with an additional suffix, `VF<vf-number>` , where `vf-number` is the VF index +1.
			For example, for the output in the example above:
			- PF0 VUID = MT2125X03335MLNXS0D0F0
			- PF1 VUID = MT2125X03335MLNXS0D0F1
			- VUID of VF0 on PF0 = MT2125X03335MLNXS0D0F0VF1
			VUIDs are persistent even on reset.

	`d`	`dma`	DMA device name to use
	`o`	`osym`	Path to the pre-generated `symbols.json` file transferred from the host
	`t`	`time`	Number of seconds to sleep between scans

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

15.20.7.4 Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

15.20.8 Application Code Flow

1. Parse application argument.
   a. Initialize arg parser resources and register DOCA general parameters.

      ```
      doca_argp_init();
      ```

   b. Register application parameters.

      ```
      register_apsh_params();
      ```

   c. Parse the arguments.

      ```
      doca_argp_start();
      ```

2. Initialize DOCA App Shield lib context.
   a. Create lib context.

      ```
      doca_apsh_create();
      ```

   b. Set DMA device for lib.

      ```
      open_doca_device_with_ibdev_name();
      doca_apsh_dma_dev_set();
      ```

   c. Start the context

      ```
      doca_apsh_start();
      apsh_system_init();
      ```

3. Initialize DOCA App Shield lib system context handler.
   a. Get the representor of the remote PCIe function exposed to the system.

      ```
      open_doca_device_rep_with_vuid();
      ```

   b. Create and start the system context handler.

      ```
      doca_apsh_system_create();
      doca_apsh_sys_os_symbol_map_set();
      doca_apsh_sys_mem_region_set();
      doca_apsh_sys_dev_set();
      doca_apsh_sys_os_type_set();
      doca_apsh_system_start();
      ```

4. Telemetry initialization.

```
telemetry_start();
```

   a. Initialize a new telemetry schema.
   b. Register YARA type event.
   c. Set up output to file (in addition to default IPC).
   d. Start the telemetry schema.
   e. Initialize and start a new DTS source with the `gethostname()` name as source ID.

5. Loop until YARA rule is matched.
   a. Get all processes from the host.

   ```
   doca_apsh_processes_get();
   ```

   b. Check for YARA rule identification and send a DTS event if there is a match.

   ```
   doca_apsh_yara_get();
   if (yara_matches_size != 0) {
       /* event fill logic
       doca_telemetry_exporter_source_report();
   DOCA_LOG_INFO();
   sleep();
   ```

6. Telemetry destroy.

   ```
   telemetry_destroy();
   ```

7. YARA inspection clean-up.

   ```
   doca_apsh_system_destroy();
   doca_apsh_destroy();
   doca_dev_close();
   doca_dev_rep_close();
   ```

8. Arg parser destroy.

   ```
   doca_argp_destroy();
   ```

15.20.9 References

- `/opt/mellanox/doca/applications/yara_inspection/`

16 DOCA Tools

This is an overview of the set of tools provided by DOCA and their purpose.

16.1 Introduction

DOCA tools are a set of executables/scripts that are needed to produce inputs to some of the DOCA libraries and applications.

All tools are installed with DOCA, as part of the doca-tools package, and can either be directly accessed from the terminal or can be found at `/opt/mellanox/doca/tools`. Refer to NVIDIA DOCA Installation Guide for Linux for more information.

> ⓘ For questions, comments, and feedback, please contact us at DOCA-Feedback@exchange.nvidia.com.

16.2 Tools

16.2.1 DOCA Bench

CLI name: `doca_bench`

DOCA Bench is a tool that allows a user to evaluate the performance of DOCA applications, with reasonable accuracy for real-world applications. It provides a flexible architecture to evaluate multiple features in series with multi-core scaling to provide detailed throughput and latency analysis.

16.2.2 Capabilities Print Tool

CLI name: `doca_caps`

DOCA Capabilities Print tool is used to print the available devices and their representor devices (in the DPU), all their capabilities, and the available DOCA libraries.

16.2.3 DPA Tools

DOCA DPA tools are a set of executables that enable the DPA application developer and the system administrator to manage and monitor DPA resources and to debug DPA applications.

16.2.4 PCC Counter

CLI name: `pcc_counters.sh`

DOCA PCC Counter tool is used to print PCC-related hardware counters. The output counters help debug the PCC user algorithm embedded in the DOCA PCC application.

16.2.5 Socket Relay

CLI name: `doca_socket_relay`

DOCA Socket Relay tool allows Unix Domain Socket (AF_UNIX family) server applications to be offloaded to Bluefield while communication between the two sides is proxied by DOCA Comm Channel.

16.3 NVIDIA DOCA Bench

16.3.1 Introduction

NVIDIA DOCA Bench allows users to evaluate the performance of DOCA applications, with reasonable accuracy for real-world applications. It provides a flexible architecture to evaluate multiple features in series with multi-core scaling to provide detailed throughput and latency analysis.

This tool can be used to evaluate the performance of multiple DOCA operations, gain insight into each stage in complex DOCA operations and understand how items such as buffer sizing, scaling, and GGA configuration affect throughput and latency.

16.3.2 Feature Overview

DOCA Bench is designed as a unified testing tool for all BlueField accelerators. It, therefore, provides these major features:

- BlueField execution, utilizing the Arm cores and GGAs "locally"
- Host (x86) execution, utilizing x86 cores and the GGAs on the BlueField over PCIe
- Support for following DOCA/DPU features:
    - DOCA AES GCM
    - DOCA Comch
    - DOCA Compress
    - DOCA DMA
    - DOCA EC
    - DOCA Eth
    - DOCA RDMA
    - DOCA SHA
- Multi-core/multi-thread support
- Schedule executions based on time, job counts, etc.
- Ability to construct complex pipelines with multiple GGAs (where data moves serially through the pipeline)
- Various data sources (random data, file data, groups of files, etc.)
- Remote memory operations
    - Use data location on the host x86 platform as input to GGAs
- Comprehensive output to screen or CSV
- Query function to report supported software and hardware feature
- Sweeping of parameters between a start and end value, using a specific increment each time

- Specific attributes can be set per GGA instance, allowing fine control of GGA operation

16.3.3 Installation

DOCA Bench is installed and available in both DOCA-for-Host and DOCA BlueField Arm packages. It is located under the `/opt/mellanox/doca/tools` folder.

16.3.3.1 Prerequisites

DOCA 2.7.0 and higher.

16.3.3.2 Granular Build Support

DOCA Bench supports a granular build environment which allows users to determine which DOCA libraries are installed on any target system. During initialization, DOCA Bench probes all available and supported DOCA libraries, and provides the ability to test those libraries. For example, if the DOCA SHA library is not present then DOCA Bench does not allow SHA to be tested.

DOCA Bench provides a query system where device capabilities can be queried to see if the library is indeed installed and supported (under the "installed : yes / no" section of each library). Please see section "Queries" for details.

16.3.4 Operating Modes

DOCA Bench measures performance of either throughput (bandwidth) or latency.

16.3.5 Throughput Measurements

In this mode, DOCA Bench measures the maximum performance of a given pipeline (see "Core Principles"). At the end of the execution, a short summary along with more detailed statistics is presented:

```
Aggregate stats
        Duration:      3000049 micro seconds
        Enqueued jobs: 17135128
        Dequeued jobs: 17135128
        Throughput:    5712042 Operations/s
        Ingress rate:  063.832 Gib/s
        Egress rate:   063.832 Gib/s
```

16.3.5.1 Latency Measurements

Latency is the measurement of time taken to perform a particular operation. In this instance, DOCA Bench measures the time taken between submitting a job and receiving a response.

DOCA Bench provides two different types of latency measurement figures:

- Bulk latency mode – attempts to submit a group of jobs in parallel to gain maximum throughput, while reporting latency as the time between the first job submitted in the group and the last job received.
- Precision latency mode – used to ensure that only one job is submitted and measured before the next job is scheduled.

16.3.5.1.1 Bulk Latency

This latency mode effectively runs the pipelines at full rate, trying to maintain the maximum throughput of any pipeline while also recording latency figures for jobs submitted.

To record latency, while operating at the pipeline's maximum throughput, users must place the latency figures inside groups or "buckets" (rather than record each individual job latency). Using this method, users can avoid the large memory and CPU overheads associated with recording millions of latency figures per second (which would otherwise significantly reduce the performance).

As each pipeline operation is different, and therefore has different latency characteristics, the user can supply the boundaries of the latency measure. DOCA Bench internally creates 100 buckets, of which the user can specify the starting value and the width or size of each bucket. The first and last bucket have significance:

- The first bucket contains all jobs that executed faster than the starting period
- The last bucket contains a count of jobs that took longer than the maximum time allowed

The command line option `--latency-bucket-range` is used to supply two values representing the starting time period of the first bucket, and the width of each sequential bucket. For example, `--latency-bucket-range 10us,100us` would start with the lowest bucket measuring <10μs response times, then 100 buckets which are 100μs wide, and a final bucket for results taking longer than 10010μs.

The report generated by bulk mode visualizes the latency data in two methods:

1. A bar graph is provided to visually show the spread of values across the range specified by the `--latency-bucket-range` option:

```
Latency report:
                   :
                   :
                   :
                   :
                   :
                   ::
                   ::
                   ::
                   ::
                  .::.  .    .    ..
-------------------------------------------------------------------------------------------------------
```

2. A breakdown of the number of jobs per bucket is presented. This example shortens the output to show that the majority of values lie between 27000ns and 31000ns.

```
[25000ns -> 25999ns]: 0
[26000ns -> 26999ns]: 0
[27000ns -> 27999ns]: 128
[28000ns -> 28999ns]: 2176
[29000ns -> 29999ns]: 1152
[30000ns -> 30999ns]: 128
[31000ns -> 31999ns]: 0
[32000ns -> 32999ns]: 0
[33000ns -> 33999ns]: 128
[34000ns -> 34999ns]: 0
[35000ns -> 35999ns]: 0
```

16.3.5.1.2 Precision Latency

This latency mode operates on a single job at a time. At the cost of greatly reduced throughput, this allows the minimum latency to be precisely recorded. As shown below, the statistics generated are precise and include various fields such as min, max, median, and percentile values.

```
Aggregate stats

        ....
        min:          1878 ns
        max:          4956 ns
        median:       2134 ns
        mean:         2145 ns
        90th %ile:    2243 ns
        95th %ile:    2285 ns
        99th %ile:    2465 ns
        99.9th %ile:  3193 ns
        99.99th %ile: 4487 ns
```

16.3.6 Core Principles

The following subsections elaborate on principles which are essential to understand how DOCA Bench operates.

16.3.6.1 Host or BlueField Arm Execution

Whether executing DOCA Bench on an x86 host or BlueField Arm, the behavior of DOCA Bench is identical. The performance measured is dependent on the environment.

> Only execution on x86 hosts is supported.

16.3.6.2 Pipelines

DOCA Bench is a highly flexible tool, providing the ability to configure how and what operations occur and in what order. To accomplish this, DOCA Bench uses a pipeline of operations, which are termed "steps". These steps can be a particular function (e.g., Ethernet receive, SHA hash generation, data compression). Therefore, a pipeline of steps can accomplish a number of sequential operations. DOCA Bench can measure the throughput performance or latency of these pipelines, whether running on single or multiple cores/threads.

> Currently, DOCA supports running only one pipeline at a time.

16.3.6.3 Warm-up Period

To ensure correct measurement, the pipelines must be run "hot" (i.e., any initial memory, caches, and hardware subsystems must be running prior to actual performance measurements begin). This is known as the "warm-up" period and, by default, runs approximately 250 jobs through the pipeline before starting measurements.

16.3.6.4 Defaults

DOCA Bench has a large number of parameters but, to simplify execution, only a few must be supplied to commence a performance measurement. Therefore various parameters have defaults which should be sufficient for most cases. To fine tune performance, users should pay close attention to any default parameters which may affect their pipeline's operation.

> When executed, DOCA Bench reports a full list of all parameters and configured values.

16.3.6.5 Optimizing Performance

To obtain maximum performance, a certain amount of tuning is required for any given environment. While outside the scope of this documentation, it is recommended for users to:

- Avoid using CPU 0 as most OS processes and interrupt request (IRQ) handlers are scheduled to execute on this core
- Enable CPU/IRQ isolation in the kernel boot parameters to remove kernel activities from any cores they wish to execute performance tests on
- On hosts, ensure to not cross any non-uniform memory access (NUMA) regions when addressing the BlueField
- Understand the memory allocation requirements of scenarios, to avoid over-allocating or running into near out-of-memory situations

16.3.7 Supported BlueField Feature Matrix

DOCA Bench can be executed on both host and BlueField Arm environments, and can target BlueField networking platforms.

The following table shows which operations are possible using either DOCA Bench. It also provides two columns showing whether remote memory can be used as an input or output to that operation. For example, DMA operations on the BlueField Arm can access remote memory as an input to pull memory from the host into the BlueField Arm).

	BlueField-2 Networking Platform	BlueFIeld-3 Networking Platform	Execute on Host Side	Execute on BlueField Arm	Remote Memory as Input Allowed?	Remote Memory as Output Allowed?
doca_compress::compress	✓		✓	✓	✓	✓
doca_compress::decompress	✓	✓	✓	✓	✓[1]	✓
doca_dma	✓	✓	✓	✓	✓	✓
doca_ec::create		✓	✓	✓	✓	✓
doca_ec::recover		✓	✓	✓	✓	✓
doca_ec::update		✓	✓	✓	✓	✓
doca_sha	✓		✓	✓	✓	
doca_rdma::send	✓	✓	✓	✓	✓	✓
doca_rdma::receive	✓	✓	✓	✓	✓	✓

doca_aes_gcm::encrypt		✓	✓	✓	✓	✓
doca_aes_gcm::decrypt		✓	✓	✓	✓	✓
doca_cc::client_producer	✓	✓	✓	✓		
doca_cc::client_consumer	✓	✓	✓	✓		
doca_eth::rx	✓	✓	✓			
doca_eth::tx	✓	✓	✓			

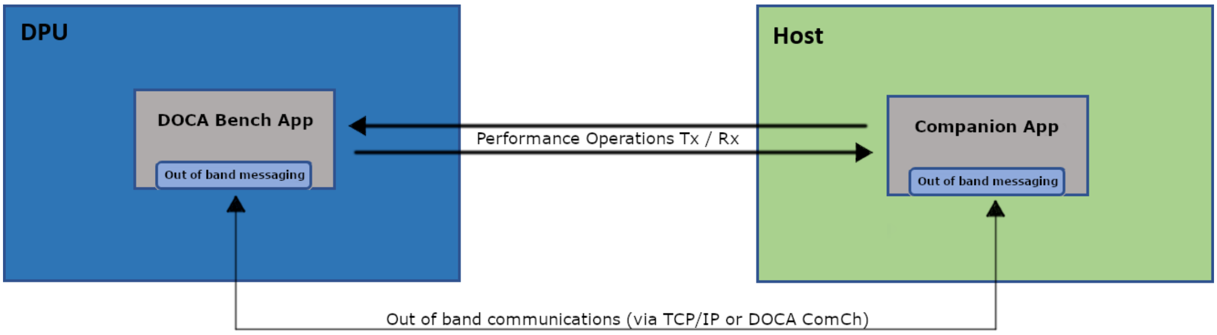
1. Input remote memory is not supported for lz4 decompression ↩

16.3.8 Remote Operations

A subset of BlueField operations have a remote element, whether this is an RDMA connection, Ethernet connectivity, or memory residing on an x86 host. All these operations require an agent to be present on the far side to facilitate the benchmarking of that particular feature.

In DOCA Bench, this agent is an additional standalone application called the "companion app". It provides the remote benchmarking facilities and is part of the standard DOCA Bench installation.

The following diagram provides an overview of the function and communications between DOCA Bench and the companion app:


DPU
DOCA Bench App
Out of band messaging
Performance Operations Tx / Rx
Host
Companion App
Out of band messaging
Out of band communications (via TCP/IP or DOCA ComCh)

In this particular setup, the BlueField executes "DOCA Bench" while the host (x86) is executes the companion App.

DOCA Bench also acts as the controller of the tests, instructing the companion app to perform the necessary operations as required. There is an out-of-band communications channel operating between the two applications that utilizes either standard TCP/IP sockets or a DOCA Comch channel (depending on the test scenario/user preferences).

DOCA Bench tool is not intended to be used in a production deployment. If choosing to do so, please be aware that the out-of-band communications might contain sensitive information and thus should be done over a secure channel when using the standard TCP/IP sockets

16.3.9 CPU Core and Thread Selection

Selection of the correct CPU cores and threads has a significant impact on the performance or latency obtained. Read this section carefully.

A key requirement to scaling any application is the number of CPU cores or threads allocated to any given activity. DOCA Bench provides the ability to specify the numbers of cores, and the number of threads to be created per core, to maximize the number of jobs submitted to a given pipeline.

The following care should be given when selecting the number of CPU's or threads:

- Threads that are on cores located on distant NUMA regions (i.e., not the same NUMA region the BlueField is connected to) will experience lower performance and higher latency
- Core 0 is often most used by the OS and should be avoided
- Standard Linux Kernel installations allow the OS to move processes on any CPU core resulting in unexpected drops in performance, or higher latency, due to process switching

The selection of CPU cores is provided through the `--core-mask` , `--core-list` , `--core-count` parameters, while thread selection is made via the `--threads-per-core` parameter.

16.3.10 Device Selection

When executing from a host (x86) environment DOCA Bench can target one or more BlueField devices within an installed environment. When executing from the BlueField Arm, the target is always the local BlueField.

The default method of targeting a given BlueField from either the host or the BlueField Arm is using the `--device` or `-A` parameters, which can be provided as:

- Device PCIe address (i.e., `03:00.0`);
- Device IB name (`mlx5_0`); or
- Device interface name (`ens4f0`)

From the BlueField Arm environment, DOCA Bench should be targeted at the local PCIe address (i.e., `--device 03:00.0`) or the IB device name (i.e., `mlx5_0`).

16.3.11 Input Data Selection and Sizing of Jobs

DOCA Bench supports different methods of supplying data to jobs and providing information on the amount of data to process per job. These are referred to as "Data Providers".

16.3.11.1 Input Data Selection

The following subsections provide the modes available to provide data for input into any operation.

16.3.11.1.1 File

A single file is used as input to the operation. The contents of the file are not important for certain operations (e.g., DMA, SHA, etc.) but must be valid and specific for others (e.g., decompress, etc). The data may be used multiple times and repeated if the operations required more data than the single file contains. For more information on how file data is handled in complex operations, see section "Command-line Parameters".

16.3.11.1.2 File Sets

File sets are a group of files that are primarily used for structured data. The data in the file set is effectively a list of files, separated by a new line that is used sequentially as input data for jobs. Each file pointed to by the file set would have its entire contents read into a single buffer. This is useful for operations that require structured data (i.e., a complete valid block of data, such as decompression or AES).

16.3.11.1.3 Random Data

Random data is provided when the actual data required for the given operation is not specific (e.g., DMA).

⚠ The use of random data for certain operations may reduce the maximum performance obtained. For example, compressing random data results in lower performance than compressing actual file data (due to the lack of repeating patterns in random data).

16.3.11.2 Job Sizing

Each job in DOCA Bench consists of three buffers: An original input buffer, an output, and an intermediate buffer.

The input buffer is provided by the data provider for the first step in the pipeline to use, after which the following steps use the output and intermediate buffers (can be sized by using `--job-output-buffer-size`) in a ping-pong fashion. This means, the pipeline can always start with the same deterministic data while allowing for each step to provide its newly generated output data to be used as input to the next step.

The input buffer is specified in one of two ways: using uniform-job-size to make every input buffer the exact same size, or using a file set to size each buffer based on the size of the selected input data file(s). Users should ensure the data generated by each step in the pipeline will fit in the provided output buffer.

16.3.12 Controlling Test Duration

DOCA Bench has a variety of ways to control the length of executing tests—whether based on data or time limit.

16.3.12.1 Limit to Specific Number of Seconds

Using the `--run-limit-seconds` or `-s` parameter ensures that the execution continues for a specific number of seconds.

16.3.12.2 Limited Through Total Number of Jobs

It may be desirable to measure a specific number of jobs passing through a pipeline. The `--run-limit-jobs` or `-J` parameter is used to specify the exact number of jobs submitted to the pipeline and allowed to complete before execution finishes.

16.3.13 GGA-specific Attributes

As DOCA Bench supports a wide range of both GGA and software based DOCA libraries, the ability to fine tune their invocation is important. Command-line parameters are generally used for configuration options that apply to all aspects of DOCA Bench, without being specific to a particular DOCA library.

Attributes are the method of providing configuration options to a particular DOCA Library, whilst some shared attributes exist the majority of libraries have specific attributes designed to control their specific behavior.

For example, the attribute `doca_ec.data_block_count` allows you to set the data block count for the DOCA EC library, whilst the attribute `doca_sha.algorithm` controls the selection of the SHA algorithm.

For a full list of support attributes, see the "Command-line Parameters" section.

> ⓘ Due to batching it is possible that more than the supplied jobs are executed.

16.3.14 Command-line Parameters

DOCA Bench allows users to specify a series of operations to be performed and then scale that workload across multiple CPU cores/threads to get an estimation of how that workload performs and some insight into which stage(s), if any, cause performance problems for them. The user can then modify various configuration properties to explore how issues can be tuned to better serve their need.

When running, DOCA Bench creates a number of execution threads with affinities to the specific CPU specified by the user. Each thread creates, uniquely for themselves, a jobs pool (with job data initialized by a data provider) and a pipeline of workload steps.

16.3.14.1 CPU Core and Thread Count Configuration

There are many factors involved when carrying out performance tests, one of these is the CPU selection:

- The user should consider NUMA regions when selecting which cores to use, as using a CPU which is distant from the device under test can impact the performance achievable
- The user may also wish to avoid core 0 as this is typically the default core for kernel interrupt handlers.

⚠ CPU core selection has an impact on the total memory footprint of the test. See section "Test Memory Footprint" for more details.

16.3.14.1.1 --core-mask

Default value: 0x02

Core mask is the simplest way to specify which cores to use but is limited in that it can only specify up to 32 CPUs (0-31). Usage example: `--core-mask 0xF001` selects CPU cores 0, 12, 13, 14, and 15.

16.3.14.1.2 --core-list

Core list can specify any/all CPU cores in a given system as a list, range, or combination of the two. Usage example: `--core-list 0,3,6-10` selects CPU cores 0, 3, 6, 7, 8, 9, and 10.

16.3.14.1.3 --core-count

The user can select the first N cores from a given core set (list or mask) if desired. Usage example: `--core-count N`.

ⓘ Sweep testing is supported. See section "Sweep Tests" for more details.

16.3.14.1.4 --threads-per-core -t

To test the impacts of contention within a single CPU core, the user can specify this value so that instead of only one thread being created per core, N threads are created with their affinity mask set to the given core for each core selected. For example, 3 cores and 2 threads per core create 6 threads total.

ⓘ Sweep testing is supported. See section "Sweep Tests" for more details.

16.3.14.2 Device Configuration

The test requires the use of at least one BlueField to execute. With remote system testing, a second device may be required.

16.3.14.2.1 --device -A

Specify the device to use from the perspective of the system under test. The value can be for any one of either the device PCIe address (e.g., `03:00.0`), the device IB device name (e.g., `mlx5_0`), or the device interface name (e.g., `ens4f0`).

16.3.14.2.2 --representor -R

This option is used only when performing remote memory operations between a BlueField device and its host using DOCA Comch. This is typically automated by the companion connection string but exists for some developer debug use-cases.

> This option used to be important before the companion connection string property was introduced but now is rarely used.

16.3.14.3 Input Data and Buffer Size Configuration

DOCA Bench supports multiple methods of acquiring data to use to initialize job buffers. The user can also configure the output/intermediate buffers associated with each job.

> Input data and buffer size configuration has an impact on the total memory footprint of the test. See section "Test Memory Footprint" for more details.

16.3.14.3.1 --data-provider -I

DOCA Bench supports several different input data sources:

- `file`
- `file-set`
- `random-data`

16.3.14.3.1.1 File Data Provider

The file data provider produces uniform/non-structured data buffers by using a single input file. The input data is stripped and or repeated to fill each data buffer as required, returning back to the start of the file each time it is exhausted to collect more data. This is desirable when the performance of the component(s) under test is meant to show different performance characteristics depending on the input data supplied.

For example, `doca_dma` and `doca_sha` would execute in constant time regardless of the input data. Whereas `doca_compress` would be faster with data with more duplication and slower for truly random data and would produce different output depending on the input data.

Example 1 – Small Input File with Large Buffers

Given a small input data (i.e., smaller than the data buffer size), the file contents are repeated until the buffer is filled and then continue onto the next buffer(s). So, if the input file contained the data `012345` and the user requested two 20-byte buffers, the buffers would appear as follows:

- `01234501234501234501`
- `23450123450123450123`

Example 2 – Large Input File with Smaller Buffers

Given a large input data (i.e., greater than the data buffer size), the file contents are distributed across the data buffers. If the the input file contained the data `0123456789abcdef` and the user requested three 12-byte buffers, the buffers would appear as follows:

- `0123456789ab`
- `cdef01234567`
- `89abcdef0123`

16.3.14.3.1.2 File Set Data Provider

The file set data provider produces structured data. The file set input file itself is a file containing one or more filenames (relative to the input `"command working directory (cwd)"` not relative to the file set file). Each file listed inside the file set would have its entire contents used as a job buffer. This is useful for operations where the data must be a complete valid data block for the operation to succeed like decompression with `doca_compress` or decryption with `doca_aes`.

Example – File Set and Its Contents

Given a file set in the `"command working directory (cwd)"` referring to `data_1.bin` and `data_2.bin` (one file name per line), and `data_1.bin` contains 33 bytes and `data_2.bin` contains 69 bytes, then the data required by the buffers would be filled with these two files in a round-robin manner until the buffers are full. Unlike uniform (non-structured) data each task can have different lengths.

16.3.14.3.1.3 Random-data Data Provider

The random data data provider provides uniform (non-structured) data from a random data source. Each buffer will have unique (pseudo) random bytes of content.

16.3.14.3.2 --data-provider-job-count

Default value: 128

Each thread in DOCA Bench has its own allocation of job data buffers to avoid memory contention issues. Users may select how many jobs should be created per thread using this parameter.

ⓘ Sweep testing is supported. See section "Sweep Tests" for more details.

16.3.14.3.3 --data-provider-input-file

For data providers which use an input file, the filename can be specified here. The filename is relative to the `input_cwd`.

> Sweep testing is supported. See section "Sweep Tests" for more details.

16.3.14.3.4 --uniform-job-size

Specify the size of uniform input buffers (in bytes) that should be created.

> Does not apply and should not be specified when using structured data input sources.

> Sweep testing is supported. See section "Sweep Tests" for more details.

16.3.14.3.5 --job-output-buffer-size

Default value: 16384

Specify the size of output/intermediate buffers (in bytes). Each job has 3 buffers: immutable input buffer and two output/intermediate buffers. This allows for a pipeline to mutate the data an infinite number of times throughout the pipeline, while allowing for it to be reset and re-used at the end and allowing any step to use the new mutated data created by the previous step.

16.3.14.3.6 --input-cwd -i

To ease configuration management, the user may opt to use a separate folder for the input data for a given scenario outside of the DOCA build/install directory.

> It is recommended to use relative file paths for the input files.

16.3.14.3.6.1 Example 1 - Running DOCA Bench from Current Working Directory

Considering a user executing DOCA Bench from `/home/bob/doca/build`, values specified in `--data-provider-input-file` and filenames within a file set would search relative to the shell's "command working directory (cwd)": `/home/bob/doca/build`. Their command might look something like:

```
doca_bench --data-provider file-set --data-provider-input-file my_file_set.txt
```

And assuming `my_file_set.txt` contains `data_1.bin`, the files that would be loaded by DOCA Bench after path resolution would be:

- `/home/bob/doca/build/my_file_set.txt`
- `/home/bob/doca/build/data_1.bin`

16.3.14.3.6.2 Example 2 – Running DOCA Bench from Another Directory

Considering the user executed that same test from one level up. Something like:

```
build/doca_bench --data-provider file-set --data-provider-input-file build/my_file_set.txt
```

The files to be loaded would be:

- `/home/bob/doca/build/my_file_set.txt`
- `/home/bob/doca/data_1.bin`

Notice how both files were loaded relative to the `"command working directory (cwd)"` and the data file was not loaded relative to the file set.

16.3.14.3.6.3 Example 3 – Example 2 Revisited Using input-cwd

The user can solve this easily by keeping all input files in a single directory and then referring to that directory using the parameter `input-cwd`. In this case, the command like may look something like:

```
build/doca_bench --data-provider file-set --data-provider-input-file my_file_set.txt --input-cwd build
```

Note that the value for `--data-provider-input-file` also changed to be relative to the new `"command working directory (cwd)"`.

The files loaded this time are back to being what is expected:

- `/home/bob/doca/build/my_file_set.txt`
- `/home/bob/doca/build/data_1.bin`

16.3.14.4 Test Execution Control

DOCA Bench supports multiple test modes and run execution limits to allow the user to configure the test type and duration.

16.3.14.4.1 --mode

Default value: throughput

Select which type of test is to be performed.

16.3.14.4.1.1 Throughput Mode

Throughput mode is optimized to increase the volume of data processed in a given period with little or no regard for latency impact. Throughput mode tries to keep each component under test as busy as possible. A summary of the bandwidth and job execution rate are provided as output.

16.3.14.4.1.2 Bulk-latency Mode

Bulk latency mode strikes a balance between throughout and latency, submitting a batch of jobs and waiting for them all to complete to measure the latency of each job. This mode uses a bucketing mechanism to allow DOCA Bench to handle many millions of jobs worth of results. DOCA Bench keeps a count of the number of jobs that complete within each bucket to allow it to run for long periods of time. A summery of the distribution of results with an ASCII histogram of the results are provided as output. The latency reported is the time taken between the first job submission (for a batch of jobs) until the final job response is received (for that same batch of jobs).

16.3.14.4.1.3 Precision-latency Mode

Precision latency mode executes one job at a time to allow DOCA Bench to calculate the minimum possible latency of the jobs. This causes the components which can process many jobs in parallel to be vastly underutilized and so greatly reduces bandwidth. As this mode records every result individually, it should not be used to execute more than several thousand jobs. Precision latency mode requires 8 bytes of storage for each result, so be mindful of the memory overhead of the number of jobs to be executed.

A statistical analysis including minimum, maximum, mean, median and some percentiles of the latency value are provided as output.

16.3.14.4.2 --latency-bucket-range

Default value: 100ms,10ms

Only applicable to bulk-latency mode. Allows the user to specify the starting value of the buckets, and the width of each bucket. There are 100 buckets of the given size and an under flow and overflow bucket for results that fall outside of the central range.

For example:

```
--latency-bucket-range 10us,100us
```

This would start with the lowest bucket measuring <10µs response times, then 100 buckets which are 100µs wide, and a final bucket for results taking longer than >10010µs.

16.3.14.5 Blocking Mode

DOCA supports two methods of waiting on completion of tasks:

- Busy-wait (or polling) mode
- Notification-driven mode

> Refer to "DOCA SDK Architecture" documentation for more information.

By default, DOCA Bench uses the busy-wait to ensure maximum bandwidth (and low latency) for any given pipeline and its tasks with high utilization of any allocated CPU resources.

As with all high-performance software, utilizing GGAs or hardware accelerators, performance is usually CPU-bound at smaller packet sizes (i.e., at smaller payload sizes, the CPU spends a long time generating tasks and dealing with completions). For larger packet sizes, the CPU submits less tasks, as each task contains more data, therefore it may easily submit more data than the GGA or hardware accelerator can accept, resulting in periods where the CPU is busy-waiting on completions before being able to submit further tasks.

To execute any tests using an "notification-driven mode", use the options detailed in the following subsections.

16.3.14.5.1 --use-blocking-mode

This option causes DOCA Bench to use the "notification-drive mode" method of waiting on task completion.

At smaller packet sizes, the benchmark may still be CPU bound.

16.3.14.5.2 --record-cpu-usage

If specified, this option reports CPU statistics for any CPU cores DOCA Bench is executing on. This provides guidance on how much CPU time is returned, and thus available to other processes or threads, should the "notification-driven" mode be active.

Short duration tests may not result in sufficient produced data to generate CPU usage statistics.

The statistics provided include min, max, median, and mean values for the CPU usage. Also included are a number of percentile results, showing 90th, 95th, and a number of 99th percentile values. Example output:

```
CPU Usage stats
        min:            25%
        max:            50%
        median:         50%
        mean:           45.8333%
        90th %ile:      50%
        95th %ile:      50%
        99th %ile:      50%
        99.9th %ile:    50%
        99.99th %ile:   50%
```

16.3.14.6 Execution Limits

By default, a test runs forever. This is typically undesirable so the user can specify a limit to the test.

Precision-latency mode only supports job limited execution.

16.3.14.6.1 --run-limit-seconds -s

Runs the test for N seconds as specified by the user.

16.3.14.6.2 --run-limit-jobs -J

Runs the test until at least N jobs have been submitted, then allowing in-flight jobs to complete before exiting. More jobs than N may be executed based on batch size.

16.3.14.6.3 --run-limit-bytes -b

Runs the test until at least N bytes of data have been submitted, then allowing in-flight jobs to complete before exiting. More data may be processed than desired if the limit is not a multiple of the job input buffer size.

16.3.14.7 Gather/Scatter Support

Gather support involved breaking incoming input data from a single buffer into multiple buffers, which are "gathered" into a single gather list. Currently only gather is supported.

16.3.14.7.1 --gather-value

Default value: 1

Specifies the partitioning of input data from a single buffer into a gather list. The value can be specified in two flavors:

- `--gather-value 4` - splits input buffers into 4 parts as evenly as possible with odd bytes in the last segment
- `--gather-value 4KiB` - splits buffers after each 4KB of data. See `doca_bench/utility/byte_unit.hpp` for the list of possible units.

16.3.14.8 Stats Output

16.3.14.8.1 --rt-stats-interval

By default, DOCA Bench emits the results of an iteration once it completes. The user can ask for transient snapshots of the stats as the test progresses by providing the `--rt-stats-interval` argument with a value representing the number of milliseconds between stat prints. The end-result of the run is still displayed as normal.

> ⚠ This may produce a large amount of console output.

16.3.14.8.2 --csv-output-file

DOCA Bench can produce an output file as part of its execution which can contain stats and the configuration values used to produce that stat. This is enabled by specifying the `--csv-output-file` argument with a file path as the value. Providing a value for this argument enables CSV stats output (in addition to the normal console output). When performing a sweep test, one line per iteration of the sweep test is populated.

By default, the CSV output contains every possible value. The user can tune this by applying a filter.

16.3.14.8.3 --csv-stats

Provide one or more filters (positive or negative) to tune which stats are displayed. The value for this argument is a comma-separated list of filter strings. Negative filters start with a minus sign ('`-`').

16.3.14.8.3.1 Example 1 - Emit Only Statistical Values (No Configuration Values)

```
--csv-stats "stats.*"
```

> The quotes around the `*` prevent the shell from interpreting it as a wild card for filenames in the command.

16.3.14.8.3.2 Example 2 - Emit Statistical Values and Some Configuration Values (Remove Attribute Values)

```
--csv-stats "stats.*,-attribute*
```

16.3.14.8.4 --csv-append-mode

Default: false

When enabled, DOCA Bench appends to a CSV file if it exists or creates a new one. It is assumed that all invocation uses the exact same set of output values. This is not verified by DOCA Bench. The user must ensure that all tests that append to the CSV use the same set of output values.

16.3.14.8.5 --csv-separate-dynamic-values

A special case which creates a non-standard CSV file. All values that are not supported by sweep tests are reported only once first, then a new line of headers for values emitted during the test, then a row for each test result. This is reserved for an internal use case and should not be relied upon by anyone else.

16.3.14.8.6 --enable-environment-information

Instructs DOCA Bench to collect some detailed system information as part of the test startup procedure which are then made available for output in the CSV. These also gather the same details from the companion side if the companion is in use.

> This collection can take a long time (up to a few minutes in some circumstances) to complete, so it is not recommended unless you know you need it.

16.3.14.9 Remote Memory Testing

Some libraries (e.g., `doca_dma`) support the use of remote memory. To enable this, the user can specify one or both of the remote memory flags `--use-remote-input-buffers` and `--use-remote-output-buffers`. This tells DOCA Bench to use the companion to create a remote mmap. This remote mmap is then used to create buffers that are submitted to the component under test.

> ⚠ These flags should be used with caution and an understanding that if the underlying components under test can support this scenario, there is no automated checking. It is user responsibility to ensure these are used appropriately.

16.3.14.9.1 --use-remote-input-buffers

Specifies that the memory used for the initial immutable job input buffers into a pipeline should be backed by an mmap on the remote side.

> ⚠ Requires the companion app to be configured.

16.3.14.9.2 --use-remote-output-buffers

Specifies that all output and translation buffers in use are backed by an mmap on the remote side.

> ⚠ Requires the companion app to be configured.

16.3.15 Network Options

16.3.15.1 --mtu-size

For use with `doca_rdma`. Value is an enum: `256B` `512B` `1KB` `2KB` `4KB` or `raw_eth`.

16.3.15.2 --receive-queue-size

For use with `doca_rdma`. Configure the RDMA RQ size independently of the SQ size.

16.3.15.3 --send-queue-size

For use with `doca_rdma`. Configure the RDMA SQ size independently of the RQ size.

16.3.15.4 DOCA Lib Configuration Options

16.3.15.4.1 --task-pool-size

Default value: 1024

Configure the maximum task pool size used when libraries initialize task pools.

16.3.15.5 Pipeline Configuration

DOCA Bench is based on a pipeline of operations, This allow for complex test scenarios where multiple components are tested in parallel. Currently only a single chain of operations in a pipeline is supported (but scaled across multiple cores or threads), future versions will allow for varied pipeline's per CPU core.

A pipeline is described as a series of steps. All steps have a few general characteristics:

- Step type: `doca_dma` , `doca_sha` , `doca_compress` , etc.
- An operation category - transformative or non-transformative
- An input data category - structured or non structured

Individual step types may also have some additional metadata information or configuration as defined on a per step basis.

Metadata examples:

- `doca_compress` requires an operation type: `compress` or `decompress`
- `doca_aes` requires an operation type: `encrypt` or `decrypt`
- `doca_ec` requires an operation type: `create` , `recover` or `update`
- `doca_rdma` requires a direction: `send` , `receive` or `bidir`

Configuration examples:

- `--pipeline-steps doca_dma`
- `--pipeline-steps doca_compress::compress,doca_compress::decompress`

16.3.15.5.1 --pipeline-steps

Define the step(s) (comma-separated list) to be executed by each thread of the test.

The following is the list of supported steps:

- `doca_compress::compress`
- `doca_compress::decompress`
- `doca_dma`
- `doca_ec::create`
- `doca_ec::recover`
- `doca_ec::update`
- `doca_sha`
- `doca_rdma::send`
- `doca_rdma::receive`
- `doca_rdma::bidir`
- `doca_aes_gcm::encrypt`
- `doca_aes_gcm::decrypt`
- `doca_cc::client_producer`
- `doca_cc::client_consumer`
- `doca_eth::rx`

- `doca_eth::tx`

> ⓘ Some modules may be unavailable if they were not compiled as part of DOCA when DOCA Bench was compiled.

16.3.15.5.2 --attribute

Some of the options are very niche or specific to a single step/mmo type, so they are defined simply as attributes instead of a unique command-line argument.

The following is the list of supported options:

- `doption.mmp.log_qp_depth`
- `doption.mmo.log.num_qps`
- `doption.companion_app.path`
- `doca_compress.algorithm`
- `doca_ec.matrix_count`
- `doca_ec.data_block_count`
- `doca_ec.redundancy_block_count`
- `doca_sha.algorithm`
- `doca_rdma.gid-index`
- `doca_eth.max_burst_size`
- `doca_eth.l3_chksum_offload`
- `doca_eth.l4_chksum_offload`

16.3.15.5.3 --warm-up-jobs

Default value: 100

Warm-up serves two purposes:

- Firstly, it runs N tasks in a round robin fashion to get the data path code, tasks memory, and tasks data buffers memory into the CPU caches before the measurement of the test begins
- Secondly, it uses `doca_task_try_submit` instead of `doca_task_submit` to validate the jobs. This validation is not desirable during the proper hot path as it costs time revalidating the task each execution.

The user should ensure their warmup count equals or exceeds the number of tasks being used per thread (see `--data-provider-job-count`).

16.3.15.6 Companion Configuration

Some tests require a remote system to function. For this purpose, DOCA Bench comes bundled with a companion application (this application is installed as part of the DOCA-for-Host or BlueField packages). The companion is responsible for providing services to DOCA Bench such as creating a `doca_mmap` on the remote side and exporting it for use with remote operations like `doca_dma` / `doca_sha`, or other `doca_libs` that support remote memory input buffers. DOCA Bench can also provide remote worker processes for libraries that require them such as `doca_rdma` and `doca_cc`. The companion is enabled by providing the `--companion-connection-string` argument. Companion remote workers are enabled by providing either of the arguments `--companion-core-list` or `--companion-core-mask`.

> DOCA Bench requires that an SSH key is configured to allow the user specified to SSH without a password to the remote system using the supplied address (to launch the companion). Refer to your OS's documentation for information on how to achieve this.

The companion connection may also specify the `no-launch` option.

> This is reserved for expert developer use.

The user may also specify a path to a specific companion binary to allow them to test companion binaries not in the default install path using the following command:

```
--attribute doption.companion_app.path=/tmp/my_doca_build/tools/bench/doca_bench_companion
```

> This is reserved for expert developer use.

16.3.15.6.1 --companion-connection-string

Specifies the details required to establish a connection to and execute the companion process.

- Example of running DOCA Bench from the host side using the BlueField for the remote side using `doca_comch` as the communications method:

  ```
  --companion-connection-string "proto=dcc,mode=DPU,user=bob,addr=172.17.0.1,dev=03:00.0,rep=d8:00.0"
  ```
- Example of running DOCA Bench from the BlueField side using the host for the remote side using `doca_comch` as the communications method:

  ```
  --companion-connection-string "proto=dcc,mode=host,user=bob,addr=172.17.0.1,dev=d8:00.0"
  ```
- Example of running DOCA Bench on one host with the companion on another host using TCP as the communications method:

  ```
  --companion-connection-string "proto=tcp,user=bob,addr=172.17.0.1,port=12345,dev=d8:00.0"
  ```

  > For `doca_rdma` only.

16.3.15.6.2 --companion-core-list

Works the same way as `--core-list` but defines the cores to be used on the companion side.

⚠ Must be at least as large as the `--core-list`.

16.3.15.6.3 --companion-core-mask

Works the same way as `--core-mask` but defines the cores to be used on the companion side.

⚠ Must be at least as large as the `--core-mask`.

16.3.15.7 Sweep Tests

16.3.15.7.1 --sweep

DOCA Bench supports executing a set of tests based on a number of value ranges. For example, to understand the performance of multi-threading, the user may wish to run the same test for various CPU core counts. They may also wish to vary more than one aspect of the test. Providing one or more `--sweep` parameters activates sweep test mode where every combination of values is tested with a single invocation of DOCA Bench.

The following is a list of the supported sweep test options:

- `core-count`
- `data-provider-input-file`
- `data-provider-job-count`
- `gather-value`
- `mtu-size`
- `receive-queue-size`
- `send-queue-size`
- `threads-per-core`
- `task-pool-size`
- `uniform-job-size`
- `doption.mmo.log_qp_depth`
- `doption.mmo.log_num_qps`
- `doca_rdma.transport-type`
- `doca_rdma.gid-index`

Sweep test argument values take one of three forms:

- `--sweep param,start_value,end_value,+N`
- `--sweep param,start_value,end_value,*N`
- `--sweep param,value1,...,valueN`

Sweep core count and input file example:

```
--sweep core-count,1,8,*2 -sweep data-provider-input-file,file1.bin,file2.bin
```

This would sweep cores 1-8, inclusive, multiplying the value each time as 1,2,4,8 and two different input files resulting in a cumulative 8 test cases:

Iteration Number	Core Count	Input File
1	1	`file1.bin`
2	2	`file1.bin`
3	4	`file1.bin`
4	8	`file1.bin`
5	1	`file2.bin`
6	2	`file2.bin`
7	4	`file2.bin`
8	8	`file2.bin`

16.3.15.8 Queries

16.3.15.8.1 Device Capabilities

DOCA Bench allows the querying of a device to report which step types are available as well as information of valid configuration options for each step. A device must be specified:

```
tools/bench/doca_bench --device 03:00.0 --query device-capabilities
```

For each supported library, this would report:

- Capable – if that library is enabled in DOCA Bench at compile time (if not capable, installing the library would not make it become available to bench)
- Installed – if the library is installed on the machine executing the query (if not installed, installing it would make it available to bench)
- Library wide attributes
- A list of supported task types (~= step name)
    - If the task type is supported
    - Task specific attributes/capabilities

```
doca_compress:
    Capable: yes
    Installed: yes
    Tasks:
        compress::deflate:
            Supported: no
        compress::lz4:
            Supported: no
        compress::lz4_stream:
            Supported: no
        decompress::deflate:
            Supported: yes
            Max buffer length: 134217728
        decompress::lz4:
            Supported: yes
            Max buffer length: 134217728
        decompress::lz4_stream:
            Supported: yes
            Max buffer length: 134217728
```

16.3.15.8.2 Supported Sweep Attributes

Shows the possible parameters that can be used with the sweep test parameter

```
tools/bench/doca_bench --query sweep-properties
```

Example output:

```
Supported query properties: [
        core-count
        threads-per-core
        uniform-job-size
        task-pool-size
        data-provider-job-count
        gather-value
        mtu-size
        send-queue-size
        receive-queue-size
        doption.mmo.log_qp_depth
        doption.mmo.log_num_qps
        doca_rdma.transport-type
        doca_rdma.gid-index
]
```

16.3.16 Test Memory Footprint

DOCA Bench allocates memory for all the tasks required by the test based on the input buffer size, output/intermediate buffer size, number of cores, number of threads, and number of jobs in use. All jobs contain an input buffer, an output buffer, and an intermediate buffer. The input buffer is immutable and sized based on the data provider in use. The output and intermediate buffers are sized based on the users specification or automatically calculated at the users request. For a library which produces the same amount of output as it consumes (e.g., `doca_dma`), typically the user should set the buffers all to the same size to make things as efficient as possible.

The memory footprint for job buffers can be calculated as: `(number-of-tasks) * (number-of-cores) * (number-of-threads-per-core) * (input-buffer-size + (output/intermediate-buffer-size * 2))`. For a 1KB job with the default of 32 jobs, 1 core, and 1 core per thread, the memory footprint would be 96KB.

For sweep testing and structured data input, it can be difficult to pick a suitable output buffer size so the user may choose to specify 0 and have DOCA Bench try all the tasks once to calculate the required output buffer sizes. This only has a cost in terms of time taken to perform the calculation. After this, there is no difference between auto-sizing and manually sizing the jobs output buffers.

> When running DOCA Bench on the BlueField and on some host OSs, it may be necessary to increase the limit of how much memory the process can acquire. Consult your OS's documentation for details of how to do this.

16.3.17 DOCA Bench Sample Invocations

16.3.17.1 Overview

This guide provides examples of various invocations of the tool to help provide guidance and insight into it and the feature under test.

To make the samples clearer, certain verbose output and repeated information has been removed or shortened, in particular to output of the configuration or defaults when DOCA Bench is first executed is removed.

The command line options may need to be updated to suit your environment (e.g., TCP addresses, port numbers, interface names, usernames). See the "Command-line Parameters" section for more information.

16.3.17.2 DOCA Eth Receive Sample

- This test invokes DOCA Bench to run in Ethernet receive mode, configured to receive Ethernet frames of size 1500 bytes.
- The test runs for 3 seconds using a single core and use a maximum burst size of 512 frames.
- The test runs in the default throughput mode, with throughput figures displayed at the end of the test run.
- The companion application uses 6 cores to continuously transmit Ethernet frames of size 1500 bytes until it is stopped by DOCA Bench.

16.3.17.2.1 Command Line

```
doca_bench --core-mask 0x02 \
                                --pipeline-steps doca_eth::rx \
                                    --device b1:00.1 \
                                    --data-provider random-data \
                                    --uniform-job-size 1500 \
                                    --run-limit-seconds 3 \
                                    --attribute doca_eth.max-burst-size=512 \
                                    --companion-connection-string proto=tcp,addr=10.10.10.10,port=12345,user=bob,de
v=ens4f1np1 \
                                    --attribute doption.companion_app.path=/opt/mellanox/doca/tools/
doca_bench_companion \
                                    --companion-core-list 6 \
                                    --job-output-buffer-size 1500 \
                                    --mtu-size raw_eth
```

16.3.17.2.2 Results Output

```
[main] doca_bench : 2.7.0084
[main] release build
+ + + + + + + + + + + + + + + + + + + + + + + + + +
DOCA bench supported modules: [doca_comm_channel, doca_compress, doca_dma, doca_ec, doca_eth, doca_sha, doca_comch,
doca_rdma, doca_aes_gcm]
+ + + + + + + + + + + + + + + + + + + + + + + + + +

DOCA bench configuration
Static configuration: [
        Attributes: [doca_eth.l4-chksum-offload:false, doca_eth.max-burst-size:512, doption.companion_app.path:/
opt/mellanox/doca/tools/doca_bench_companion, doca_eth.l3-chksum-offload:false]
        Companion configuration: [
                Device: ens4f1np1
                Remote IP address: "bob@10.10.10.10"
                Core set: [6]
        ]
        Pipelines: [
                Steps: [
                        name: "doca_eth::rx"
                        attributes: []
                ]
                Use remote input buffers: no
                Use remote output buffers: no
                Latency bucket_range: 10000ns-110000ns
        ]
        Run limits: [
                Max execution time: 3seconds
                Max jobs executed: -- not configured --
                Max bytes processed: -- not configured --
        ]
        Data provider: [
                Name: "random-data"
                Job output buffer size: 1500
        ]
        Device: "b1:00.1"
        Device representor: "-- not configured --"
        Warm up job count: 100
        Input files dir: "-- not configured --"
        Output files dir: "-- not configured --"
        Core set: [1]
        Benchmark mode: throughput
        Warnings as errors: no
        CSV output: [
                File name: -- not configured --
                Selected stats: []
                Deselected stats: []
                Separate dynamic values: no
                Collect environment information: no
                Append to stats file: no
        ]
]
Test permutations: [
        Attributes: []
        Uniform job size: 1500
        Core count: 1
        Per core thread count: 1
        Task pool size: 1024
        Data provider job count: 128
        MTU size: ETH_FRAME
        SQ depth: -- not configured --
        RQ depth: -- not configured --
        Input data file: -- not configured --
]

[main] Initialize framework...
[main] Start execution...
Preparing...
EAL: Detected CPU lcores: 36
EAL: Detected NUMA nodes: 4
EAL: Detected shared linkage of DPDK
EAL: Multi-process socket /run/user/48679/dpdk/rte/mp_socket
EAL: Selected IOVA mode 'PA'
EAL: VFIO support initialized
TELEMETRY: No legacy callbacks, legacy socket not created
EAL: Probe PCI driver: mlx5_pci (15b3:a2d6) device: 0000:b1:00.1 (socket 2)
[08:19:32:110524][398304][DOCA][WRN][engine_model.c:90][adapt_queue_depth] adapting queue depth to 128.
Executing...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Cleanup...
[main] Completed! tearing down...
Aggregate stats
        Duration:      3000633 micro seconds
        Enqueued jobs: 611215
        Dequeued jobs: 611215
        Throughput:    000.204 MOperations/s
        Ingress rate:  002.276 Gib/s
        Egress rate:   002.276 Gib/s
```

16.3.17.2.3 Results Overview

As a single core is specified, there is a single section of statistics output displayed.

16.3.17.3 DOCA Eth Send Sample

- This test invokes DOCA Bench to run in Ethernet send mode, configured to transmit Ethernet frames of size 1500 bytes
- Random data is used to populate the Ethernet frames
- The test runs for 3 seconds using a single core and uses a maximum burst size of 512 frames
- L3 and L4 checksum offloading is not enabled
- The test runs in the default throughput mode, with throughput figures displayed at the end of the test run
- The companion application uses 6 cores to continuously receive Ethernet frames of size 1500 bytes until it is stopped by DOCA Bench

16.3.17.3.1 Command Line

```
doca_bench --core-mask 0x02 \
                                    --pipeline-steps doca_eth::tx \
                                    --device b1:00.1 \
                                    --data-provider random-data \
                                    --uniform-job-size 1500 \
                                    --run-limit-seconds 3 \
                                    --attribute doca_eth.max-burst-size=512 \
                                    --attribute doca_eth.l4-chksum-offload=false \
                                    --attribute doca_eth.l3-chksum-offload=false \
                                    --companion-connection-string proto=tcp,addr=10.10.10.10,port=12345,user=bob,de
v=ens4f1np1 \
                                    --attribute doption.companion_app.path=/opt/mellanox/doca/tools/
doca_bench_companion \
                                    --companion-core-list 6 \
                                    --job-output-buffer-size 1500
```

16.3.17.3.2 Results Output

```
[main] doca_bench : 2.7.0084
[main] release build
+ + + + + + + + + + + + + + + + + + + + + + + + + +
DOCA bench supported modules: [doca_comm_channel, doca_compress, doca_dma, doca_ec, doca_eth, doca_sha, doca_comch,
doca_rdma, doca_aes_gcm]
+ + + + + + + + + + + + + + + + + + + + + + + + + +

DOCA bench configuration
Static configuration: [
        Attributes: [doca_eth.l4-chksum-offload:false, doca_eth.max-burst-size:512, doption.companion_app.path:/
opt/mellanox/doca/tools/doca_bench_companion, doca_eth.l3-chksum-offload:false]
        Companion configuration: [
                Device: ens4f1np1
                Remote IP address: "bob@10.10.10.10"
                Core set: [6]
        ]
        Pipelines: [
                Steps: [
                        name: "doca_eth::tx"
                        attributes: []
                ]
                Use remote input buffers: no
                Use remote output buffers: no
                Latency bucket_range: 10000ns-110000ns
        ]
        Run limits: [
                Max execution time: 3seconds
                Max jobs executed: -- not configured --
                Max bytes processed: -- not configured --
        ]
        Data provider: [
                Name: "random-data"
                Job output buffer size: 1500
        ]
        Device: "b1:00.1"
        Device representor: "-- not configured --"
        Warm up job count: 100
        Input files dir: "-- not configured --"
        Output files dir: "-- not configured --"
        Core set: [1]
        Benchmark mode: throughput
        Warnings as errors: no
        CSV output: [
                File name: -- not configured --
                Selected stats: []
                Deselected stats: []
                Separate dynamic values: no
                Collect environment information: no
                Append to stats file: no
        ]
]
Test permutations: [
        Attributes: []
        Uniform job size: 1500
        Core count: 1
        Per core thread count: 1
        Task pool size: 1024
        Data provider job count: 128
        MTU size: -- not configured --
        SQ depth: -- not configured --
        RQ depth: -- not configured --
        Input data file: -- not configured --
]

[main] Initialize framework...
[main] Start execution...
Preparing...
Executing...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Cleanup...
[main] Completed! tearing down...
Aggregate stats
        Duration:      3000049 micro seconds
        Enqueued jobs: 17135128
        Dequeued jobs: 17135128
        Throughput:    005.712 MOperations/s
        Ingress rate:  063.832 Gib/s
        Egress rate:   063.832 Gib/s
```

16.3.17.3.3 Results Overview

As a single core is specified, there is a single section of statistics output displayed.

16.3.17.4 Host-side AES-GCM Decrypt Sample

- This test invokes DOCA Bench on the x86 host side to run the AES-GM Decryption step
- A file-set file is used to indicate which file is to be decrypted. The content of the file-set file lists the filename to be decrypted.
- The key to be used for the encryption and decryption is specified using the `doca_aes_gcm.key` -file attribute. This contains the key to be used.
- It will run until 5000 jobs have been processed
- It runs in the precision-latency mode, with latency and throughput figures displayed at the end of the test run
- A core mask is specified to indicate that cores 12, 13, 14, and 15 are to be used for this test

16.3.17.4.1 Command Line

```
doca_bench  --mode precision-latency \
            --core-mask 0xf000 \
            --warm-up-jobs 32 \
            --device 17:00.0 \
            --data-provider file-set \
            --data-provider-input-file aes_64_128.fileset \
            --run-limit-jobs 5000 \
            --pipeline-steps doca_aes_gcm::decrypt \
            --attribute doca_aes_gcm.key-file='aes128.key' \
            --job-output-buffer-size 80
```

16.3.17.4.2 Results Output

```
[main] Completed! tearing down...
Worker thread[0](core: 12) stats:
        Duration:      10697 micro seconds
        Enqueued jobs: 5000
        Dequeued jobs: 5000
        Throughput:    000.467 MOperations/s
        Ingress rate:  000.265 Gib/s
        Egress rate:   000.223 Gib/s
Worker thread[1](core: 13) stats:
        Duration:      10700 micro seconds
        Enqueued jobs: 5000
        Dequeued jobs: 5000
        Throughput:    000.467 MOperations/s
        Ingress rate:  000.265 Gib/s
        Egress rate:   000.223 Gib/s
Worker thread[2](core: 14) stats:
        Duration:      10733 micro seconds
        Enqueued jobs: 5000
        Dequeued jobs: 5000
        Throughput:    000.466 MOperations/s
        Ingress rate:  000.264 Gib/s
        Egress rate:   000.222 Gib/s
Worker thread[3](core: 15) stats:
        Duration:      10788 micro seconds
        Enqueued jobs: 5000
        Dequeued jobs: 5000
        Throughput:    000.463 MOperations/s
        Ingress rate:  000.262 Gib/s
        Egress rate:   000.221 Gib/s
Aggregate stats
        Duration:      10788 micro seconds
        Enqueued jobs: 20000
        Dequeued jobs: 20000
        Throughput:    001.854 MOperations/s
        Ingress rate:  001.050 Gib/s
        Egress rate:   000.884 Gib/s
        min:           1878 ns
        max:           4956 ns
        median:        2134 ns
        mean:          2145 ns
        90th %ile:     2243 ns
        95th %ile:     2285 ns
        99th %ile:     2465 ns
        99.9th %ile:   3193 ns
        99.99th %ile:  4487 ns
```

16.3.17.4.3 Results Overview

Since a core mask is specified but no core count, then all cores in the mask are used.

There is a section of statistics displayed for each core used as well as the aggregate statistics.

16.3.17.5 BlueField-side AES-GCM Encrypt Sample

- This test invokes DOCA Bench on the BlueField side to run the AES-GM encryption step
- A text file of size 2KB is the input for the encryption stage
- The key to be used for the encryption and decryption is specified using the `doca_aes_gcm.key` attribute
- It runs until 2000 jobs have been processed
- It runs in the bulk-latency mode, with latency and throughput figures displayed at the end of the test run
- A single core is specified with 2 threads

16.3.17.5.1 Command Line

```
doca_bench  --mode bulk-latency \
            --core-list 3 \
            --threads-per-core 2 \
            --warm-up-jobs 32 \
            --device 03:00.0 \
            --data-provider file \
            --data-provider-input-file plaintext_2k.txt \
            --run-limit-jobs 2000 \
            --pipeline-steps doca_aes_gcm::encrypt \
            --attribute doca_aes_gcm.key="0123456789abcdef0123456789abcdef" \
            --uniform-job-size 2048 \
            --job-output-buffer-size 4096
```

16.3.17.5.2 Results Output

```
[main] Completed! tearing down...
Worker thread[0](core: 3) stats:
        Duration:      501 micro seconds
        Enqueued jobs: 2048
        Dequeued jobs: 2048
        Throughput:    004.082 MOperations/s
        Ingress rate:  062.279 Gib/s
        Egress rate:   062.644 Gib/s
Worker thread[1](core: 3) stats:
        Duration:      466 micro seconds
        Enqueued jobs: 2048
        Dequeued jobs: 2048
        Throughput:    004.386 MOperations/s
        Ingress rate:  066.922 Gib/s
        Egress rate:   067.314 Gib/s
Aggregate stats
        Duration:      501 micro seconds
        Enqueued jobs: 4096
        Dequeued jobs: 4096
        Throughput:    008.163 MOperations/s
        Ingress rate:  124.558 Gib/s
        Egress rate:   125.287 Gib/s
Latency report:
                   :
                   :
                   :
                   :
                   :
                   ::
                   ::
                   ::
                   ::
                  .::.  .    .    ..
------------------------------------------------------------------------------------------------------
```

```
[<10000ns]: 0
.. OUTPUT RETRACTED (SHORTENED) ..
[26000ns -> 26999ns]: 0
[27000ns -> 27999ns]: 128
[28000ns -> 28999ns]: 2176
[29000ns -> 29999ns]: 1152
[30000ns -> 30999ns]: 128
[31000ns -> 31999ns]: 0
[32000ns -> 32999ns]: 0
[33000ns -> 33999ns]: 128
[34000ns -> 34999ns]: 0
[35000ns -> 35999ns]: 0
[36000ns -> 36999ns]: 0
[37000ns -> 37999ns]: 0
[38000ns -> 38999ns]: 128
[39000ns -> 39999ns]: 0
[40000ns -> 40999ns]: 0
[41000ns -> 41999ns]: 0
[42000ns -> 42999ns]: 0
[43000ns -> 43999ns]: 128
[44000ns -> 44999ns]: 128
[45000ns -> 45999ns]: 0
.. OUTPUT RETRACTED (SHORTENED) ..
[>110000ns]: 0
```

16.3.17.5.3 Results Overview

Since a single core is specified, there is a single section of statistics output displayed.

16.3.17.6 Host-side AES-GCM Encrypt and Decrypt Sample

- This test invokes DOCA Bench on the host side to run 2 AES-GM steps in the pipeline, first to encrypt a text file and then to decrypt the associated output from the encrypt step
- A text file of size 2KB is the input for the encryption stage
- The `input-cwd` option instructs DOCA Bench to look in a different location for the input file, in the parent directory in this case
- The key to be used for the encryption and decryption is specified using the `doca_aes_gcm.key`-file attribute, indicating that the key can be found in the specified file
- It runs until 204800 bytes have been processed
- It runs in the default throughput mode, with throughput figures displayed at the end of the test run

16.3.17.6.1 Command Line

```
doca_bench  --core-mask 0xf00 \
            --core-count 1 \
            --warm-up-jobs 32 \
            --device 17:00.0 \
            --data-provider file \
            --input-cwd ../. \
            --data-provider-input-file plaintext_2k.txt \
            --run-limit-bytes 204800 \
            --pipeline-steps doca_aes_gcm::encrypt,doca_aes_gcm::decrypt \
            --attribute doca_aes_gcm.key-file='aes128.key' \
            --uniform-job-size 2048 \
            --job-output-buffer-size 4096
```

16.3.17.6.2 Results Output

```
Executing...
Worker thread[0](core: 8) [doca_aes_gcm::encrypt>>doca_aes_gcm::decrypt] started...
Worker thread[0] Executing 32 warm-up tasks using 32 unique tasks
Cleanup...
[main] Completed! tearing down...
Aggregate stats
        Duration:      79 micro seconds
        Enqueued jobs: 214
        Dequeued jobs: 214
        Throughput:    002.701 MOperations/s
        Ingress rate:  041.214 Gib/s
        Egress rate:   041.214 Gib/s
```

16.3.17.6.3 Results Overview

Since a single core is specified, there is a single section of statistics output displayed.

16.3.17.7 Host-side SHA with CSV Output File Sample

- This test invokes DOCA Bench on the host side to execute the SHA operation using the SHA256 algorithm and to create a CSV file containing the test configuration and statistics
- A list of 1 core is provided with a count of 2 threads per core

16.3.17.7.1 Command Line

```
doca_bench  --core-mask 2 \
            --threads-per-core 2 \
            --pipeline-steps doca_sha \
            --device d8:00.0 \
            --data-provider random-data \
            --uniform-job-size 2048 \
            --job-output-buffer-size 2048 \
            --run-limit-seconds 3 \
            --attribute doca_sha.algorithm=sha256 \
            --warm-up-jobs 100 \
            --csv-output-file /tmp/sha_256_test.csv
```

16.3.17.7.2 Results Output

```
Executing...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Data path thread [1] started...
WT[1] Executing 100 warm-up tasks using 100 unique tasks
Cleanup...
[main] Completed! tearing down...
Stats for thread[0](core: 1)
        Duration:      3000064 micro seconds
        Enqueued jobs: 3713935
        Dequeued jobs: 3713935
        Throughput:    001.238 MOperations/s
        Ingress rate:  018.890 Gib/s
        Egress rate:   000.295 Gib/s
Stats for thread[1](core: 1)
        Duration:      3000056 micro seconds
        Enqueued jobs: 3757335
        Dequeued jobs: 3757335
        Throughput:    001.252 MOperations/s
        Ingress rate:  019.110 Gib/s
        Egress rate:   000.299 Gib/s
Aggregate stats
        Duration:      3000064 micro seconds
        Enqueued jobs: 7471270
        Dequeued jobs: 7471270
        Throughput:    002.490 MOperations/s
        Ingress rate:  038.000 Gib/s
        Egress rate:   000.594 Gib/s
```

16.3.17.7.3 Results Overview

As a single core has been specified with a thread count of 2, there are statistics displayed for each thread as well as the aggregate statistics.

It can also be observed that 2 threads are started on core 1 with each thread executing the warm-up jobs.

The contents of the `/tmp/sha_256_test.csv` are shown below. It can be seen that the configuration used for the test and the associated statistics from the test run are listed:

```
cfg.companion.connection_string,cfg.pipeline.steps,cfg.pipeline.use_remote_input_buffers,cfg.pipeline.use_remote_ou
tput_buffers,cfg.pipeline.bulk_latency.lower_bound,cfg.pipeline.bulk_latency.bucket_width,cfg.run_limit.duration,cf
g.r
un_limit.jobs,cfg.run_limit.bytes,cfg.data_provider.type,cfg.data_provider.output_buffer_size,cfg.device.pci_addres
s,cfg.input.cwd,cfg.output.cwd,cfg.warmup_job_count,cfg.core_set,cfg.benchmark_mode,cfg.warnings_are_errors,cfg.att
rib
ute.doca_compress.algorithm,cfg.attribute.doca_ec.matrix_type,cfg.attribute.doca_ec.data_block_count,cfg.attribute.
doca_ec.redundancy_block_count,cfg.attribute.doca_ec.use_precomputed_matrix,cfg.attribute.doca_eth.l3_chksum_offloa
d,c
fg.attribute.doca_eth.l4_chksum_offload,cfg.attribute.doca_sha.algorithm,cfg.uniform_job_size,cfg.core_count,cfg.pe
r_core_thread_count,cfg.task_pool_size,cfg.data_provider_job_count,cfg.sg_config,cfg.mtu-size,cfg.send-queue-
size,cfg.    receive-queue-size,cfg.data-provider-input-
file,cfg.attribute.mmo.log_qp_depth,cfg.attribute.mmo.log_num_qps,stats.input.job_count,stats.output.job_count,stat
s.input.byte_count,stats.output.byte_count,stats.input.throughput.bytes,sta
ts.output.throughput.bytes,stats.input.throughput.rate,stats.output.throughput.rate
,[doca_sha],0,0,10000,1000,3,,,random-data,2048,d8:00.0,,,100,[1],throughput,0,,,,,,,,sha256,2048,1,2,1024,128,1 fr
agments,,,,,,,7471270,7471270,15301160960,239109312,038.000 Gib/s,000.594 Gib/s,2.490370 MOperations/s,2.490370 MOp
era    tions/s
```

16.3.17.8 Host-side SHA with CSV Appended Output File Sample

- This test invokes DOCA Bench on the Host side to execute the SHA operation using the SHA512 algorithm and to create a csv file containing the test configuration and statistics,
- The command is repeated with the added option of csv-append-mode. This instructs DOCA Bench to append the test run statistics to the existing csv file.
- A list of 1 core is provided with a count of 2 threads per core.

16.3.17.8.1 Command Line

1. Create the initial `/tmp/sha_512_test.csv` file:

   ```
   doca_bench    --core-list 2 \
                   --threads-per-core 2 \
                   --pipeline-steps doca_sha \
                   --device d8:00.0 \
                   --data-provider random-data \
                   --uniform-job-size 2048 \
                   --job-output-buffer-size 2048 \
                   --run-limit-seconds 3 \
                   --attribute doca_sha.algorithm=sha512 \
                   --warm-up-jobs 100 \
                   --csv-output-file /tmp/sha_512_test.csv
   ```

2. The second command is:

   ```
   ./doca_bench    --core-list 2 \
                   --threads-per-core 2 \
                   --pipeline-steps doca_sha \
                   --device d8:00.0 \
                   --data-provider random-data \
                   --uniform-job-size 2048 \
                   --job-output-buffer-size 2048 \
                   --run-limit-seconds 3 \
                   --attribute doca_sha.algorithm=sha512 \
                   --warm-up-jobs 100 \
                   --csv-output-file /tmp/sha_512_test.csv \
                   --csv-append-mode
   ```

   This causes DOCA Bench to append the configuration and statistics from the second command run to the `/tmp/sha_512_test.csv` file.

16.3.17.8.2 Results Output

This is a snapshot of the results output from the first command run:

```
Executing...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Data path thread [1] started...
WT[1] Executing 100 warm-up tasks using 100 unique tasks
Cleanup...
[main] Completed! tearing down...
Stats for thread[0](core: 2)
        Duration:      3015185 micro seconds
```

```
        Enqueued jobs: 3590717
        Dequeued jobs: 3590717
        Throughput:    001.191 MOperations/s
        Ingress rate:  018.171 Gib/s
        Egress rate:   000.568 Gib/s
Stats for thread[1](core: 2)
        Duration:      3000203 micro seconds
        Enqueued jobs: 3656044
        Dequeued jobs: 3656044
        Throughput:    001.219 MOperations/s
        Ingress rate:  018.594 Gib/s
        Egress rate:   000.581 Gib/s
Aggregate stats
        Duration:      3015185 micro seconds
        Enqueued jobs: 7246761
        Dequeued jobs: 7246761
        Throughput:    002.403 MOperations/s
        Ingress rate:  036.673 Gib/s
        Egress rate:   001.146 Gib/s
```

This is a snapshot of the results output from the second command run:

```
Executing...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Data path thread [1] started...
WT[1] Executing 100 warm-up tasks using 100 unique tasks
Cleanup...
[main] Completed! tearing down...
Stats for thread[0](core: 2)
        Duration:      3000072 micro seconds
        Enqueued jobs: 3602562
        Dequeued jobs: 3602562
        Throughput:    001.201 MOperations/s
        Ingress rate:  018.323 Gib/s
        Egress rate:   000.573 Gib/s
Stats for thread[1](core: 2)
        Duration:      3000062 micro seconds
        Enqueued jobs: 3659148
        Dequeued jobs: 3659148
        Throughput:    001.220 MOperations/s
        Ingress rate:  018.611 Gib/s
        Egress rate:   000.582 Gib/s
Aggregate stats
        Duration:      3000072 micro seconds
        Enqueued jobs: 7261710
        Dequeued jobs: 7261710
        Throughput:    002.421 MOperations/s
        Ingress rate:  036.934 Gib/s
        Egress rate:   001.154 Gib/s
```

16.3.17.8.3 Results Overview

Since a single core has been specified with a thread count of 2, there are statistics displayed for each thread as well as the aggregate statistics.

It can also be observed that 2 threads are started on core 1 with each thread executing the warm-up jobs.

The contents of the `/tmp/sha_256_test.csv`, after the first command has been run, are shown below. It can be seen that the configuration used for the test and the associated statistics from the test run are listed:

```
cfg.companion.connection_string,cfg.pipeline.steps,cfg.pipeline.use_remote_input_buffers,cfg.pipeline.use_remote_ou
tput_buffers,cfg.pipeline.bulk_latency.lower_bound,cfg.pipeline.bulk_latency.bucket_width,cfg.run_limit.duration,cf
g.run_limit.jobs,cfg.run_limit.bytes,cfg.data_provider.type,cfg.data_provider.output_buffer_size,cfg.device.pci_add
ress,cfg.input.cwd,cfg.output.cwd,cfg.warmup_job_count,cfg.core_set,cfg.benchmark_mode,cfg.warnings_are_errors,cfg.
attribute.doca_compress.algorithm,cfg.attribute.doca_ec.matrix_type,cfg.attribute.doca_ec.data_block_count,cfg.attr
ibute.doca_ec.redundancy_block_count,cfg.attribute.doca_ec.use_precomputed_matrix,cfg.attribute.doca_eth.l3_chksum_
offload,cfg.attribute.doca_eth.l4_chksum_offload,cfg.attribute.doca_sha.algorithm,cfg.uniform_job_size,cfg.core_cou
nt,cfg.per_core_thread_count,cfg.task_pool_size,cfg.data_provider_job_count,cfg.sg_config,cfg.mtu-size,cfg.send-
queue-size,cfg.receive-queue-size,cfg.data-provider-input-
file,cfg.attribute.mmo.log_qp_depth,cfg.attribute.mmo.log_num_qps,stats.input.job_count,stats.output.job_count,stat
s.input.byte_count,stats.output.byte_count,stats.input.throughput.bytes,stats.output.throughput.bytes,stats.input.t
hroughput.rate,stats.output.throughput.rate
,[doca_sha],0,0,10000,1000,3,,,random-data,2048,d8:00.0,,,100,[2],throughput,0,,,,,,,,sha512,2048,1,2,1024,128,1 fr
agments,,,,,,,7246761,7246761,14841366528,463850048,036.673 Gib/s,001.146 Gib/s,2.403422 MOperations/s,2.403422 MOp
erations/s
```

The contents of the `/tmp/sha_256_test.csv`, after the second command has been run, are shown below. It can be seen that a second entry has been added detailing the configuration used for the test and the associated statistics from the test run:

```
cfg.companion.connection_string,cfg.pipeline.steps,cfg.pipeline.use_remote_input_buffers,cfg.pipeline.use_remote_ou
tput_buffers,cfg.pipeline.bulk_latency.lower_bound,cfg.pipeline.bulk_latency.bucket_width,cfg.run_limit.duration,cf
g.run_limit.jobs,cfg.run_limit.bytes,cfg.data_provider.type,cfg.data_provider.output_buffer_size,cfg.device.pci_add
ress,cfg.input.cwd,cfg.output.cwd,cfg.warmup_job_count,cfg.core_set,cfg.benchmark_mode,cfg.warnings_are_errors,cfg.
attribute.doca_compress.algorithm,cfg.attribute.doca_ec.matrix_type,cfg.attribute.doca_ec.data_block_count,cfg.attr
ibute.doca_ec.redundancy_block_count,cfg.attribute.doca_ec.use_precomputed_matrix,cfg.attribute.doca_eth.l3_chksum_
offload,cfg.attribute.doca_eth.l4_chksum_offload,cfg.attribute.doca_sha.algorithm,cfg.uniform_job_size,cfg.core_cou
nt,cfg.per_core_thread_count,cfg.task_pool_size,cfg.data_provider_job_count,cfg.sg_config,cfg.mtu-size,cfg.send-
queue-size,cfg.receive-queue-size,cfg.data-provider-input-
file,cfg.attribute.mmo.log_qp_depth,cfg.attribute.mmo.log_num_qps,stats.input.job_count,stats.output.job_count,stat
s.input.byte_count,stats.output.byte_count,stats.input.throughput.bytes,stats.output.throughput.bytes,stats.input.t
hroughput.rate,stats.output.throughput.rate
,[doca_sha],0,0,10000,1000,3,,,random-data,2048,d8:00.0,,,100,[2],throughput,0,,,,,,,,sha512,2048,1,2,1024,128,1 fr
agments,,,,,,,7246761,7246761,14841366528,463850048,036.673 Gib/s,001.146 Gib/s,2.403422 MOperations/s,2.403422 MOp
erations/s
,[doca_sha],0,0,10000,1000,3,,,random-data,2048,d8:00.0,,,100,[2],throughput,0,,,,,,,,sha512,2048,1,2,1024,128,1 fr
agments,,,,,,,7261710,7261710,14871982080,464806784,036.934 Gib/s,001.154 Gib/s,2.420512 MOperations/s,2.420512 MOp
erations/s
```

16.3.17.9 BlueField-side SHA with Transient Statistics Sample

- This test invokes DOCA Bench on the BlueField side to execute the SHA operation using the SHA1 algorithm and to display statistics every 2000 milliseconds during the test run
- A list of 3 cores is provided with a count of 2 threads per core and a core-count of 1
- The core-count instructs DOCA Bench to use the first core number in the core list, in this case core number 2

16.3.17.9.1 Command Line

```
doca_bench    --core-list 2,3,4 \
                --core-count 1 \
                --threads-per-core 2 \
                --pipeline-steps doca_sha \
                --device 03:00.0 \
                --data-provider random-data \
                --uniform-job-size 2048 \
                --job-output-buffer-size 2048 \
                --run-limit-seconds 3 \
                -attribute doca_sha.algorithm=sha1 \
                --warm-up-jobs 100 \
                --rt-stats-interval 2000
```

16.3.17.9.2 Results Output

```
Executing...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Data path thread [1] started...
WT[1] Executing 100 warm-up tasks using 100 unique tasks
Stats for thread[0](core: 2)
        Duration:      965645 micro seconds
        Enqueued jobs: 1171228
        Dequeued jobs: 1171228
        Throughput:    001.213 MOperations/s
        Ingress rate:  018.505 Gib/s
        Egress rate:   000.181 Gib/s
Stats for thread[1](core: 2)
        Duration:      965645 micro seconds
        Enqueued jobs: 1171754
        Dequeued jobs: 1171754
        Throughput:    001.213 MOperations/s
        Ingress rate:  018.514 Gib/s
        Egress rate:   000.181 Gib/s
Aggregate stats
        Duration:      965645 micro seconds
        Enqueued jobs: 2342982
        Dequeued jobs: 2342982
        Throughput:    002.426 MOperations/s
        Ingress rate:  037.019 Gib/s
        Egress rate:   000.362 Gib/s
Stats for thread[0](core: 2)
        Duration:      2968088 micro seconds
        Enqueued jobs: 3653691
        Dequeued jobs: 3653691
        Throughput:    001.231 MOperations/s
        Ingress rate:  018.783 Gib/s
        Egress rate:   000.183 Gib/s
Stats for thread[1](core: 2)
        Duration:      2968088 micro seconds
        Enqueued jobs: 3689198
        Dequeued jobs: 3689198
        Throughput:    001.243 MOperations/s
        Ingress rate:  018.965 Gib/s
        Egress rate:   000.185 Gib/s
Aggregate stats
        Duration:      2968088 micro seconds
        Enqueued jobs: 7342889
        Dequeued jobs: 7342889
        Throughput:    002.474 MOperations/s
        Ingress rate:  037.748 Gib/s
        Egress rate:   000.369 Gib/s
Cleanup...
[main] Completed! tearing down...
Stats for thread[0](core: 2)
        Duration:      3000122 micro seconds
        Enqueued jobs: 3694128
        Dequeued jobs: 3694128
        Throughput:    001.231 MOperations/s
        Ingress rate:  018.789 Gib/s
        Egress rate:   000.184 Gib/s
Stats for thread[1](core: 2)
        Duration:      3000089 micro seconds
        Enqueued jobs: 3751128
        Dequeued jobs: 3751128
        Throughput:    001.250 MOperations/s
        Ingress rate:  019.079 Gib/s
        Egress rate:   000.186 Gib/s
Aggregate stats
        Duration:      3000122 micro seconds
        Enqueued jobs: 7445256
        Dequeued jobs: 7445256
        Throughput:    002.482 MOperations/s
        Ingress rate:  037.867 Gib/s
        Egress rate:   000.370 Gib/s
```

16.3.17.9.3 Results Overview

Although a core list of 3 cores has been specified, the core-count value of 1 instructs DOCA Bench to use the first entry in the core list.

It can be seen that as a thread-count of 2 has been specified, there are 2 threads created on core 2.

A transient statistics interval of 2000 milliseconds has been specified, and the transient statistics per thread can be seen, as well as the final aggregate statistics.

16.3.17.10 Host-side Local DMA with Core Sweep Sample

- This test invokes DOCA Bench to execute a local DMA operation on the host
- It specifies that a core sweep should be carried out using core counts of 1, 2, and 4 using the option `--sweep core-count,1,4,*2`
- Test output is to be saved in a CSV file `/tmp/dma_sweep.csv` and a filter is applied so that only statistics information is recorded. No configuration information is to be recorded.

16.3.17.10.1 Command Line

```
doca_bench    --core-mask 0xff \
                --sweep core-count,1,4,*2 \
                --pipeline-steps doca_dma \
                --device d8:00.0 \
                --data-provider random-data \
                --uniform-job-size 2048 \
                --job-output-buffer-size 2048 \
                --run-limit-seconds 5 \
                --csv-output-file /tmp/dma_sweep.csv \
                --csv-stats "stats.*"
```

16.3.17.10.2 Results Overview

```
Test permutations: [
        Attributes: []
        Uniform job size: 2048
        Core count: 1
        Per core thread count: 1
        Task pool size: 1024
        Data provider job count: 128
        MTU size: -- not configured --
        SQ depth: -- not configured --
        RQ depth: -- not configured --
        Input data file: -- not configured --
        --------------------------------
        Attributes: []
        Uniform job size: 2048
        Core count: 2
        Per core thread count: 1
        Task pool size: 1024
        Data provider job count: 128
        MTU size: -- not configured --
        SQ depth: -- not configured --
        RQ depth: -- not configured --
        Input data file: -- not configured --
        --------------------------------
        Attributes: []
        Uniform job size: 2048
        Core count: 4
        Per core thread count: 1
        Task pool size: 1024
        Data provider job count: 128
        MTU size: -- not configured --
        SQ depth: -- not configured --
        RQ depth: -- not configured --
        Input data file: -- not configured --
]

[main] Initialize framework...
[main] Start execution...
Preparing permutation 1 of 3...
Executing permutation 1 of 3...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Cleanup permutation 1 of 3...
Aggregate stats
        Duration:      5000191 micro seconds
        Enqueued jobs: 22999128
        Dequeued jobs: 22999128
        Throughput:    004.600 MOperations/s
        Ingress rate:  070.185 Gib/s
        Egress rate:   070.185 Gib/s
Preparing permutation 2 of 3...
Executing permutation 2 of 3...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Data path thread [1] started...
WT[1] Executing 100 warm-up tasks using 100 unique tasks
Cleanup permutation 2 of 3...
Stats for thread[0](core: 0)
        Duration:      5000066 micro seconds
        Enqueued jobs: 14409794
        Dequeued jobs: 14409794
        Throughput:    002.882 MOperations/s
        Ingress rate:  043.975 Gib/s
        Egress rate:   043.975 Gib/s
Stats for thread[1](core: 1)
        Duration:      5000188 micro seconds
        Enqueued jobs: 14404708
        Dequeued jobs: 14404708
        Throughput:    002.881 MOperations/s
        Ingress rate:  043.958 Gib/s
        Egress rate:   043.958 Gib/s
Aggregate stats
        Duration:      5000188 micro seconds
        Enqueued jobs: 28814502
        Dequeued jobs: 28814502
        Throughput:    005.763 MOperations/s
        Ingress rate:  087.932 Gib/s
        Egress rate:   087.932 Gib/s
Preparing permutation 3 of 3...
Executing permutation 3 of 3...
Data path thread [1] started...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
WT[1] Executing 100 warm-up tasks using 100 unique tasks
Data path thread [3] started...
WT[3] Executing 100 warm-up tasks using 100 unique tasks
Data path thread [2] started...
WT[2] Executing 100 warm-up tasks using 100 unique tasks
Cleanup permutation 3 of 3...
[main] Completed! tearing down...
Stats for thread[0](core: 0)
        Duration:      5000092 micro seconds
        Enqueued jobs: 7227025
        Dequeued jobs: 7227025
        Throughput:    001.445 MOperations/s
        Ingress rate:  022.055 Gib/s
        Egress rate:   022.055 Gib/s
```

```
Stats for thread[1](core: 1)
        Duration:       5000081 micro seconds
        Enqueued jobs: 7223269
        Dequeued jobs: 7223269
        Throughput:     001.445 MOperations/s
        Ingress rate:   022.043 Gib/s
        Egress rate:    022.043 Gib/s
Stats for thread[2](core: 2)
        Duration:       5000047 micro seconds
        Enqueued jobs: 7229678
        Dequeued jobs: 7229678
        Throughput:     001.446 MOperations/s
        Ingress rate:   022.063 Gib/s
        Egress rate:    022.063 Gib/s
Stats for thread[3](core: 3)
        Duration:       5000056 micro seconds
        Enqueued jobs: 7223037
        Dequeued jobs: 7223037
        Throughput:     001.445 MOperations/s
        Ingress rate:   022.043 Gib/s
        Egress rate:    022.043 Gib/s
Aggregate stats
        Duration:       5000092 micro seconds
        Enqueued jobs: 28903009
        Dequeued jobs: 28903009
        Throughput:     005.780 MOperations/s
        Ingress rate:   088.203 Gib/s
        Egress rate:    088.203 Gib/s
```

16.3.17.10.3 Results Overview

The output gives a summary of the permutations being carried out and then proceeds to display the statistics for each of the permutations.

The CSV output file contents can be seen to contain only statistics information. Configuration information is not included.

There is an entry for each of the sweep permutations:

```
stats.input.job_count,stats.output.job_count,stats.input.byte_count,stats.output.byte_count,stats.input.throughput.
bytes,stats.output.throughput.bytes,stats.input.throughput.rate,stats.output.throughput.rate
22999128,22999128,47102214144,47102214144,070.185 Gib/s,070.185 Gib/s,4.599650 MOperations/s,4.599650 MOperations/s
28814502,28814502,59012100096,59012100096,087.932 Gib/s,087.932 Gib/s,5.762683 MOperations/s,5.762683 MOperations/s
28903009,28903009,59193362432,59193362432,088.203 Gib/s,088.203 Gib/s,5.780495 MOperations/s,5.780495 MOperations/s
```

16.3.17.11 Host-side Local DMA with Job Size Sweep Sample

This test invokes DOCA Bench to execute a local DMA operation on the host.

It specifies that a uniform job size sweep should be carried out using job sizes 1024 and 2048 using the option `--sweep uniform-job-size,1024,2048`.

Test output is to be saved in a CSV file `/tmp/dma_sweep_job_size.csv` and collection of environment information is enabled.

16.3.17.11.1 Command Line

```
doca_bench    --core-mask 0xff \
                --core-count 1 \
                --pipeline-steps doca_dma \
                --device d8:00.0 \
                --data-provider random-data \
                --sweep uniform-job-size,1024,2048 \
                --job-output-buffer-size 2048 \
                --run-limit-seconds 5 \
                --csv-output-file /tmp/dma_sweep_job_size.csv \
                --enable-environment-information
```

16.3.17.11.2 Results Overview

```
Test permutations: [
        Attributes: []
        Uniform job size: 1024
        Core count: 1
        Per core thread count: 1
        Task pool size: 1024
        Data provider job count: 128
        MTU size: -- not configured --
        SQ depth: -- not configured --
        RQ depth: -- not configured --
        Input data file: -- not configured --
        --------------------------------
        Attributes: []
        Uniform job size: 2048
        Core count: 1
        Per core thread count: 1
        Task pool size: 1024
        Data provider job count: 128
        MTU size: -- not configured --
        SQ depth: -- not configured --
        RQ depth: -- not configured --
        Input data file: -- not configured --
]

[main] Initialize framework...
[main] Start execution...
Preparing permutation 1 of 2...
Executing permutation 1 of 2...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Cleanup permutation 1 of 2...
Aggregate stats
        Duration:      5000083 micro seconds
        Enqueued jobs: 23645128
        Dequeued jobs: 23645128
        Throughput:    004.729 MOperations/s
        Ingress rate:  036.079 Gib/s
        Egress rate:   036.079 Gib/s
Preparing permutation 2 of 2...
Executing permutation 2 of 2...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Cleanup permutation 2 of 2...
[main] Completed! tearing down...
Aggregate stats
        Duration:      5000027 micro seconds
        Enqueued jobs: 22963128
        Dequeued jobs: 22963128
        Throughput:    004.593 MOperations/s
        Ingress rate:  070.078 Gib/s
        Egress rate:   070.078 Gib/s
```

16.3.17.11.3 Results Overview

The output gives a summary of the permutations being carried out and then proceeds to display the statistics for each of the permutations.

The CSV output file contents can be seen to contain statistics information and the environment information.

There is an entry for each of the sweep permutations.

```
cfg.companion.connection_string,cfg.pipeline.steps,cfg.pipeline.use_remote_input_buffers,cfg.pipeline.use_remote_ou
tput_buffers,cfg.pipeline.bulk_latency.lower_bound,cfg.pipeline.bulk_latency.bucket_width,cfg.run_limit.duration,cf
g.run_limit.jobs,cfg.run_limit.bytes,cfg.data_provider.type,cfg.data_provider.output_buffer_size,cfg.device.pci_add
ress,cfg.input.cwd,cfg.output.cwd,cfg.warmup_job_count,cfg.core_set,cfg.benchmark_mode,cfg.warnings_are_errors,cfg.
attribute.doca_compress.algorithm,cfg.attribute.doca_ec.matrix_type,cfg.attribute.doca_ec.data_block_count,cfg.attr
ibute.doca_ec.redundancy_block_count,cfg.attribute.doca_ec.use_precomputed_matrix,cfg.attribute.doca_eth.l3_chksum_
offload,cfg.attribute.doca_eth.l4_chksum_offload,cfg.attribute.doca_sha.algorithm,cfg.uniform_job_size,cfg.core_cou
nt,cfg.per_core_thread_count,cfg.task_pool_size,cfg.data_provider_job_count,cfg.sg_config,cfg.mtu-size,cfg.send-
queue-size,cfg.receive-queue-size,cfg.data-provider-input-
file,cfg.attribute.mmo.log_qp_depth,cfg.attribute.mmo.log_num_qps,stats.input.job_count,stats.output.job_count,stat
s.input.byte_count,stats.output.byte_count,stats.input.throughput.bytes,stats.output.throughput.bytes,stats.input.t
hroughput.rate,stats.output.throughput.rate,host.pci.3.address,host.pci.3.ext_tag,host.pci.3.link_type,host.pci.2.e
xt_tag,host.pci.2.address,host.cpu.0.model,host.ofed_version,host.pci.4.max_read_request,host.pci.2.width,host.cpu.
1.logical_cores,host.pci.2.eswitch_mode,host.pci.3.max_read_request,host.pci.4.address,host.pci.2.link_type,host.pc
i.1.max_read_request,host.pci.4.link_type,host.cpu.socket_count,host.pci.0.ext_tag,host.pci.6.port_speed,host.cpu.0
.physical_cores,host.pci.7.port_speed,host.memory.dimm_slot_count,host.cpu.1.model,host.pci.0.max_payload_size,host
.pci.6.relaxed_ordering,host.doca_host_package_version,host.pci.6.max_payload_size,host.pci.0.gen,host.pci.4.width,
host.pci.2.gen,host.pci.1.max_payload_size,host.pci.4.relaxed_ordering,host.pci.3.width,host.cpu.0.logical_cores,ho
st.cpu.0.arch,host.pci.4.port_speed,host.pci.4.eswitch_mode,host.pci.7.address,host.pci.5.eswitch_mode,host.pci.5.a
ddress,host.cpu.1.arch,host.pci.0.eswitch_mode,host.pci.7.width,host.pci.7.link_type,host.pci.1.link_type,host.pci.
3.gen,host.pci.7.max_read_request,host.pci.7.eswitch_mode,host.pci.6.gen,host.pci.2.port_speed,host.pci.7.gen,host.
pci.2.relaxed_ordering,host.pci.6.width,host.pci.4.gen,host.pci.6.address,host.hostname,host.pci.5.link_type,host.p
ci.6.link_type,host.pci.6.max_read_request,host.pci.7.max_payload_size,host.pci.5.gen,host.pci.6.eswitch_mode,host.
pci.5.width,host.pci.3.relaxed_ordering,host.pci.4.ext_tag,host.pci.0.width,host.pci.5.port_speed,host.pci.2.max_pa
```

```
yload_size,host.pci.3.max_payload_size,host.pci.5.max_payload_size,host.pci.2.max_read_request,host.pci.0.address,h
ost.pci.gen,host.os.family,host.pci.1.gen,host.pci.5.relaxed_ordering,host.pci.1.port_speed,host.pci.7.ext_tag,host
.pci.1.address,host.pci.3.eswitch_mode,host.pci.3.port_speed,host.pci.0.max_read_request,host.pci.1.ext_tag,host.pc
i.0.relaxed_ordering,host.pci.0.link_type,host.pci.5.max_read_request,host.pci.4.max_payload_size,host.pci.device_c
ount,host.memory.populated_dimm_count,host.memory.installed_capacity,host.pci.6.ext_tag,host.os.kernel_version,host
.pci.0.port_speed,host.pci.1.width,host.pci.7.relaxed_ordering,host.pci.1.relaxed_ordering,host.os.version,host.os.
name,host.cpu.1.physical_cores,host.numa_node_count,host.pci.5.ext_tag,host.pci.1.eswitch_mode
,[doca_dma],0,0,10000,1000,5,,,random-data,2048,d8:00.0,,,100,"[0, 1, 2, 3, 4, 5, 6, 7]",throughput,0,,,,,,,,,1024,1,1
024,128,1 fragments,,,,,,,23645128,23645128,24212611072,24212611072,036.079 Gib/s,036.079 Gib/s,4.728947 MOperation
s/s,4.728947 MOperations/s,0000:5e:00.1,true,Infiniband,true,0000:5e:00.0,N/A,OFED-internal-24.04-0.4.8,N/A,x63,N/
A,N/A,N/A,0000:af:00.0,Infiniband,N/A,Ethernet,2,true,N/A,N/A,N/A,N/A,N/A,N/A,true,<none>,N/A,Gen15,x63,Gen15,N/A,t
rue,x63,N/A,x86_64,104857600000,N/A,0000:d8:00.1,N/A,0000:af:00.1,x86_64,N/A,x63,Ethernet,Infiniband,Gen15,N/A,N/
A,Gen15,N/A,Gen15,true,x63,Gen15,0000:d8:00.0,zibal,Ethernet,Ethernet,N/A,N/A,Gen15,N/A,x63,true,true,x63,104857600
000,N/A,N/A,N/A,N/A,0000:3b:00.0,N/A,Linux,Gen15,true,N/A,true,0000:3b:00.1,N/A,N/A,N/A,true,true,Infiniband,N/A,N/
A,8,N/A,270049112064,true,5.4.0-174-generic,N/A,x63,true,true,20.04.1 LTS (Focal Fossa),Ubuntu,N/A,2,true,N/A
,[doca_dma],0,0,10000,1000,5,,,random-data,2048,d8:00.0,,,100,"[0, 1, 2, 3, 4, 5, 6, 7]",throughput,0,,,,,,,,,2048,1,1
024,128,1 fragments,,,,,,,22963128,22963128,47028486144,47028486144,070.078 Gib/s,070.078 Gib/s,4.592600 MOperation
s/s,4.592600 MOperations/s,0000:5e:00.1,true,Infiniband,true,0000:5e:00.0,N/A,OFED-internal-24.04-0.4.8,N/A,x63,N/
A,N/A,N/A,0000:af:00.0,Infiniband,N/A,Ethernet,2,true,N/A,N/A,N/A,N/A,N/A,N/A,true,<none>,N/A,Gen15,x63,Gen15,N/A,t
rue,x63,N/A,x86_64,104857600000,N/A,0000:d8:00.1,N/A,0000:af:00.1,x86_64,N/A,x63,Ethernet,Infiniband,Gen15,N/A,N/
A,Gen15,N/A,Gen15,true,x63,Gen15,0000:d8:00.0,zibal,Ethernet,Ethernet,N/A,N/A,Gen15,N/A,x63,true,true,x63,104857600
000,N/A,N/A,N/A,N/A,0000:3b:00.0,N/A,Linux,Gen15,true,N/A,true,0000:3b:00.1,N/A,N/A,N/A,true,true,Infiniband,N/A,N/
A,8,N/A,270049112064,true,5.4.0-174-generic,N/A,x63,true,true,20.04.1 LTS (Focal Fossa),Ubuntu,N/A,2,true,N/A
```

16.3.17.12 BlueField-side Remote DMA Sample

- This test invokes DOCA Bench to execute a remote DMA operation on the host
- It specifies the companion connection details to be used on the host and that remote output buffers are to be used

16.3.17.12.1 Command Line

```
doca_bench  --core-list 12 \
            --pipeline-steps doca_dma \
            --device 03:00.0 \
            --data-provider random-data \
            --uniform-job-size 2048 \
            --job-output-buffer-size 2048 \
            --use-remote-output-buffers \
            --companion-connection-string proto=tcp,port=12345,mode=host,dev=17:00.0,user=bob,addr=10.10.10.10 \
            --run-limit-seconds 5
```

16.3.17.12.2 Results Overview

```
Executing...
Worker thread[0](core: 12) [doca_dma] started...
Worker thread[0] Executing 100 warm-up tasks using 100 unique tasks
Cleanup...
[main] Completed! tearing down...
Aggregate stats
        Duration:      5000073 micro seconds
        Enqueued jobs: 32202128
        Dequeued jobs: 32202128
        Throughput:    006.440 MOperations/s
        Ingress rate:  098.272 Gib/s
        Egress rate:   098.272 Gib/s
```

16.3.17.12.3 Results Overview

None.

16.3.17.13 Compress BlueField-side Sample

⚠ This test is relevant for BlueField-2 only.

- This test invokes DOCA Bench to run compression using random data as input
- The compression algorithm specified is "deflate"

16.3.17.13.1 Command Line

```
doca_bench    --core-list 2 \
                --pipeline-steps doca_compress::compress \
                --device 03:00.0 \
                --data-provider random-data \
                --uniform-job-size 2048 \
                --job-output-buffer-size 4096 \
                --run-limit-seconds 3 \
                --attribute doca_compress.algorithm="deflate"
```

16.3.17.13.2 Result Output

```
Executing...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Cleanup...
[main] Completed! tearing down...
Aggregate stats
        Duration:      3000146 micro seconds
        Enqueued jobs: 5340128
        Dequeued jobs: 5340128
        Throughput:    001.780 MOperations/s
        Ingress rate:  027.160 Gib/s
        Egress rate:   027.748 Gib/s
```

16.3.17.13.3 Results Overview

None

16.3.17.14 BlueField-side Decompress LZ4 Sample

- This test invokes DOCA Bench to run decompression using random data as input
- This test specifies a data provider of file set which contains the filename of an LZ4 compressed file
- Remote input buffers are specified to be used for the input jobs
- It specifies the companion connection details to be used on the host for the remote input buffers

16.3.17.14.1 Command Line

```
doca_bench --core-list 12 \
            --pipeline-steps doca_compress::decompress \
            --device 03:00.0 \
            --data-provider file-set \
            --data-provider-input-file lz4_compressed_64b_buffers.fs \
            --job-output-buffer-size 4096 \
            --run-limit-seconds 3 \
            --attribute doca_compress.algorithm="lz4" \
            --use-remote-output-buffers \
            --companion-connection-string proto=tcp,port=12345,mode=host,dev=17:00.0,user=bob,addr=10.10.10.10
```

16.3.17.14.2 Results Output

```
Executing...
Worker thread[0](core: 12) [doca_compress::decompress] started...
Worker thread[0] Executing 100 warm-up tasks using 100 unique tasks
Cleanup...
[main] Completed! tearing down...
Aggregate stats
        Duration:      3000043 micro seconds
        Enqueued jobs: 15306128
        Dequeued jobs: 15306128
        Throughput:    005.102 MOperations/s
        Ingress rate:  003.155 Gib/s
        Egress rate:   002.433 Gib/s
```

16.3.17.14.3 Results Comment

None

16.3.17.15 Host-side EC Creation in Bulk Latency Mode Sample

- This test invokes DOCA Bench to run the EC creation step.
- It runs in bulk latency mode and specifies the `doca_ec` attributes of `data_block_count`, `redundancy_block_count`, and `matrix_type`

16.3.17.15.1 Command Line

```
doca_bench --mode bulk-latency \
            --core-list 12 \
            --pipeline-steps doca_ec::create \
            --device 17:00.0 \
            --data-provider random-data \
            --uniform-job-size 1024 \
            --job-output-buffer-size 1024 \
            --run-limit-seconds 3 \
            --attribute doca_ec.data_block_count=16 \
            --attribute doca_ec.redundancy_block_count=16 \
            --attribute doca_ec.matrix_type=cauchy
```

16.3.17.15.2 Results Output

Bulk latency output will be similar to that presented in section "BlueField-side Decompress LZ4 Sample".

16.3.17.15.3 Results Comment

Bulk latency output will be similar to that presented earlier on this page.

16.3.17.16 BlueField-side EC Creation in Precision Latency Mode Sample

- This test invokes DOCA Bench to run the EC creation step
- It runs in precision latency mode and specifies the `doca_ec` attributes of `data_block_count`, `redundancy_block_count`, and `matrix_type`

16.3.17.16.1 Command Line

```
doca_bench    --mode precision-latency \
                --core-list 12 \
                --pipeline-steps doca_ec::create \
                --device 03:00.0 \
                --data-provider random-data \
                --uniform-job-size 1024 \
                --job-output-buffer-size 1024 \
                --run-limit-jobs 5000 \
                --attribute doca_ec.data_block_count=16 \
                --attribute doca_ec.redundancy_block_count=16 \
                --attribute doca_ec.matrix_type=cauchy
```

16.3.17.16.2 Results Output

None

16.3.17.16.3 Results Comment

Precision latency output will be similar to that presented earlier on this page.

16.3.17.17 Comch Consumer from Host Side Sample

- This test invokes DOCA Bench in Comch consumer mode using a core-list on host side and BlueField side
- The run-limit is 500 jobs

16.3.17.17.1 Command Line

```
./doca_bench --core-list 4 --warm-up-jobs 32 --pipeline-steps doca_comch::consumer --device ca:00.0 --data-provider
random-data --run-limit-jobs 500 --core-count 1 --uniform-job-size 4096 --job-output-buffer-size 4096 --companion-
connection-string proto=tcp,mode=dpu,dev=03:00.0,user=bob,addr=10.10.10.10,port=12345 --attribute
dopt.companion_app.path=<path to DPU doca_bench_companion application location> --data-provider-job-count 256 --
companion-core-list 12
```

16.3.17.17.2 Results Output

```
[main] Completed! tearing down...
Aggregate stats
        Duration:      1415 micro seconds
        Enqueued jobs: 500
        Dequeued jobs: 500
        Throughput:    000.353 MOperations/s
        Ingress rate:  000.000 Gib/s
        Egress rate:   010.782 Gib/s
```

16.3.17.17.3 Results Comment

The aggregate statistics show the test completed after 500 jobs were processed.

16.3.17.18 Host-side Comch Producer Sample

- This test invokes DOCA Bench in Comch producer mode using a core-mask on the host side and BlueField side
- The run-limit is 1000 jobs

16.3.17.18.1 Command Line

```
doca_bench    --core-list 4 \
                --warm-up-jobs 32 \
                --pipeline-steps doca_comch::producer \
                --device ca:00.0 \
                --data-provider random-data \
                --run-limit-jobs 500 \
                --core-count 1 \
                --uniform-job-size 4096 \
                --job-output-buffer-size 4096 \
                --companion-connection-string proto=tcp,mode=dpu,dev=03:00.0,user=bob,addr=10.10.10.10,port=12345 \
                --attribute dopt.companion_app.path=<path to DPU doca_bench_companion location> \
                --data-provider-job-count 256 \
                --companion-core-list 12
```

16.3.17.18.2 Results Overview

```
[main] Completed! tearing down...
Aggregate stats
        Duration:      407 micro seconds
        Enqueued jobs: 500
        Dequeued jobs: 500
        Throughput:    001.226 MOperations/s
        Ingress rate:  037.402 Gib/s
        Egress rate:   000.000 Gib/s
```

16.3.17.18.3 Results Comment

The aggregate statistics show the test completed after 500 jobs were processed.

16.3.17.19 Host-side RDMA Send Sample

- This test invokes DOCA Bench in RDMA send mode using a core-list on the send and receive side
- The send queue size is configured to 50 entries

16.3.17.19.1 Command Line

```
doca_bench  --pipeline-steps doca_rdma::send \
            --device d8:00.0 \
            --data-provider random-data \
            --uniform-job-size 2048 \
            --job-output-buffer-size 2048 \
            --run-limit-seconds 3 \
            --send-queue-size 50 \
            --companion-connection-string proto=tcp,addr=10.10.10.10,port=12345,user=bob,dev=ca:00.0 \
            --companion-core-list 12 \
            --core-list 12
```

16.3.17.19.2 Results Output

```
Test permutations: [
        Attributes: []
        Uniform job size: 2048
        Core count: 1
        Per core thread count: 1
        Task pool size: 1024
        Data provider job count: 128
        MTU size: -- not configured --
        SQ depth: 50
        RQ depth: -- not configured --
        Input data file: -- not configured --
]
```

16.3.17.19.3 Results Comment

The configuration output shows the send queue size configured to 50.

16.3.17.20 Host-side RDMA Receive Sample

- This test invokes DOCA Bench in RDMA receive mode using a core-list on the send and receive side
- The receive queue size is configured to 100 entries

16.3.17.20.1 Command Line

```
doca_bench  --pipeline-steps doca_rdma::receive \
            --device d8:00.0 \
            --data-provider random-data \
            --uniform-job-size 2048 \
            --job-output-buffer-size 2048 \
            --run-limit-seconds 3 \
            --receive-queue-size 100 \
            --companion-connection-string proto=tcp,addr=10.10.10.10,port=12345,user=bob,dev=ca:00.0 \
            --companion-core-list 12 \
            --core-list 12
```

16.3.17.20.2 Results Output

```
Test permutations: [
        Attributes: []
        Uniform job size: 2048
        Core count: 1
        Per core thread count: 1
        Task pool size: 1024
        Data provider job count: 128
        MTU size: -- not configured --
        SQ depth: -- not configured --
        RQ depth: 100
        Input data file: -- not configured --
]
```

16.3.17.20.3 Results Overview

The configuration output shows the receive queue size configured to 100.

16.4 NVIDIA DOCA Capabilities Print Tool

This document provides instruction on the usage of the DOCA Capabilities Print Tool.

16.4.1 Introduction

This tool is used to print all the available DOCA libraries and devices. For each DOCA device, the tool prints its representor devices and the capabilities it supports in each DOCA library.

16.4.2 Prerequisites

DOCA 2.6.0 and higher.

16.4.3 Description

This tool can be executed on the host or Arm sides.

The following capabilities are supported by this tool:

- DOCA device list - print the PCIe device of every available DOCA device and its capabilities
- DOCA representor device list - for every DOCA device, print the PCIe device of every available DOCA representor device and its capabilities
- DOCA library list - print the available DOCA libraries supported by the running OS and their availability for specific OSs

- DOCA library capabilities – for every DOCA device, print the capabilities it supports in every DOCA library

16.4.4 Execution

- To print all the available DOCA devices and their capabilities, run:

```
/opt/mellanox/doca/tools/doca_caps --list-devs
```

Printing the capabilities of a specific DOCA device can be done using the `--pci-addr` flag.

Example output:

```
/opt/mellanox/doca/tools/doca_caps --list-devs
PCI: 0000:03:00.0
        ibdev_name                                    mlx5_0
        iface_name                                    p0
        mac_addr                                      94:6d:ae:5c:9e:04
        ipv4_addr                                     0.0.0.0
        ipv6_addr                                     fe80:0000:0000:0000:966d:aeff:fe5c:9e04
        gid_table_size                                255
        GID[0]                                        fe80:0000:0000:0000:966d:aeff:fe5c:9e04
PCI: 0000:03:00.1
        ibdev_name                                    mlx5_1
        iface_name                                    p1
        mac_addr                                      94:6d:ae:5c:9e:05
        ipv4_addr                                     0.0.0.0
        ipv6_addr                                     fe80:0000:0000:0000:966d:aeff:fe5c:9e05
        gid_table_size                                255
        GID[0]                                        fe80:0000:0000:0000:966d:aeff:fe5c:9e05
PCI: 0000:03:00.0
        ibdev_name                                    mlx5_2
        iface_name                                    enp3s0f0s0
        mac_addr                                      02:c6:d0:fd:56:d7
        ipv4_addr                                     0.0.0.0
        ipv6_addr                                     fe80:0000:0000:0000:00c6:d0ff:fefd:56d7
        gid_table_size                                255
        GID[0]                                        fe80:0000:0000:0000:00c6:d0ff:fefd:56d7
PCI: 0000:03:00.1
        ibdev_name                                    mlx5_3
        iface_name                                    enp3s0f1s0
        mac_addr                                      02:b6:4f:a9:fa:9a
        ipv4_addr                                     0.0.0.0
        ipv6_addr                                     fe80:0000:0000:0000:00b6:4fff:fea9:fa9a
        gid_table_size                                255
        GID[0]                                        fe80:0000:0000:0000:00b6:4fff:fea9:fa9a
```

- To print all the available DOCA representor devices and their capabilities, run:

```
/opt/mellanox/doca/tools/doca_caps --list-rep-devs
```

This command is available only on the Arm side.

Printing the representor list of a specific DOCA device can be done using the `--pci-addr` flag.

Example output:

```
/opt/mellanox/doca/tools/doca_caps --list-rep-devs
PCI: 0000:03:00.0
        representor-PCI: 0000:3b:00.0
            pci_func_type                                 PF
            hotplug                                       no
            vuid                                          MT2308XZ0BN0MLNXS0D0F0
        representor-PCI: 0000:3b:00.0
```

```
            pci_func_type                                 SF
            hotplug                                       no
            vuid                                          MT2308XZ0BN0ECMLNXS0D0F0SF32800
PCI: 0000:03:00.1
        representor-PCI: 0000:3b:00.1
            pci_func_type                                 PF
            hotplug                                       no
            vuid                                          MT2308XZ0BN0MLNXS0D0F1
        representor-PCI: 0000:3b:00.1
            pci_func_type                                 SF
            hotplug                                       no
            vuid                                          MT2308XZ0BN0ECMLNXS0D0F1SF32800
PCI: 0000:03:00.0
PCI: 0000:03:00.1
```

- To print all the supported DOCA libraries by the OS and their availability status, run:

```
/opt/mellanox/doca/tools/doca_caps --list-libs
```

> Different OSs may support different DOCA libraries.

Example output:

```
/opt/mellanox/doca/tools/doca_caps --list-libs
    common                                        installed
    aes_gcm                                       installed
    apsh                                          installed
    argp                                          installed
    cc                                            installed
    comm_channel                                  installed
    compress                                      installed
    dma                                           installed
    dpa                                           installed
    dpdk_bridge                                   installed
    erasure_coding                                installed
    eth                                           installed
    ipsec                                         installed
    flow                                          installed
    flow_ct                                       installed
    pcc                                           installed
    rdma                                          installed
    sha                                           installed
    telemetry                                     installed
```

- To print all the capabilities for all the available libraries, that have capabilities, for every DOCA device, run:

```
/opt/mellanox/doca/tools/doca_caps
```

> Printing the capabilities of one specific DOCA device can be done using the `--pci-addr` flag.

> Printing the capabilities of one specific DOCA library can be done using the `--lib` flag.

Example output:

```
/opt/mellanox/doca/tools/doca_caps
PCI: 0000:03:00.0
    common
        mmap_export_pci                               supported
        mmap_create_from_export_pci                   supported
        hotplug_manager                               unsupported
        rep_filter_all                                supported
        rep_filter_net                                supported
        rep_filter_emulated                           unsupported
    aes_gcm
        task_encrypt                                  supported
        task_encrypt_get_max_iv_len                   12
        task_encrypt_tag_96                           supported
        task_encrypt_tag_128                          supported
        task_encrypt_128b_key                         supported
        task_encrypt_256b_key                         supported
        task_encrypt_max_buf_size                     2097152
        task_encrypt_max_list_buf_num_elem            128
        task_decrypt                                  supported
        task_decrypt_get_max_iv_len                   12
        task_decrypt_tag_96                           supported
        task_decrypt_tag_128                          supported
        task_decrypt_128b_key                         supported
        task_decrypt_256b_key                         supported
        task_decrypt_max_buf_size                     2097152
        task_decrypt_max_list_buf_num_elem            128
        max_num_tasks                                 65536
    cc
        server                                        supported
        client                                        supported
        max_name_len                                  120
        max_msg_size                                  4080
        max_recv_queue_size                           8192
        max_send_tasks                                8192
        max_clients                                   512
        consumer                                      supported
        consumer_max_num_tasks                        65536
        consumer_max_buf_size                         2097152
        producer                                      supported
        producer_max_num_tasks                        65536
        producer_max_buf_size                         2097152
    comm_channel
        max_service_name_len                          120
        max_message_size                              4080
        max_send_queue_size                           8192
        max_recv_queue_size                           8192
        service_max_num_connections                   512
    compress
        task_compress_deflate                         unsupported
        task_compress_deflate_get_max_buf_size        0
        task_compress_deflate_get_max_buf_list_len    0
        task_decompress_deflate                       supported
        task_decompress_deflate_get_max_buf_size      2097152
        task_decompress_deflate_get_max_buf_list_len  128
        task_decompress_lz4                           supported
        task_decompress_lz4_get_max_buf_size          2097152
        task_decompress_lz4_get_max_buf_list_len      128
        max_num_tasks                                 65536
    dma
        task_memcpy                                   supported
        max_buf_size                                  2097152
        max_buf_list_len                              64
        max_num_tasks                                 65536
    dpa
        dpa                                           supported
        max_threads_per_kernel                        128
        kernel_max_run_time                           12
    erasure_coding
        task_galois_mul                               supported
        task_create                                   supported
        task_update                                   supported
        task_recover                                  supported
        max_block_size                                1048576
        max_buf_list_len                              128
    eth
        rxq_cyclic_cpu                                unsupported
        rxq_cyclic_gpu                                supported
        rxq_managed_mempool_cpu                       unsupported
        rxq_managed_mempool_gpu                       supported
        rxq_regular_cpu                               unsupported
        rxq_regular_gpu                               supported
        rxq_max_recv_buf_list_len                     32
        rxq_max_packet_size                           16384
        rxq_max_burst_size                            32768
        txq_regular_cpu                               unsupported
        txq_regular_gpu                               supported
        txq_max_send_buf_list_len                     48
        txq_max_lso_header_size                       256
        txq_txq_max_lso_msg_size                      262144
        txq_l3_chksum_offload                         supported
        txq_l4_chksum_offload                         supported
        txq_wait_on_time_type                         unsupported
    flow_ct
        flow_ct                                       supported
    ipsec
        task_sa_create                                supported
        task_sa_destroy                               supported
    nvrd_transport
        task_write                                    supported
        rc_max_src_buf_list_len                       0
        dc_max_src_buf_list_len                       0
    pcc
        pcc                                           unsupported
        pcc_np                                        unsupported
        min_num_threads                               0
        max_num_threads                               0
    rdma
        task_send                                     supported
        task_send_imm                                 supported
        task_read                                     supported
        task_write                                    supported
        task_write_imm                                supported
        task_atomic_cmp_swp                           supported
        task_atomic_fetch_add                         supported
        task_receive                                  supported
        rc_transport_type                             supported
```

```
        dc_transport_type                             unsupported
        rc_task_receive_get_max_dst_buf_list_len      31
        dc_task_receive_get_max_dst_buf_list_len      0
        task_remote_net_sync_event_get                supported
        task_remote_net_sync_event_notify_set         supported
        task_remote_net_sync_event_notify_add         supported
        max_send_queue_size                           32768
        max_recv_queue_size                           32768
        max_send_buf_list_len                         13
        max_message_size                              1073741824
    sha
        sha1                                          unsupported
        sha256                                        unsupported
        sha512                                        unsupported
        sha1_partial                                  unsupported
        sha256_partial                                unsupported
        sha512_partial                                unsupported
        max_list_num_elem                             0
        max_src_buf_size                              0
        sha1_min_dst_buf_size                         0
        sha256_min_dst_buf_size                       0
        sha512_min_dst_buf_size                       0
        sha1_partial_hash_block_size                  0
        sha256_partial_hash_block_size                0
        sha512_partial_hash_block_size                0
PCI: 0000:03:00.1
    common
        mmap_export_pci                               supported
        mmap_create_from_export_pci                   supported
        hotplug_manager                               unsupported
        rep_filter_all                                supported
        rep_filter_net                                supported
        rep_filter_emulated                           unsupported
    aes_gcm
        task_encrypt                                  supported
        task_encrypt_get_max_iv_len                   12
        task_encrypt_tag_96                           supported
        task_encrypt_tag_128                          supported
        task_encrypt_128b_key                         supported
        task_encrypt_256b_key                         supported
        task_encrypt_max_buf_size                     2097152
        task_encrypt_max_list_buf_num_elem            128
        task_decrypt                                  supported
        task_decrypt_get_max_iv_len                   12
        task_decrypt_tag_96                           supported
        task_decrypt_tag_128                          supported
        task_decrypt_128b_key                         supported
        task_decrypt_256b_key                         supported
        task_decrypt_max_buf_size                     2097152
        task_decrypt_max_list_buf_num_elem            128
        max_num_tasks                                 65536
    cc
        server                                        supported
        client                                        supported
        max_name_len                                  120
        max_msg_size                                  4080
        max_recv_queue_size                           8192
        max_send_tasks                                8192
        max_clients                                   512
        consumer                                      supported
        consumer_max_num_tasks                        65536
        consumer_max_buf_size                         2097152
        producer                                      supported
        producer_max_num_tasks                        65536
        producer_max_buf_size                         2097152
    comm_channel
        max_service_name_len                          120
        max_message_size                              4080
        max_send_queue_size                           8192
        max_recv_queue_size                           8192
        service_max_num_connections                   512
    compress
        task_compress_deflate                         unsupported
        task_compress_deflate_get_max_buf_size        0
        task_compress_deflate_get_max_buf_list_len    0
        task_decompress_deflate                       supported
        task_decompress_deflate_get_max_buf_size      2097152
        task_decompress_deflate_get_max_buf_list_len  128
        task_decompress_lz4                           supported
        task_decompress_lz4_get_max_buf_size          2097152
        task_decompress_lz4_get_max_buf_list_len      128
        max_num_tasks                                 65536
    dma
        task_memcpy                                   supported
        max_buf_size                                  2097152
        max_buf_list_len                              64
        max_num_tasks                                 65536
    dpa
        dpa                                           supported
        max_threads_per_kernel                        128
        kernel_max_run_time                           12
    erasure_coding
        task_galois_mul                               supported
        task_create                                   supported
        task_update                                   supported
        task_recover                                  supported
        max_block_size                                1048576
        max_buf_list_len                              128
    eth
        rxq_cyclic_cpu                                unsupported
        rxq_cyclic_gpu                                supported
        rxq_managed_mempool_cpu                       unsupported
        rxq_managed_mempool_gpu                       supported
        rxq_regular_cpu                               unsupported
        rxq_regular_gpu                               supported
        rxq_max_recv_buf_list_len                     32
        rxq_max_packet_size                           16384
        rxq_max_burst_size                            32768
        txq_regular_cpu                               unsupported
        txq_regular_gpu                               supported
        txq_max_send_buf_list_len                     48
        txq_max_lso_header_size                       256
        txq_txq_max_lso_msg_size                      262144
        txq_l3_chksum_offload                         supported
        txq_l4_chksum_offload                         supported
        txq_wait_on_time_type                         unsupported
    flow_ct
        flow_ct                                       supported
    ipsec
        task_sa_create                                supported
        task_sa_destroy                               supported
    nvrd_transport
        task_write                                    supported
        rc_max_src_buf_list_len                       0
        dc_max_src_buf_list_len                       0
    pcc
        pcc                                           unsupported
        pcc_np                                        unsupported
        min_num_threads                               0
        max_num_threads                               0
    rdma
        task_send                                     supported
        task_send_imm                                 supported
        task_read                                     supported
        task_write                                    supported
        task_write_imm                                supported
        task_atomic_cmp_swp                           supported
        task_atomic_fetch_add                         supported
        task_receive                                  supported
        rc_transport_type                             supported
        dc_transport_type                             unsupported
        rc_task_receive_get_max_dst_buf_list_len      31
        dc_task_receive_get_max_dst_buf_list_len      0
        task_remote_net_sync_event_get                supported
        task_remote_net_sync_event_notify_set         supported
        task_remote_net_sync_event_notify_add         supported
        max_send_queue_size                           32768
        max_recv_queue_size                           32768
        max_send_buf_list_len                         13
        max_message_size                              1073741824
    sha
        sha1                                          unsupported
        sha256                                        unsupported
        sha512                                        unsupported
        sha1_partial                                  unsupported
        sha256_partial                                unsupported
        sha512_partial                                unsupported
        max_list_num_elem                             0
        max_src_buf_size                              0
        sha1_min_dst_buf_size                         0
        sha256_min_dst_buf_size                       0
        sha512_min_dst_buf_size                       0
        sha1_partial_hash_block_size                  0
        sha256_partial_hash_block_size                0
        sha512_partial_hash_block_size                0
PCI: 0000:03:00.0
    common
        mmap_export_pci                               supported
        mmap_create_from_export_pci                   supported
        hotplug_manager                               unsupported
        rep_filter_all                                unsupported
        rep_filter_net                                unsupported
        rep_filter_emulated                           unsupported
    aes_gcm
        task_encrypt                                  supported
        task_encrypt_get_max_iv_len                   12
        task_encrypt_tag_96                           supported
        task_encrypt_tag_128                          supported
        task_encrypt_128b_key                         supported
        task_encrypt_256b_key                         supported
        task_encrypt_max_buf_size                     2097152
        task_encrypt_max_list_buf_num_elem            128
        task_decrypt                                  supported
        task_decrypt_get_max_iv_len                   12
        task_decrypt_tag_96                           supported
        task_decrypt_tag_128                          supported
        task_decrypt_128b_key                         supported
        task_decrypt_256b_key                         supported
        task_decrypt_max_buf_size                     2097152
        task_decrypt_max_list_buf_num_elem            128
        max_num_tasks                                 65536
    cc
        server                                        unsupported
        client                                        supported
        max_name_len                                  120
        max_msg_size                                  4080
        max_recv_queue_size                           8192
        max_send_tasks                                8192
        max_clients                                   0
        consumer                                      supported
        consumer_max_num_tasks                        65536
        consumer_max_buf_size                         2097152
        producer                                      supported
        producer_max_num_tasks                        65536
        producer_max_buf_size                         2097152
    comm_channel
```

```
        max_service_name_len                          120
        max_message_size                              4080
        max_send_queue_size                           8192
        max_recv_queue_size                           8192
        service_max_num_connections                   0
    compress
        task_compress_deflate                         unsupported
        task_compress_deflate_get_max_buf_size        0
        task_compress_deflate_get_max_buf_list_len    0
        task_decompress_deflate                       supported
        task_decompress_deflate_get_max_buf_size      2097152
        task_decompress_deflate_get_max_buf_list_len  128
        task_decompress_lz4                           supported
        task_decompress_lz4_get_max_buf_size          2097152
        task_decompress_lz4_get_max_buf_list_len      128
        max_num_tasks                                 65536
    dma
        task_memcpy                                   supported
        max_buf_size                                  2097152
        max_buf_list_len                              64
        max_num_tasks                                 65536
    dpa
        dpa                                           supported
        max_threads_per_kernel                        128
        kernel_max_run_time                           12
    erasure_coding
        task_galois_mul                               supported
        task_create                                   supported
        task_update                                   supported
        task_recover                                  supported
        max_block_size                                1048576
        max_buf_list_len                              128
    eth
        rxq_cyclic_cpu                                supported
        rxq_cyclic_gpu                                supported
        rxq_managed_mempool_cpu                       supported
        rxq_managed_mempool_gpu                       supported
        rxq_regular_cpu                               supported
        rxq_regular_gpu                               supported
        rxq_max_recv_buf_list_len                     32
        rxq_max_packet_size                           16384
        rxq_max_burst_size                            32768
        txq_regular_cpu                               supported
        txq_regular_gpu                               supported
        txq_max_send_buf_list_len                     48
        txq_max_lso_header_size                       256
        txq_txq_max_lso_msg_size                      262144
        txq_l3_chksum_offload                         supported
        txq_l4_chksum_offload                         supported
        txq_wait_on_time_type                         unsupported
    flow_ct
        flow_ct                                       unsupported
    ipsec
        task_sa_create                                unsupported
        task_sa_destroy                               unsupported
    nvrd_transport
        task_write                                    supported
        rc_max_src_buf_list_len                       0
        dc_max_src_buf_list_len                       0
    pcc
        pcc                                           unsupported
        pcc_np                                        unsupported
        min_num_threads                               0
        max_num_threads                               0
    rdma
        task_send                                     supported
        task_send_imm                                 supported
        task_read                                     supported
        task_write                                    supported
        task_write_imm                                supported
        task_atomic_cmp_swp                           supported
        task_atomic_fetch_add                         supported
        task_receive                                  supported
        rc_transport_type                             supported
        dc_transport_type                             unsupported
        rc_task_receive_get_max_dst_buf_list_len      31
        dc_task_receive_get_max_dst_buf_list_len      0
        task_remote_net_sync_event_get                supported
        task_remote_net_sync_event_notify_set         supported
        task_remote_net_sync_event_notify_add         supported
        max_send_queue_size                           32768
        max_recv_queue_size                           32768
        max_send_buf_list_len                         13
        max_message_size                              1073741824
    sha
        sha1                                          unsupported
        sha256                                        unsupported
        sha512                                        unsupported
        sha1_partial                                  unsupported
        sha256_partial                                unsupported
        sha512_partial                                unsupported
        max_list_num_elem                             0
        max_src_buf_size                              0
        sha1_min_dst_buf_size                         0
        sha256_min_dst_buf_size                       0
        sha512_min_dst_buf_size                       0
        sha1_partial_hash_block_size                  0
        sha256_partial_hash_block_size                0
        sha512_partial_hash_block_size                0
PCI: 0000:03:00.1
    common
        mmap_export_pci                               supported
        mmap_create_from_export_pci                   supported
        hotplug_manager                               unsupported
        rep_filter_all                                unsupported
        rep_filter_net                                unsupported
        rep_filter_emulated                           unsupported
    aes_gcm
        task_encrypt                                  supported
        task_encrypt_get_max_iv_len                   12
        task_encrypt_tag_96                           supported
        task_encrypt_tag_128                          supported
        task_encrypt_128b_key                         supported
        task_encrypt_256b_key                         supported
        task_encrypt_max_buf_size                     2097152
        task_encrypt_max_list_buf_num_elem            128
        task_decrypt                                  supported
        task_decrypt_get_max_iv_len                   12
        task_decrypt_tag_96                           supported
        task_decrypt_tag_128                          supported
        task_decrypt_128b_key                         supported
        task_decrypt_256b_key                         supported
        task_decrypt_max_buf_size                     2097152
        task_decrypt_max_list_buf_num_elem            128
        max_num_tasks                                 65536
    cc
        server                                        unsupported
        client                                        supported
        max_name_len                                  120
        max_msg_size                                  4080
        max_recv_queue_size                           8192
        max_send_tasks                                8192
        max_clients                                   0
        consumer                                      supported
        consumer_max_num_tasks                        65536
        consumer_max_buf_size                         2097152
        producer                                      supported
        producer_max_num_tasks                        65536
        producer_max_buf_size                         2097152
    comm_channel
        max_service_name_len                          120
        max_message_size                              4080
        max_send_queue_size                           8192
        max_recv_queue_size                           8192
        service_max_num_connections                   0
    compress
        task_compress_deflate                         unsupported
        task_compress_deflate_get_max_buf_size        0
        task_compress_deflate_get_max_buf_list_len    0
        task_decompress_deflate                       supported
        task_decompress_deflate_get_max_buf_size      2097152
        task_decompress_deflate_get_max_buf_list_len  128
        task_decompress_lz4                           supported
        task_decompress_lz4_get_max_buf_size          2097152
        task_decompress_lz4_get_max_buf_list_len      128
        max_num_tasks                                 65536
    dma
        task_memcpy                                   supported
        max_buf_size                                  2097152
        max_buf_list_len                              64
        max_num_tasks                                 65536
    dpa
        dpa                                           supported
        max_threads_per_kernel                        128
        kernel_max_run_time                           12
    erasure_coding
        task_galois_mul                               supported
        task_create                                   supported
        task_update                                   supported
        task_recover                                  supported
        max_block_size                                1048576
        max_buf_list_len                              128
    eth
        rxq_cyclic_cpu                                supported
        rxq_cyclic_gpu                                supported
        rxq_managed_mempool_cpu                       supported
        rxq_managed_mempool_gpu                       supported
        rxq_regular_cpu                               supported
        rxq_regular_gpu                               supported
        rxq_max_recv_buf_list_len                     32
        rxq_max_packet_size                           16384
        rxq_max_burst_size                            32768
        txq_regular_cpu                               supported
        txq_regular_gpu                               supported
        txq_max_send_buf_list_len                     48
        txq_max_lso_header_size                       256
        txq_txq_max_lso_msg_size                      262144
        txq_l3_chksum_offload                         supported
        txq_l4_chksum_offload                         supported
        txq_wait_on_time_type                         unsupported
    flow_ct
        flow_ct                                       unsupported
    ipsec
        task_sa_create                                unsupported
        task_sa_destroy                               unsupported
    nvrd_transport
        task_write                                    supported
        rc_max_src_buf_list_len                       0
        dc_max_src_buf_list_len                       0
    pcc
        pcc                                           unsupported
        pcc_np                                        unsupported
        min_num_threads                               0
        max_num_threads                               0
    rdma
        task_send                                     supported
```

```
        task_send_imm                                 supported
        task_read                                     supported
        task_write                                    supported
        task_write_imm                                supported
        task_atomic_cmp_swp                           supported
        task_atomic_fetch_add                         supported
        task_receive                                  supported
        rc_transport_type                             supported
        dc_transport_type                             unsupported
        rc_task_receive_get_max_dst_buf_list_len      31
        dc_task_receive_get_max_dst_buf_list_len      0
        task_remote_net_sync_event_get                supported
        task_remote_net_sync_event_notify_set         supported
        task_remote_net_sync_event_notify_add         supported
        max_send_queue_size                           32768
        max_recv_queue_size                           32768
        max_send_buf_list_len                         13
        max_message_size                              1073741824
    sha
        sha1                                          unsupported
        sha256                                        unsupported
        sha512                                        unsupported
        sha1_partial                                  unsupported
        sha256_partial                                unsupported
        sha512_partial                                unsupported
        max_list_num_elem                             0
        max_src_buf_size                              0
        sha1_min_dst_buf_size                         0
        sha256_min_dst_buf_size                       0
        sha512_min_dst_buf_size                       0
        sha1_partial_hash_block_size                  0
        sha256_partial_hash_block_size                0
        sha512_partial_hash_block_size                0
```

16.5 NVIDIA DOCA Comm Channel Admin Tool

This document provides instructions on the usage of the DOCA Comm Channel Admin Tool.

16.5.1 Introduction

The Comm Channel Admin Tool is used to print a snapshot of DOCA Comch (comm channel) connections:

- On the BlueField Arm side, it includes DOCA Comch servers and their current connection information
- On the host side, it includes all active client connections and the server they are connected to
- Only client-to-server control channels are reported; fast path producer/consumer channels are not.

16.5.2 Prerequisites

The Comm Channel Admin Tool is for Linux only and requires an up-to-date BFB bundle or DOCA host packages of at least 2.7, which include in the Resource dump binary.

16.5.3 Description and Execution

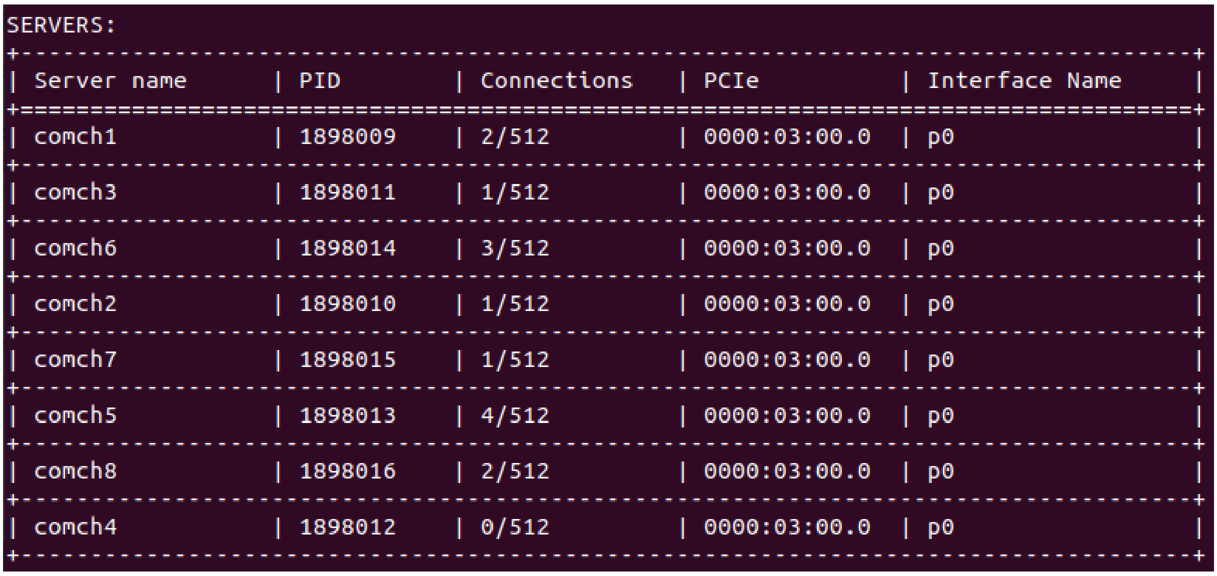
The Comm Channel Admin Tool can be executed on the host or Arm CPUs. By default, the tool scans all available PCIe slots to detect supported DOCA devices and reports any Comch information available.

The tool can be run on BlueField Arm or x86 host using the following command:

```
/opt/mellanox/doca/tools/doca_comm_channel_admin
```

16.5.3.1 Sample Output from BlueField Arm

On the BlueField Arm side, any active DOCA Comch servers are be reported:



```
SERVERS:
+---------------------------------------------------------------------------------------+
| Server name        | PID        | Connections     | PCIe           | Interface Name    |
+=======================================================================================+
| comch1             | 1898009    | 2/512           | 0000:03:00.0   | p0                |
+---------------------------------------------------------------------------------------+
| comch3             | 1898011    | 1/512           | 0000:03:00.0   | p0                |
+---------------------------------------------------------------------------------------+
| comch6             | 1898014    | 3/512           | 0000:03:00.0   | p0                |
+---------------------------------------------------------------------------------------+
| comch2             | 1898010    | 1/512           | 0000:03:00.0   | p0                |
+---------------------------------------------------------------------------------------+
| comch7             | 1898015    | 1/512           | 0000:03:00.0   | p0                |
+---------------------------------------------------------------------------------------+
| comch5             | 1898013    | 4/512           | 0000:03:00.0   | p0                |
+---------------------------------------------------------------------------------------+
| comch8             | 1898016    | 2/512           | 0000:03:00.0   | p0                |
+---------------------------------------------------------------------------------------+
| comch4             | 1898012    | 0/512           | 0000:03:00.0   | p0                |
+---------------------------------------------------------------------------------------+
```

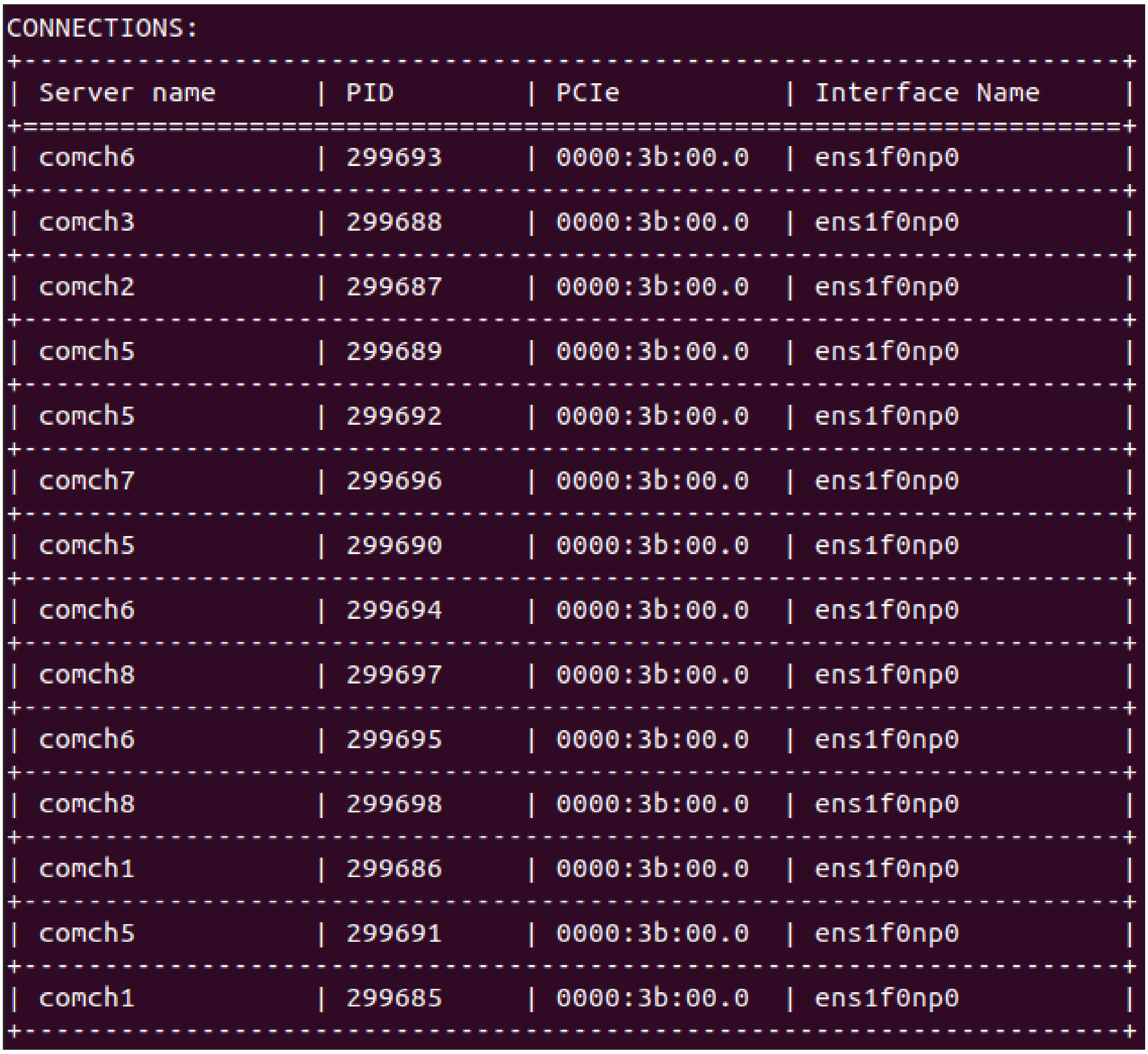
The following information is available:

- Server Name - the name assigned to the server
- PID – the Linux process ID of the application which created the server
- Connections – the number of connections active on the server out of the total allowed (e.g., 2/512 means 2 active connections of a maximum of 512)
- PCIe - the PCIe address of the device which the server has been detected on
- Interface Name – the interface name associated with the PCIe address

⚠ Connections may also be displayed on the BlueField Arm like on x86. This occurs if SF ports are detected here. The interface name associated with the PCIe address indicates the SF port.

16.5.3.2 Sample Output from x86

The x86 host cannot run DOCA Comch servers. Therefore, individual client connections are reported:



```
CONNECTIONS:
+--------------------------------------------------------------------+
| Server name      | PID        | PCIe         | Interface Name     |
+====================================================================+
| comch6           | 299693     | 0000:3b:00.0  | ens1f0np0          |
+--------------------------------------------------------------------+
| comch3           | 299688     | 0000:3b:00.0  | ens1f0np0          |
+--------------------------------------------------------------------+
| comch2           | 299687     | 0000:3b:00.0  | ens1f0np0          |
+--------------------------------------------------------------------+
| comch5           | 299689     | 0000:3b:00.0  | ens1f0np0          |
+--------------------------------------------------------------------+
| comch5           | 299692     | 0000:3b:00.0  | ens1f0np0          |
+--------------------------------------------------------------------+
| comch7           | 299696     | 0000:3b:00.0  | ens1f0np0          |
+--------------------------------------------------------------------+
| comch5           | 299690     | 0000:3b:00.0  | ens1f0np0          |
+--------------------------------------------------------------------+
| comch6           | 299694     | 0000:3b:00.0  | ens1f0np0          |
+--------------------------------------------------------------------+
| comch8           | 299697     | 0000:3b:00.0  | ens1f0np0          |
+--------------------------------------------------------------------+
| comch6           | 299695     | 0000:3b:00.0  | ens1f0np0          |
+--------------------------------------------------------------------+
| comch8           | 299698     | 0000:3b:00.0  | ens1f0np0          |
+--------------------------------------------------------------------+
| comch1           | 299686     | 0000:3b:00.0  | ens1f0np0          |
+--------------------------------------------------------------------+
| comch5           | 299691     | 0000:3b:00.0  | ens1f0np0          |
+--------------------------------------------------------------------+
| comch1           | 299685     | 0000:3b:00.0  | ens1f0np0          |
+--------------------------------------------------------------------+
```

The following information is available:

- Server Name – the name of the BlueField Arm server that a client has connected to
- PID – the Linux process ID of the application running a DOCA Comch client
- PCIe – the PCIe address of the BlueField networking platform which the destination server is running on
- Interface Name – the interface name associated with the PCIe address

16.6 NVIDIA DPA Tools

16.6.1 Introduction

DPA tools are a set of executables that enable the DPA application developer and the system administrator to manage and monitor DPA resources and to debug DPA applications.

16.6.2 DPA Tools

16.6.2.1 DPACC Compiler

CLI name: `dpacc`

DPACC is a high-level compiler for the DPA processor. It compiles code targeted for the DPA processor into an executable and generates a DPA program.

The DPA program is a host library with interfaces encapsulating the DPA executable. This DPA program can be linked with the host application to generate a host executable where the DPA code is invoked through the FlexIO runtime API.

16.6.2.2 DPA EU Management Tool

CLI name: `dpaeumgmt`

This tool allows users to manage the DPA's EUs which are the basic resource of the DPA. The tool enables the resource control of EUs to optimize the usage of computation resources of the DPA. Using this tool, users may query, create, and destroy EU partitions and groups, thus ensuring proper EU allocation between devices.

16.6.2.3 DPA GDB Server Tool

CLI name: `dpa-gdbserver`

The DPA GDB Server tool enables debugging FlexIO DEV programs.

16.6.2.4 DPA PS Tool

CLI name: `dpa-ps`

This tool allows users to monitor running DPA processes and threads.

16.6.2.5 DPA Statistic Tool

CLI name: `dpa-statistics`

This tool allows users to monitor and obtain statistics on thread execution per running DPA process and thread.

16.6.3 NVIDIA DOCA DPACC Compiler

This document describes DOCA DPACC compiler and instructions about DPA toolchain setup and usage.

16.6.3.1 Introduction

DPACC is a high-level compiler for the DPA processor which compiles code targeted for the data-path accelerator (DPA) processor into a device executable and generates a DPA program.

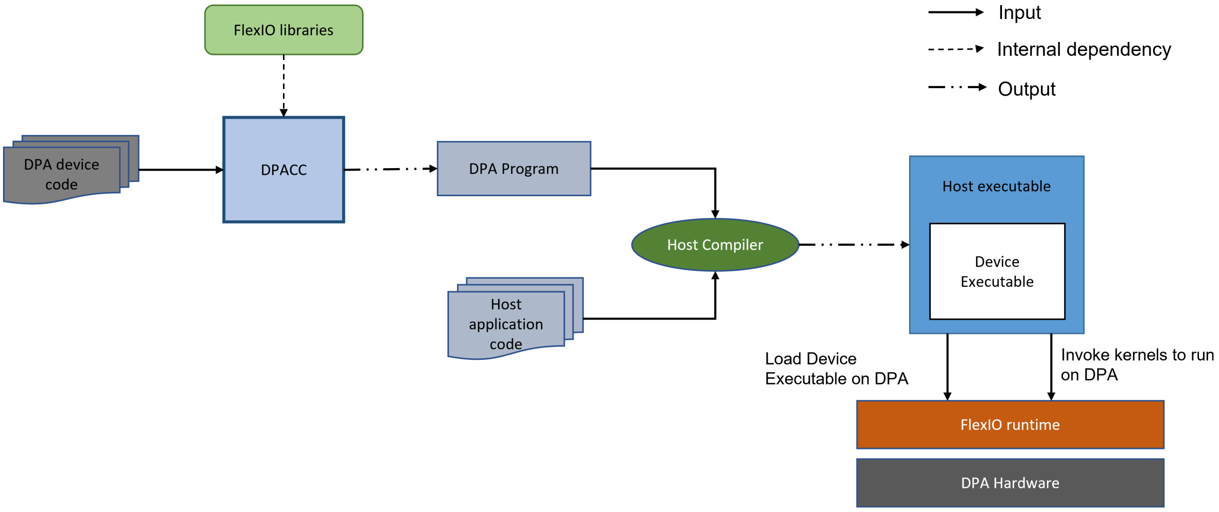
The DPA program is a host library with interfaces encapsulating the device executable. This DPA program is linked with the host application to generate a host executable. The host executable can invoke the DPA code through FlexIO runtime API.

DPACC uses DPA compiler (`dpa-clang`) to compile code targeted for DPA. dpa-clang is part of the DPA toolchain package which is an LLVM-based cross-compiling bare-metal toolchain. It provides Clang compiler, LLD linker targeting DPA architecture, and other utilities.

16.6.3.1.1 Glossary

Term	Definition
Device	DPA as present on the BlueField DPU
Host	CPU that launches the device code to run on the DPA
Device function	Any C function that runs on the DPA device
DPA global function	Device function that is the point of entry when offloading any work on DPA
Host compiler	Compiler used to compile the code targeting the host CPU
Device compiler	Compiler used to compile code targeting the DPA
DPA program	Host library that encapsulates the DPA device executable (`.elf`) and host stubs which are used to access the device executable

16.6.3.1.2 Offloading Work on DPA



To invoke a DPA function from host, the following things are required:

- DPA device code - C programs, targeted to run on the DPA. DPA device code may contain one or more entry functions.
- Host application code - the corresponding host application. For more information, refer to DPA Subsystem documentation.
- Runtime - FlexIO or DOCA DPA library provides the runtime

The generated DPA program, when linked with a host application results in a host executable which also contains the device executable. The host application oversees loading the device executable on the device.

16.6.3.1.3 DPACC Predefined Macros

DPACC predefines the following macros:

Macro	Description
`__DPA__`	Defined when compiling device code file
`__NV_DPA`	Defined to the target DPA hardware identifier macros. See Architecture Macros for more details.
`__DPA_MAJOR__`	Defined to the major version number of DPACC
`__DPA_MINOR__`	Defined to the minor version number of DPACC
`__DPA_PATCH__`	Defined to the patch version number of DPACC

16.6.3.1.4 Writing DPA Applications

DPA device code is a C code with some restrictions and special definitions.

FlexIO or DOCA-DPA APIs provide interfaces to DPA.

16.6.3.1.4.1 Language Support

The DPA is programmed using a subset of the C11 language standard. The compiler documents any constructs that are not available. Language constructs, where available, retain their standard definitions.

16.6.3.1.4.2 Restrictions on DPA Code

- Use of C thread local storage is not allowed for any variables
- Identifiers with `_dpacc` prefix are reserved by the compiler. Use of such identifiers may result in an error or undefined behavior
- DPA processor does not have native floating-point support; use of floating point operations is disabled

16.6.3.1.4.3 DPA RPC Functions

A remote procedure call function is a synchronous call that triggers work in DPA and waits for its completion. These functions return a type `uint64_t` value. They are annotated with a `__dpa_rpc__` attribute.

16.6.3.1.4.4 DPA Global Functions

A DPA global function is an event handler device function referenced from the host code. These functions do not return anything. They are annotated with a `__dpa_global__` attribute.

For more information, refer to DPA Subsystem documentation.

16.6.3.1.4.5 Characteristics of Annotated Functions

- Global functions must have `void` return type and RPC functions must have `uint64_t` return type
- Annotated functions cannot accept C pointers and arrays as arguments (e.g., `void my_global (int *ptr, int arr[])`)
- Annotated functions cannot accept a variable number of arguments
- Inline specifier is not allowed on annotated functions

16.6.3.1.4.6 Handling User-defined Data Types

User-defined data types, when used as global function arguments, require special handling. They must be annotated with a `__dpa_global__` attribute.

If the user-defined data type is `typedef`'d, the `typedef` statement must be annotated with a `__dpa_global__` attribute along the data type itself.

16.6.3.1.4.7 Characteristics of Annotated Types

- They must have a copy of the definition in all translation units where they are used as global function arguments
- They cannot have pointers, variable length arrays, and flexible arrays as members
- Fixed-size arrays as C structure members are supported
- These characteristics apply recursively to any user-defined/ `typedef`'d types that are members of an annotated type

DPACC processes all annotated functions along with annotated types and generates host and device interfaces to facilitate the function launch.

16.6.3.1.4.8 DPA Intrinsics

DPA features such as fences and processor-specific instructions are exposed via intrinsics by the DPA compiler. All intrinsics defined in the header file `dpaintrin.h` are guarded by the `DPA_INTRIN_VERSION_USED` macro. The current `DPA_INTRIN_VERSION` is `1.3`.

Example:

```
#define DPA_INTRIN_VERSION_USED (DPA_INTRIN_VERSION(1, 3))
#include <dpaintrin.h>
…
__dpa_thread_writeback_window();   // Fence for write barrier
```

For more information, refer to DPA Subsystem documentation.

16.6.3.2 Prerequisites

Package	Instructions
Host compiler	Compiler specified through `hostcc` option. Both `gcc` and `clang` are supported. ⚠ Minimum supported version for clang as hostcc is `clang 3.8.0`.
Device compiler	The default device compiler is the "DPA compiler". Installing the DPACC package also installs the DPA compiler binaries `dpa-clang`, `dpa-ar`, `dpa-nm` and `dpa-objdump`. ⚠ `dpa-clang` is the only supported device compiler.
FlexIO SDK and C library	Available as part of the DOCA software package. DPA toolchain does not provide C library and corresponding headers. Users are expected to use the C library for DPA from the FlexIO SDK.

16.6.3.2.1 Supported Versions

- DPACC version 1.8.0
- Refer to DPA Subsystem documentation for other component versions

16.6.3.3 Description

16.6.3.3.1 DPACC Inputs and Outputs

DPACC can produce DPA programs in a single command by accepting all source files as input. DPACC also offers the flexibility of producing DPA object files or libraries from input files.

DPA object files contain both host stub objects (DPACC-generated interfaces) and device objects. These DPA object files can later be given to DPACC as input to produce the DPA library.

Phase	Option Name	Default Output File Name
Compile input device code files to DPA object files	`--compile` or `-c`	`.dpa.o` appended to the name of each input source file
Compile and link the input device code files/ DPA object files, and produce a DPA program	No specific option	No default name, output file name must be specified
Compile and build DPA library from input device code files/DPA object files	`--gen-libs` or `-gen-libs`	No default name, output library name must be specified

DPACC can accept the following file types as input:

Input File Extension	File Type	Description
`.c`	C source file	DPA device code

Input File Extension	File Type	Description
`.dpa.o`	DPA object file	Object file generated by DPACC, containing both host and device objects
`.a`	DPA object archive	An archive of DPA object files. User can generate this archive from DPACC-generated DPA objects.

Based on the mode of operations, DPACC can generate the following output files:

Output File Type	Input Files
DPA object file	C source files
DPA program	C source files, DPA object files, and/or DPA object archives
DPA library (DPA host library and DPA device library)	C source files, DPA object files, and/or DPA object archives

The following provides the commands to generate different kinds of supported output file types for each input file type:

Input	Output	DPACC Command
C source file	DPA program	`dpacc -hostcc=gcc in.c -o libprog.a`
	DPA object	`dpacc -hostcc=gcc in.c -c`
	DPA library	`dpacc -hostcc=gcc in.c -o lib<name> -gen-libs`
DPA object	DPA program	`dpacc -hostcc=gcc in.dpa.o -o libprog.a`
	DPA library	`dpacc -hostcc=gcc in.dpa.o -o lib<name> -gen-libs`
DPA object archive	DPA program	`dpacc -hostcc=gcc in.a -o libprog.a`
	DPA library	`dpacc -hostcc=gcc in.a -o lib<name> -gen-libs`

16.6.3.3.1.1 DPA Program

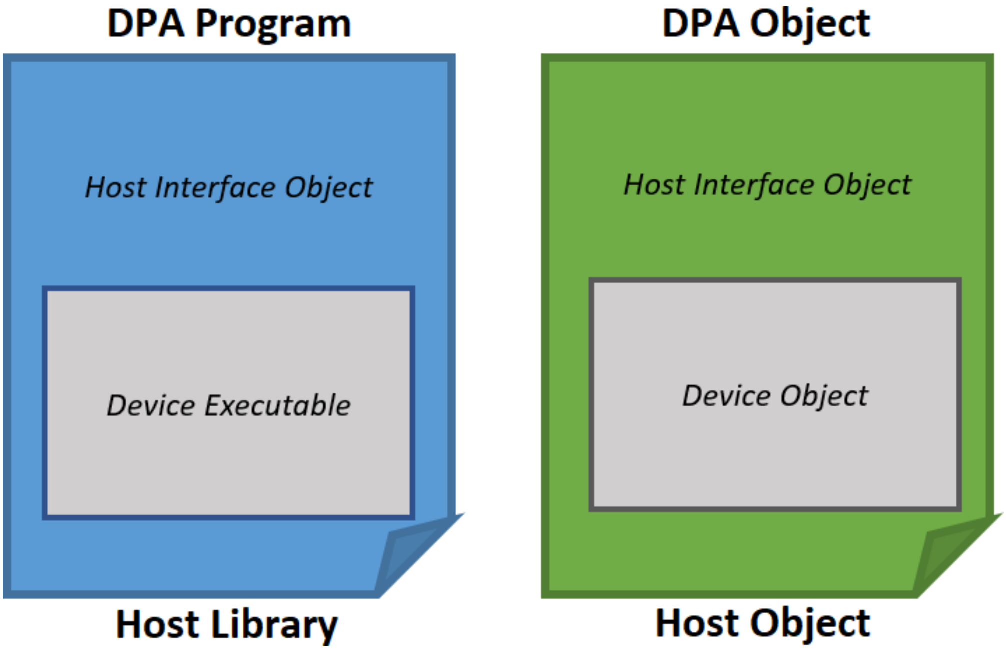
DPACC produces a DPA program in compile-and-link mode. A DPA program is a host library which contains:

- DPACC-generated host stubs which facilitate invoking a DPA global function from the host application
- Device executable, generated by DPACC by compiling input DPA device code

DPA program library must be linked with the host application that contains appropriate runtime APIs to load the device executable onto DPA memory.

16.6.3.3.1.2 DPA Object

DPACC produces DPA object files in compile-only mode. A DPA object is an object file for the host machine. In a DPA object, the device object generated by compiling the input device code file is placed inside a specific section of the generated host stubs object. This process is repeated for each input file.

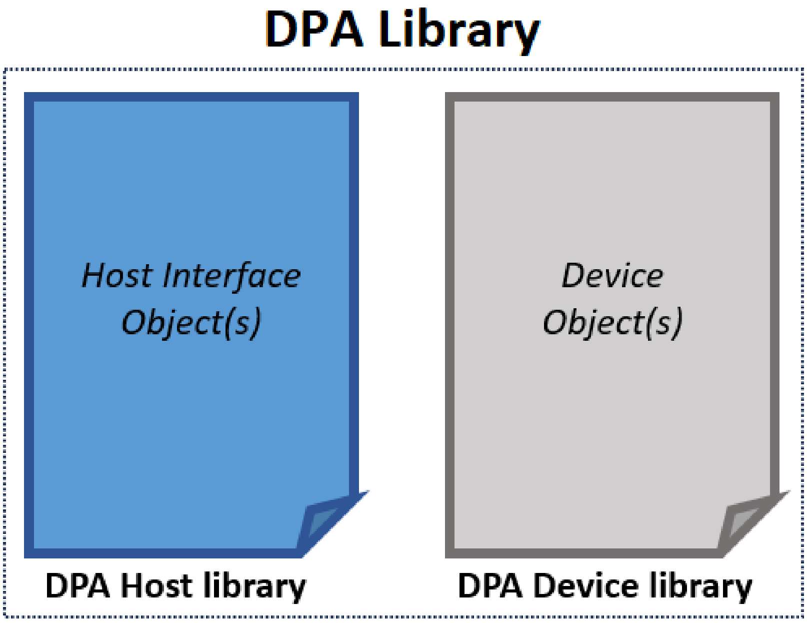


16.6.3.3.1.3 DPA Library

A DPA library is a collection of two individual libraries:

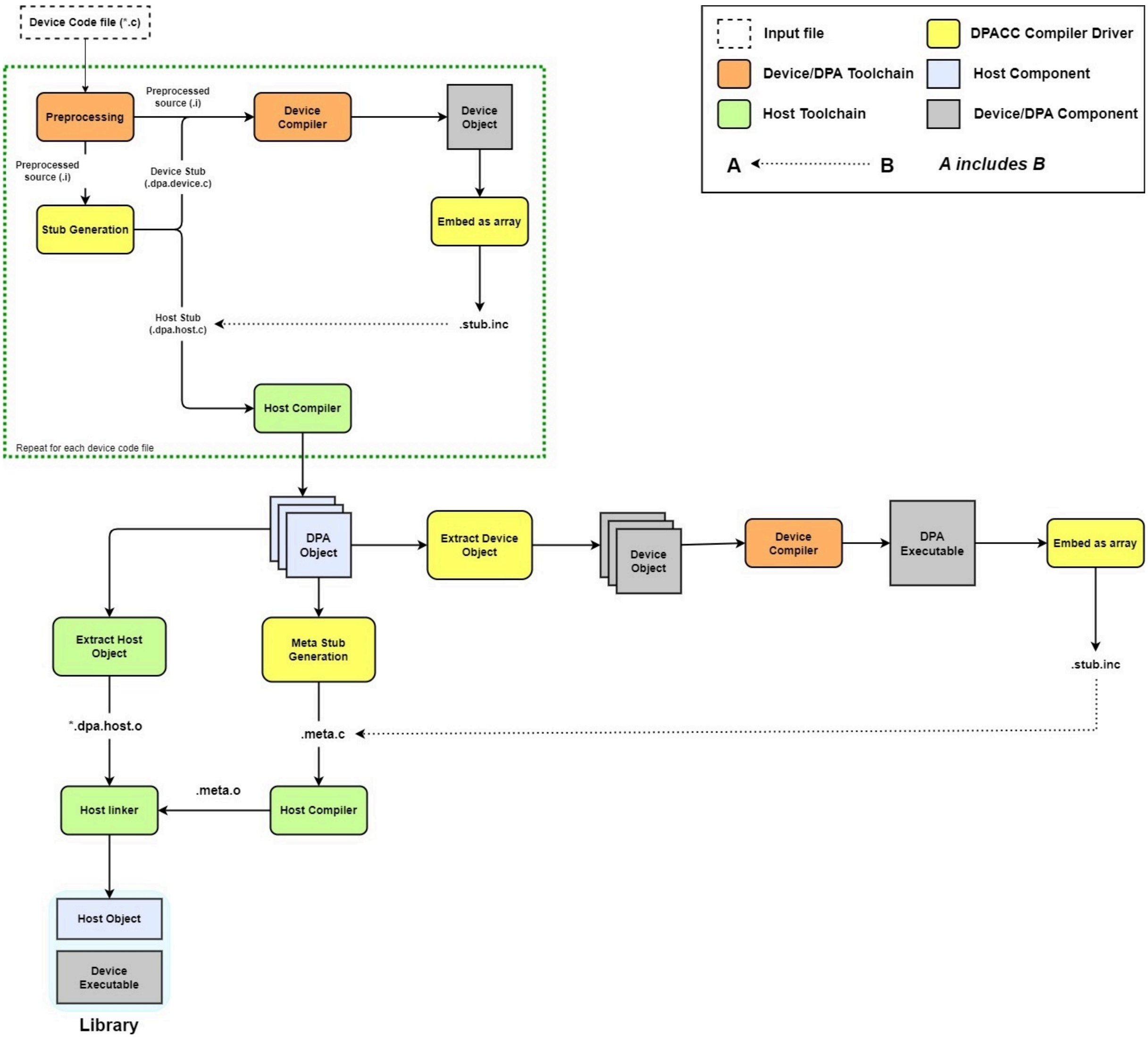
- DPA device library – contains device objects generated from input files
- DPA host library – contains host interface objects corresponding to the device objects in DPA device library

The DPA device library is consumed by DPACC during DPA-program generation and the DPA host library can optionally be linked with other host code and be distributed as the host library. Both libraries are generated as static archives.



16.6.3.3.2 DPACC Trajectory

The following diagram illustrates DPACC compile-and-link mode trajectory.



16.6.3.3.3 Modes of Operation

16.6.3.3.3.1 Compile-and-link Mode

This is a one-step mode that accepts C source files or DPA object files and produces the DPA program. Specifying the output library name is mandatory in this mode.

Example commands:

```
$ dpacc in1.c in2.c -o myLib1.a -hostcc=gcc          # Takes C sources to produce myLib1.a library
$ dpacc in3.dpa.o in4.dpa.o -o myLib2.a -hostcc=gcc  # Takes DPA object files to produce myLib2.a library
$ dpacc in1.c in3.dpa.o -o myLib3.a -hostcc=gcc      # Takes C source and DPA object to produce myLib3.a library
```

16.6.3.3.3.2 Compile-only Mode

This mode accepts C source code and produces `.dpa.o` object files. These files can be given to DPACC to produce the DPA program. The mode is invoked by the `--compile` or `-c` option.

The user can explicitly provide the output object file name using the `--output-file` or `-o` option.

Example commands:

```
$ dpacc -c input1.c -hostcc=gcc                 # Produces input1.dpa.o
$ dpacc -c input3.c input4.c -hostcc=gcc        # Produces input3.dpa.o and input4.dpa.o
$ dpacc -c input2.c -o myObj.dpa.o -hostcc=gcc  # Produces myObj.dpa.o
```

16.6.3.3.3.3 Library Generation Mode

This mode accepts C source files or DPA object files and produces the DPA program. Specifying the output DPA library name is mandatory in this mode.

Example commands:

```
$ dpacc in1.c in2.c -o libdummy1 -hostcc=gcc -gen-libs              # Takes C sources to produce libdummy1_host.a
and libdummy_device.a archives
$ dpacc in3.dpa.o in4.dpa.o -o libdummy2 -hostcc=gcc -gen-libs      # Takes DPA object files to produce
libdummy2_host.a and libdummy2_device.a archives
$ dpacc in1.c in3.dpa.o -o outdir/libdummy3 -hostcc=gcc -gen-libs   # Takes C source and DPA object to produce
outdir/libdummy3_host.a and outdir/libdummy3_device.a archives
```

16.6.3.4 Execution

To execute DOCA DPACC compiler:

```
Usage: dpacc <list-of-input-files> -hostcc=<path> [other options]
Helper Flags:
  -h, --help                           Print help information about DPACC
  -V, --version                        Print DPACC version information
  -v, --verbose                        List the compilation commands generated by this invocation while also
executing every command in verbose mode
  -dryrun, --dryrun                    Only list the compilation commands generated by DPACC, without executing them
  -keep, --keep                        Keep all intermediate files that are generated during internal compilation
steps in the current directory
  -keep-dir, --keep-dir                Keep all intermediate files that are generated during internal compilation
steps in the given directory
  -optf, --options-file <file>,...    Include command line options from the specified file
```

16.6.3.4.1 Mandatory Arguments

Flag	DPACC Mode	Description
List of one or more input files	All	List of C source files or DPA object file names. Specifying at least one input file is mandatory. A file with an unknown extension is treated as a DPA object file.
`-hostcc`, `--hostcc <path>`	All	Specify the host compiler. This is typically the native compiler present on the host system. ⚠ The host compiler used to link the host application with the DPA program must be link-compatible with the `hostcc` compiler provided here.
`-o`, `--output-file <file>`	Compile-and-link/library generation	Specify name and location of the output file.

16.6.3.4.2 Commonly Used Arguments

✓ Use `--help` option for a list of all supported options.

Flag	Description
`-app-name`, `--app-name <name>`	Specify DPA application name for the DPA program. This option is required if multiple DPA programs are part of a host application because each DPA application must have a unique name. Default name is `__dpa_a_out`.
`-mcpu=<target_cpu>`	Specify the target DPA hardware for code generation. See DPA Hardware Architectures for more details. Supported values: `nv-dpa-bf3`, `nv-dpa-cx7`
`-flto`, `--flto`	Enable link-time optimization (LTO) for device code. Specify this option during compilation along with an optimization level in `devicecc-options`.
`-devicecc-options`, `--devicecc-options <options>`,...	Specify the list of options to pass to the device compiler.
`-devicelink-options`, `--devicelink-options <options>`,...	Specify the list of options to pass during device linking stage.
`-device-libs`, `--device-libs '-L<path> -l<name>'`,...	Specify the list of device libraries including their names (in `-l`) and their paths (in `-L`). FlexIO libraries are linked by default.
`-I`, `--common-include-path <path>`,...	Specify include search paths common to host and device code compilation. FlexIO headers paths are included by DPACC by default.
`-o`, `--output-file <file>`	Specify name and location of the output file. • Compile-only mode - name of the output DPA object file. If not specified, `.dpa.o` is generated for each `.c` file. • Compiler-and-link mode - name of the output DPA program. This is a mandatory option in compiler-and-link mode. • Library generation mode - name of the output library. This is a mandatory option for this mode. Output files `<name>_device.a` and `<name>_host.a` are generated.
`-hostcc-options`, `--hostcc-options <options>`,...	Specify the list of options to pass to the host compiler.
`-gen-libs`, `--gen-libs`	Generate a DPA library from input files
`-ldoca_dpa`, `--ldoca_dpa`	Link with DOCA-DPA libraries

Using machine dependent options (e.g., `-mcpu`, `-march`, `-mabi`) through `-devicecc-options` to influence compiler code generation is not supported.

The `devicecc-options` option allows passing any option to the device compiler. However, passing options that prevent compilation of the input file may lead to unexpected behavior (e.g., `-devicecc-options="-version"` makes the device compiler print the version and not process input files).

Incompatible options that affect DPA global function argument sizes during DPACC invocation and host application compilation may lead to undefined behavior during execution (e.g., passing `-hostcc-options="-fshort-enums"` to DPACC and missing this option when building the host application).

16.6.3.4.3 DPA Hardware Architectures

The following table mentions the DPA architectures, the associated values supported in the compiler through the `-mcpu` option, and the macros defined by the compiler to identify these architectures.

Hardware	Value	Macro
ConnectX-7	`nv-dpa-cx7`	`__NV_DPA_CX7`
BlueField-3	`nv-dpa-bf3`	`__NV_DPA_BF3`

Since ConnectX-7 and BlueField-3 share the same DPA hardware, `nv-dpa-cx7` is treated as an alias of `nv-dpa-bf3` by the compiler.

16.6.3.4.4 Architecture Macros

As described in section "DPA Hardware Architectures", the compiler defines identifier macros for each version of DPA hardware. Each identifier macro has a unique integer value which is strictly greater than that of macros for older DPA CPU models. Known aliases such as BlueField-3 DPA and ConnectX-7 DPA share the same integer value. The macro `__NV_DPA` is defined to the value of current compilation target. This can be used to write device code specific to a DPA hardware generation as shown in the following:

```
#if __NV_DPA == __NV_DPA_BF3
// Code for Bluefield-3 here
#elif __NV_DPA > __NV_DPA_BF3
// Code for devices after Bluefield-3 here
#endif
```

The ordering established by the value of the hardware version identifier macros does not imply an ordering of features supported by hardware. It is the user responsibility to ensure that features used in the code which are specific for a DPA version are actually supported on the hardware.

16.6.3.4.5 LTO Usage Guidelines

16.6.3.4.5.1 Restrictions

- Only the default linker script is supported with LTO
- Using options `-fPIC` / `-fpic` / `-shared` / `-mcmodel=large` through `-devicecc-options` is not supported when LTO is enabled
- Fat objects containing both LLVM bitcode and ELF representation are not supported
- Thin LTO is not supported

16.6.3.4.5.2 Compatibility

During compilation, LLVM generates the object as bitcode IR (intermediate representation) when LTO is enabled instead of ELF representation. The bitcode IR generated by the DPA compiler is only guaranteed to be compatible within the same version of DPACC. All objects involved in link-time optimization (enabled with `-flto`) must be built with the same version of DPACC.

16.6.3.4.6 Deprecated Features

- The `-ldpa` option which links with DOCA-DPA libraries is deprecated and will be removed in future releases. Use the option `-ldoca_dpa` instead.

16.6.3.4.7 Examples

This section provides some common use cases of DPACC and showcases the `dpacc` command.

16.6.3.4.7.1 Building Libraries

This example shows how to build DPA libraries using DPACC. Libraries for DPA typically contain two archives, one for the host and one for the device.

```
dpacc input.c -hostcc=gcc -o lib<name> -gen-libs -hostcc-options="-fPIC"
```

This command generates the output files `lib<name>_host.a` and `lib<name>_device.a`.

The host stub archive can be linked with other host code to generate a shared/static host library.

- Generating a static host library:

```
ar x lib<name>_host.a                    # Extract objects to generate *.o
ar cr lib<name>.a <*src.host.o> *.o      # Generate final static archive with all objects
```

- Generating a shared host library:

```
gcc -shared -o lib<name>.so <*src.host.o> -Wl,-whole-archive -l<name>_host -Wl,-no-whole-archive       #
Link the generated archive to build a shared library
```

16.6.3.4.7.2 Linking with DPA Device Library

The DPA device library generated by DPACC using `-gen-libs` as part of a DPA library can be consumed by DPACC using the `-device-libs` option.

```
dpacc input.c -hostcc=gcc -o libInput.a -device-libs="-L <path-to-library> -l<libName>"
```

16.6.3.4.7.3 Enabling Link-time Optimizations

Link-time optimizations can be enabled using `-flto` along with an optimization level specified for device compilation.

```
dpacc input1.c -hostcc=gcc -c -flto -devicecc-options="-O2" dpacc input2.c -hostcc=gcc -c -flto -devicecc-
options="-O2" dpacc input1.dpa.o input2.dpa.o -hostcc=gcc -o libInput.a
```

16.6.3.4.7.4 Including Headers

This example includes headers for device compilation using `devicecc-options` and host compilation using `hostcc-options`. You may also specify headers for any compilation on both the host and device side using the `-I` option.

```
dpacc input.c -hostcc=gcc -o libInput.a -I <common-headers-path> -devicecc-options="-I <device-headers-path>"
-hostcc-options="-I <host-headers-path>"
```

16.6.3.4.7.5 Generating Output as Source Code

DPACC provides an option, `-src-output`, to generate the output as host source code. This source can be compiled by the host compiler to generate functionally equivalent output which DPACC would have generated directly.

This example shows how to build various outputs of DPACC as source using this option and how to compile the generated source.

DPA-program Source

Generate DPA-program source by passing the following option to DPACC:

```
dpacc input.c -hostcc=gcc -o libfoo.c -src-output
```

Compile the generated source using host compiler to generate an object and build an archive with this object. A macro `__DPACC_SRC_TARGET__` must be defined when building this object to remove code which is unnecessary when building from source:

```
$ gcc libfoo.c -c -I /opt/mellanox/flexio/include -Wno-attributes -Wno-pedantic -Wno-unused-parameter -Wno-return-
type -Wno-implicit-function-declaration -D__DPACC_SRC_TARGET__
$ ar cr libfoo.a libfoo.o
```

DPA-library Source

Generate DPA-library source by passing the following option to DPACC:

```
dpacc input.c -hostcc=gcc -o libfoo -gen-libs -src-output
```

This generates the device archive `libfoo_device.a` and host code files `libfoo.lib.c` and `input.dpa.c`.

The host archive of DPA-library is generated by compiling these sources and building an archive. The `__DPACC_SRC_TARGET__` macro must be defined in this instance to remove unnecessary code:

```
$ gcc libfoo.lib.c input.dpa.c -c -I /opt/mellanox/flexio/include -Wno-attributes -Wno-pedantic -Wno-unused-
parameter -Wno-return-type -Wno-implicit-function-declaration -D__DPACC_SRC_TARGET__
$ ar cr libfoo_host.a libfoo.lib.o input.dpa.o
```

DPA-object Source

Generate DPA-object source by passing the following option to DPACC:

```
dpacc input.c -hostcc=gcc -c -src-output
```

This generates a single file, `input.dpa.c`.

Compile the host file to generate an object:

```
gcc input.dpa.c -c -I /opt/mellanox/flexio/include -Wno-attributes -Wno-pedantic -Wno-unused-parameter -Wno-return-
type -Wno-implicit-function-declaration
```

16.6.3.4.8 DPA Compiler Usage

dpa-clang is a compiler driver for accessing the Clang/LLVM compiler, assembler, and linker which accepts C code files or object files and generates an output according to different usage modes.

> Invoking the compiler, assembler, or linker directly may lead to unexpected errors.

Refer to the following resources for more detailed information on Clang:

- Clang Compiler User's Manual
- Clang command line argument reference
- Target-dependent compilation options

16.6.3.4.8.1 Compiler Driver Command-line Options

```
dpa-clang <list-of-input-files> [other-options]
```

16.6.3.4.8.2 Linker Command Line Options

LLD is the default linker provided in the DPA toolchain. Linker-related options are passed to through the compiler driver.

```
dpa-clang -Wl,<linker-option>
```

For more information, please refer to the LLD command line reference.

16.6.3.4.8.3 dpacc-extract Command Line Options

`dpacc-extract` is a tool for extracting a device executable out of a DPA program or a host executable containing DPA program(s).

To execute dpacc-extract:

```
Usage: dpacc-extract <input-file> -o=<output-file> [other options]
Helper Flags:

  -o, --output-file                   Specify name of the output file
  -app-name, --app-name <name>        Specify name of the DPA application to extract
  -h, --help                          Print help information about dpacc-extract
  -V, --version                       Print dpacc-extract version information
  -optf, --options-file <file>,...    Include command line options from the specified file
```

Mandatory arguments:

Flag	Description
Input file	DPA program or host executable containing DPA program. Specifying one input file is mandatory.
`-o`, `--output-file <file>`	Specify name and location of the output device executable.
`-app-name`, `--app-name <name>`	Specify name of the DPA application to extract. Mandatory if input file has multiple DPA apps.

16.6.3.4.8.4 Objdump Command Line Options

The dpa-objdump utility prints the contents of object files and final linked images named on the command line.

For more information, please refer to the Objdump command line reference.

Commonly used dpa-objdump options:

Flag	Description
`--mcpu=nv-dpa-bf3`	Option to choose micro-architecture for DPA processor. `nv-dpa-bf3` is the default CPU for dpa-objdump.

16.6.3.4.8.5 Archiver Command Line Options

dpa-ar is a Unix ar-compatible archiver.

For more information, please refer to the Archiver command line reference.

16.6.3.4.8.6 NM Tool Command Line Options

The dpa-nm utility lists the names of symbols from object files and archives.

For more information, please refer to the NM tool command line reference.

16.6.3.4.8.7 Common Compiler Options

Flag	Description
`--mcpu=nv-dpa-bf3`	Option to choose micro-architecture and ABI for DPA processor. `nv-dpa-bf3` is the default CPU for the compiler.
`-mrelax` / `-mno-relax`	Option to enable/disable linker relaxations.
`-I <dir>`	Option to include header files present in `<dir>`.

16.6.3.4.8.8 Common Linker Options

Flag	Description
`-Wl,-L <path-to-library> -Wl,-l<library-name>`	Option to link against libraries

⚠ Linker options are provided through the compiler driver dpa-clang.

⚠ The LLD linker script is honored in addition to the default configuration rather than replacing the whole configuration like in GNU lD. Hence, additional options may be required to override some default behaviors.

16.6.3.4.8.9 Debugging Options

Flag	Description
`-fdebug-macro`	Option to emit macro debugging information. This option enables macro-debugging similar to GCC option `-g3`.

16.6.3.4.8.10 Miscellaneous Notes

- Objects produced by LLD are not compatible with those generated by any other linker.
- The default debugging standard of the DPA compiler is DWARFv5. GDB versions <10.1 have issues processing some DWARFv5 features. Use the option `-devicecc-options="-gdwarf-4"` with DPACC to debug with GDB versions <10.1.

16.6.4 NVIDIA DOCA DPA Execution Unit Management Tool

This document describes the DPA Execution Unit (EU) management tool, `dpaeumgmt`.

⚠ Execution unit partitions will be supported in future releases.

16.6.4.1 Introduction

This table introduces important terms for understanding this document:

Term	Definition
DPA	Data-path accelerator; an auxiliary processor designed to accelerate data-path operations.
DPA partition manager	PCIe device function capable of controlling the entire system's EUs. On NVIDIA® BlueField®-3 it is the ECPF. The DPA partition manager is by default associated with the default partition.
EU	Hardware execution unit; a logical DPA processing unit.
EU group	Collection/subset of EUs which could be created using `dpaeumgmt`. EU groups are created under an EU partition and could only be formed from the pool of EUs under that partition.
EU object	EU partition or EU group.
EU partition	An isolated pool of EUs which may be created using `dpaeumgmt`. Only when a partition is created and associated with other vHCAs are they able to use hardware resources and execute a DPA software thread.
EU affinity	The method by which a DPA thread is paired with a DPA EU. DPA supports three types of affinity: • `none` - selects an EU from a pool of all available EUs • `strict` - select only the specified EU (by ID) • `group` - select an EU from all the EUs in the specified group

The DPA EU management tool can run either on the host machine or on the target DPU and allows users to manage the DPA's EUs which are the basic resource of the DPA. The tool enables the resource control of EUs to optimize computation resources usage of the DPA before using DOCA FlexIO SDK API.

Without EU allocation, a DPA software thread would lack access to the hardware pipeline/CPU time resource, and consequently not be able to execute.

`dpaeumgmt` serves the following main usages:

- Running a DPA software thread with `strict` affinity on a DPA EU (i.e., running a DPA thread using only the specific preselected EU). For this purpose, `dpaeumgmt` provides an option to query the maximum EU ID allowed to use.
- Allowing a DPA software thread to run over a DPA EU from a group of EUs:
  - Once an EU group is created, it is allocated a subset of EUs.
  - `dpaeumgmt` provides an ID to the created group which can be used to run DPA applications with `group` affinity where the affinity ID would be the same as that group's ID.
- EU partition management - the ability to manage EU partitions.

When the software stack wishes to run a DPA thread with `group` affinity type, one of the available EUs from the group's collection is used for the execution.

> ⚠ A DPA thread may execute if and only if there is an available EU for it.

16.6.4.2 Execution Unit Objects

Upon boot, a default EU partition is automatically created. The default EU partition possesses all the system's EUs. The DPA partition manager function is the only function that belongs to it and can therefore control the entire resources of the system.

When running a DPA thread with `none` affinity, the EU chosen for the DPA thread to run with comes from the partition's pool of EUs. Namely, from the EUs belonging only to the DPA device's current partition which were not assigned to any EU groups (on the current partition). If the aforementioned group of EUs (i.e., the partition's default EU group) is empty, the DPA thread would fail to run with `none` affinity.

16.6.4.3 dpaeumgmt Commands

`dpaeumgmt` enables users to create, destroy, and query EU objects.

> ⚠ `dpaeumgmt` tool must run with root privileges and users must execute `sudo mst start` before using it.

Top-level `dpaeumgmt` command syntax:

```
Usage: dpaeumgmt {help|version|eu_group|partition}
Type "./dpaeumgmt help" for detailed help
```

16.6.4.3.1 General Commands

- Print basic usage information for the tool:

  ```
  dpaeumgmt -h
  ```

- Print a detailed help menu of the tool's commands:

  ```
  dpaeumgmt help
  ```

- Print version information:

  ```
  dpaeumgmt version
  ```

16.6.4.3.2 Execution Unit Group Commands

The commands listed in the following subsections are used to configure EU groups.

16.6.4.3.2.1 EU Group Command Flags and Arguments

The following table lists the flags relevant to eu `_group` commands. Arguments for the flags must be used within quotes (if more than one) and without extra spaces.

Short Option	Long Option	Description
`-h`	`--help`	Print out basic tool usage information.
`-d`	`--dpa_device`	The device interface name (MST/PCI/RDMA/NET).
`-r`	`--range_eus`	The range of EUs to allocate an EU group or a partition. The argument must be provided within quotes.
`-g`	`--id_group`	Group ID number. This number must be positive and less than or equal to the `max_num_dpa_eu_group` parameter which may be retrieved using the command eu `_group info -d <device>` .
`-n`	`--name_group`	Group name; 15-character string. The argument must be provided within quotes.
`-f`	`--file_groups`	Full path or only the filename if it is located in the same directory as the executable directory (where `dpaeumgmt` is).

16.6.4.3.2.2 Info EU Group

Print information on the relevant DPA resources for the EU groups:

```
dpaeumgmt eu_group info --dpa_device <device>
```

Example:

```
$ sudo ./dpaeumgmt eu_group info -d mlx5_0
Max number of DPA EU groups: 15
Max number of DPA EUs in one DPA EU group: 190
Max DPA EU number available to use: 190
Max EU group name length is 15 chars
```

16.6.4.3.2.3 Create EU Group

Create an EU group with the specified name on the provided device's partition. The EUs indicated by the range are taken from the DPA device's EU partition.

```
dpaeumgmt eu_group create --dpa_device <device> --name_group <name> --range_eus <range>
```

Example:

```
$ sudo ./dpaeumgmt eu_group create -d mlx5_0 -n "HG hello world1" -r "6-8,16,55,70"
Group created successfully-
EU group ID: 1
EU group name: HG hello world
Member EUs are: 6,7,8,16,55,70
```

After successfully creating an EU group, users can run a DPA thread using `group` affinity with the affinity type set to the group's ID.

16.6.4.3.2.4 Destroy EU Group

Destroy an EU group that exists on the device's partition with either the provided group name or ID.

```
dpaeumgmt eu_group destroy --dpa_device <device> [--name_group <name> | --id_group <id>]
```

Example:

```
$ sudo ./dpaeumgmt eu_group destroy -d mlx5_0 -g 1
Group with group id: 1, was destroyed successfully
```

16.6.4.3.2.5 Query EU Group

Query EU groups residing on the provided device's partition. If one of the optional parameters is used, the command only queries the specific group and prints it if it exists:

```
dpaeumgmt eu_group query --dpa_device <device> [--name_group <name> | --id_group <id>]
```

Example:

```
$ sudo ./dpaeumgmt eu_group query -d mlx5_0
1) EU group ID: 1
EU group name: HG hello world
Member EUs are: 6,7,8,16,55,70

In total there are 1 EU groups configured.
```

More options:

```
$ sudo ./dpaeumgmt eu_group query -d mlx5_0 -n "HG hello world"
$ sudo ./dpaeumgmt eu_group query -d mlx5_0 -g 1
```

16.6.4.3.2.6 Apply EU Group

Apply the EU groups provided in the file on the device's partition:

```
dpaeumgmt eu_group apply --dpa_device <device> --file_groups <file>
```

File format example:

```
{
        "eu_groups": [
                { "name": "hg1", "range": "178-180"},
                { "name": "hg2", "range": "2-10"}
        ]
}
```

The command removes all the previous EU groups defined on the EU partition that the DPA device belongs to and applies the ones from the file.

Example:

```
$ sudo ./dpaeumgmt eu_group apply -d mlx5_0 --file_groups example.json
1) EU group ID: 1
EU group name: hg1
Member EUs are: 178,179,180

1) EU group ID: 2
EU group name: hg2
Member EUs are: 2,3,4,5,6,7,8,9,10

In total there are 2 EU groups configured.
```

16.6.4.3.3 EU Partition Commands

The commands listed in the following subsections are used to configure EU partitions.

16.6.4.3.3.1 EU Partition Command Flags and Arguments

The following table lists the flags relevant to EU `partition` commands. Arguments for the flags must be used within quotes (if more than one) and without extra spaces.

Short Option	Long Option	Description
`-h`	`--help`	Print out basic tool usage information.
`-d`	`--dpa_device`	The device interface name (MST/PCI/RDMA/NET).
`-r`	`--range_eus`	The range of EUs to allocate an EU group or a partition. The argument must be provided within quotes.
`-p`	`--id_partition`	Partition ID number. This number must be positive and less than or equal to the value of `max_num_dpa_eu_partition` which may be retrieved using the command `partition info -d <device>`.
`-v`	`--vhca_list`	The vHCA IDs to be associated with the partition. The argument must be provided within quotes.
`-m`	`--max_num_eu_group`	The number of EU groups to reserve for the partition upon its creation.

16.6.4.3.3.2 Info EU Partition

Print the relevant DPA resources of the EU partitions:

```
dpaeumgmt partition info --dpa_device <device>
```

Example:

```
$ sudo ./dpaeumgmt partition info -d mlx5_0
Max number of DPA EU partitions: 15
Max number of VHCAs associated with a single partition: 32
Max number of DPA EU groups: 15
Note- an allocation of a partition consumes from the number of DPA EU *groups* available to create
Max DPA EU number available to use: 190
```

16.6.4.3.3.3 Create EU Partition

Create an EU partition on the DPA device:

```
dpaeumgmt partition create --dpa_device <device> --vhca_list <id_list> --range_eus <range> --max_num_eu_group
<max_num>
```

Example:

```
$ sudo ./dpaeumgmt partition create -d mlx5_0 -v 1 -r 10-20 -m 2
Partition created successfully-
EU Partition ID: 1
Maximal number of groups: 2
The partition has a total of 1 associated VHCA IDs, namely: 1
Partition's member EUs are: 10,11,12,13,14,15,16,17,18,19,20
```

16.6.4.3.3.4 Destroy EU Partition

Destroy an EU partition that exists on the device's partition:

```
dpaeumgmt partition destroy --dpa_device <device> --id_partition <id>
```

Example:

```
$ sudo ./dpaeumgmt partition destroy -d mlx5_0 -p 1
Partition with partition id: 1, was destroyed successfully
```

16.6.4.3.3.5 Query EU Partition

Query EU partitions that reside on the provided device's partition and print out the partition if it exists:

```
dpaeumgmt partition query --dpa_device <device> [--id_partition <id>]
```

Example:

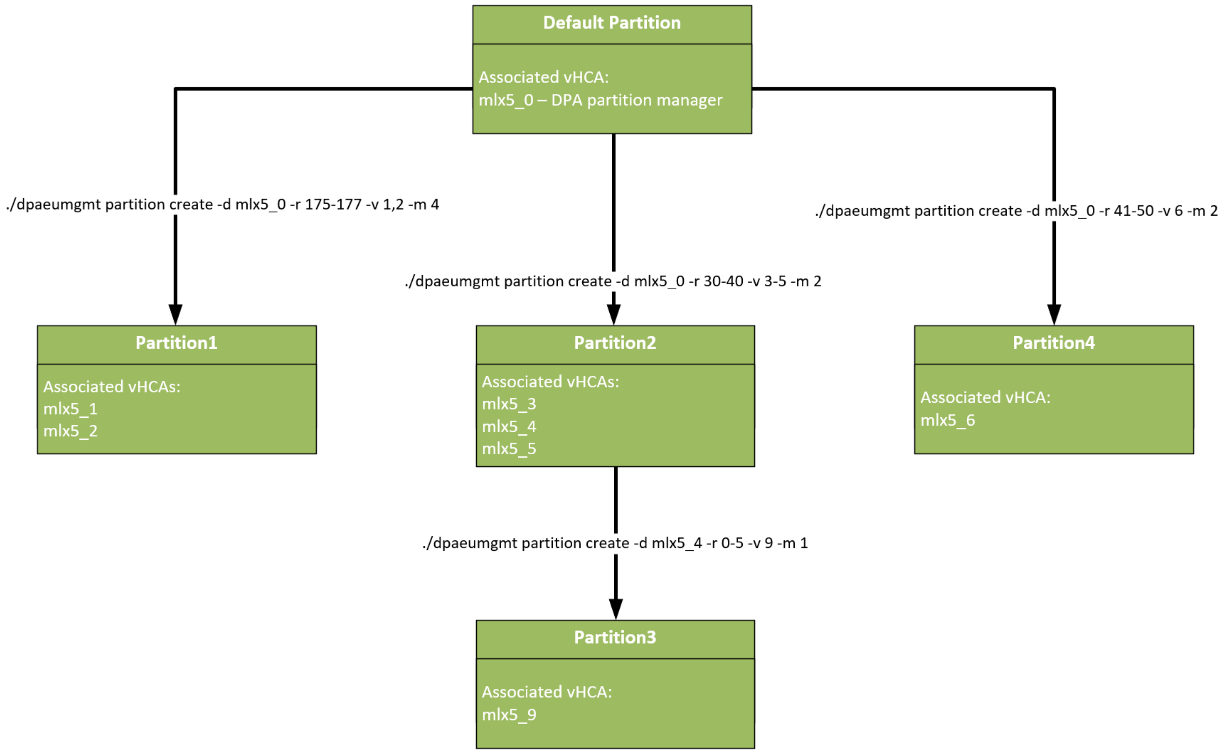
```
$ sudo ./dpaeumgmt partition query -d mlx5_0 -p 1
EU Partition ID: 1
Maximal number of groups: 2
The partition has a total of 1 associated VHCA IDs, namely: 1
Partition's member EUs are: 10,11,12,13,14,15,16,17,18,19,20
```

More options:

```
$ sudo ./dpaeumgmt partition query -d mlx5_0
```

16.6.4.4 vHCAs and Partitions

The following diagram illustrates the ownership and control of a partition by a vHCA and also which vHCAs have claim to (i.e., can use) a partition.


Default Partition
Associated vHCA:
mlx5_0 – DPA partition manager
./dpaeumgmt partition create -d mlx5_0 -r 175-177 -v 1,2 -m 4
./dpaeumgmt partition create -d mlx5_0 -r 30-40 -v 3-5 -m 2
./dpaeumgmt partition create -d mlx5_0 -r 41-50 -v 6 -m 2
Partition1
Associated vHCAs:
mlx5_1
mlx5_2
Partition2
Associated vHCAs:
mlx5_3
mlx5_4
mlx5_5
Partition4
Associated vHCA:
mlx5_6
./dpaeumgmt partition create -d mlx5_4 -r 0-5 -v 9 -m 1
Partition3
Associated vHCA:
mlx5_9

16.6.4.5 Known Limitations

- Currently, `dpaeumgmt` is only supported on the DPU not the host
- `dpaeumgmt` should run before creating a DPA process so all resources are configured ahead of time
    - Running the tool over a device with an existing DPA process results in failure
- The EU group name assigned by the user must be unique for every EU group on a specific partition or the EU group create command fails
- The creation of an EU partition consumes from the number of EU groups allowed on the vHCA's partition it is created on:
    - 1 group for the partition itself due to a default group created for each partition
    - `<max_num>` of groups which is the user's input provided upon partition creation
- Creating groups or running DPA threads in general (with any affinity) on interfaces other than ECPF, requires a configuration of a valid partition for the specific vHCA
- Only the default partition is exposed to the real EU numbers, all other partitions the user creates use virtual EUs
    - For example, if a user creates a partition with the range of EUs 20-40, querying the partition info from one of its virtual HCAs (vHCAs) would display EUs from 0-20. Therefore, the EU whose real number is 39 in this example would correspond to the virtual EU number 19.
- Group IDs on a non-default partition are virtual.
    - Different partitions can have completely distinct groups, even if they have the same ID.
    - The affinity ID parameter, specified on the FlexIO API, can distinguish between the groups according to the vHCA an application is running on.
- vHCA ID overlap is not allowed on EU partitions

- It is not possible to query vHCA IDs with `dpaeumgmt`, these are assumed to be known by the user beforehand
- Partition destruction fails if there are EU objects that exist on that partition
- It is not possible to know which EU has been chosen to run on
- Every vHCA sees the partition it belongs to, and its resources, as the entire world. It only sees:
    - Groups and partitions it created
    - The number of EUs it was given
    - The `max_num_eu_group` of the partition it belongs to
- No guarantee regarding EU group ID that will be given on group creation
- The default groups (of every partition) cannot be managed by the user
- The EU numbers available are between 0 and the max DPA EU number available to use minus 1 (the upper limit can be queried using the info command specified above)
- `dpaeumgmt` does not support virtual functions (VFs)
- It is not possible to create partitions on other vHCAs other than the DPA partition manager function
- There are at most 16 hardware EU group entities

16.6.5 NVIDIA DOCA DPA GDB Server Tool

This document describes the DPA GDB Server tool.

ⓘ The DPA GDB Server Tool is currently supported at beta level.

16.6.5.1 Introduction

The DPA GDB Server tool (`dpa-gdbserver`) enables debugging FlexIO DEV programs.

DEV programs for debugging are selected using a token (8-byte value) provided by the FlexIO process owner.

ⓘ Any GDB, familiar with RISC-V architecture, can be used for the debug. Refer to this page for information how to work with GDB.

16.6.5.1.1 Glossary

Term	Description
PUD	Process under debug. DEV-side processes intended for debug.
EU	Execution unit (similar to hardware CPU core)
DPA	Data path accelerator
RPC	Remote process communication. Mechanism used in FlexIO to run DEV-side code instantly. Runtime is limited to 6 seconds.
HOST	x86 or aarch64 Linux OS which manages dev-side code (i.e., DEV)

Term	Description
DEV	RISC-V code, loaded by HOST into the DPA's device. Triggered to run by different types of interrupts. DEV side is directly connected to ConnectX adapter card.
GDB	GNU Project debugger. Allows users to monitor another program while it executes.
GDBSERVER	Tool for remote debug programs
RTOS	Real-time operation system running on RISC-V core. Manages handling of interrupts and calls to DEV user processes routines.
RSP	Remote serial protocol. Used for interaction between GDB and GDBSERVER.

16.6.5.1.2 Known Limitations

- DPA GDB technology does not catch fatal errors. Therefore, if a fatal error occurs, core dump (created by `flexio_coredump_create()`) should be used.
- DPA GDB technology does not support Outbox access. GDB users cannot write to Doorbell or to Window configuration areas.
- DPA GDB technology does not support Window access. Read/write to Window memory does not work properly.

16.6.5.2 DPA-specific Notes

16.6.5.2.1 Token

The process under debug (PUD) can expose a debugging token. Every external process, using this token, get full access to the process with given token. To not show it constantly (e.g., for security reasons), users can modify their host application temporary. See `flexio_process_udbg_token_get()` .

16.6.5.2.2 Connection on Application Launch

If the code which needs debugging begins to run immediately after launch, the user should modify the host application to stop upon start to give the user time to run `dpa-gdbserver` . One possible way of doing this is to place function `getchar()` immediately after process creation.

16.6.5.2.3 Dummy Thread Concept

Something to consider with DPA debugging is that a PUD does not have a running thread all time (e.g., the process's thread may exist but be waiting for incoming packets). In a regular Linux application, this scenario is not possible and GDB does not support such cases.

Therefore, when no thread is running, `dpa-gdbserver` reports a dummy thread:

gdb

```
(gdb) info thread
  Id   Target Id                              Frame
* 1    Thread 1.805378433 (Dummy Flexio thread) 0x0800000000000000 in ?? ()
(gdb)
```

In this case user can inspect memory, create breakpoints, and give the `continue` command.

Commands like `step`, `next`, and `stepi` can not be executed for the Dummy thread.

16.6.5.2.4 Watchdog Issues

The RTOS has a watchdog timer that limits DEV code interrupt processes to 120 seconds. This timer is stopped when the user connects to DEV with GDB. Therefore users will have no time limitation for debugging.

16.6.5.3 Tool TCP Port and Execution Unit (EU)

By default, `dpa-gdbserver` uses TCP port 1981 and runs on EU 29. If this conflicts with another application (or if other instances of `dpa-gdbserver` are running), users should change the defaults as follows:

Bash

```
$> dpa-gdbserver mlx5_0 -T <token> -s <port> -E <eu_id>
```

16.6.5.4 Debugging

16.6.5.4.1 Preparation for Debug

Modify your FlexIO application if needed. Make sure the HOST code prints `udbg_token` and waits for GDB connection if needed:

C code. Host side. diff

```
+   uint64_t udbg_token;

    flexio_process_create(..., &flexio_process);

+   udbg_token = flexio_process_udbg_token_get(flexio_process);
+   if (udbg_token)
+       printf("Process created. Use token >>> %#lx <<< for debug\n", udbg_token);

+   printf("Stop point for waiting of GDB connection. Press Enter to continue..."); /* Usually you don't need this
stop point */
+   fflush(stdout);
+   getchar();
```

Extract the DPA application from the FlexIO application. For example:

Bash

```
$> dpacc-extract cc-host/app/host/flexio_app_name -o flexio_app_name.rv5
```

16.6.5.4.2 Start Debugging

1. Run your FlexIO application. It should expose the debug token:

   Bash

   ```
   $> flexio_app_name mlx5_0
   Process created. Use token >>> 0xd6278388ce4e682c <<< for debug
   ```

2. Run `dpa-gdbserver` with the debug token received:

   Bash

   ```
   $> dpa-gdbserver mlx5_0 -T 0xd6278388ce4e682c
   Registered on device mlx5_0
   Listening for GDB connection on port 1981
   ```

3. Run any GDB with RISC-V support. For example, `gdb-multiarch`:

   Bash

   ```
   $> gdb-multiarch -q flexio_app_name.rv5
   Reading symbols from flexio_app_name.rv5...
   (gdb)
   ```

4. Connect to the gdbserver using proper TCP port and hostname, if needed:

   gdb

   ```
   (gdb) target remote :1981
   Remote debugging using :1981
   0x0800000000000000 in ?? ()
   ```

16.6.5.4.3 DPA-specific Debugging Techniques

16.6.5.4.3.1 Easy Example of Transitioning from Dummy to Real Thread

Transitioning between the dummy thread and a real thread is not standard practice for debugging under GDB. In an ideal situation, the user would know exactly the entry points for all their routines and can set breakpoints for all of them. Then the user may run the `continue` command:

gdb

```
(gdb) target remote :1981
Remote debugging using :1981
0x0800000000000000 in ?? ()
(gdb) info threads
  Id   Target Id                                        Frame
```

```
* 1    Thread 1.805378433 (Dummy Flexio thread) 0x0800000000000000 in ?? ()
(gdb) b foo
Breakpoint 1 at 0x400000b2: file ../tests/path/hello.c, line 58.
(gdb) b bar
Breakpoint 2 at 0x40000518: file ../tests/path/hallo.c, line 113.
(gdb) continue
Continuing.
```

Initiate interrupts for your DEV program (depends your task), and GDB should catch a breakpoint and now the real thread of the PUD appear instead of the dummy:

gdb

```
(gdb) continue
Continuing.
(gdb) [New Thread 1.2]
[New Thread 1.130]
[New Thread 1.258]
[New Thread 1.386]
[Switching to Thread 1.2]

Thread 2 hit Breakpoint 1, foo(thread_arg=9008)
    at ../tests/path/hello.c:58
58              struct host_data *hdata = NULL;
(gdb) info threads
  Id   Target Id                                   Frame
* 2    Thread 1.2 (Process 0 thread 0x1 GVMI 0)     foo (arg=9008) at ../tests/path/hello.c:58
  3    Thread 1.130 (Process 0 thread 0x81 GVMI 0)  foo (arg=9264) at ../tests/path/hello.c:58
  4    Thread 1.258 (Process 0 thread 0x101 GVMI 0) foo (arg=9648) at ../tests/path/hello.c:58
  5    Thread 1.386 (Process 0 thread 0x181 GVMI 0) foo (arg=9904) at ../tests/path/hello.c:58
(gdb)
```

From this point, you may examine memory and trace your code as usual.

16.6.5.4.3.2 Complicated Example of Transitioning from Dummy to Real Thread

In a more complicated situation, the interrupt happens after GDB connection. In this case, the real thread should start running but cannot because the PUD is in HALT state. The user can type the command `info threads`, see new thread instead of the old dummy, and then switch to the new thread manually:

gdb

```
 1  (gdb) target remote :1981
 2  Remote debugging using :1981
 3  0x0800000000000000 in ?? ()
 4  (gdb) info threads
 5    Id   Target Id                                   Frame
 6  * 1    Thread 1.805378433 (Dummy Flexio thread) 0x0800000000000000 in ?? ()
 7  (gdb) info threads
 8  [New Thread 1.32769]
 9    Id   Target Id                                          Frame
10    2    Thread 1.32769 (Process 0 thread 0x8000 GVMI 0) bar (arg=0xc0, len=0)
11      at /path/lib/src/stub.c:167
12
13  The current thread <Thread ID 1> has terminated.  See `help thread'.
14  (gdb) thread 2
15  [Switching to thread 2 (Thread 1.32769)]
16  #0  bar (arg=0xc0, len=0)
17      at /path/lib/src/stub.c:167
18  167     {
19  (gdb) bt
20  #0  bar (arg=0xc0, len=0)
21      at /path/lib/src/stub.c:167
22  #1  0x000000004000017a in foo (thread_arg=3221)
23      at ../path/dev/hello.c:182
24  #2  0x0000000000000000 in ?? ()
25  Backtrace stopped: frame did not save the PC
26  (gdb)
```

⚠ The same command `info threads` in lines 4 and 7 gives different results. This happens because the interrupt occurs between the instances and the real code begins to run.

The user must switch to the new thread manually (see line 14). After this, they can trace/debug the flow as usual (i.e., using the commands `step`, `next`, `stepi`).

16.6.5.4.3.3 Finishing Real Thread without Finishing PUD

Every interrupt handler at some point finishes its way and returns the CPU resources to RTOS. The most common way to do this is to call function `flexio_dev_thread_reschedule()`. The command `next` on this function will have the same effect as the command `continue`:

gdb

```
205         __dpa_thread_fence(__DPA_MEMORY, __DPA_W, __DPA_W);
(gdb) next
206         flexio_dev_cq_arm(dtctx, app_ctx.rq_cq_ctx.cq_idx, app_ctx.rq_cq_ctx.cq_number);
(gdb) next
208         if ((dev_errno = flexio_dev_get_and_rst_errno(dtctx))) {
(gdb) next
213         print_sim_str("Nothing to do. Wait for next duar\n", 0);
(gdb) next
214         flexio_dev_thread_reschedule();
(gdb) next
```

GDB waits until the user types `^C` or a breakpoint is reached after the next interrupt occurred.

16.6.5.5 Error Reporting

The DPA GDB server tool has been validated with `gdb-multiarch` (version 9.2) and with GDB version 12.1 from RISC-V tool chain.

The GDB server should support all commands described in GDB RSP (remote serial protocol) for GDB stubs. But only the most common GDB commands are supported.

Should a dpa-gdbserver bug occur, please provide the following data:

- Used GDB (name and version)
- Commands sequence to reproduce the issue
- DPA GDB server tool console output
- DPA GDB server tool log directory content (see next part for details)
- Optional - output data printed when `dpa-gdbserver` is run in verbose mode

16.6.5.5.1 Tool Log Directory

For every run, a temporary directory is created with the template `/tmp/flexio_gdbs.XXXXXX`.

To locate the latest one, run the following command:

Bash

```
$> ls -ldtr /tmp/flexio_gdbs.* | tail
```

16.6.5.5.2 Verbosity Level of gdbserver

By default, `dpa-gdbserver` does not print any log information to screen. Adding `-v` option to command line increases verbosity level, printing additional info to `dpa-gdbserver` terminal display. Verbosity level is incremented according to number of 'v' in command line switch (i.e. `-vv`, `-vvv` etc.).

One `-v` shows the RSP exchange. This is a textual protocol, so users can read and understand requests from GDB and answers from the GDB server:

gdbserver.log -v

```
<<<<< "qTStatus"
>>>>> ""
<<<<< "?"
>>>>> "S05"
<<<<< "qfThreadInfo"
>>>>> "mp01.30011981"
<<<<< "qsThreadInfo"
>>>>> "l"
<<<<< "qAttached:1"
>>>>> "1"
<<<<< "Hc-1"
>>>>> "OK"
<<<<< "qC"
>>>>> "QCp01.30011981"
```

In the examples, `<<<<<` and `>>>>>` are used to indicate data received from GDB and transmitted to GDB, respectively.

When running with a higher verbosity level (e.g., run `dpa-gdbserver` with option `-vv` or higher), the exchange with the RTOS module is shown:

gdbserver.log -vv

```
<<<<< "qfThreadInfo"
/  2/dgdbs_handler - cmd 0x5
/  2/dgdbs_handler - retval 0x4
>>>>> "mp01.30011981"
<<<<< "qsThreadInfo"
/  2/dgdbs_handler - cmd 0x5
/  2/dgdbs_handler - retval 0x5
>>>>> "l"
<<<<< "m800000000000000,4"
/  2/dgdbs_handler - cmd 0xc
/  2/dgdbs_handler - retval 0x9
>>>>> "E0a"
<<<<< "m7ffffffffffffffc,4"
/  2/dgdbs_handler - cmd 0xc
/  2/dgdbs_handler - retval 0x9
>>>>> "E0a"
<<<<< "qSymbol::"
>>>>> "OK"
```

Lines beginning with `/   #/` provide the number of internal RTOS threads printed from the DEV side.

16.6.5.6 Useful Info Regarding Work with GDB

This section provides useful information about commands and methods which can help users when performing DPA debug. This is not related to the `dpa-gdbserver` itself. But this is about remote debugging and FlexIO sources.

16.6.5.6.1 Command "directory"

GDB can run on a different host from the one where compilation was done. For example, users may have compiled and run their application on `host1` and run their instance of GDB on `host2`. In this case, users will see the error message `../xxx/yyy/zzz/your_file.c: No such file or directory`. To solve this problem, copy sources to the host running GDB (`host2` in the example). Make sure to save the original code hierarchy. Use GDB command `directory` to inform where the sources are to GDB:

gdb on host2

```
host2~$> gdb-multiarch -q /tmp/my_riscv.elf
Reading symbols from /tmp/my_riscv.elf...
(gdb) b foo
Breakpoint 1 at 0x4000016c: file ../xxx/yyy/zzz/my_file.c, line 182.
(gdb) target remote host1:1981
Remote debugging using host1:1981
0x0800000000000000 in ?? ()
(gdb) c
Continuing.
[New Thread 1.32769]
[Switching to Thread 1.32769]

Thread 2 hit Breakpoint 1, foo (thread_arg=5728) at ../xxx/yyy/zzz/my_file.c:182
182     ../xxx/yyy/zzz/my_file.c: No such file or directory.
(gdb) directory /tmp/apps/
Source directories searched: /tmp/apps:$cdir:$cwd
(gdb) list
179             struct flexio_dev_thread_ctx *dtctx;
180             uint64_t dev_errno;
181
182             print_sim_str("=====> NET event handler started\n", 0);
183
184             flexio_dev_print("Hello GDB user\n");
185
```

Pay attention to the exact path reported by GDB. The argument for the command `directory` should point to the start point for this path. For example, if GDB looks for `../xxx/yyy/zzz` and you placed the sources in local directory `/tmp/copy_of_worktree`, then the command should be `(gdb) directory /tmp/copy_of_worktree/`**`xxx/`** and not `(gdb) directory /tmp/copy_of_worktree/`.

Sometimes, the `*.elf` file provides a global path from the root. In this case, use the command `set substitute-path <from> <to>`. For example, if the file `/foo/bar/baz.c` was moved to `/mnt/cross/baz.c`, then the command `(gdb) set substitute-path /foo/bar /mnt/cross` instructs GDB to replace `/foo/bar` with `/mnt/cross`, which allows GDB to find the file `baz.c` even though it was moved.

See this page of GDB documentation for more examples of specifying source directories.

16.6.5.6.2 Core Dump Usage

If the code runs into a fatal error even though the host side of your project is implemented correctly, a core dump is saved which allows analyzing the core. It should point exactly to where the fatal error occurred. The command `backtrace` can be used to examine the memory and its registers. Change the frame to see local variables of every function on the backtrace list:

gdb

```
$> gdb-multiarch -q -c crash_demo.558184.core /tmp/my_riscv.elf
Reading symbols from /tmp/my_riscv.elf...

[New LWP 1]
#0  0x000000004000126e in read_test (line=153, ptr=0x30) at /xxx/yyy/zzz/my_file.c:109
109             val = *(volatile uint64_t *)ptr;
(gdb) bt
#0  0x000000004000126e in read_test (line=153, ptr=0x30) at /xxx/yyy/zzz/my_file.c:109
#1  0x000000004000031a in tlb_miss_test (op_code=1) at /xxx/yyy/zzz/my_file.c:153
#2  0x0000000040000144 in test_thread_err_events_entry_point (h2d_daddr=3221258560) at /xxx/yyy/zzz/my_file.c:588
#3  0x00000000400013fc in _dpacc_flexio_dev_arg_unpack_test_err_events_dev_test_thread_err_events_entry_point
(argbuf=0xc0008228, func=0x400000b0 <test_thread_err_events_entry_point>)
    at /tmp/dpacc_xExkvE/test_err_events_dev.dpa.device.c:67
#4  0x0000000040001680 in flexio_hw_rpc (host_arg=3221258752) at /local_home/www/flexio-sdk/libflexio-dev/src/
flexio_dev_entry_point.c:75
#5  0x0000000000000000 in ?? ()
Backtrace stopped: frame did not save the PC
(gdb) frame 4
#4  0x0000000040001680 in flexio_hw_rpc (host_arg=3221258752) at /local_home/igorle/flexio-sdk/libflexio-dev/src/
flexio_dev_entry_point.c:75
75                       retval = unpack_cb(&data_from_host->func_params.arg_buf,
(gdb) p /x *data_from_host
$2 = {poll_lkey = 0x1ff2b1, window_id = 0x3, poll_haddr = 0x55dc0f40b900, entry_point = 0x400013d8, func_params =
{func_wo_pack = 0x0, dev_func_entry = 0x400000b0, arg_buf = 0xc0008140}}
(gdb)
```

16.6.5.6.3 Debug of Optimized Code

Usually highly optimized code is compiled and run.

Two types of mistakes in code can be considered:

- Logical errors
- Optimization-related errors

Logical errors (e.g., using `&` instead of `&&`) are reproduced on the non-optimized version of the code. Optimization related errors (e.g., forgetting volatile classification, non-usage of memory barriers) only impact optimization. Non-optimized code is much easier for tracing with GDB, because every C instruction is translated directly to assembly code.

It is good practice to check if an issue can be reproduced on non-optimized code. That helps observing the application flow:

Bash

```
$> build.sh -O 0
```

For tracing this code, using GDB commands `next` and `step` should be sufficient.

But if an issue can only be reproduced on on optimized code, you should start debugging it. This would require reading disassembly code and using the GDB command `stepi` because it becomes a challenge to understand exactly which C-code line executed.

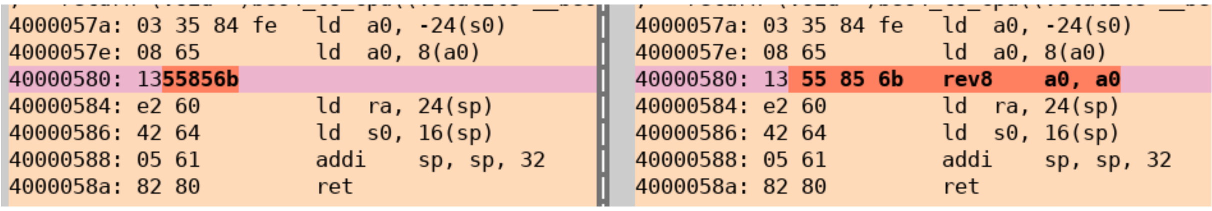
16.6.5.6.4 Disassembly of Advanced RISC-V Commands

DPA core runs on a RISC-V CPU with an extended instruction set. The GDB may not be familiar with some of those instructions. Therefore, `asm` view mode shows numbers instead of disassembly. In this case it is recommended to disassemble your RISC-V binary code manually. Use the `dpa-objdump` utility with the additional option `--mcpu=nv-dpa-bf3`.

bash

```bash
$> dpa-objdump -sSdxl --mcpu=nv-dpa-bf3 my_riscv.elf > my_riscv.asm
```

The following screenshot shows the difference:



```
4000057a: 03 35 84 fe    ld  a0, -24(s0)
4000057e: 08 65          ld  a0, 8(a0)
40000580: 1355856b
40000584: e2 60          ld  ra, 24(sp)
40000586: 42 64          ld  s0, 16(sp)
40000588: 05 61          addi    sp, sp, 32
4000058a: 82 80          ret
```

```
4000057a: 03 35 84 fe    ld  a0, -24(s0)
4000057e: 08 65          ld  a0, 8(a0)
40000580: 13 55 85 6b    rev8    a0, a0
40000584: e2 60          ld  ra, 24(sp)
40000586: 42 64          ld  s0, 16(sp)
40000588: 05 61          addi    sp, sp, 32
4000058a: 82 80          ret
```

16.6.6 NVIDIA DOCA DPA PS Tool

16.6.6.1 Introduction

DOCA `dpa-ps` is a CLI tool which allows users to monitor running DPA processes and threads. The tool presents sorted lists of the currently running DPA processes and threads.

The process ID output of the `dpa-ps` tool may be used as the input parameter for the `dpa-statistics` tool.

This tool is supported for NVIDIA® BlueField®-3 only.

16.6.6.2 Command Flags and Arguments

The following table lists the flags for the `dpa-ps` tool.

Short Option	Long Option	Description
`-h`	`--help`	Help information
`-d`	`--device`	Device interface name (MST/RDMA)
`-p`	`--process-id`	Hexadecimal process ID for filtering
`-t`	`--threads`	Show threads info for each process
`-i`	`--suppress-header-info`	Suppress print header info

Arguments for the flags must be used within quotes (if more than one) and without extra spaces.

16.6.6.3 Example

```
$ sudo ./dpa-ps -d mlx5_0 -t
ProcessID
    ThreadID
0
    5
    6
1
    3
    4
2
3
    0
    1
    2
4
```

16.6.6.4 Known Limitations

- The `dpa-ps` and `dpa-statistics` tools cannot be run at the same time on the same device

16.6.7 NVIDIA DOCA DPA Statistics Tool

16.6.7.1 Introduction

DOCA `dpa-statistics` is a CLI tool which allows users to monitor and obtain statistics on thread execution per running DPA process and thread. The tool is used to expose information about the running DPA processes and threads and to collect statistics on DPA thread performance.

The tool presents performance information for running DPA threads, including the number of cycles and instructions executed in a time period. The tool enables initiating and stopping collection of statistics and displaying the data collected per thread.

The process ID output of the `dpa-ps` tool may be used as the input parameter for the `dpa-statistics` tool.

This tool is supported for NVIDIA® BlueField®-3 only.

16.6.7.2 Collecting Performance Statistics Data

The command `collect` works on four mutually exclusive modes:

- Enable mode - start collecting performance data
- Disable mode - stop collecting performance data
- Timeout mode - start collecting, wait with a timeout, stop collect and print info. User could break the wait with Ctrl-C command and then the timeout will be canceled and tool will disable statistics collection and prints the info with the actual time of the collect operation.

- Infinite mode - no special flags. Same as timeout mode but with infinite timeout. The tool awaits the Ctrl-C command to stop.

The following table lists the `collect` command's flags and arguments:

Short Option	Long Option	Description
`-h`	`--help`	Help information
`-d`	`--device`	Device interface name (MST/RDMA)
`-p`	`--process-id`	Hexadecimal process ID for filtering This flag indicates a specific command for the command to operate on. Otherwise, statistics are collected from all processes.
`-i`	`--suppress-header-info`	Suppress print header info
`-n`	`--enable`	Enable collect info
`-o`	`--disable`	Disable collect info
`-t`	`--timeout`	Enable collect, wait with timeout, disable collect and print info Timeout value is in milliseconds. Examples for inputting timeout value: • 45 - 45 milliseconds • 45.55 - 45 milliseconds and 550,000 nanoseconds • .0005 - 500 nanoseconds • 45m55n - 45 milliseconds and 55 nanoseconds • 66n - 66 nanoseconds
`-r`	`--reset`	Reset counters before operation starting collect operation

16.6.7.3 Presenting Statistics List

Presenting performance statistics is applicable after initiating data collection.

The following table lists the `show` command's flags and arguments:

Short Option	Long Option	Description
`-h`	`--help`	Help information
`-d`	`--device`	Device interface name (MST/RDMA)
`-p`	`--process-id`	Hexadecimal process ID for filtering
`-i`	`--suppress-header-info`	Suppress print header info

Output example:

```
$ sudo ./dpa-statistics show -d mlx5_0 -p 1
ProcessID
    ThreadID    Cycles             Instruction        Time               Executions
1
    3           266268             18193              164                41
    4           411571             32727              252                47
```

Where:

- `ProcessID` - The `dpa_process_object_id` to which the threads belongs
- `ThreadID` - DPA thread object ID
- `Cycles` - Total EU cycles the thread used
- `Instruction` - Total number of instructions the thread executed
- `Time` - Total time in ticks the thread was active
- `Executions` - Total number of thread invocations

16.6.7.3.1 Examples

- Example of `collect` in infinite mode for process 0 with suppress header info:

```
$ sudo ./dpa-statistics collect -d mlx5_0 -p 0 -i
...^C
Data collected for 4606 milliseconds 0 nanoseconds
0
    5           223964             13754              140                31
    6           190130             13754              114                31
```

- Example of `collect` in timeout mode with a timeout of 1 second and half a millisecond.

```
$ sudo ./dpa-statistics collect -d mlx5_0 -t 1000.500
Data collected for 1000 milliseconds 500000 nanoseconds
ProcessID
    ThreadID    Cycles             Instruction        Time               Executions
0
    5           223964             13754              140                31
    6           190130             13754              114                31
1
    3           266268             18193              164                41
    4           411571             32727              252                47
2
3
    0           223205             13754              137                31
    1           189896             13754              113                31
    2           191796             13754              117                31
4
```

- Example of enabling statistics collection with reset of counters.

```
$ sudo ./dpa-statistics collect -d mlx5_0 -n -r
```

- Example of disabling statistics collection.

```
$ sudo ./dpa-statistics collect -d mlx5_0 -o
```

16.6.7.4 Known Limitations

- Reading large statistics counter blocks takes a long time
- The `dpa-ps` and `dpa-statistics` tools cannot be run at the same time on the same device

16.7 NVIDIA DOCA PCC Counter Tool

This document provides instruction on the usage of the PCC Counter tool.

16.7.1 Introduction

The PCC Counter tool is used to print PCC-related hardware counters. The output counters help debug the PCC user algorithm embedded in the DOCA PCC application.

16.7.2 Prerequisites

DOCA 2.2.0 and higher.

16.7.3 Description

If NVIDIA® BlueField®-3 is operating in DPU mode, the script must be executed on the Arm side. If BlueField-3 is operating in NIC mode, the script must be executed on the host side.

> ⓘ Refer to NVIDIA BlueField Modes of Operation for more information on the DPU's modes of operation.

The following performance counters are supported for PCC:

- `MAD_RTT_PERF_CONT_REQ` - the number of RTT requests received in total
- `MAD_RTT_PERF_CONT_RES` - the number of RTT responses received in total
- `SX_EVENT_WRED_DROP` - the number of TX events dropped due to the CC event queue being full
- `SX_RTT_EVENT_WRED_DROP` - the number of "TX event with RTT request sent indication" dropped due to the CC event queue being full
- `ACK_EVENT_WRED_DROP` - the number of Ack events dropped due to the CC event queue being full
- `NACK_EVENT_WRED_DROP` - the number of Nack events dropped due to the CC event queue being full
- `CNP_EVENT_WRED_DROP` - the number of CNP events dropped due to the CC event queue being full
- `RTT_EVENT_WRED_DROP` - the number of RTT events dropped due to the CC event queue being full
- `HANDLED_SXW_EVENTS` - the number of handled CC events related to SXW
- `HANDLED_RXT_EVENTS` - the number of handled CC events related to RXT
- `DROP_RTT_PORT0_REQ` - the number of RTT requests dropped in total from port 0
- `DROP_RTT_PORT1_REQ` - the number of RTT requests dropped in total from port 1
- `DROP_RTT_PORT0_RES` - the number of RTT responses dropped in total from port 0
- `DROP_RTT_PORT1_RES` - the number of RTT responses dropped in total from port 1
- `RTT_GEN_PORT0_REQ` - the number of RTT requests sent in total from port 0
- `RTT_GEN_PORT1_REQ` - the number of RTT requests sent in total from port 1
- `RTT_GEN_PORT0_RES` - the number of RTT responses sent in total from port 0
- `RTT_GEN_PORT1_RES` - the number of RTT responses sent in total from port 1
- `PCC_CNP_COUNT` - the number of CNP received in total, regardless of whether it is handled or ignored

16.7.4 Execution

To use the PCC Counter:

1. Initialize all supported hardware counters. Run:

```
sudo ./pcc_counters.sh set /dev/mst/mt41692_pciconf0
```

> Counters are zeroed after each `set` command.

2. Query all supported hardware counters. Run:

```
sudo ./pcc_counters.sh query /dev/mst/mt41692_pciconf0
```

> The output counters are counted from the time the `set` command is executed to the time when the `query` command is issued.

Example output:

```
sudo ./pcc_counters.sh query /dev/mst/mt41692_pciconf0
-----------------PCC Counters-----------------
Counter: MAD_RTT_PERF_CONT_REQ  Value: 000000000028b85b
Counter: MAD_RTT_PERF_CONT_RES  Value: 000000000028b85a
Counter: SX_EVENT_WRED_DROP     Value: 0000000000000000
Counter: SX_RTT_EVENT_WRED_DROP Value: 0000000000000000
Counter: ACK_EVENT_WRED_DROP    Value: 0000000000ccdf4f
Counter: NACK_EVENT_WRED_DROP   Value: 0000000000000000
Counter: CNP_EVENT_WRED_DROP    Value: 0000000000000000
Counter: RTT_EVENT_WRED_DROP    Value: 0000000000000000
Counter: HANDLED_SXW_EVENTS     Value: 000000000932543a
Counter: HANDLED_RXT_EVENTS     Value: 000000000028b85c
Counter: DROP_RTT_PORT0_REQ     Value: 0000000000000000
Counter: DROP_RTT_PORT1_REQ     Value: 0000000000000000
Counter: DROP_RTT_PORT0_RES     Value: 0000000000000000
Counter: DROP_RTT_PORT1_RES     Value: 0000000000000000
Counter: RTT_GEN_PORT0_REQ      Value: 0000000000000000
Counter: RTT_GEN_PORT1_REQ      Value: 000000000028b85c
Counter: RTT_GEN_PORT0_RES      Value: 0000000000000000
Counter: RTT_GEN_PORT1_RES      Value: 000000000028b85d
Counter: PCC_CNP_COUNT          Value: 0000000000000000
```

16.8 NVIDIA DOCA Socket Relay

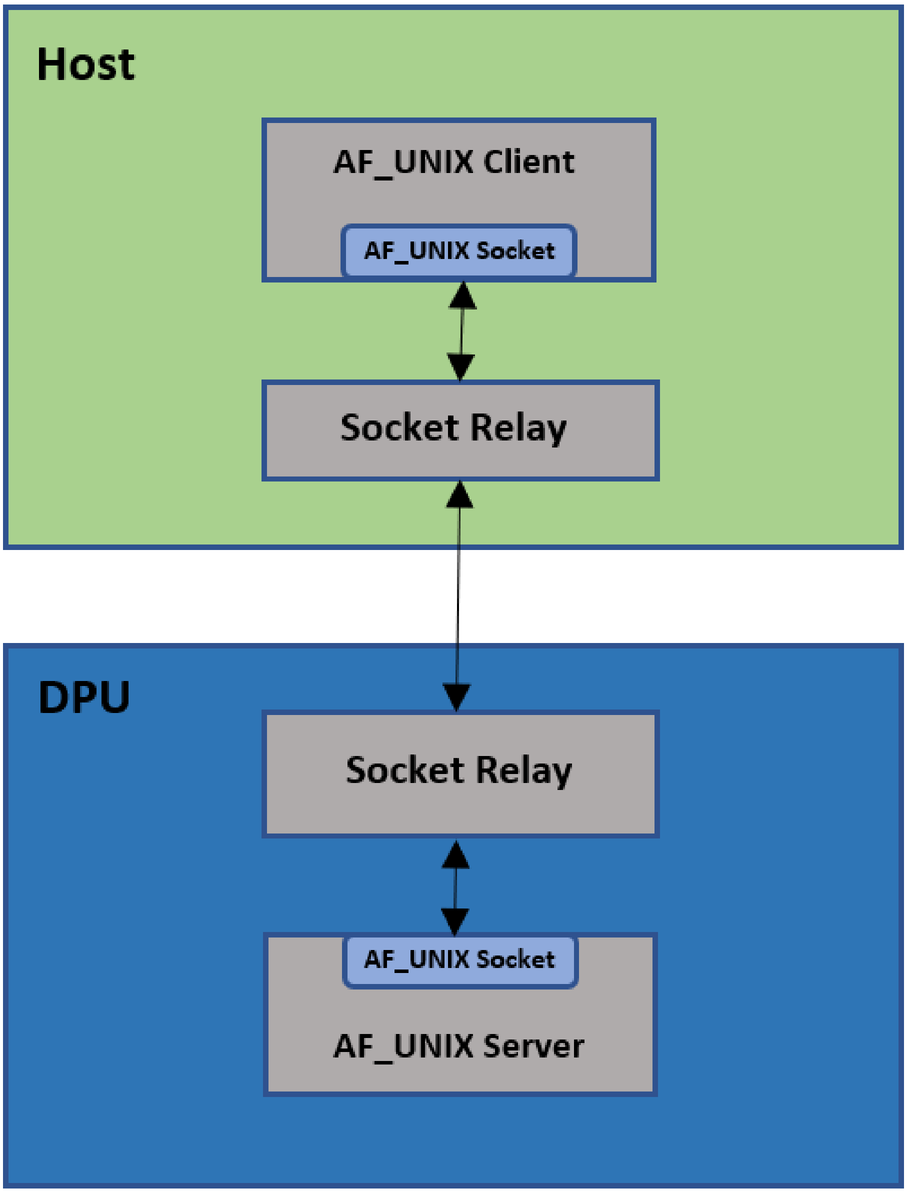
This document describes DOCA Socket Relay architecture, usage, etc.

16.8.1 Introduction

DOCA Socket Relay allows Unix Domain Socket (AF_UNIX family) server applications to be offloaded to the DPU while communication between the two sides is proxied by DOCA Comch.

Socket relay only supports SOCK_STREAM communication with a limit of 512 AF_UNIX application clients.

The tool is coupled to the client AF_UNIX server application. That is, a socket relay instance should be initiated per AF_UNIX server application.



Socket relay is transparent to the application except for the following TCP flows:

- Connection termination must be done by the host side application only
- Once a FIN packet (shutdown system call has been made) is sent by the host side application, data cannot be transferred between the DPU and the host, and the connection must be closed.

The following details the communication flow between the client and server:

- The AF_UNIX client application connects to the socket relay AF_UNIX server in the same way as in the original flow
- The AF_UNIX client application sends SOCK_STREAM packets
- The socket relay (host) AF_UNIX server receives the client application packets, and the Comm Channel client sends them on the channel
- The socket relay (DPU) Comm Channel server receives the client application packets and the AF_UNIX client sends them to the user's AF_UNIX server application

16.8.2 Prerequisites

Windows 10 build 17063 is the minimal Windows version to run DOCA Socket Relay on a Windows host.

16.8.3 Dependencies

NVIDIA® BlueField®-2 firmware version 24.35.1012 or higher.

16.8.4 Execution

To execute DOCA Socket Relay:

```
Usage: doca_socket_relay [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                    Print a help synopsis
  -v, --version                 Print program version information
  -l, --log-level               Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level               Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>             Parse all command flags from an input json file

Program Flags:
  -s, --socket                  Unix domain socket path, host side will bind to and DPU connect to
  -n, --cc-name                 Comm Channel service name
  -p, --pci-addr                DOCA Comm Channel device PCI address
  -r, --rep-pci                 DOCA Comm Channel device representor PCI address (needed only on DPU)
```

For example (DPU side):

```
doca_socket_relay -s /tmp/sr_server.socket -n cc_channel -p 03:00.0 -r b1:00.0
```

To run `doca_socket_relay` using a JSON file:

```
doca_socket_relay --json [json_file]
```

For example:

```
doca_socket_relay --json /tmp/doca_socket_relay.json
```

16.8.5 Arg Parser DOCA Flags

Refer to the DOCA Arg Parser for more information.

Flag Type	Short Flag	Long Flag/ JSON Key	Description	JSON Content
General flags	`h`	`help`	Prints a help synopsis	N/A
	`v`	`version`	Prints program version information	N/A

Flag Type	Short Flag	Long Flag/ JSON Key	Description	JSON Content
	`l`	`log-level`	Set the log level for the application: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70 (requires compilation with `TRACE` log level support)	`"log-level": 60`
	N/A	`sdk-log-level`	SDK log events are currently unsupported for this tool	N/A
	`j`	`json`	Parse all command flags from an input JSON file	N/A
Program flags	`s`	`socket`	AF_UNIX (`SOCK_STREAM`) path. On the host, this is the path of the socket relay AF_UNIX server for the client's application to connect to. On the DPU, this is the path of the client AF_UNIX server application. ⚠ This flag is mandatory.	`"socket": "/tmp/uds-server.socket"`
	`n`	`cc-name`	Comm Channel service name ⚠ This flag is mandatory.	`"cc-name": sr_channel`
	`p`	`pci-addr`	DOCA Comm Channel device PCIe address ⚠ This flag is mandatory.	`"pci-addr": b1:00.1`
	`r`	`rep-pci`	DOCA Comm Channel device representor PCIe address ⚠ This flag is available and mandatory only on the DPU.	`"rep-pci": b1:02.2`

17 DOCA Services

This is an overview of the set of services provided by DOCA and their purpose.

17.1 Introduction

DOCA services are DOCA-based products, wrapped in a container for fast and easy deployment on top of the NVIDIA® BlueField® DPU. DOCA services leverage DPU capabilities to offer telemetry, time synchronization, networking solutions, and more.

Services containers can be found under the official NGC catalog, labeled under the "DOCA" and "DPU" NGC labels, as well as the built-in NVIDIA platform option ("DOCA") on the container catalog.

> ⚠ The following services are not available in the NGC catalog:
> - DOCA Management Service
> - NVIDIA OpenvSwitch Acceleration (OVS in DOCA)

For information on the deployment of the services, refer to the NVIDIA BlueField Container Deployment Guide.

17.2 Development Lifecycle

DOCA-based containers consist of two main categories:

- DOCA Base Images – containerized DOCA environments for both runtime and development. Used either by developers for their development environment or in the process of containerizing a DOCA-based solution.
- DOCA Services – containerized DOCA-based products

The process of developing and containerizing a DOCA-based product is described in the following sections.

17.2.1 Development

Before containerizing a product, users must first design and develop it using the same process for a bare-metal deployment on the BlueField DPU.

This process consists of the steps:

1. Identifying the requirements for the DOCA-based solution.
2. Reviewing the feature set offered by the DOCA SDK libraries, as shown in detail in their respective programming guides.
3. Starting the development process by following our Developer Guide to make the best use of our provided tips and tools.
4. Testing the developed solution.

Once the developed product is mature enough, it is time to start containerizing it.

17.2.2 Containerization

In this process, it is recommended to make use of DOCA's provided base-images, as available on DOCA's NGC page.

Three image flavors are provided:

- `base-rt` - includes the DOCA runtime, using the most basic runtime environment required by DOCA's SDK
- `full-rt` - builds on the previous image and includes the full list of runtime packages, which are all user-mode components that can be found under the doca-runtime package
- `devel` - builds on the previous image and adds headers and development tools for developing and debugging DOCA applications. This image is particularly useful for multi-stage builds.

All images are preconfigured to use to the DOCA repository of the matching DOCA version. This means that installing an additional DOCA package as part of a Dockerfile / within the development container can be done using the following commands:

```
apt update
apt install <package name>
```

For DOCA and CUDA environments, there are similar flavors for these images combined with CUDA's images:

- `base-rt` (DOCA) + `base` (CUDA)
- `full-rt` (DOCA) + `runtime` (CUDA)
- `devel` (DOCA) + `devel` (CUDA)

Once the containerized solution is mature enough, users may start profiling it in preparation for a production-grade deployment.

> ⚠ DOCA provides base images for both the DPU and the Host. For host-related DOCA base images, please refer to the image tag suffixed with "-host".

17.2.3 Profiling

As mentioned in the NVIDIA BlueField Container Deployment Guide, the current deployment model of containers on top of the DPU is based on kubelet-standalone. And more specifically, this Kubernetes-based deployment makes use of YAML files to describe the resources required by the pod such as:

- CPU
- RAM
- Huge pages

It is recommended to profile your product so as to estimate the resources it requires (under regular deployments, as well as under stress testing) so that the YAML would contain an accurate "resources" section. This allows an administrator to better understand what the requirements are for deploying your service, as well as allow the k8s infrastructure to ensure that the service is not misbehaving once deployed.

Once done, the containerized DOCA-based product is ready for the final testing rounds, after which it will be ready for deployment in production environments.

17.3 Services

17.3.1 Container Deployment

The NVIDIA BlueField Container Deployment Guide provides an overview and deployment configuration of DOCA containers for NVIDIA® BlueField® DPU.

17.3.2 DOCA BlueMan

DOCA BlueMan service runs in the DPU as a standalone web dashboard and consolidates all the basic information, health, and telemetry counters into a single interface. This friendly, easy-to-use web dashboard acts as a one-stop shop for all the information needed to monitor the DPU.

17.3.3 DOCA Firefly

DOCA Firefly service provides precision time protocol (PTP) based time syncing services to the BlueField DPU. PTP is used to synchronize clocks in a network which, when used in conjunction with hardware support, PTP is capable of sub-microsecond accuracy, which is far better than what is normally obtainable with network time protocol (NTP).

17.3.4 DOCA Flow Inspector

DOCA Flow Inspector service allows monitoring real-time data and extraction of telemetry components which can be utilized by various services for security, big data and more.

Specific mirrored packets can be transferred to Flow Inspector for parsing and analyzing. These packets are forwarded to DTS, which gathers predefined statistics determined by various telemetry providers.

17.3.5 DOCA HBN

DOCA Host-Based Networking service orchestrates network connectivity of dynamically created VMs/ containers on cloud servers. HBN service is a BGP router that supports EVPN extension to enable multi-tenant clouds.

At its core, HBN is the Linux networking acceleration driver of the DPU, Netlink-to-DOCA daemon which seamlessly accelerates Linux networking using DOCA hardware programming APIs.

17.3.6 DOCA Management Service

DOCA Management service (DMS) is a one-stop shop for the user to configure and operate NVIDIA BlueField Networking Platforms and NVIDIA ConnectX Adapters (NICs). DMS governs all scripts/tools of NVIDIA with an easy open API created by the OpenConfig community. The user can configure

BlueField or ConnectX for any mode whether locally (ssh) or remotely (grpc). It makes it easy to migrate and bootstrap any customer for any NVIDIA network device.

17.3.7 OpenvSwitch Acceleration (OVS in DOCA)

OVS-DOCA is a virtual switch service, designed to work with NVIDIA NICs and DPUs to utilize ASAP^2 (Accelerated Switching and Packet Processing) technology for data-path acceleration, providing the most efficient performance and feature set due to its architecture and use of DOCA libraries.

17.3.8 DOCA Telemetry

DOCA Telemetry service (DTS) collects data from built-in providers and from external telemetry applications. Collected data is stored in binary format locally on the DPU and can be propagated onwards using Prometheus endpoint pulling, pushing to Fluent Bit, or using other supported providers. Exporting NetFlow packets collected using the DOCA Telemetry NetFlow API is a great example of DTS usage.

17.3.9 DOCA UROM

DOCA UROM service provides a framework for offloading significant portions of HPC software stack directly from the host and to the BlueField networking platform.

ⓘ For questions, comments, and feedback, please contact us at DOCA-Feedback@exchange.nvidia.com.

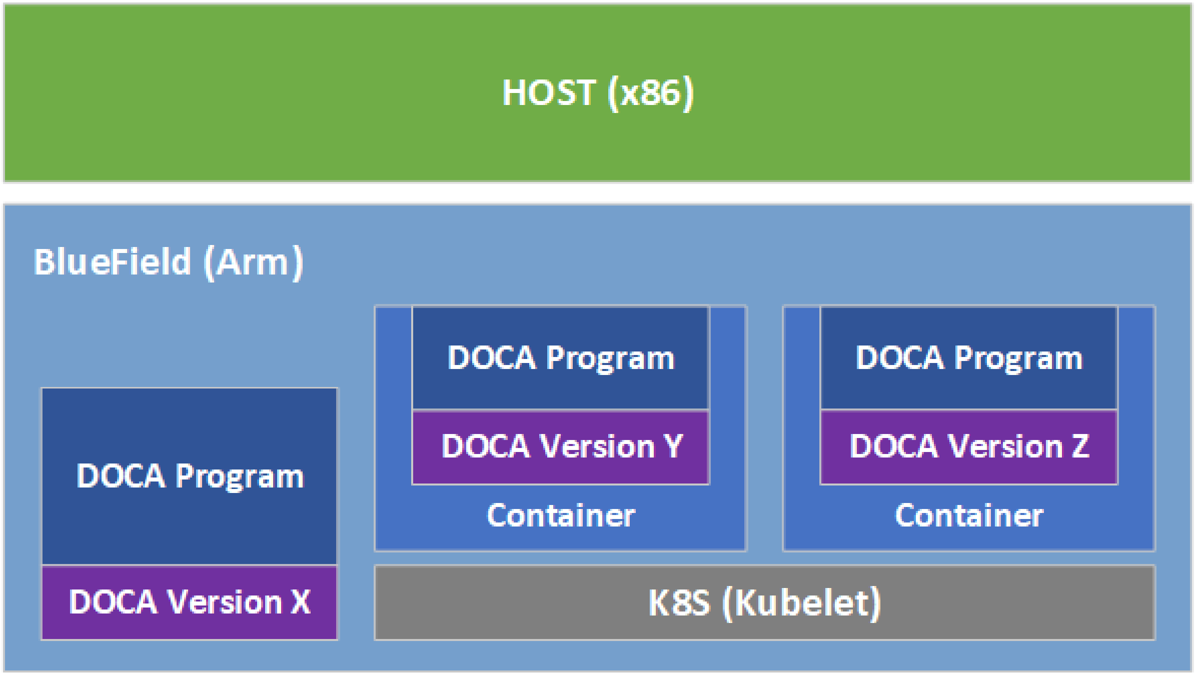
17.4 NVIDIA BlueField Container Deployment Guide

This guide provides an overview and deployment configuration of DOCA containers for NVIDIA® BlueField® DPU.

17.4.1 Introduction

DOCA containers allow for easy deployment of ready-made DOCA environments to the DPU, whether it is a DOCA service bundled inside a container and ready to be deployed, or a development environment already containing the desired DOCA version.

Containerized environments enable the users to decouple DOCA programs from the underlying BlueField software. Each container is pre-built with all needed libraries and configurations to match the specific DOCA version of the program at hand. One only needs to pick the desired version of the service and pull the ready-made container of that version from NVIDIA's container catalog.



The different DOCA containers are listed on NGC, NVIDIA's container catalog, and can be found under both the "DOCA" and "DPU" labels.

17.4.2 Prerequisites

- Refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField related software
- BlueField image version required is 3.9.0 and higher

> ⚠ Container deployment based on standalone Kubelet, as presented in this guide, is currently in alpha version and is subject to change in future releases.

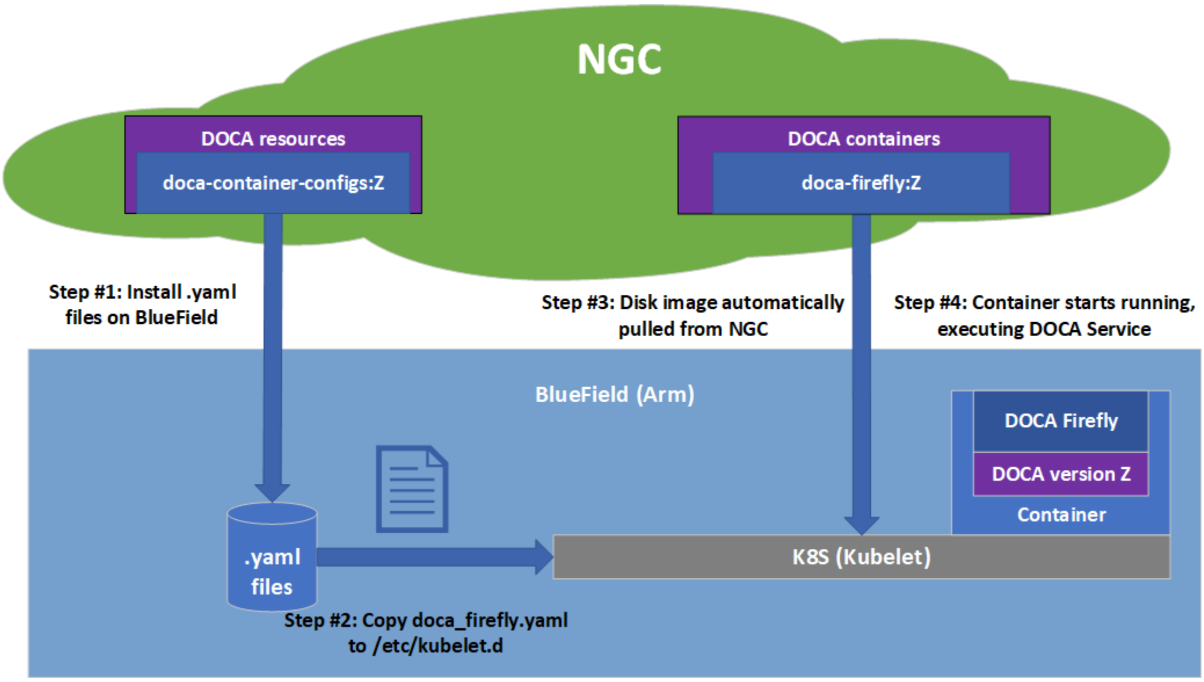
17.4.3 Container Deployment

Deploying containers on top of the BlueField DPU requires the following setup sequence:

1. Pull the container `.yaml` configuration files.
2. Modify the container's `.yaml` configuration file.
3. Deploy the container. The image is automatically pulled from NGC.

Some of the steps must only be performed once, while others are required before the deployment of each container.

What follows is an example of the overall setup sequence using the DOCA Firefly container as an example.


NGC
DOCA resources
doca-container-configs:Z
DOCA containers
doca-firefly:Z
Step #1: Install .yaml files on BlueField
Step #3: Disk image automatically pulled from NGC
Step #4: Container starts running, executing DOCA Service
BlueField (Arm)
DOCA Firefly
DOCA version Z
Container
.yaml files
K8S (Kubelet)
Step #2: Copy doca_firefly.yaml to /etc/kubelet.d

17.4.3.1 Pull Container YAML Configurations

> ⚠ This step pulls the `.yaml` configurations from NGC. If you have already performed this step for other DOCA containers you may skip to the next section.

To pull the latest resource version:

1. Pull the entire resource as a `*.zip` file:

```
wget https://api.ngc.nvidia.com/v2/resources/nvidia/doca/doca_container_configs/versions/2.8.0v1/zip -O
doca_container_configs_2.8.0v1.zip
```

2. Unzip the resource:

```
unzip -o doca_container_configs_2.8.0v1.zip -d doca_container_configs_2.8.0v1
```

More information about additional versions can be found in the NGC resource page.

17.4.3.2 Container-specific Instructions

Some containers require specific configuration steps for the resources used by the application running inside the container and modifications for the `.yaml` configuration file of the container itself.

Refer to the container-specific instructions listed under the container's relevant page on NGC.

17.4.3.3 Structure of NGC Resource

The DOCA NGC resource downloaded in section "Pull Container YAML Configurations" contains a `configs` directory under which a dedicated folder per DOCA version is located. For example, `2.0.2` will include all currently available `.yaml` configuration files for DOCA 2.0.2 containers.

```
doca_container_configs_2.0.2v1
    configs
        1.2.0
        ...
        2.0.2
            doca_application_recognition.yaml
            doca_blueman.yaml
            doca_devel.yaml
            doca_devel_cuda.yaml
            doca_firefly.yaml
            doca_flow_inspector.yaml
            doca_hbn.yaml
            doca_ips.yaml
            doca_snap.yaml
            doca_telemetry.yaml
            doca_url_filter.yaml
```

In addition, the resource also contains a `scripts` directory under which services may choose to provide additional helper-scripts and configuration files to use with their services.

The folder structure of the `scripts` directory is as follows:

```
+ doca_container_configs_2.0.2v1
+-+ configs
| +--  ...
+-+ scripts
  +-+ doca_firefly               <== Name of DOCA Service
  +-+ doca_hbn                   <== Name of DOCA Service
  | +-+ 1.3.0
  | | +-- ...                    <== Files for the DOCA HBN version "1.3.0"
  | +-+ 1.4.0
  | | +-- ...                    <== Files for the DOCA HBN version "1.4.0"
```

A user wishing to deploy an older version of the DOCA service would still have access to the suitable YAML file (per DOCA release under `configs`) and scripts (under the service-specific version folder which resides under `scripts`).

17.4.3.4 Spawn Container

Once the desired `.yaml` file is updated, simply copy the configuration file to Kubelet's input folder. Here is an example using the `doca_firefly.yaml`, corresponding to the DOCA Firefly service.

```
cp doca_firefly.yaml /etc/kubelet.d
```

Kubelet automatically pulls the container image from NGC and spawns a pod executing the container. In this example, the DOCA Firelfy service starts executing right away and its printouts would be seen via the container's logs.

17.4.3.5 Review Container Deployment

When deploying a new container, it is recommended to follow this procedure to ensure successful completion of each step in the deployment:

1. View currently active pods and their IDs:

```
sudo crictl pods
```

It may take up to 20 seconds for the pod to start.

When deploying a new container, search for a matching line in the command's output:

```
POD ID              CREATED             STATE               NAME
NAMESPACE           ATTEMPT             RUNTIME
06bd84c07537e       4 seconds ago       Ready               doca-firefly-my-dpu
default             0                   (default)
```

2. If a matching line fails to appear, it is recommended to view Kubelet's logs to get more information about the error:

   ```
   sudo journalctl -u kubelet --since -5m
   ```

   Once the issue is resolved, proceed to the next steps.

   For more troubleshooting information and tips, refer to the matching section in our Troubleshooting Guide.

3. Verify that the container image is successfully downloaded from NGC into the DPU's container registry (download time may vary based on the size of the container image):

   ```
   sudo crictl images
   ```

   Example output:

   ```
   IMAGE                              TAG                 IMAGE ID            SIZE
   k8s.gcr.io/pause                   3.9                 829e9de338bd5       268kB
   nvcr.io/nvidia/doca/doca_firefly   1.1.0-doca2.0.2     134cb22f34611       87.4MB
   ```

4. View currently active containers and their IDs:

   ```
   sudo crictl ps
   ```

   Once again, find a matching line for the deployed container (boot time may vary depending on the container's image size):

   ```
   CONTAINER           IMAGE               CREATED             STATE               NAME
   ATTEMPT             POD ID              POD
   b505a05b7dc23       134cb22f34611       4 minutes ago       Running             doca-firefly                 0
   06bd84c07537e       doca-firefly-my-dpu
   ```

5. In case of failure, to see a line matching the container, check the list of all recent container deployments:

   ```
   sudo crictl ps -a
   ```

   It is possible that the container encountered an error during boot and exited right away:

   ```
   CONTAINER           IMAGE               CREATED             STATE               NAME
   ATTEMPT             POD ID              POD
   de2361ec15b61       134cb22f34611       1 second ago        Exited              doca-firefly                 1
   4aea5f5adc91d       doca-firefly-my-dpu
   ```

6. During the container's lifetime, and for a short timespan after it exits, once can view the containers logs as were printed to the standard output:

```
sudo crictl logs <container-id>
```

In this case, the user can learn from the log that the wrong configuration was passed to the container:

```
$ sudo crictl logs de2361ec15b61
Starting DOCA Firefly - Version 1.1.0
...
Requested the following PTP interface: p10
Failed to find interface "p10". Aborting
```

For additional information and guides on using `crictl`, refer to the Kubernetes documentation.

17.4.3.6 Stop Container

The recommended way to stop a pod and its containers is as follows:

1. Delete the `.yaml` configuration file for Kubelet to stop the pod:

```
rm /etc/kubelet.d/<file name>.yaml
```

2. Stop the pod directly (only if it still shows "Ready"):

```
sudo crictl stopp <pod-id>
```

3. Once the pod stops, it may also be necessary to stop the container itself:

```
sudo crictl stop <container-id>
```

17.4.4 Troubleshooting Common Errors

This section provides a list of common errors that may be encountered when spawning a container. These account for the vast majority of deployment errors and are easy to verify first before trying to parse the Kubelet journal log.

If more troubleshooting is required, refer to the matching section in the Troubleshooting Guide.

17.4.4.1 Yaml Syntax

The syntax of the `.yaml` file is extremely sensitive and minor indentation changes may cause it to stop working. The file uses spaces (' ') for indentations (two per indent). Using any other number of spaces causes an undefined behavior.

17.4.4.2 Huge Pages

The container only spawns once all the required system resources are allocated on the DPU and can be reserved for the container. The most notable resource is huge pages.

1. Before deploying the container, make sure that:
    a. Huge pages are allocated as required per container.
    b. Both the amount and size of pages match the requirements precisely.
2. Once huge pages are allocated, it is recommended to restart the container service to apply the change:

    ```
    sudo systemctl restart kubelet.service
    sudo systemctl restart containerd.service
    ```

3. Once the above operations are completed successfully, the container could be deployed (YAML can be copied to `/etc/kubelet.d`).

17.4.5 Advanced Troubleshooting

17.4.5.1 Manual Execution from Within Container - Debugging

> ⚠ The deployment described in this section requires an in-depth knowledge of the container's structure. As this structure might change from version to version, it is only recommended to use this deployment for debugging, and only after other debugging steps have been attempted.

Although most containers define the `entrypoint.sh` script as the container's ENTRYPOINT, this option is only valid for interaction-less sessions. In some debugging scenarios, it is useful to have better control of the programs executed within the container via an interactive shell session. Hence, the `.yaml` file supports an additional execution option.

Uncommenting (i.e., removing `#` from) the following 2 lines in the `.yaml` file causes the container to boot without spawning the container's entrypoint script.

```
# command: ["sleep"]
# args: ["infinity"]
```

In this execution mode, users can attach a shell to the spawned container:

```
crictl exec -it <container-id> /bin/bash
```

Once attached, users get a full shell session enabling them to execute internal programs directly at the scope of the container.

17.4.6 Air-gapped Container Deployment

Container deployment on the BlueField DPU can be done in air-gapped networks and does not require an Internet connection. As explained previously, per DOCA service container, there are 2 required components for successful deployment:

- Container image – hosted on NVIDIA's NGC catalog
- YAML file for the container

From an infrastructure perspective, one additional module is required:

- `k8s.gcr.io/pause` container image

17.4.6.1 Pulling Container for Offline Deployment

When preparing an air-gapped environment, users must pull the required container images in advance so they could be imported locally to the target machine:

```
docker pull <container-image:tag>
docker save <container-image:tag> > <name>.tar
```

The following example pulls DOCA Firefly `1.1.0-doca2.0.2`:

```
docker pull nvcr.io/nvidia/doca/doca_firefly:1.1.0-doca2.0.2
docker save nvcr.io/nvidia/doca/doca_firefly:1.1.0-doca2.0.2 > firefly_v1.1.0.tar
```

> Some of DOCA's container images support multiple architectures, causing the `docker pull` command to pull the image according to the architecture of the machine on which it is invoked. Users may force the operation to pull an Arm image by passing the `--platform` flag:
>
> ```
> docker pull --platform=linux/arm64 <container-image:tag>
> ```

17.4.6.2 Importing Container Image

After exporting the image from the container catalog, users must place the created `*.tar` files on the target machine on which to deploy them. The import command is as follows:

```
ctr --namespace k8s.io image import <name>.tar
```

For example, to import the firefly `.tar` file pulled in the previous section:

```
ctr --namespace k8s.io image import firefly_v1.1.0.tar
```

Examining the status of the operation can be done using the image inspection command:

```
crictl images
```

17.4.6.3 Built-in Infrastructure Support

The DOCA image comes pre-shipped with the `k8s.gcr.io/pause` image:

```
/opt/mellanox/doca/services/infrastructure/
    docker_pause_3_9.tar
    enable_offline_containers.sh
```

This image is imported by default during boot as part of the automatic activation of DOCA Telemetry Service (DTS).

> ⚠ Importing the image independently of DTS can be done using the `enable_offline_container.sh` script located under the same directory as the image's `*.tar` file.

This image can also be pulled and imported manually, using the following instructions:

- To export the image:

  ```
  docker pull k8s.gcr.io/pause:3.9
  docker save k8s.gcr.io/pause:3.9 > docker_pause_3_9.tar
  ```

- To import the image:

  ```
  ctr --namespace k8s.io image import docker_pause_3_9.tar
  crictl images
  IMAGE                              TAG                 IMAGE ID            SIZE
  k8s.gcr.io/pause                   3.9                 829e9de338bd5       268kB
  ```

17.4.7 DOCA Services for Host

A subset of the DOCA services is available for host-based deployment as well. This is indicated in those services' deployment and can also be identified by having container tags on NGC with the `*-host` suffix.

In contrast to the managed DPU environment, the deployment of DOCA services on the host is based on docker. This deployment can be extended further based on the user's own container runtime solution.

17.4.7.1 Docker Deployment

DOCA services for the host are deployed directly using Docker.

1. Make sure Docker is installed on your host. Run:

   ```
   docker version
   ```

   If it is not installed, visit the official Install Docker Engine webpage for installation instructions.
2. Make sure the Docker service is started. Run:

```
sudo systemctl daemon-reload
sudo systemctl start docker
```

3. Pull the container image directly from NGC (can also be done using the `docker run` command):
    a. Visit the NGC page of the desired container.
    b. Under the "Tags" menu, select the desired tag and click the paste icon so it is copied to the clipboard.
    c. The docker pull command will be as follows:

    ```
    sudo docker pull <NGC container tag here>
    ```

    For example:

    ```
    sudo docker pull nvcr.io/nvidia/doca/doca_firefly:1.1.0-doca2.0.2-host
    ```

    ⚠ For DOCA services with deployments on both DPU and host, make sure to select the tag ending with `-host`.

4. Deploy the DOCA service using Docker:
    a. The deployment is performed using the following command:

    ```
    sudo docker run --privileged --net=host -v <host directory>:<container directory> -e <env
    variables> -it <container tag> /entrypoint.sh
    ```

    ⓘ For more information, refer to Docker's official documentation.

    b. The specific deployment command for each DOCA service is listed in their respective deployment guide.

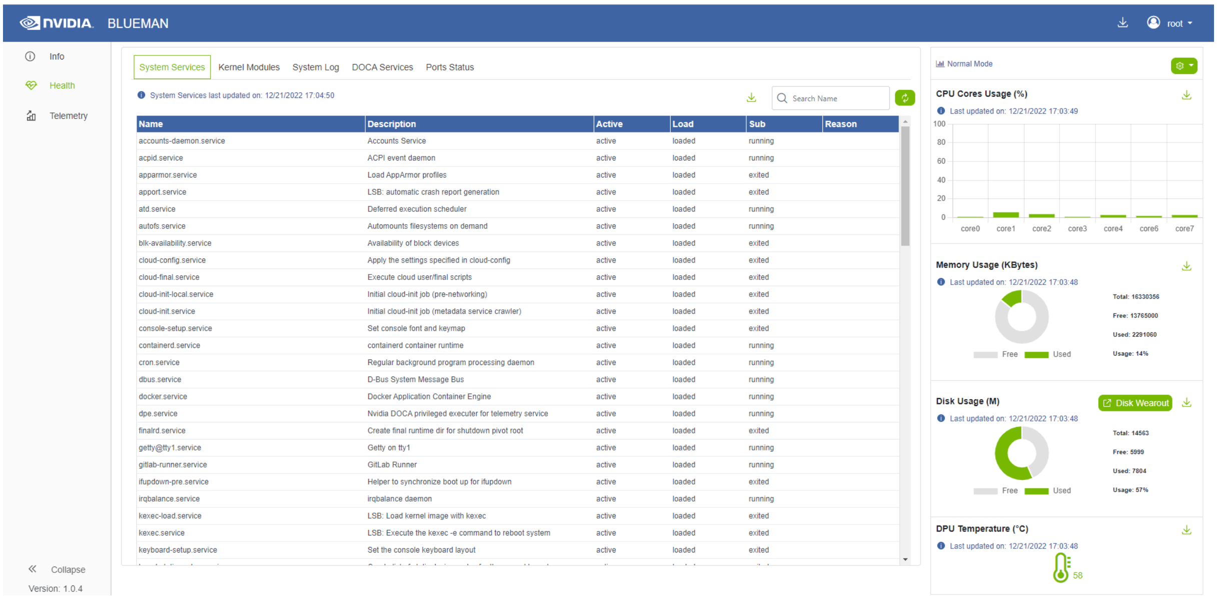
17.5 NVIDIA DOCA BlueMan Service Guide

This guide provides instructions on how to use the DOCA BlueMan service on top of NVIDIA® BlueField® DPU.

17.5.1 Introduction

DOCA BlueMan runs in the DPU as a standalone web dashboard and consolidates all the basic information, health, and telemetry counters into a single interface.

All the information that BlueMan provides is gathered from the DOCA Telemetry Service (DTS), starting from DTS version 1.11.1-doca1.5.1.



System Services last updated on: 12/21/2022 17:04:50

Name	Description	Active	Load	Sub	Reason
accounts-daemon.service	Accounts Service	active	loaded	running	
acpid.service	ACPI event daemon	active	loaded	running	
apparmor.service	Load AppArmor profiles	active	loaded	exited	
apport.service	LSB: automatic crash report generation	active	loaded	exited	
atd.service	Deferred execution scheduler	active	loaded	running	
autofs.service	Automounts filesystems on demand	active	loaded	running	
blk-availability.service	Availability of block devices	active	loaded	exited	
cloud-config.service	Apply the settings specified in cloud-config	active	loaded	exited	
cloud-final.service	Execute cloud user/final scripts	active	loaded	exited	
cloud-init-local.service	Initial cloud-init job (pre-networking)	active	loaded	exited	
cloud-init.service	Initial cloud-init job (metadata service crawler)	active	loaded	exited	
console-setup.service	Set console font and keymap	active	loaded	exited	
containerd.service	containerd container runtime	active	loaded	running	
cron.service	Regular background program processing daemon	active	loaded	running	
dbus.service	D-Bus System Message Bus	active	loaded	running	
docker.service	Docker Application Container Engine	active	loaded	running	
dpe.service	Nvidia DOCA privileged executer for telemetry service	active	loaded	running	
finalrd.service	Create final runtime dir for shutdown pivot root	active	loaded	exited	
getty@tty1.service	Getty on tty1	active	loaded	running	
gitlab-runner.service	GitLab Runner	active	loaded	running	
ifupdown-pre.service	Helper to synchronize boot up for ifupdown	active	loaded	exited	
irqbalance.service	irqbalance daemon	active	loaded	running	
kexec-load.service	LSB: Load kernel image with kexec	active	loaded	exited	
kexec.service	LSB: Execute the kexec -e command to reboot system	active	loaded	exited	
keyboard-setup.service	Set the console keyboard layout	active	loaded	exited	

CPU Cores Usage (%)

Last updated on: 12/21/2022 17:03:49

Memory Usage (KBytes)

Last updated on: 12/21/2022 17:03:48

Total: 16330356

Free: 13765000

Used: 2291060

Usage: 14%

Disk Usage (M)

Last updated on: 12/21/2022 17:03:48

Total: 14563

Free: 5999

Used: 7804

Usage: 57%

DPU Temperature (°C)

Last updated on: 12/21/2022 17:03:48

17.5.2 Requirements

- BlueField image version 3.9.3.1 or higher
- DTS and the DOCA Privileged Executer (DPE) daemon must be up and running

17.5.2.1 Verifying DTS Status

All the information that BlueMan provides is gathered from DTS.

Verify that the state of the DTS pod is `ready`:

```
$ crictl pods --name doca-telemetry-service
```

Verify that the state of the DTS container is `running`:

```
$ crictl ps --name doca-telemetry-service
```

17.5.2.2 Verifying DPE Status

All the information that DTS gathers for BlueMan is from the the DPE daemon.

Verify that the DPE daemon is `active`:

```
$ systemctl is-active dpe.service
active
```

If the daemon is inactive, activate it by starting the `dpe.service`:

```
$ systemctl start dpe.service
```

17.5.3 Service Deployment

For information about the deployment of DOCA containers on top of the BlueField DPU, refer to the NVIDIA DOCA Container Deployment Guide.

17.5.3.1 DOCA Service on NGC

BlueMan is available on NGC, NVIDIA's container catalog. Service-specific configuration steps and deployment instructions can be found under the service's container page.

17.5.3.2 Default Deployment – BlueField BSP

BlueMan service is located under `/opt/mellanox/doca/services/blueman`/.

The following is a list of the files under the BlueMan directory:

```
doca_blueman_fe_service_<version>-doca<version>_arm64.tar
doca_blueman_conv_service_<version>-doca<version>_arm64.tar
doca_blueman_standalone.yaml
bring_up_doca_blueman_service.sh
```

17.5.3.2.1 Enabling BlueMan Service

17.5.3.2.1.1 Using Script

Run `bring_up_doca_blueman_service.sh`:

```
$ chmod +x /opt/mellanox/doca/services/blueman/bring_up_doca_blueman_service.sh
$ /opt/mellanox/doca/services/blueman/bring_up_doca_blueman_service.sh
```

17.5.3.2.1.2 Manual Procedure

1. Import images to crictl images:

   ```
   $ cd /opt/mellanox/doca/services/blueman/
   $ ctr --namespace k8s.io image import doca_blueman_fe_service_<version>-doca<version>_arm64.tar
   $ ctr --namespace k8s.io image import doca_blueman_conv_service_<version>-doca<version>_arm64.tar
   ```

2. Verify that the DPE daemon is active:

   ```
   $ systemctl is-active dpe.service
   active
   ```

   If the daemon is inactive, activate it by starting the `dpe.service`:

   ```
   $ systemctl start dpe.service
   ```

3. Copy `blueman_standalone.yaml` to `/etc/kubelet.d/`:

   ```
   $ cp doca_blueman_standalone.yaml /etc/kubelet.d/
   ```

17.5.3.3 Verifying Deployment Success

1. Verify that the DPE daemon is active:

```
$ systemctl is-active dpe.service
```

2. Verify that the state of the DTS container is `running`:

```
$ crictl ps --name doca-telemetry-service
```

3. Verify that the state of the BlueMan service container is `running`:

```
$ crictl ps --name doca-blueman-fe
$ crictl ps --name doca-blueman-conv
```

Configuration

The configuration of the BlueMan back end is located under `/opt/mellanox/doca/services/telemetry/config/blueman_config.ini`. Users can interact with the `blueman_config.ini` file which contains the default range values of the Pass, Warning, and Failed categories which are used in the health page. Changing these values gets reflected in the BlueMan webpage within 60 seconds.

Example of `blueman_config.ini`:

```
;Health Cpu usages Pass, warning, Failed
[Health:CPU_Usages:Pass]
range = 0,80
[Health:CPU_Usages:Warning]
range = 80,90
[Health:CPU_Usages:Failed]
range = 90,100
```

17.5.4 Collected Data

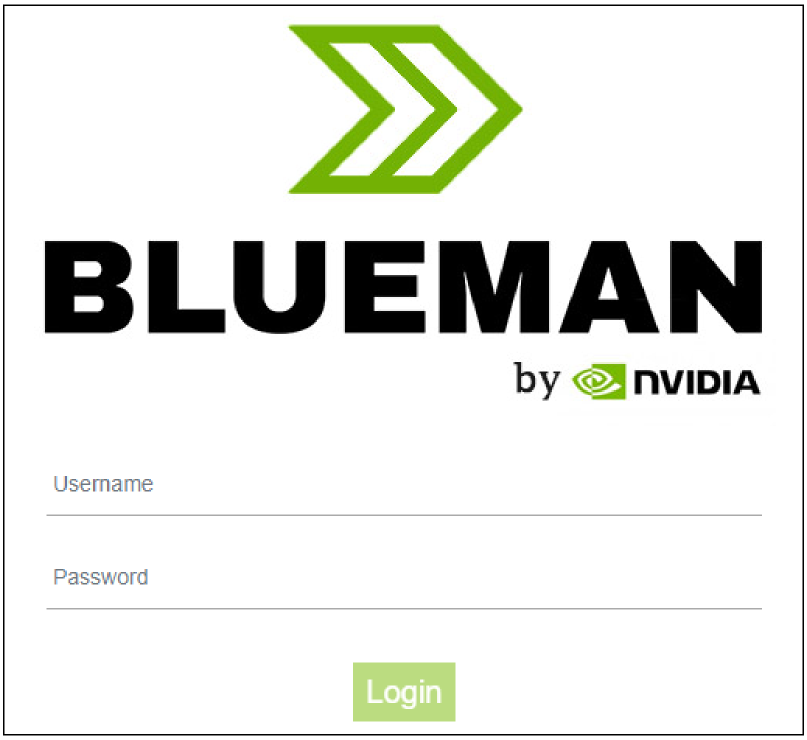
- Info
  - General info – OS name, kernel, part number, serial number, DOCA version, driver, board ID, etc.
  - Installed packages – list of all installed packages on the DPU including their version
  - CPU info – vendor, cores, model, etc.
  - FW info – all the mlxconfig parameters with default/current/next boot data
  - DPU operation mode
- Health
  - System service
  - Kernel modules
  - Dmesg
  - DOCA services
  - Port status of the PF and OOB
  - Core usage and processes running on each core
  - Memory usage
  - Disk usage
  - Temperature

- Telemetry – all telemetry counters that come from DTS according to the enabled providers displayed on tables
  - Users have the ability to build graphs of specific counters

17.5.5 Connecting to BlueMan Web Interface

To log into BlueMan, enter the IP address of the DPU's OOB interface (`http://<DPU_OOB_IP>`) to a web browser located in the same network as the DPU.

The login credentials to use are the same pair used for the SSH connection to the DPU.



17.5.6 Troubleshooting

For general troubleshooting, refer to the NVIDIA DOCA Troubleshooting Guide.

For container-related troubleshooting, refer to the "Troubleshooting" section in the NVIDIA DOCA Container Deployment Guide.

The following are additional troubleshooting tips for DOCA BlueMan:

- The following error message in the login page signifies a failure to connect to the DPE daemon: "The service is currently unavailable. Please check server up and running."
  a. Restart the DPE daemon:

  ```
  $ systemctl restart dpe.service
  ```

  b. Verify that DTS is up and running by following the instructions in section "Verifying DTS Status".
- If the message "Invalid Credentials" appears in the login page, verify that the username and password are the same ones used to SSH to the DPU.

- If all of the above is configured as expected and there is still some failure to log in, it is recommended to check if there are any firewall rules that block the connection.
- For other issues, check the `/var/log/syslog` and `/var/log/doca/telemetry/blueman_service.log` log file.

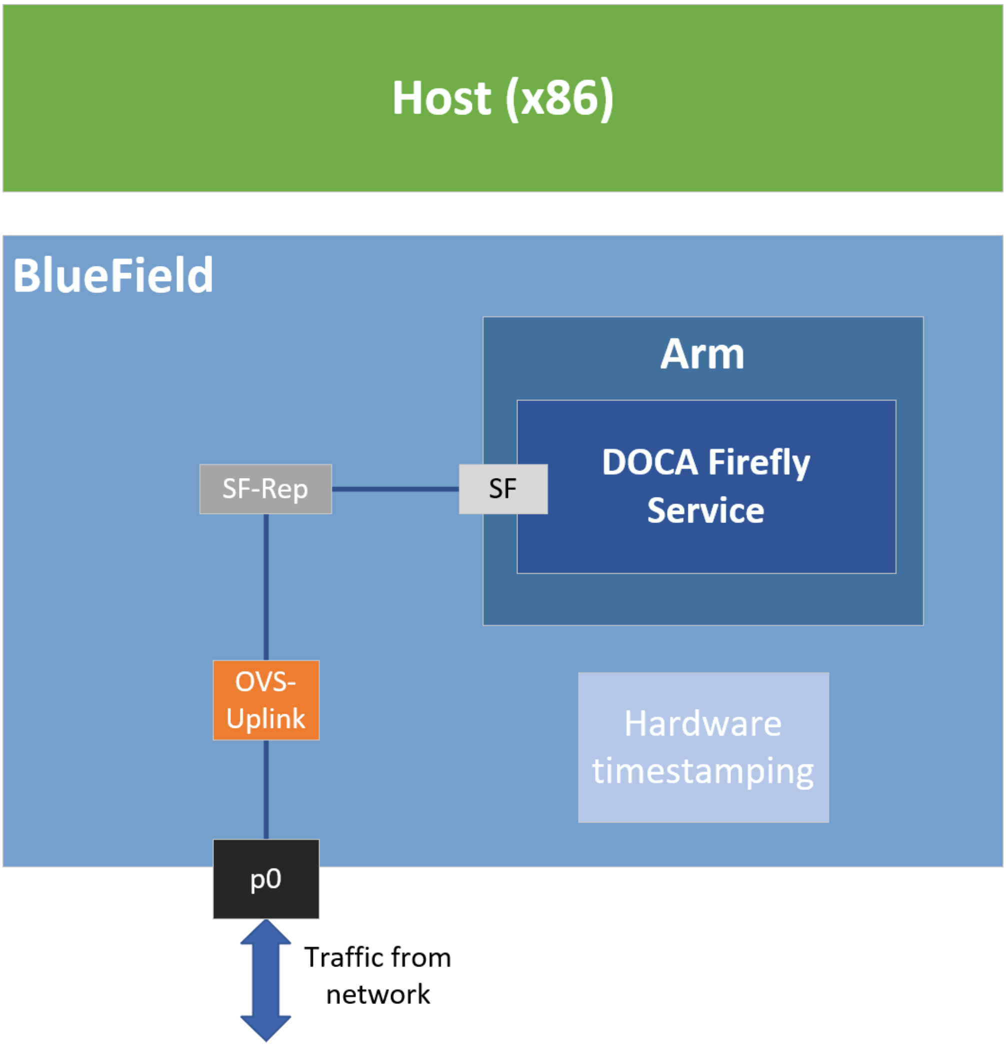
17.6 NVIDIA DOCA Firefly Service Guide

This guide provides instructions on how to use the DOCA Firefly service container on top of NVIDIA® BlueField® DPU.

17.6.1 Introduction

DOCA Firefly Service provides precision time protocol (PTP) based time syncing services to the BlueField DPU.

PTP is a protocol used to synchronize clocks in a network. When used in conjunction with hardware support, PTP is capable of sub-microsecond accuracy, which is far better than is what is normally obtainable with network time protocol (NTP). PTP support is divided between the kernel and user space. The ptp4l program implements the PTP boundary clock and ordinary clock. With hardware time stamping, it is used to synchronize the PTP hardware clock to the master clock.



17.6.2 Requirements

Some of the features provided by Firefly require specific BlueField DPU hardware capabilities:

- PTP – Supported by all BlueField DPUs
- PPS – Requires BlueField DPU with PPS capabilities
- SyncE - Requires converged card BlueField DPUs

Failure to run PPS due to missing hardware support will be noted in the service's output. However, the service will continue to run the timing services it can provide on the provided hardware.

17.6.2.1 Firmware Version

Firmware version must be 24.34.1002 or higher.

17.6.2.2 BlueField BSP Version

Supported BlueField image versions are 3.9.0 and higher.

17.6.2.3 Embedded Mode

17.6.2.3.1 Configuring Firmware Settings on DPU for Embedded Mode

1. Set the DPU to embedded mode (default mode):

   ```
   sudo mlxconfig -y -d 03:00.0 s INTERNAL_CPU_MODEL=1
   ```

2. Enable the real time clock (RTC):

   ```
   sudo mlxconfig -d 03:00.0 set REAL_TIME_CLOCK_ENABLE=1
   ```

3. Graceful shutdown and power cycle the DPU to apply the configuration.
4. You may check the DPU mode using the following command:

   ```
   sudo mlxconfig -d 03:00.0 q | grep INTERNAL_CPU_MODEL
   # Example output
            INTERNAL_CPU_MODEL                    EMBEDDED_CPU(1)
   ```

17.6.2.3.2 Ensuring OVS Hardware Offload

DOCA Firefly requires that hardware offload is activated in Open vSwitch (OVS). This is enabled by default as part of the BFB image installed on the DPU.

To verify the hardware offload configuration in OVS:

```
sudo ovs-vsctl get Open_vSwitch . other_config | grep hw-offload
# Example output
         {hw-offload="true"}
```

If inactive:

1. Activate hardware offloading by running:

   ```
   sudo ovs-vsctl set Open_vSwitch . other_config:hw-offload=true;
   ```

2. Restart the OVS service:

   ```
   sudo /etc/init.d/openvswitch-switch restart
   ```

3. Graceful shutdown and power cycle the DPU to apply the configuration.

17.6.2.3.3 Helper Scripts

Firefly's deployment contains a script to help with the configuration steps required for the network interface in embedded mode:

- `scripts/doca_firefly/<firefly-version>/prepare_for_embedded_mode.sh`
- `scripts/doca_firefly/<firefly-version>/set_new_sf.sh`

The latest DOCA Firefly version is `1.4.0`.

Both scripts are included as part of DOCA's container resource which can be downloaded according to the instructions in the NVIDIA DOCA Container Deployment Guide. For more information about the structure of the DOCA container resource, refer to section "Structure of NGC Resource" in the deployment guide.

> ⚠ Due to technical limitations of the NGC resource, both scripts are provided without execute (+x) permissions. This could be resolved by running the following command:
>
> ```
> chmod +x scripts/doca_firefly/<firefly-version>/*.sh
> ```

17.6.2.3.3.1 prepare_for_embedded_mode.sh

This script automates all the steps mentioned in section "Setting Up Network Interfaces for Embedded Mode" and configures a freshly installed BFB image to the settings required by DOCA Firefly.

Notes:

- The script deletes all previous OVS settings and creates a single OVS bridge that matches the definitions in section "Setting Up Network Interfaces for Embedded Mode"
- The script should only be run once when connecting to the DPU for the first time or after a power cycle
- The only manual step required after using this script is configuring the IP address for the created network interface (step 5 in section "Setting Up Network Interfaces for Embedded Mode")
- The script automatically uses port 0 (`p0`). Configurations for port 1 should be done manually based on the commands listed in sections "set_new_sf.sh" and "Setting Up Network Interfaces for DPU Mode".

Script arguments:

- SF number (checks if already exists)

Examples:

- Prepare OVS settings using an SF indexed 4:

```
chmod +x ./*.sh
./prepare_for_embedded_mode.sh 4
```

The script makes use of `set_new_sf.sh` as a helper script.

17.6.2.3.3.2 set_new_sf.sh

Creates a new trusted SF and marks it as "trusted".

Script arguments:

- PCIe address
- SF number (checks if already exists)
- MAC address (if absent, a random address is generated)

Examples:

- Create SF with number "4" over port 0 of the DPU:

```
./set_new_sf.sh 0000:03:00.0 4
```

- Create SF with number "5" over port 0 of the DPU and a specific MAC address:

```
./set_new_sf.sh 0000:03:00.0 5 aa:bb:cc:dd:ee:ff
```

- Create SF with number "4" over port 1 of the DPU:

```
./set_new_sf.sh 0000:03:00.1 4
```

The first two examples should work out of the box for a BlueField-2 device and create SF4 and SF5 respectively.

17.6.2.3.4 Setting Up Network Interfaces for DPU Mode

1. Create a trusted SF to be used by the service according to the Scalable Function Setup Guide.

   > ⚠ The following instructions assume that the SF has been created using index 4.

2. Create the required OVS setting as is shown in the architecture diagram:

```
$ sudo ovs-vsctl add-br uplink
$ sudo ovs-vsctl add-port uplink p0
$ sudo ovs-vsctl add-port uplink en3f0pf0sf4
# This port is needed to ensure we have traffic host<->network as well
$ sudo ovs-vsctl add-port uplink pf0hpf
```

3. Verify the OVS settings:

```
sudo ovs-vsctl show
    Bridge uplink
        Port pf0hpf
            Interface pf0hpf
        Port en3f0pf0sf4
            Interface en3f0pf0sf4
        Port p0
            Interface p0
        Port uplink
            Interface uplink
                type: internal
```

4. Enable TX timestamping on the SF interface (not the representor):

```
# tx port timestamp offloading
sudo ethtool --set-priv-flags enp3s0f0s4 tx_port_ts on
```

5. Enable the interface and set an IP address for it:

```
# configure ip for the interface:
sudo ifconfig enp3s0f0s4 <ip-addr> up
```

6. Configure OVS to support TX timestamping over this SF and multicast traffic in general:

```
# Multicast-related definitions
```

```
$ sudo ovs-vsctl set Bridge uplink mcast_snooping_enable=true
$ sudo ovs-vsctl set Bridge uplink other_config:mcast-snooping-disable-flood-unregistered=true
$ sudo ovs-vsctl set Port p0 other_config:mcast-snooping-flood=true
$ sudo ovs-vsctl set Port p0 other_config:mcast-snooping-flood-reports=true
# PTP-related definitions
$ sudo ovs-ofctl add-flow uplink in_port=en3f0pf0sf4,udp,tp_src=319,actions=output:p0
$ sudo ovs-ofctl add-flow uplink in_port=p0,udp,tp_src=319,actions=output:en3f0pf0sf4
$ sudo ovs-ofctl add-flow uplink in_port=en3f0pf0sf4,udp,tp_src=320,actions=output:p0
$ sudo ovs-ofctl add-flow uplink in_port=p0,udp,tp_src=320,actions=output:en3f0pf0sf4
```

> If your OVS bridge uses a name other than `uplink`, make sure that the used name is reflected in the `ovs-vsctl` and `ovs-ofctl` commands. For instance:
>
> ```
> $ sudo ovs-vsctl set Bridge <bridge-name> mcast_snooping_enable=true
> ```

17.6.2.4 Separated Mode

17.6.2.4.1 Configuring Firmware Settings on DPU for Separated Mode

1. Set the BlueField mode of operation to "Separated":

   ```
   sudo mlxconfig -y -d 03:00.0 s INTERNAL_CPU_MODEL=0
   ```

2. Enable RTC:

   ```
   sudo mlxconfig -d 03:00.0 set REAL_TIME_CLOCK_ENABLE=1
   ```

3. Graceful shutdown and power cycle the DPU to apply the configuration.
4. You may check the BlueField's operation mode using the following command:

   ```
   sudo mlxconfig -d 03:00.0 q | grep INTERNAL_CPU_MODEL
   # Example output
            INTERNAL_CPU_MODEL                  SEPARATED_HOST(0)
   ```

17.6.2.4.2 Setting Up Network Interfaces for Separated Mode

1. Make sure that `p0` is not connected to an OVS bridge:

   ```
   sudo ovs-vsctl show
   ```

2. Enable TX timestamping on the `p0` interface:

   ```
   # TX port timestamp offloading (assuming PTP interface is p0)
   sudo ethtool --set-priv-flags p0 tx_port_ts on
   ```

3. Enable the interface and set an IP address for it:

   ```
   # Configure IP for the interface
   sudo ifconfig p0 <ip-addr> up
   ```

17.6.2.5 Host-based Deployment

Host-based deployment requires the same configuration described under section "Separated Mode".

17.6.3 Service Deployment

17.6.3.1 DPU Deployment

For information about the deployment of DOCA containers on top of the BlueField DPU, refer to NVIDIA DOCA Container Deployment Guide.

Service-specific configuration steps and deployment instructions can be found under the service's container page.

> ⚠ DOCA Firefly can also be deployed on DPUs not connected to the Internet. For instructions, refer to the relevant section in the NVIDIA DOCA Container Deployment Guide.

17.6.3.2 Host Deployment

DOCA Firefly has a version adapted for host-based deployments. For more information about the deployment of DOCA containers on top of a host, refer to the NVIDIA BlueField DPU Container Deployment Guide.

The following is the docker command for deploying DOCA Firefly on the host:

```
sudo docker run --privileged --net=host -v /var/log/doca/firefly:/var/log/firefly -v /etc/firefly:/etc/firefly -e
PTP_INTERFACE='eth2' -it nvcr.io/nvidia/doca/doca_firefly:1.4.0-doca2.7.0-host /entrypoint.sh
```

Where:

- Additional YAML configs may be passed as environment variables as additional `-e` key-value pairs as done with `PTP_INTERFACE` above
- The exact container tag should be the desired tag as chosen on DOCA Firefly's NGC page

17.6.4 Configuration

All modules within the service have configuration files that allow customizing various settings, both general and PTP-related.

17.6.4.1 Built-In Config File

Each profile has its own base PTP configuration file for `ptp4l`. For example, the Media profile PTP configuration file is `ptp4l-media.conf`.

The built-in PTP configuration files can be found in section "PTP Profile Default Config Files". For ease-of-use, those files are provided as part of DOCA's container resource as downloaded from NGC and are placed under Firefly's `configs` directory (`scripts/doca_firefly/<firefly version>/configs`).

> ⚠ When using a built-in configuration file, Firefly uses the files as stored within the container itself in the `/etc/linuxptp` directory. The configuration files included in the NGC resource are only provided for ease of access. Modifying them does not impact the configuration used in practice by the container. Instead, updates to the configuration should be done as described in the following sections.

17.6.4.2 Custom Config File

Instead of using a profile's base config file, users can create a file of their own, for each of the modules.

To set a custom config file, users should locate their config file in the directory `/etc/firefly` and set the config file name in DOCA Firefly's YAML file.

For example, to set a custom `linuxptp` config file, the user can set the parameter `PTP_CONFIG_FILE` in the YAML file:

```
- name: PTP_CONFIG_FILE
  value: my_custom_ptp.conf
```

In this example, `my_custom_ptp.conf` should be placed at `/etc/firefly/my_custom_ptp.conf`.

> ⚠ A config file must not define values for the UDS-related ports (`/var/run/ptp4l` and `/var/run/ptp4lro`), as those will impact internal container behavior. Such settings will prompt a warning and will be ignored when preparing the finalized configuration (See more in the next sections).

17.6.4.3 Overriding Specific Config File Parameters

Instead of replacing the entire config file, users may opt to override specific parameters. This can be done using the following variable syntax in the YAML file: `CONF_<TYPE>_<SECTION>_<PARAMETER_NAME>`.

- `TYPE` - either `PTP`, `MONITOR`, `PHC2SYS`, SYNCE, or `SERVO`
- `SECTION` - the section in the config file that the parameter should be placed in

  > ⚠ If the specified section does not already exist in the config file, a new section is created unless it refers to a PTP network interface that has not been included in the `PTP_INTERFACE` YAML field.

- `PARAMETER_NAME` - the config parameter name as should be placed in the config file

> ⚠ If the parameter name already exists in the config file, then the value is changed according to the value provided in the `.yaml` file. If the parameter name does not already exist in the config file, then it is added.

For example, the following variable in the YAML file definition changes the value of the parameter `priority1` under section `global` in the PTP config file to `64`.

```
- name: CONF_PTP_global_priority1
  value: "64"
```

> ⚠ Configuring `unicast_master_table` through the YAML file is not supported due to the structure of the table (i.e., multiple entries sharing the same key).

17.6.4.4 Ensuring and Debugging Correctness of Config Files

The previous sections describe 2 layers for the configuration file definitions:

- Basic configuration file - either a built-in config file or a custom config file
- Adding/overriding values to/from the YAML file

In practice, there are slightly more layers in place, and the precedence is as follows (presented in increasing order):

- Default configuration values of the PTP program (ptp4l for instance) – holds values of all available configuration options
- Your chosen configuration file - contains a subset of options
- Definitions from the YAML file – narrower subset
- Firefly mandatory values

When combining the supplied configuration file with the definitions from the YAML file, Firefly goes over those definitions and checks them against a predefined set of configuration options:

- Warning only - warns if a certain value leads to known issues in a supported deployment scenario
- Override - container-internal definitions that should not be set by the user and will be overridden by Firefly

Suitable log messages are provided in either case:

```
# Example for a warning
2023-01-31 11:55:13 - Firefly - Config - INFO    - Missing explicit definition "fault_reset_interval", verifying
default value instead: "4"
2023-01-31 11:55:13 - Firefly - Config - WARNING - Value "4" for definition "fault_reset_interval" will be invalid
in Embedded Mode, expected a value lesser or equal to "1"
2023-01-31 11:55:13 - Firefly - Config - WARNING - Continuing with invalid value
# Example for an override
2023-01-31 11:21:00 - Firefly - Config - WARNING - Invalid value "/var/run/ptp4l2" for definition "uds_address",
expected "/var/run/ptp4l"
2023-01-31 11:21:00 - Firefly - Config - INFO    - Setting definition "uds_address" value to the following: "/var/
run/ptp4l"
```

At the end of this process, an updated configuration file is generated by Firefly to be used later by the various time providers. To avoid accidental modification of a user-supplied configuration file or permission issues, the finalized file is generated within the container under the `/tmp` directory.

For instance, if using a custom configuration file named `my_custom_ptp.conf` under the `/etc/firefly` directory on the DPU, the updated file will reside within the container at the following path: `/tmp/my_custom_ptp.conf`.

For troubleshooting possible issues with the configuration file, one can do one of the following:

- Connect to the container directly as is explained in the debugging finalized configuration file bullet under "Troubleshooting".
- Map the container's `/tmp` directory to the DPU using the built-in support in the YAML file:
  - Before the change:

```
  # Uncomment when debugging the finalized configuration files used - Part #1
  #- name: debug-firefly-volume
  #  hostPath:
  #    path: /tmp/firefly
  #    type: DirectoryOrCreate
containers:
  ...
    volumeMounts:
    - name: logs-firefly-volume
      mountPath: /var/log/firefly
    - name: conf-firefly-volume
      mountPath: /etc/firefly
    # Uncomment when debugging the finalized configuration files used - Part #2
    #- name: debug-firefly-volume
    #  mountPath: /tmp
```

  - After the change:

```
  # Uncomment when debugging the finalized configuration files used - Part #1
  - name: debug-firefly-volume
    hostPath:
      path: /tmp/firefly
      type: DirectoryOrCreate
containers:
  ...
    volumeMounts:
    - name: logs-firefly-volume
      mountPath: /var/log/firefly
    - name: conf-firefly-volume
      mountPath: /etc/firefly
    # Uncomment when debugging the finalized configuration files used - Part #2
    - name: debug-firefly-volume
      mountPath: /tmp
```

⚠ The finalized configuration file keeps the sections and config options in the same order as they appear in the original file, yet the file is stripped from spare new lines or comment lines. This should be taken into considerations when directly accessing it during a debugging session.

17.6.5 Description

17.6.5.1 Providers

DOCA Firefly Service uses the following third-party providers to provide time syncing services:

- Linuxptp - Version v4.2
  - `PTP` - PTP service, provided by the PTP4L program
  - `PHC2SYS` - OS time calibration, provided by the PHC2SYS program

- Testptp
    - `PPS` - PPS settings service

In addition, DOCA Firefly Service also makes use of the following NVIDIA modules:

- SyncE
    - `SYNCE` - Synchronous Ethernet Deamon (`synced`)
- Firefly
    - `MONITOR` - Firefly PTP Monitor
- Firefly
    - `SERVO` - Firefly PTP Servo

Each of the providers can be enabled, disabled, or set to use the setting defined by the configuration profile:

- YAML setting - `<provider name>_STATE`
- Supported values - `enable` , `disable` , `defined_by_profile`

> ⚠ For the default profile settings per provider, refer to the table under section "Profiles".

An example YAML setting for specifically disabling the `phc2sys` provider is the following:

```
- name: PHC2SYS_STATE
  value: "disable"
```

> ⚠ The `defined_by_profile` setting is only available for well-defined profiles. As such, it cannot be used when the `custom` profile is selected. For more information about the profile settings, refer to the table under section "Profiles".

17.6.5.2 Profiles

DOCA Firefly Service includes profiles which represent common use cases for the Firefly service that provide a different default configuration per profile:

	Default	Media	Telco (L2)	Custom
Purpose	Any user that requires PTP	Media productions	Telco networks	Custom configuration for a dedicated user scenario
PTP	Enabled	Enabled	Enabled	No default. Enable/disable should be set by the user.
PTP profile	PTP default profile	SMPTE 2059-2	G.8275.1	Set by the user
PTP Client/Server [1]	Both	Client-only	Both	Set by the user
PHC2SYS	Enabled	Enabled	Enabled	No default. Enable/disable should be set by the user.

	Default	Media	Telco (L2)	Custom
PPS (in/out)	Enabled	Enabled	Enabled	No default. Enable/ disable should be set by the user.
PTP Monitor	Disabled	Disabled	Disabled	No default. Enable/ disable should be set by the user.
SyncE	Disabled	Disabled	Enabled	No default. Enable/ disable should be set by the user.
Servo	Disabled	Disabled	Disabled	No default. Enable/ disable should be set by the user.

1. Client-only is only relevant to a single PTP interface. If more than one PTP interface is provided in the YAML file, both modes are enabled. ↩

17.6.5.3 Outputs

17.6.5.3.1 Container Output

While running, the full output of the DOCA Firefly Service container can be viewed using the following command:

```
sudo crictl logs <CONTAINER-ID>
```

Where `CONTANIER-ID` can be retrieved using the following command:

```
sudo crictl ps
```

For example, in the following output, the container ID is `8f368b98d025b`.

```
$ sudo crictl ps
CONTAINER           IMAGE               CREATED             STATE               NAME                ATTEMPT
    POD ID              POD
8f368b98d025b       289809f312b4c       2 seconds ago       Running             doca-firefly        0
    5af59511b4be4       doca-firefly-some-computer-name
```

The output of the container depends on the services supported by the hardware and enabled by configuration and the selected profile. However, note that any of the configurations runs PTP, so when DOCA FireFly is running successfully expect to see the line "`Running ptp4l`".

The following is an example of the expected container output when running the default profile on a DPU that supports PPS:

```
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Starting DOCA Firefly - Version 1.4.0
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Selected features:
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] PTP     - Enabled - ptp4l will be used
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] MONITOR - Enabled - PTP Monitor will be used
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] PHC2SYS - Enabled - phc2sys will be used
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [-] SyncE   - Disabled
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [-] SERVO   - Disabled
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] PPS     - Enabled - testptp will be used (if supported by
hardware)
```

```
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Going to analyze the configuration files
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Requested the following PTP interface: p0
2023-09-07 14:04:23 - Firefly
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Starting PPS configuration
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] PPS is supported by hardware
2023-09-07 14:04:23 - Firefly - Init    - INFO     - set pin function okay
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] PPS in - Activated
2023-09-07 14:04:23 - Firefly - Init    - INFO     - set pin function okay
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] PPS out - Activated
2023-09-07 14:04:23 - Firefly - Init    - INFO     - name mlx5_pps0 index 0 func 1 chan 0
2023-09-07 14:04:23 - Firefly - Init    - INFO     - name mlx5_pps1 index 1 func 2 chan 0
2023-09-07 14:04:23 - Firefly - Init    - INFO     - periodic output request okay
2023-09-07 14:04:23 - Firefly
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Running ptp4l
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Running Firefly PTP Monitor
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Running phc2sys
```

The following is an example of the expected container output when running the default profile on a DPU that does not support PPS:

```
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Starting DOCA Firefly - Version 1.3.0
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Selected features:
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] PTP     - Enabled - ptp4l will be used
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] MONITOR - Enabled - PTP Monitor will be used
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] PHC2SYS - Enabled - phc2sys will be used
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [-] SyncE   - Disabled
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [-] SERVO   - Disabled
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] PPS     - Enabled - testptp will be used (if supported by
hardware)
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Going to analyze the configuration files
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Requested the following PTP interface: p0
2023-09-07 14:04:23 - Firefly
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Starting PPS configuration
2023-09-07 14:04:23 - Firefly - Init    - WARNING  - [-] PPS capability is missing, seems that the card doesn't
support PPS
2023-09-07 14:04:23 - Firefly - Init    - INFO     - capabilities:
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   50000000 maximum frequency adjustment (ppb)
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   0 programmable alarms
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   0 external time stamp channels
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   0 programmable periodic signals
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   0 pulse per second
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   0 programmable pins
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   0 cross timestamping
2023-09-07 14:04:23 - Firefly
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Running ptp4l
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Running Firefly PTP Monitor
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Running phc2sys
```

17.6.5.3.2 Firefly Output

On top of the container's log, Firefly defines an additional, non-volatile log that can be found in `/var/log/doca/firefly/firefly.log`.

This file contains the same output described in section "Container Output" and is useful for debugging deployment errors should the container stop its execution.

> To avoid disk space issues, the `/var/log/doca/firefly/firefly.log` file only contains the log from Firefly's initialization, and not the logs of the rest of the modules (ptp4l, phc2sys, etc.) or that of the PTP monitor. The latter is still included in the container log and can be inspected using the command `sudo crictl logs <CONTAINER-ID>`.

17.6.5.3.3 ptp4l Output

The ptp4l output can be found in the file `/var/log/doca/firefly/ptp4l.log`.

Example output:

```
ptp4l[192710.691]: rms 1 max 1 freq -114506 +/- 0 delay -15 +/- 0
ptp4l[192712.692]: rms 6 max 9 freq -114501 +/- 3 delay -15 +/- 0
ptp4l[192714.692]: rms 7 max 9 freq -114511 +/- 3 delay -13 +/- 0
ptp4l[192716.692]: rms 5 max 7 freq -114502 +/- 1 delay -13 +/- 0
ptp4l[192718.693]: rms 4 max 6 freq -114509 +/- 2 delay -13 +/- 0
```

```
ptp4l[192720.693]: rms 3 max 3 freq -114506 +/- 2 delay -13 +/- 0
ptp4l[192722.694]: rms 4 max 6 freq -114510 +/- 3 delay -12 +/- 0
ptp4l[192724.694]: rms 5 max 7 freq -114510 +/- 5 delay -12 +/- 1
ptp4l[192726.695]: rms 4 max 5 freq -114508 +/- 3 delay -11 +/- 0
ptp4l[192728.695]: rms 6 max 9 freq -114504 +/- 4 delay -11 +/- 0
```

17.6.5.3.4 phc2sys Output

The phc2sys output can be found in the file `/var/log/doca/firefly/phc2sys.log`.

Example output:

```
phc2sys[1873325.928]: reconfiguring after port state change
phc2sys[1873325.928]: selecting CLOCK_REALTIME for synchronization
phc2sys[1873325.928]: selecting enp3s0f0s4 as the master clock
phc2sys[1873325.928]: CLOCK_REALTIME phc offset      1378 s2 freq -165051 delay    255
phc2sys[1873326.928]: CLOCK_REALTIME phc offset      1378 s2 freq -163673 delay    240
phc2sys[1873327.928]: port 62b785.fffe.0c9369-1 changed state
phc2sys[1873327.929]: CLOCK_REALTIME phc offset        14 s2 freq -164624 delay    255
phc2sys[1873328.936]: CLOCK_REALTIME phc offset        89 s2 freq -164545 delay    240
```

17.6.5.3.5 SyncE Output

The SyncE output can be found in the file `/var/log/doca/firefly/synced.log`.

Example output:

```
INFO     [05/09/2023 05:11:01.493414]: SyncE Group #0: is in TRACKING holdover acquired mode on p0, frequency_diff:
0 (ppb)
INFO     [05/09/2023 05:11:02.502963]: SyncE Group #0: is in TRACKING holdover acquired mode on p0, frequency_diff:
-113 (ppb)
INFO     [05/09/2023 05:11:03.512491]: SyncE Group #0: is in TRACKING holdover acquired mode on p0, frequency_diff:
37 (ppb)
```

The verbosity of the output from the `SYNCE` module is limited by default. To set the output to be more verbose, set the `verbose` option to `1` (True).

Before:

```
# Example #4 - Overwrite the value of verbose in the [global] section of the SyncE configuration file.
#- name: CONF_SYNCE_global_verbose
#   value: "1"
```

After:

```
# Example #4 - Overwrite the value of verbose in the [global] section of the SyncE configuration file.
- name: CONF_SYNCE_global_verbose
  value: "1"
```

17.6.5.3.6 Firefly Servo Output

The Firefly servo output can be found in the file `/var/log/doca/firefly/servo.log`.

Example output:

```
2024-03-18 09:04:22 - Firefly - SERVO  - INFO     - offset   +8 +/- 2  freq   -5.66 +/- 0.41  delay -48 +/- 2
2024-03-18 09:04:24 - Firefly - SERVO  - INFO     - offset   +4 +/- 2  freq   -6.35 +/- 0.36  delay -47 +/- 2
2024-03-18 09:04:26 - Firefly - SERVO  - INFO     - offset   +2 +/- 2  freq   -6.75 +/- 0.41  delay -47 +/- 1
2024-03-18 09:04:28 - Firefly - SERVO  - INFO     - offset   +0 +/- 2  freq   -6.97 +/- 0.35  delay -47 +/- 1
2024-03-18 09:04:30 - Firefly - SERVO  - INFO     - offset   +0 +/- 3  freq   -7.30 +/- 0.60  delay -47 +/- 1
2024-03-18 09:04:33 - Firefly - SERVO  - INFO     - offset   +1 +/- 2  freq   -6.93 +/- 0.41  delay -47 +/- 1
```

```
2024-03-18 09:04:35 - Firefly - SERVO  - INFO     - offset   +1 +/- 2  freq   -6.81 +/- 0.48  delay -47 +/- 1
2024-03-18 09:04:37 - Firefly - SERVO  - INFO     - offset   +2 +/- 2  freq   -6.76 +/- 0.52  delay -48 +/- 2
```

17.6.5.4 Tx Timestamping Support on DPU Mode

When the BlueField is operating in DPU mode, additional OVS configuration is required as mentioned in step 6 of section "Setting Up Network Interfaces for DPU Mode". This configuration achieves the following:

- Proper support for incoming/outgoing multicast traffic
- Enabling Tx timestamping

Firefly only gets the packet timestamping for outgoing PTP messages (Tx timestamping) when they are offloaded to the hardware. As such, when working with OVS, users must ensure this traffic flow is properly recognized and offloaded. If offloading does not take place, Firefly gets stuck in a fault loop while waiting to receive the Tx timestamp events:

```
ptp4l[2912.797]: timed out while polling for tx timestamp
ptp4l[2912.797]: increasing tx_timestamp_timeout may correct this issue, but it is likely caused by a driver bug
ptp4l[2912.797]: port 1 (enp3s0f0s4): send sync failed
ptp4l[2923.528]: timed out while polling for tx timestamp
ptp4l[2923.528]: increasing tx_timestamp_timeout may correct this issue, but it is likely caused by a driver bug
ptp4l[2923.528]: port 1 (enp3s0f0s4): send sync failed
```

The solution to this issue:

- Activation of hardware offloading in OVS
- OpenFlow rules that ensure OVS properly recognizes the traffic and offloads it to the hardware
- Modification to the `fault_reset_interval` configuration value to ensure timely recovery from the fault induced by the first packet being always treated by software (until the rule is offloaded to hardware). As such, Firefly requires that the `fault_reset_interval` value is 1 or less. Proper warnings are raised if an improper value is detected. The value is updated accordingly in the built-in profiles.

When these configurations are in order, Firefly includes a report for a single fault during boot, but recovers from it and continues as usual:

```
ptp4l[3715.687]: timed out while polling for tx timestamp
ptp4l[3715.687]: increasing tx_timestamp_timeout may correct this issue, but it is likely caused by a driver bug
ptp4l[3715.687]: port 1 (enp3s0f0s4): send delay request failed
```

17.6.5.4.1 Troubleshooting Tx Timestamp Issues

As explained earlier, there are several layers required to ensure Tx timestamping works as necessary by Firefly. The following is a list of commands to debug the state of each layer:

1. Inspect the OpenFlow rules:

   ```
   $ sudo ovs-ofctl dump-flows uplink
   cookie=0x0, duration=4075.576s, table=0, n_packets=2437, n_bytes=209582, udp,in_port=en3f0pf0sf4,tp_src=319
   actions=output:p0
   cookie=0x0, duration=4075.549s, table=0, n_packets=1216, n_bytes=109420, udp,in_port=p0,tp_src=319
   actions=output:en3f0pf0sf4
   cookie=0x0, duration=4075.521s, table=0, n_packets=13, n_bytes=1242, udp,in_port=en3f0pf0sf4,tp_src=320
   actions=output:p0
   cookie=0x0, duration=4074.604s, table=0, n_packets=3034, n_bytes=297376, udp,in_port=p0,tp_src=320
   actions=output:en3f0pf0sf4
   cookie=0x0, duration=4075.856s, table=0, n_packets=184, n_bytes=12901, priority=0 actions=NORMAL
   ```

2. Inspect hardware TC rules while DOCA Firefly is deployed (the rules age out after 10 seconds without traffic):

```
$ sudo tc -s -d filter show dev en3f0pf0sf4 egress
filter ingress protocol ip pref 4 flower chain 0
filter ingress protocol ip pref 4 flower chain 0 handle 0x1
  eth_type ipv4
  ip_proto udp
  src_port 320
  ip_flags nofrag
  in_hw in_hw_count 1
    action order 1: mirred (Egress Redirect to device p0) stolen
    index 3 ref 1 bind 1 installed 7 sec used 7 sec
    Action statistics:
    Sent 0 bytes 0 pkt (dropped 0, overlimits 0 requeues 0)
    backlog 0b 0p requeues 0
    cookie bec8bd6ede4e86341e9045a6edb58ca2
    no_percpu

filter ingress protocol ip pref 4 flower chain 0 handle 0x2
  eth_type ipv4
  ip_proto udp
  src_port 319
  ip_flags nofrag
  in_hw in_hw_count 1
    action order 1: mirred (Egress Redirect to device p0) stolen
    index 4 ref 1 bind 1 installed 6 sec used 6 sec
    Action statistics:
    Sent 0 bytes 0 pkt (dropped 0, overlimits 0 requeues 0)
    backlog 0b 0p requeues 0
    cookie c568d97efd400de98608fbbf86ccdf3c
    no_percpu
```

> If no TC rules are present when Firefly is running, this usually indicates that hardware offloading is disabled at the OVS level, in which case it should be activated as explained under "Ensuring OVS Hardware Offload".

17.6.5.5 PTP

Firefly uses the `ptp4l` utility to handle the Precision Time Protocol (IEEE 1588).

Through the YAML file, users can configure the network interfaces used for the protocol:

```
# Network interfaces to be used (For multiple interfaces use a space (" ") separated list)
- name: PTP_INTERFACE
  # Set according to used interfaces on the local setup
  value: "p0"
```

Before the deployment of the container, users should configure this field to point at the desired network interface(s) configured in the previous steps.

17.6.5.6 PHC2SYS

Firefly uses the `phc2sys` utility to synchronize the OS's clock to the accurate time stamps received by `ptp4l`.

Through the YAML file, users can configure the command-line arguments used by the `phc2sys` program:

```
- name: PHC2SYS_ARGS
  value: "-a -r"
```

Firefly adds the following command-line arguments on top of the user-selected flags:

- Use of chosen configuration file (empty configuration file by default, or user-supplied file if specified in the YAML file)
- Redirection of output to a log file using the `-m` command line option

`phc2sys` must use the same `domainNumber` setting used by `ptp4l`. If the same `domainNumber` is not set by the user, Firefly does that automatically.

`phc2sys` is only able to accurately sync the clock of the hosting environment (usually the DPU, but may also be the host if deployed there) if other timing services, such as NTP, are disabled.

So, for instance, on Ubuntu 22.04, users must ensure that the NTP timing service is disabled by running:

```
systemctl stop systemd-timesyncd
```

17.6.5.7 SYNCE

Firefly uses the proprietary `synced` utility to implement the Synchronous Ethernet protocol, aimed at ensuring synchronization of the clock's frequency with the reference clock. Once achieved, both clocks are declared as "syntonized".

Through the YAML file, users can configure the network interfaces used for the protocol:

```
# Network interfaces to be used (For multiple interfaces use a space (" ") separated list)
- name: SYNCE_INTERFACE
  # Set according to used interfaces on the local setup
  value: "p0"
```

Before the deployment of the container, one should configure this field to point at the desired network interface(s) configured in the previous steps.

DOCA includes synced support for the `"dpll"` backend (default) which adds support for SFs and VFs. The `"dpll"` backend is the default backend used. If DOCA detects the system does not support it, it will automatically falls back to the `"mft"` backend.

In versions older than kernel 6.8 or BlueField Platform Software 2.8.0, only PFs are supported and only using the `"mft"` backend.

The backend option can be explicitly set using the YAML file by uncommenting the following lines:

Before

```
# Example #5 - Explicitly specify the used backend in the [global] section of the SyncE configuration file.
#- name: CONF_SYNCE_global_backend
#  # Options are "mft"/"dpll". If nothing is specified in YAML, "dpll" is taken as the default
#  value: "mft"
```

After

```
# Example #5 - Explicitly specify the used backend in the [global] section of the
SyncE configuration file.
- name: CONF_SYNCE_global_backend
  # Options are "mft"/"dpll". If nothing is specified in YAML, "dpll" is taken as
the default
  value: "mft"
```

The following is an example for the OVS commands required to route the SyncE-related traffic when using a SF on top of the `"dpll"` backend:

```
$ sudo ovs-ofctl add-flow uplink dl_dst=01:80:c2:00:00:02,in_port=en3f0pf0sf4,actions=p0
$ sudo ovs-ofctl add-flow uplink dl_dst=01:80:c2:00:00:02,in_port=p0,actions=en3f0pf0sf4
$ sudo ovs-ofctl add-flow uplink dl_dst=01:80:c2:00:00:02,actions=controller
```

This example uses the same OVS settings used earlier in the guide:

- `uplink` - bridge name
- `en3f0pf0sf4` - SF representor
- `p0` - PF interface we are working (port 0)

If your deployment uses different values make sure to adjust the above commands accordingly.

If the kernel version does not yet support this feature, and SF/VF are used, the following error is printed:

```
...
mlx5 DPLL kernel support appears to be missing
Falling back to MFT tools backend
...
```

If this error is shown, only PFs can be used, and `synced` falls back to using the `"mft"` backend.

17.6.5.8 PTP Monitor

PTP monitor periodically queries for various PTP-related information and prints it to the container's log.

The following is a sample output of this tool:

```
gmIdentity:                48:B0:2D:FF:FE:5C:4D:24 (48b02d.fffe.5c4d24)
portIdentity:              48:B0:2D:FF:FE:5C:53:44 (48b02d.fffe.5c5344-1)
port_state:                Active
domainNumber:              2
master_offset:             avg: 1   max:    -8  rms:    3
gmPresent:                 true
ptp_stable:                Recovered
UtcOffset:                 37
timeTraceable:             0
frequencyTraceable:        0
grandmasterPriority1:      128
gmClockClass:              248
gmClockAccuracy:           0x6
grandmasterPriority2:      128
gmOffsetScaledLogVariance: 0xffff
ptp_time (TAI):            Thu Sep  7 11:22:50 2023
ptp_time (UTC adjusted):   Thu Sep  7 11:22:13 2023
system_time (UTC):         Thu Sep  7 11:22:13 2023
error_count:               1
last_err_time (UTC):       Thu Sep  7 09:55:48 2023
```

Among others, this monitoring provides the following information:

- Details about the Grandmaster the DPU is syncing with
- Current PTP timestamp
- Health information such as connection errors during execution and whether they have been recovered from

PTP monitoring is disabled by default and can be activated by replacing the `disable` value with the IP address for the monitor server to use:

```
- name: MONITOR_STATE
  Value: "<IP address for the monitoring server>"
```

Once activated, the information can viewed from the container using the following command:

```
sudo crictl logs --tail=20 <CONTAINER-ID>
```

It is recommended to use the following `watch` command to actively monitor the PTP state:

```
sudo watch -n 1 crictl logs --tail=20 <CONTAINER-ID>
```

When triaging deployment issues, additional logging information can be found in the monitor's developer logs: `/var/log/doca/firefly/firefly_monitor_dev.log`.

> The monitoring feature connects to ptp4l's local UDS server to query the necessary information. This is why the configuration manager prevents users from modifying the `uds_address` and `uds_ro_address` fields used by ptp4l within the container.

17.6.5.8.1 Configuration

The PTP monitor supports configuration options which are passed through a dedicated configuration file like the rest of DOCA Firefly's modules. The built-in monitor configuration file can be found in the section "PTP Monitor". For ease of use, the file is also provided as part of DOCA's container resource as downloaded from NGC.

"Firefly Modules Configuration Options" contains a complete explanation of each of the configuration options alongside their default values.

To set a custom config file, users should locate their config file in the directory `/etc/firefly` and set the config file name in DOCA Firefly's YAML file.

```
- name: MONITOR_CONFIG_FILE
  value: my_custom_monitor.conf
```

In this example, `my_custom_monitor.conf` should be placed at `/etc/firefly/my_custom_monitor.conf`.

17.6.5.8.2 Time Representations (PTP Time vs System Time)

Under most deployment scenarios, the PTP time shown by the monitor is presented according to the International Atomic Time (TAI) standard, while the system time would most commonly use the

Coordinated Universal Time (UTC). Due to the differences between these time representation models, the monitor provides 2 different time readings (each marked accordingly):

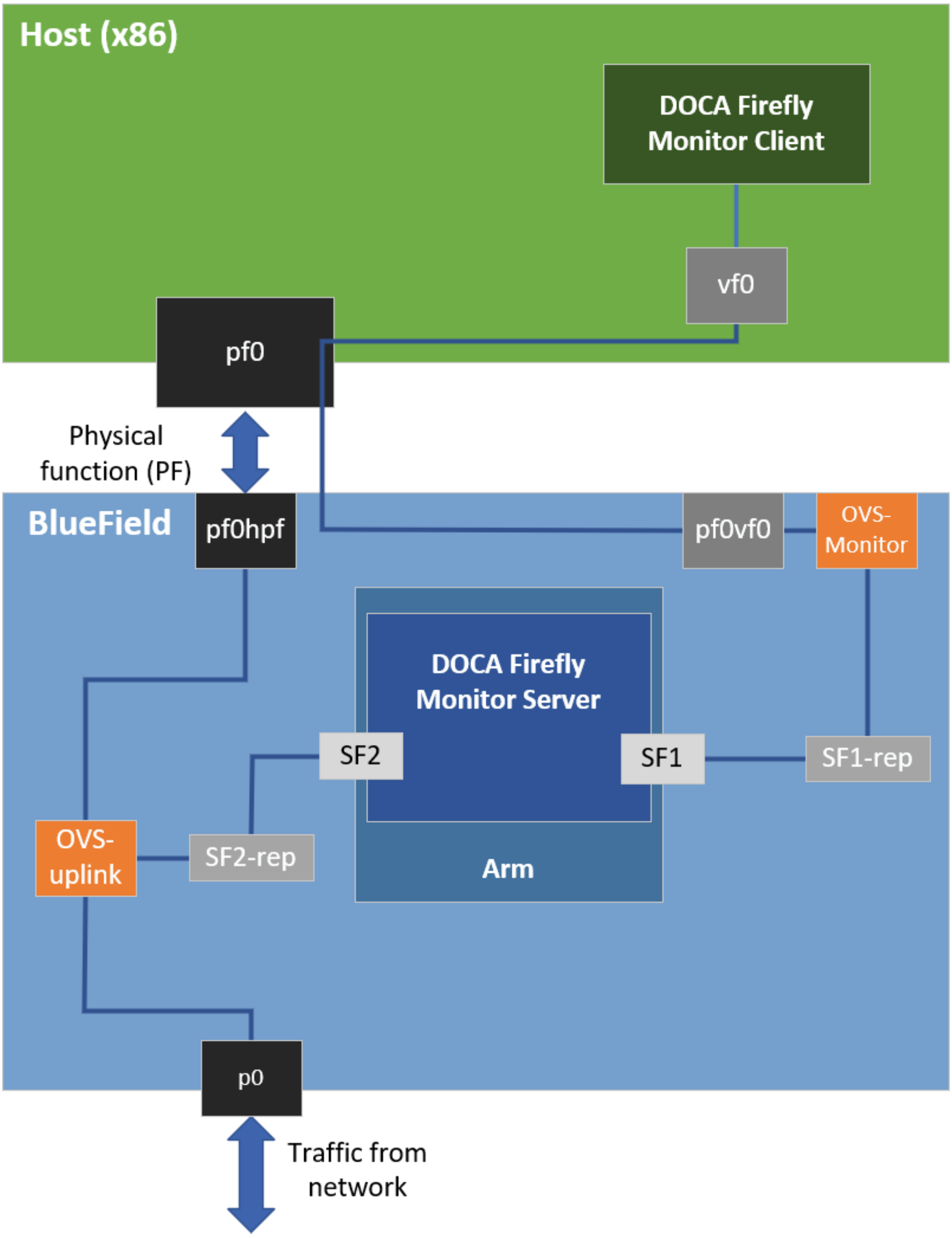
```
...
UtcOffset:                 37
...
ptp_time (TAI):            Thu Sep  7 11:22:50 2023
ptp_time (UTC adjusted):   Thu Sep  7 11:22:13 2023
system_time (UTC):         Thu Sep  7 11:22:13 2023
```

This difference (37 seconds in the above example) is intentional and stems from the amount of leap seconds since epoch. This is indicated by the `UtcOffset` field that is also included in the monitor's report.

17.6.5.8.3 Monitor Server

In addition to printing the monitoring data to the container's standard output available through the container logs, the monitoring data is also exposed through a gRPC server that clients can subscribe to. This allows a monitoring client on the host to subscribe to monitor events from the service running on top of the DPU, thus providing better visibility.

The following diagram presents the recommended deployment architecture for connecting the monitoring client (on the host) to the monitor server (on the DPU).



Based on the above, when activating the monitor feature, the user must provide the IP address to be used by the monitor server:

```
- name: MONITOR_STATE
  value: "<IP address for the monitoring server>"
```

Users can choose to only view the monitoring events through the container logs without connecting to the monitoring server. In this case, it is recommended to configure the local host IP address (127.0.0.1) in the YAML file to avoid exposing it to an unwanted network.

17.6.5.8.4 Monitor Client

The required files for the monitor client are available under the service's dedicated NGC resource "scripts" directory.

Example command line for executing the python-based monitor client from a Linux host:

```
$ sudo pip3 install click protobuf grpcio
$ ./doca_firefly_monitor_client.py <ip-address-for-the-monitoring-server>
```

> ⚠ Reference source files and the `.proto` file used for Firefly's monitor are placed under the `src/` within the NGC resource.

17.6.5.9 Firefly Servo

Firefly's Servo module can be seen as an extension to the built-in set of servos offered by `linuxptp`. When active, `linuxptp` is automatically set to "free running" and the control over the physical hardware clock (PHC) is handed over to Firefly's own servo.

The following is a sample output of this tool when using the `l2-telco` profile (16 messages per seconds):

```
2024-03-18 07:46:45 - Firefly - SERVO  - INFO     - Detected new master clock: 48b02d.fffe.5c4d24-1
2024-03-18 07:46:45 - Firefly - SERVO  - INFO     - Transition from servo state IDLE to FREE_RUNNING
2024-03-18 07:46:47 - Firefly - SERVO  - INFO     - Estimated a logSyncInterval of: -4
2024-03-18 07:46:47 - Firefly - SERVO  - INFO     - Measured offset      18691      delay -47
2024-03-18 07:46:48 - Firefly - SERVO  - INFO     - Transition from servo state FREE_RUNNING to LOCKED
2024-03-18 07:46:50 - Firefly - SERVO  - INFO     - offset +164 +/- 164 freq   -1.50 +/- 0.00  delay -48 +/- 1
2024-03-18 07:46:52 - Firefly - SERVO  - INFO     - Transition from servo state LOCKED to LOCKED_STABLE
2024-03-18 07:46:52 - Firefly - SERVO  - INFO     - offset   +0 +/- 1  freq   -1.41 +/- 0.47  delay -48 +/- 1
2024-03-18 07:46:54 - Firefly - SERVO  - INFO     - offset   -8 +/- 4  freq   -4.21 +/- 1.40  delay -47 +/- 1
2024-03-18 07:46:57 - Firefly - SERVO  - INFO     - offset  -12 +/- 2  freq   -5.46 +/- 0.73  delay -47 +/- 1
2024-03-18 07:46:59 - Firefly - SERVO  - INFO     - offset  -13 +/- 2  freq   -6.13 +/- 0.65  delay -47 +/- 1
2024-03-18 07:47:01 - Firefly - SERVO  - INFO     - offset  -13 +/- 3  freq   -6.19 +/- 1.23  delay -47 +/- 2
2024-03-18 07:47:03 - Firefly - SERVO  - INFO     - offset  -19 +/- 2  freq   -8.04 +/- 0.96  delay -47 +/- 1
2024-03-18 07:47:06 - Firefly - SERVO  - INFO     - offset  -14 +/- 3  freq   -6.46 +/- 1.11  delay -47 +/- 1
2024-03-18 07:47:08 - Firefly - SERVO  - INFO     - offset  -16 +/- 2  freq   -7.32 +/- 0.78  delay -48 +/- 2
2024-03-18 07:47:10 - Firefly - SERVO  - INFO     - offset  -15 +/- 2  freq   -7.11 +/- 0.87  delay -47 +/- 2
2024-03-18 07:47:12 - Firefly - SERVO  - INFO     - offset  -14 +/- 1  freq   -6.74 +/- 0.57  delay -47 +/- 2
2024-03-18 07:47:15 - Firefly - SERVO  - INFO     - offset  -12 +/- 3  freq   -6.20 +/- 1.01  delay -48 +/- 1
2024-03-18 07:47:17 - Firefly - SERVO  - INFO     - offset  -13 +/- 2  freq   -6.40 +/- 0.89  delay -47 +/- 1
2024-03-18 07:47:19 - Firefly - SERVO  - INFO     - offset  -11 +/- 2  freq   -5.98 +/- 0.86  delay -48 +/- 1
2024-03-18 07:47:21 - Firefly - SERVO  - INFO     - offset  -10 +/- 2  freq   -5.75 +/- 0.87  delay -46 +/- 1
2024-03-18 07:47:24 - Firefly - SERVO  - INFO     - offset   -8 +/- 1  freq   -5.15 +/- 0.42  delay -47 +/- 1
```

As can be seen, the servo's behavior is similar to that of `linuxptp`'s `ptp4l` and consists of a state machine that tracks the state of the active PTP port (`FREE_RUNNING`, `LOCKED`, `LOCKED_STABLE`, etc).

Firefly's Servo is disabled by default (in all profiles) and can be activated by replacing the `define_by_profile` value with `enable`:

```
# Activation status
- name: SERVO_STATE
  # Options are "enable"/"disable"/"defined_by_profile"
  value: "enable"
```

Once activated, the information can viewed from the module's log file `/var/log/doca/firefly/servo.log`.

17.6.5.9.1 Firefly Servo Configuration

Firefly's Servo is currently aimed for telco-related deployments, using the `l2-telco` profile including the use of SyncE. As such, the default values in the built-in configuration file are optimized for those scenarios.

The servo supports configuration options which are passed through a dedicated configuration file like the rest of DOCA Firefly's modules. The built-in servo configuration file can be found in the section "Firefly Servo". For ease of use, the file is also provided as part of DOCA's container resource as downloaded from NGC.

"Firefly Modules Configuration Options" contains a complete explanation of each of the configuration options alongside their default values.

To set a custom config file, users should locate their config file in the directory `/etc/firefly` and set the config file name in DOCA Firefly's YAML file.

```
- name: SERVO_CONFIG_FILE
  value: my_custom_servo.conf
```

In this example, `my_custom_servo.conf` should be placed at `/etc/firefly/my_custom_servo.conf`.

17.6.5.9.2 Dynamic Packet Rate Support

The servo has the ability to dynamically detect the packet rate used by the PTP grandmaster clock, so to calibrate itself accordingly incase it differs from the recommended 16 packets per seconds.

```
2024-03-18 07:46:45 - Firefly - SERVO  - INFO     - Transition from servo state IDLE to FREE_RUNNING
2024-03-18 07:46:47 - Firefly - SERVO  - INFO     - Estimated a logSyncInterval of: -4
2024-03-18 07:46:47 - Firefly - SERVO  - INFO     - Measured offset      18691      delay -47
```

In a case the message rate is constant and known in advance, the dynamic estimation can be disabled, in favour of a provided message rate:

```
- name: CONF_SERVO_global_servo_const_log_sync_interval
  value: "-2"
```

In the above example, a fixed message rate of 4 packets per seconds will be used (logSyncInterval of "-2").

⚠ While the servo was tested to produce stable results with various packets rates (2, 4, 8, 16, 32, 64, 128), it is only officially recommended for use in deployments using a packet rate of 16 packets per second.

17.6.5.10 VLAN Tagging

DOCA Firefly natively supports VLAN-tagging-enabled network interfaces.

17.6.5.10.1 Separated Mode

The name of the VLAN-enabled network interface should be the one passed through the YAML file in the `PTP_INTERFACE` field.

17.6.5.10.2 Embedded Mode

In addition to passing on the VLAN-enabled interface through the YAML as listed in the previous section, the user is also required to configure the network routing within the DPU to support the VLAN tagging:

1. The following example configures a VLAN tag of 10 to the `enp3s0f0s4` interface:

   ```
   $ sudo ip link add link enp3s0f0s4 name enp3s0f0s4.10 type vlan id 10
   $ sudo ip link set up enp3s0f0s4.10
   $ sudo ifconfig enp3s0f0s4.10 192.168.104.1 up
   ```

   In this example, `enp3s0f0s4.10` is the interface to be passed to DOCA Firefly.
2. Additional commands to route the traffic within the DPU:

   ```
   $ sudo ovs-ofctl add-flow uplink in_port=en3f0pf0sf4,dl_vlan=10,actions=output:p0
   $ sudo ovs-ofctl add-flow uplink in_port=p0,dl_vlan=10,actions=output:en3f0pf0sf4
   ```

17.6.5.11 Multiple Interfaces

DOCA Firefly can support multiple network interfaces through the following YAML file syntax:

```
- name: PTP_INTERFACE
  value: "<space (' ') separated list of interface names>"
```

For example:

```
- name: PTP_INTERFACE
  value: "p0 p1"
```

The monitoring feature is supported for multiple interfaces only when the `clientOnly` configuration is enabled.

Automatic mode (`-a`) for `phc2sys` is not supported when working with multiple interfaces. It is recommended to disable `phc2sys` in this mode.

17.6.6 Troubleshooting

When troubleshooting container deployment issues, it is highly recommended to follow the deployment steps and tips in the "Review Container Deployment" section of the NVIDIA DOCA Container Deployment Guide.

To debug the finalized configuration file used by Firefly, users can connect to the container as follows:

1. Open a shell session on the running container using the container ID:

```
sudo crictl exec -it <container-id> /bin/bash
```

2. Once connected to the container, the finalized configuration file can be found under the `/tmp` directory using the same filename as the original configuration file.

> More information regarding the configuration files can be found under section "Ensuring and Debugging Correctness of Config File".

17.6.6.1 Pod is Marked as "Ready" and No Container is Listed

17.6.6.1.1 Error

When deploying the container, the pod's STATE is marked as `Ready`, an image is listed, however no container can be seen running:

```
$ sudo crictl pods
POD ID              CREATED             STATE               NAME                                     NAMESPACE
ATTEMPT             RUNTIME
06bd84c07537e       4 seconds ago       Ready               doca-firefly-my-dpu                      default
0                   (default)

$ sudo crictl images
IMAGE                              TAG                 IMAGE ID            SIZE
k8s.gcr.io/pause                   3.2                 2a060e2e7101d       251kB
nvcr.io/nvidia/doca/doca_firefly   1.1.0-doca2.0.2     134cb22f34611       87.4MB

$ sudo crictl ps
CONTAINER           IMAGE               CREATED             STATE               NAME                     ATTEMPT
POD ID              POD
```

17.6.6.1.2 Solution

In most cases, the container did start, but immediately exited. This could be checked using the following command:

```
$ sudo crictl ps -a
CONTAINER           IMAGE               CREATED             STATE               NAME                     ATTEMPT
POD ID              POD
556bb78281e1d       134cb22f34611       7 seconds ago       Exited              doca-firefly             1
          06bd84c07537e       doca-firefly-my-dpu
```

Should the container fail (i.e., state of `Exited`) it is recommended to examine Firefly's main log at `/var/log/doca/firefly/firefly.log`.

In addition, for a short period of time after termination, the container logs could also be viewed using the the container's ID:

```
$ sudo crictl logs 556bb78281e1d
Starting DOCA Firefly - Version 1.1.0
...
Requested the following PTP interface: p10
Failed to find interface "p10". Aborting
```

17.6.6.2 Custom Config File is Not Found

17.6.6.2.1 Error

When DOCA Firefly is deployed using a custom configuration file, a deployment error occurs and the following log message appears:

```
...
2023-09-07 14:04:23 - Firefly - Init     - ERROR    - Custom config file not found: my_file.conf. Aborting
...
```

17.6.6.2.2 Solution

Check the custom file name written in the YAML file and make sure that you properly placed the file with that name under the `/etc/firefly/` directory of the DPU.

17.6.6.3 Profile is Not Supported

17.6.6.3.1 Error

When DOCA Firefly is deployed, a deployment error occurs and the following log message appears:

```
...
2023-09-07 14:04:23 - Firefly - Init     - ERROR    - profile <name> is not supported. Aborting
...
```

17.6.6.3.2 Solution

Verify that the profile selected in the YAML file matches one of the supported profiles as listed in the profiles table.

⚠ The profile name is case sensitive. The name must be specified in lower-case letters.

17.6.6.4 PPS Capability is Missing

17.6.6.4.1 Error

When DOCA Firefly is deployed and configured to use the `PPS` module, a deployment error occurs and the following log message appears:

```
...
2023-09-07 14:04:23 - Firefly - Init     - INFO     - Starting PPS configuration
2023-09-07 14:04:23 - Firefly - Init     - WARNING  - [-] PPS capability is missing, seems that the card doesn't
support PPS
2023-09-07 14:04:23 - Firefly - Init     - INFO     - capabilities:
2023-09-07 14:04:23 - Firefly - Init     - INFO     -   50000000 maximum frequency adjustment (ppb)
2023-09-07 14:04:23 - Firefly - Init     - INFO     -   0 programmable alarms
2023-09-07 14:04:23 - Firefly - Init     - INFO     -   0 external time stamp channels
2023-09-07 14:04:23 - Firefly - Init     - INFO     -   0 programmable periodic signals
2023-09-07 14:04:23 - Firefly - Init     - INFO     -   0 pulse per second
2023-09-07 14:04:23 - Firefly - Init     - INFO     -   0 programmable pins
2023-09-07 14:04:23 - Firefly - Init     - INFO     -   0 cross timestamping
...
```

17.6.6.4.2 Solution

This log indicates that the DPU hardware does not support PPS. However, PTP can still run on this hardware and you should see the line `Running ptp4l` in the container log, indicating that PTP is running successfully.

17.6.6.5 Timed Out While Polling for Tx Timestamp

17.6.6.5.1 Error

When the BlueField is operating in DPU mode, DOCA Firefly gets stuck in a fault loop while waiting to receive the Tx timestamp events:

```
ptp4l[2912.797]: timed out while polling for tx timestamp
ptp4l[2912.797]: increasing tx_timestamp_timeout may correct this issue, but it is likely caused by a driver bug
ptp4l[2912.797]: port 1 (enp3s0f0s4): send sync failed
ptp4l[2923.528]: timed out while polling for tx timestamp
ptp4l[2923.528]: increasing tx_timestamp_timeout may correct this issue, but it is likely caused by a driver bug
ptp4l[2923.528]: port 1 (enp3s0f0s4): send sync failed
```

DOCA Firefly has a known gap leading to this error appearing once, after which ptp4l recovers from it. This section only covers the case in which there is a fault loop and no recovery occurs.

17.6.6.5.2 Solution

DOCA Firefly's configurations were already adjusted to accommodate for Tx port timestamping. For more information about the reason for this error and for the designed recovery mechanism from it, refer to section "Tx Timestamping Support on DPU Mode".

17.6.6.6 Warning – Time Jumped Backwards

17.6.6.6.1 Error

When using Firefly's Servo module, the following warning log message is encountered on start:

```
 2024-01-01 14:04:23 - Firefly - SERVO   - WARNING  - Clock is going to jump backwards in time - this might have a
system-wide impact
```

17.6.6.6.2 Solution

This warning message indicates that the system's time jumped backwards with a value of at least one minute. This event is logged by Firefly given that such jumps might have system-wide implications. For more information, refer to section "Failed to Reserve Sandbox Name" in the NVIDIA DOCA Troubleshooting Guide.

Such jumps can only happen during Firefly's boot, before the Servo achieves initial time synchronization with the reference clock.

17.6.7 PTP Profile Default Config Files

17.6.7.1 Media Profile

```
#
# This config file contains configurations for media & entertainment alongside
# DOCA Firefly specific adjustments.
#

[global]
domainNumber            127
priority1               128
priority2               127
use_syslog                1
logging_level             6
tx_timestamp_timeout     30
hybrid_e2e                1
dscp_event               46
dscp_general             46
logAnnounceInterval      -2
announceReceiptTimeout    3
logSyncInterval          -3
logMinDelayReqInterval   -3
delay_mechanism         E2E
network_transport     UDPv4
# Value lesser or equal to 1 is required for Embedded Mode
fault_reset_interval      1
# Required for multiple interfaces support
boundary_clock_jbod       1
```

17.6.7.2 Default Profile

```
#
# This config file extends linuxptp default.cfg config file with DOCA Firefly
# specific adjustments.
#

[global]
# Value lesser or equal to 1 is required for Embedded Mode
fault_reset_interval                 1
# Required for multiple interfaces support
boundary_clock_jbod                  1
```

17.6.7.3 Telco (L2) Profile

```
#
# This config file extends linuxptp G.8275.1.cfg config file with DOCA Firefly
# specific adjustments.
#

[global]
dataset_comparison                       G.8275.x
G.8275.defaultDS.localPriority                128
maxStepsRemoved                               255
logAnnounceInterval                            -3
logSyncInterval                                -4
logMinDelayReqInterval                         -4
G.8275.portDS.localPriority                   128
ptp_dst_mac                     01:80:C2:00:00:0E
network_transport                              L2
domainNumber                                   24
# Value lesser or equal to 1 is required for Embedded Mode
fault_reset_interval                            1
# Required for multiple interfaces support
boundary_clock_jbod                             1
```

17.6.8 Firefly Modules Configuration Options

17.6.8.1 PTP Monitor

17.6.8.1.1 monitor-default.conf

```
#
# Default values for all of Firefly's PTP monitor configuration values.
#

[global]
# General
report_interval                     1000
# Debugging & Logging
doca_logging_level                    50
```

17.6.8.1.2 Configuration Options

- `report_interval` - the time interval (in milliseconds) for when the monitor should publish a report to all defined output providers (standard output, gRPC clients, etc). Default: 1000 (1 second).
- `doca_logging_level` - Logging level for the module, based on DOCA's logging levels. Default is 50 (INFO). Valid options:
    - 10=DISABLE
    - 20=CRITICAL
    - 30=ERROR
    - 40=WARNING
    - 50=INFO
    - 60=DEBUG

17.6.8.2 Firefly Servo

17.6.8.2.1 servo-default.conf

```
#
# Default values for all of Firefly's servo configuration values
#

[global]
# Time thresholds
offset_from_master_min_threshold   -1500
offset_from_master_max_threshold    1500
init_max_time_adjustment               0
max_time_adjustment                 1500
step_adjustment_threshold              0
hold_over_timer                        0
# Sampling Window & servo logic
warmup_period                       1500
sync_filter_length                     6
delay_request_filter_length            6
servo_adjustment_interval              4
servo_init_adjustment_interval        24
servo_const_log_sync_interval       0xFF
servo_window_min_samples               2
servo_num_offset_values                5
servo_pi_cutoff_frequency         0.0159
servo_pi_dumping_factor             7.85

# Debugging & Logging
summary_interval                    2000
doca_logging_level                    50
free_running                           0
```

17.6.8.2.2 Configuration Options

- `offset_from_master_min_threshold` - Minimal threshold (in nanoseconds) for declaring time offset from the master clock as "stable". Default is -1500 (-1.5 microseconds).
- `offset_from_master_max_threshold` - Maximal threshold (in nanoseconds) for declaring time offset from the master clock as "stable". Default is +1500 (+1.5 microseconds).
- `init_max_time_adjustment` - When active, defines the maximal allowed time (step) adjustment (in nanoseconds) before the servo reaches the "locked" state. Default is 0 (disabled).
- `max_time_adjustment` - When active, defines the maximal allowed reference time adjustment (in nanoseconds) after the servo has reached the "locked" state. Default is 1500 (1.5 microseconds).
- `step_adjustment_threshold` - When active, defines the thresholds above which a time (step) adjustment (in nanoseconds) would be allowed, even after the servo has reached the "locked" state. Default is 0 (disabled).
- `hold_over_timer` - When active, defines the time duration (in seconds) in which the servo stays in "hold over" mode, until reverting back to "free running". Default is 0 ("hold over" state is disabled).
- `warmup_period` - Time span (in milliseconds) during which samples are collected to estimate the `logSyncInterval` value (packet rate). Default is 1500 (1.5 seconds).
- `sync_filter_length` - Number of `SYNC` messages in the servo's history buffer. Default is 6.
- `delay_request_filter_length` - Number of `DELAY_REQUEST` messages in the servo's history buffer. Default is 6 messages.
- `servo_adjustment_interval` - Number of `SYNC` messages after which the PHC is updated once the servo has reached the "locked" state at least once. Default is 4 messages.
- `servo_init_adjustment_interval` - Number of `SYNC` messages after which the PHC is updated before the servo has ever reached the "locked" state. Default is 24 messages.
- `servo_const_log_sync_interval` - Known fixed value to be used as the `logSyncInterval` instead of trying to estimate it at runtime. Default is 0xFF (disabled).
- `servo_window_min_samples` - Minimal number of samples needed for a servo calculation. Default is 2 messages.
- `servo_num_offset_values` - Number of consecutive timestamps within the "offset from master" threshold that are required so to transition from the "locked" state and to the "locked stable" state. Default is 5 offset values.
- `servo_pi_cutoff_frequency` - The PI servo's cutoff frequency value. Default is 0.0159.
- `servo_pi_dumping_factor` - The PI servo's dumping factor value. Default is 7.85.
- `summary_interval` - The time interval (in milliseconds) for when the servo should publish a report log event. Default is 2000 (2 seconds).
- `doca_logging_level` - Logging level for the module, based on DOCA's logging levels. Default is 50 (INFO). Valid options:
    - 10=DISABLE
    - 20=CRITICAL
    - 30=ERROR
    - 40=WARNING
    - 50=INFO

    - 60=DEBUG
- `free_running` - Tell the servo to only log the operations, without actually adjusting the PHC. Default is 0 (disabled).

17.7 NVIDIA DOCA Flow Inspector Service Guide

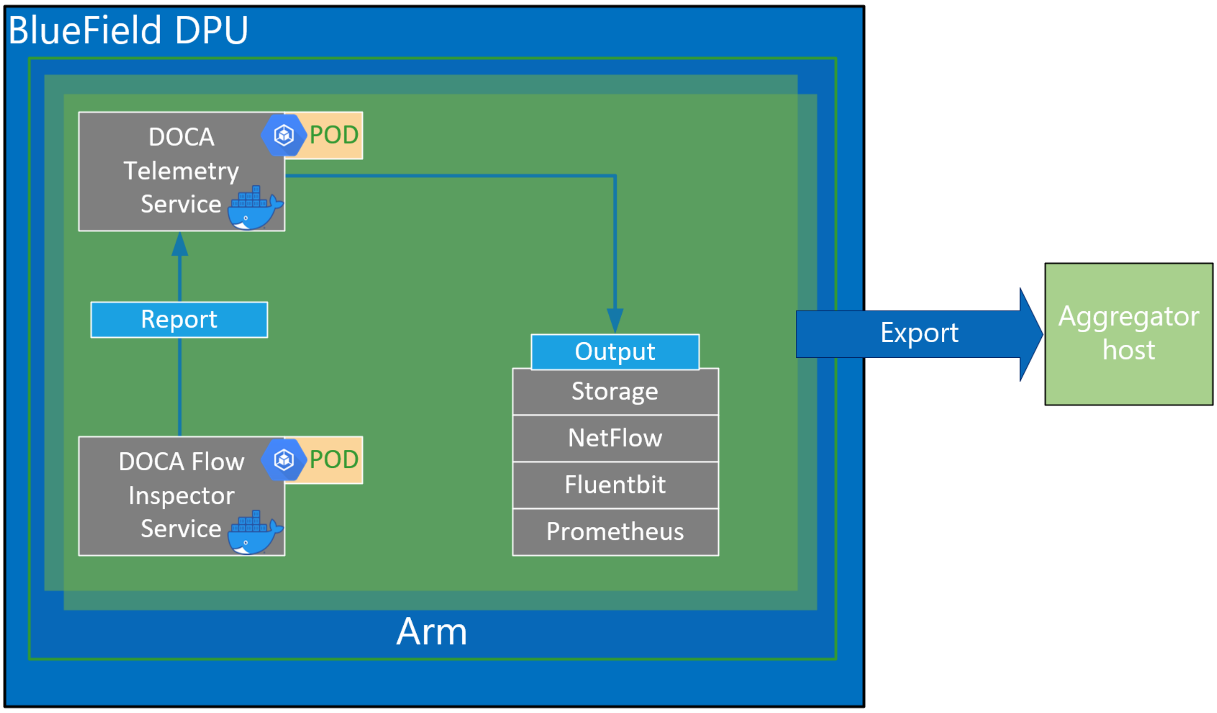
This guide provides instructions on how to use the DOCA Flow Inspector service container on top of NVIDIA® BlueField® DPU.

17.7.1 Introduction

DOCA Flow Inspector service enables real-time data monitoring and extraction of telemetry components. These components can be leveraged by various services, including those focused on security, big data, and other purposes.

DOCA Flow Inspector service is linked to DOCA Telemetry Service (DTS). It receives mirrored packets from the user parses the data, and forwards it to the DTS, which aggregates predefined statistics from various providers and sources. The service utilizes the DOCA Telemetry Exporter API to communicate with the DTS, while the DPDK infrastructure facilitates packet acquisition at a user-space layer.

DOCA Flow Inspector operates within its dedicated Kubernetes pod on BlueField, aimed at receiving mirrored packets for analysis. The received packets are parsed and transmitted, in a predefined structure, to a telemetry collector that manages the remaining telemetry aspects.



17.7.1.1 Service Flow

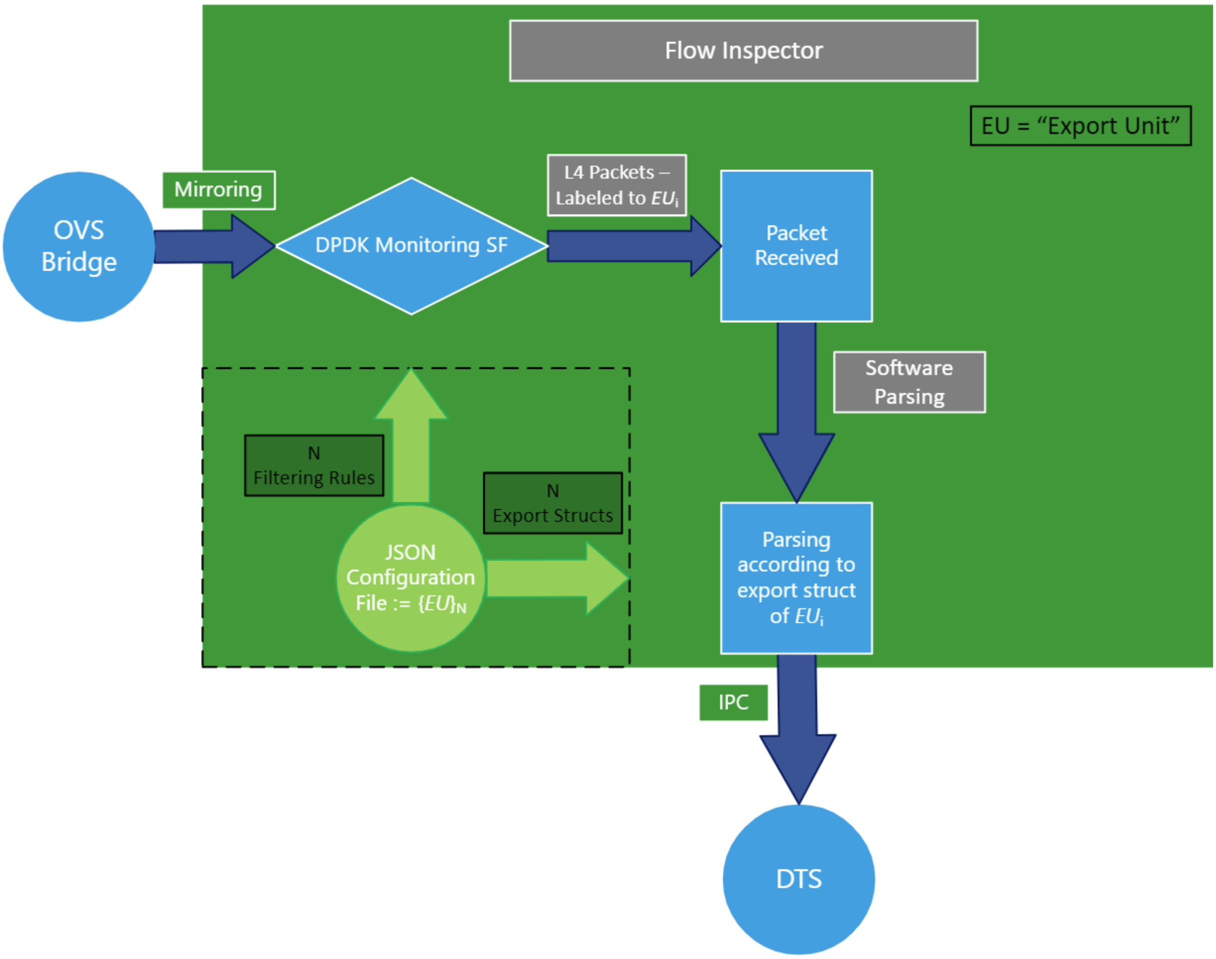
The DOCA Flow Inspector receives a configuration file in a JSON format which includes which of the mirrored packets should be filtered and which information should be sent to DTS for inspection.

The configuration file can include several export units under the "export-units" attribute. Each one is comprised of a "filter" and an "export". Each packet that matches one filter (based on the protocol and ports in the L4 header) is then parsed to the corresponding requested struct defined in the export. That information only is sent for inspection. A packet that does not match any filter is dropped.

In addition, the configuration file could contain FI optional configuration flags, see JSON format and example in the Configuration section.

The service watches for changes in the JSON configuration file in runtime and for any change that reconfigures the service.

The DOCA Flow Inspector runs on top of DPDK to acquire L4. The packets are then filtered and HW-marked with their export unit index. The packets are then parsed according to their export unit and export struct, and then forwarded to the telemetry collector using IPC.



Configuration phase:

1. A JSON file is used as input to configure the export units (i.e., filters and corresponding export structs).
2. The filters are translated to HW rules on the SF (scalable function port) using the DOCA Flow library.
3. The connection to the telemetry collector is initialized and all export structures are registered to DTS.

Inspection phase:

1. Traffic is mirrored to the relevant SF.
2. Ingress traffic is received through the configured SF.
3. Non-L4 traffic and packets that do not match any filter are dropped using hardware rules.
4. Packets matching a filter are marked with the export unit index they match and are passed to the software layer in the Arm cores.
5. Packets are parsed to the desired struct by the index of export unit.
6. The telemetry information is forwarded to the telemetry agent using IPC.
7. Mirrored packets are freed.
8. If the JSON file is changed, run the configuration phase with the updated file.

17.7.2 Requirements

Before deploying the flow inspector container, ensure that the following prerequisites are satisfied:

1. Create the needed files and directories. Folders should be created automatically. Make sure the `.json` file resides inside the folder:

   ```
   $ touch /opt/mellanox/doca/services/flow_inspector/bin/flow_inspector_cfg.json
   ```

   Validate that DTS's configuration folders exist. They should be created automatically when DTS is deployed.

   ```
   $ sudo mkdir -p /opt/mellanox/doca/services/telemetry/config
   $ sudo mkdir -p /opt/mellanox/doca/services/telemetry/ipc_sockets
   $ sudo mkdir -p /opt/mellanox/doca/services/telemetry/data
   ```

2. Allocate huge pages as needed by DPDK. This requires root privileges.

   ```
   $ sudo echo 2048 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
   ```

   Or alternatively:

   ```
   $ sudo echo '2048' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
   $ sudo mkdir /mnt/huge
   $ sudo mount -t hugetlbfs nodev /mnt/huge
   ```

   Deploy a scalable function according to NVIDIA BlueField DPU Scalable Function User Guide and mirror packets accordingly using the Open vSwitch command.
   For example:

   a. Mirror packets from `p0` to `sf4`:

      ```
      $ ovs-vsctl add-br ovsbr1
      $ ovs-vsctl add-port ovsbr1 p0
      $ ovs-vsctl add-port ovsbr1 en3f0pf0sf4
      $ ovs-vsctl -- --id=@p1 get port en3f0pf0sf4 \
                  -- --id=@p2 get port p0 \
                  -- --id=@m create mirror name=m0 select-dst-port=@p2 select-src-port=@p2 output-port=@p
      1 \
                  -- set bridge ovsbr1 mirrors=@m
      ```

   b. Mirror packets from `pf0hpf` or `p0` that pass through `sf4`:

      ```
      $ ovs-vsctl add-br ovsbr1
      $ ovs-vsctl add-port ovsbr1 pf0hpf
      $ ovs-vsctl add-port ovsbr1 p0
      $ ovs-vsctl add-port ovsbr1 en3f0pf0sf4
      ```

```
$ ovs-vsctl -- --id=@p1 get port en3f0pf0sf4 \
            -- --id=@p2 get port pf0hpf \
            -- --id=@m create mirror name=m0 select-dst-port=@p2 select-src-port=@p2 output-port=@p
1 \
            -- set bridge ovsbr1 mirrors=@m
$ ovs-vsctl -- --id=@p1 get port en3f0pf0sf4 \
            -- --id=@p2 get port p0 \
            -- --id=@m create mirror name=m0 select-dst-port=@p2 select-src-port=@p2 output-port=@p
1 \
            -- set bridge ovsbr1 mirrors=@m
```

The output of last command (creating the mirror) should output a sequence of letters and numbers similar to the following:

```
0d248ca8-66af-427c-b600-af1e286056e1
```

> The designated SF must be created as a trusted function. Additional details can be found in the NVIDIA BlueField DPU Scalable Function User Guide.

17.7.3 Service Deployment

For information about the deployment of DOCA containers on top of the BlueField DPU, refer to NVIDIA DOCA Container Deployment Guide.

DTS is available on NGC, NVIDIA's container catalog. Service-specific configuration steps and deployment instructions can be found under the service's container page.

> The order of running DTS and DOCA Flow Inspector is important. You must launch DTS, wait a few seconds, and then launch DOCA Flow Inspector.

17.7.4 Configuration

17.7.4.1 JSON Input

The DOCA Flow Inspector configuration file should be placed under `/opt/mellanox/doca/services/flow_inspector/bin/<json_file_name>.json` and be built in the following format:

```
{
    /* Optional param, time period to check for changes in JSON config file (in seconds) and flush telemetry buffer
if enabled (default is 60 seconds) */
    "config-sample-rate": <time>,

    /* Optional param, telemetry buffer size in bytes (default is 60KB) */
    "telemetry-buffer-size": <size>,

    /* Optional param, enable periodic telemetry buffer flush and defining the period time (in seconds) */
    "telemetry-flush-rate": <numeric value in seconds>,

    /* Mandatory param, Flow Inspector export units */
    "export-units":
    [

        /* Export Unit 0 */
        {
            "filter":
            {     "protocols": [<L4 protocols separated by comma>], # What L4 protocols are allowed
                  "ports":
                    [
                            [<source port>, <destination port>],
                            [<source ports range>, <destination ports range>],
                            <... more pairs of source, dest ports>
                    ]
            },
            "export":
            {
                    "fields": [<fields to be part of export struct, separated by comma>] # the Telemetry event will
contain these fields.
            }
        },
        <... More Export Units>
    ]
}
```

17.7.4.1.1 Export Unit Attributes

Allowed protocols:

- `"TCP"`
- `"UDP"`

Port range:

- It is possible to insert a range of ports for both source and destination
- Range should include borders `[start_port-end_port]`

Allowed ports:

- All ports in range `0` - `65535` as a string
- Or `*` to indicate any ports

Allowed fields in export struct:

- `timestamp` - timestamp indicating when it was received by the service
- `host_ip` - the IP of the host running the service
- `src_mac` - source MAC address
- `dst_mac` - destination MAC address
- `src_ip` - source IP
- `dst_ip` - destination IP
- `protocol` - L4 protocol
- `src_port` - source port
- `dst_port` - destination port
- `flags` - additional flags (relevant to TCP only)
- `data_len` - data payload length
- `data_short` - short version of data (payload sliced to first 64 bytes)
- `data_medium` - medium version of data (payload sliced to first 1500 bytes)
- `data_long` - long version of data (payload sliced to first 9*1024 bytes)

JSON example:

```
{
    /* Optional param, time period to check for changes in JSON config file (in seconds) and flush telemetry buffer
if enabled (default is 60 seconds) */
    "config-sample-rate": 30,

    /* Optional param, telemetry maximum buffer size in bytes */
    "telemetry-buffer-size": 70000,

    /* Optional param, enable periodic telemetry buffer flush and defining the period time (in seconds) */
    "telemetry-flush-rate": 1.5,

   /* Mandatory param, Flow Inspector export units */
    "export-units":
    [

        /* Export Unit 0 */
        {
            "filter":
```

```
            {
                "protocols": ["tcp", "udp"],
                "ports":
                    [
                        ["*","433-460"],
                        ["20480","28341"],
                        ["28341","20480"],
                        ["68", "67"],
                        ["67", "68"]
                    ]
            },
            "export":
            {
                "fields": ["timestamp", "host_ip", "src_mac", "dst_mac", "src_ip", "dst_ip", "protocol", "src_port",
                    "dst_port", "flags", "data_len", "data_long"]
            }
        },

        /* Export Unit 1 */
        {
            "filter":
            {
                "protocols": ["tcp"],
                "ports":
                    [
                        ["5-10","422"],
                        ["80","80"]
                    ]
            },
            "export":
            {
                "fields": ["timestamp","dst_ip", "host_ip", "data_len", "flags", "data_medium"]
            }
        }
    ]
}
```

If a packet header contains L4 ports or L4 protocol which are not specified in any filter, they are filtered out.

17.7.4.2 Yaml File

The `.yaml` file downloaded from NGC can be easily edited according to your needs.

```
env:
  # Set according to the local setup
  - name: SF_NUM_1
    value: "2"   # Additional EAL flags, if needed
  - name: EAL_FLAGS
    value: ""   # Service-Specific command line arguments
  - name: SERVICE_ARGS
    value: "--policy /flow_inspector/flow_inspector_cfg.json -l 60"
```

- The `SF_NUM_1` value can be changed according to the SF used in the OVS configuration and can be found using the command in NVIDIA BlueField DPU Scalable Function User Guide.
- The `EAL_FLAGS` value must be changed according to the DPDK flags required when running the container.
- The `SERVICE_ARGS` are the runtime arguments received by the service:
  - `-l`, `--log-level <value>` - sets the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  - `-p`, `--policy <json_path>` - sets the JSON path inside the container

17.7.4.3 Verifying Output

Enabling write to data in the DTS allows debugging the validity of the DOCA Flow Inspector.

To allow DTS to write locally, uncomment the following line in `/opt/mellanox/doca/services/telemetry/config/dts_config.ini`:

```
#output=/data
```

Any changes in `dts_config.ini` necessitate restarting the pod for the new settings to apply.

The schema folder contains JSON-formatted metadata files which allow reading the binary files containing the actual data. The binary files are written according to the naming convention shown in the following example:

Requires installing the `tree` runtime utility (`apt install tree`).

```
$ tree /opt/mellanox/doca/services/telemetry/data/
/opt/mellanox/doca/services/telemetry/data/
    {year}
        {mmdd}
             {hash}
                  {source_id}
                      {source_tag}{timestamp}.bin
                  {another_source_id}
                       {another_source_tag}{timestamp}.bin
    schema
        schema_{MD5_digest}.json
```

New binary files appear when:

- The service starts
- When the binary file's max age/size restriction is reached
- When JSON file is changed and new schemas of telemetry are created
- An hour passes

If no schema or no data folders are present, refer to the Troubleshooting section in NVIDIA DOCA Telemetry Service Guide.

`source_id` is usually set to the machine hostname. `source_tag` is a line describing the collected counters, and it is often set as the provider's name or name of user-counters.

Reading the binary data can be done from within the DTS container using the following command:

```
crictl exec -it <Container-ID> /opt/mellanox/collectx/bin/clx_read -s /data/schema /data/path/to/datafile.bin
```

The data written locally should be shown in the following format assuming a packet matching Export Unit 1 from the example has arrived:

```
{
    "timestamp": 1656427771076130,
    "host_ip": "10.237.69.238",
    "src_ip": "11.7.62.4",
    "dst_ip": "11.7.62.5",
    "data_len": 1152,
    "data_short": "Hello World"
}
```

17.7.5 Troubleshooting

When troubleshooting container deployment issues, it is highly recommended to follow the deployment steps and tips in the "Review Container Deployment" section of the NVIDIA DOCA Container Deployment Guide.

17.7.5.1 Pod is Marked as "Ready" and No Container is Listed

17.7.5.1.1 Error

When deploying the container, the pod's STATE is marked as `Ready` , an image is listed, however no container can be seen running:

```
$ sudo crictl pods
POD ID              CREATED             STATE               NAME
NAMESPACE           ATTEMPT             RUNTIME
3162b71e67677       4 seconds ago       Ready               doca-flow-inspector-my-dpu                            default
0                   (default)

$ sudo crictl images
IMAGE                                     TAG                 IMAGE ID            SIZE
k8s.gcr.io/pause                          3.2                 2a060e2e7101d       487kB
nvcr.io/nvidia/doca/doca_flow_inspector   1.1.0-doca2.0.2     2af1e539eb7ab       86.8MB

$ sudo crictl ps
CONTAINER           IMAGE               CREATED             STATE               NAME                     ATTEMPT
POD ID              POD
```

17.7.5.1.2 Solution

In most cases, the container did start, but immediately exited. This could be checked using the following command:

```
$ sudo crictl ps -a
CONTAINER           IMAGE               CREATED             STATE               NAME                     ATTEMPT
POD ID              POD
556bb78281e1d       2af1e539eb7ab       6 seconds ago       Exited              doca-flow-inspector      1
          3162b71e67677       doca-flow-inspector-my-dpu
```

Should the container fail (i.e., state of `Exited`), it is recommended to examine the Flow Inspector's main log at `/var/log/doca/flow_inspector/flow_inspector_fi_dev.log`.

In addition, for a short period of time after termination, the container logs could also be viewed using the container's ID:

```
$ sudo crictl logs 556bb78281e1d
...
2023-10-04 11:42:55 - flow_inspector - FI     - ERROR    - JSON file was not found <config-file-path>.
```

17.7.5.2 Pod is Not Listed

17.7.5.2.1 Error

When placing the container's YAML file in the Kubelet's input folder, the service pod is not listed in the list of pods:

```
$ sudo crictl pods
POD ID              CREATED             STATE               NAME
NAMESPACE           ATTEMPT             RUNTIME
```

17.7.5.2.2 Solution

In most cases, the pod does not start due to the absence of the requested hugepages. This can be verified using the following command:

```
$ sudo journalctl -u kubelet -e. . .
Oct 04 12:12:19 <my-dpu> kubelet[2442376]: I1004 12:12:19.905064 2442376 predicate.go:103] "Failed to admit pod, 
unexpected error while attempting to recover from admission failure" pod="default/doca-flow-inspector-<my-dpu>"
 err="preemption: error finding a set of pods to preempt: no set of running pods found to reclaim resources: [(res:
hugepages-2Mi, q: 104563999874), ]"
```

17.8 NVIDIA DOCA HBN Service Guide

This guide provides instructions on how to use the DOCA HBN Service container on top of NVIDIA® BlueField® networking platform.

17.8.1 Introduction

ⓘ

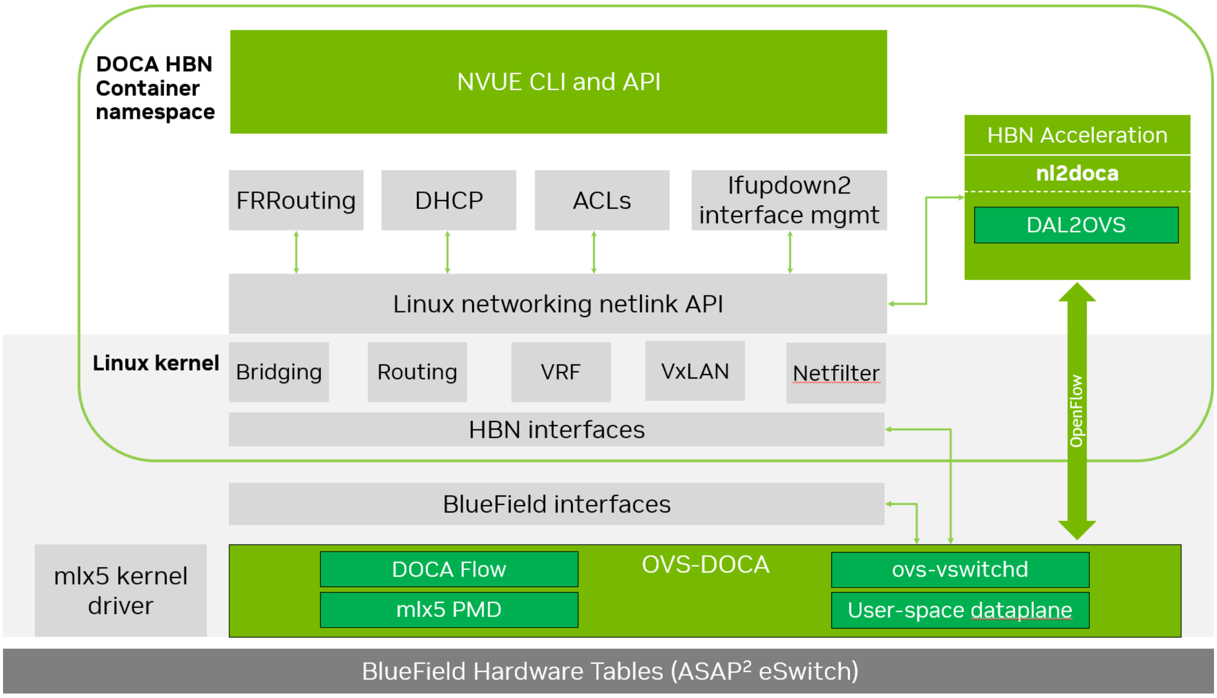
Beyond this page, the content of the HBN Service Guide is distributed across the following subpages:

- HBN Service Release Notes
- HBN Service Deployment
- HBN Service Configuration
- HBN Service Troubleshooting

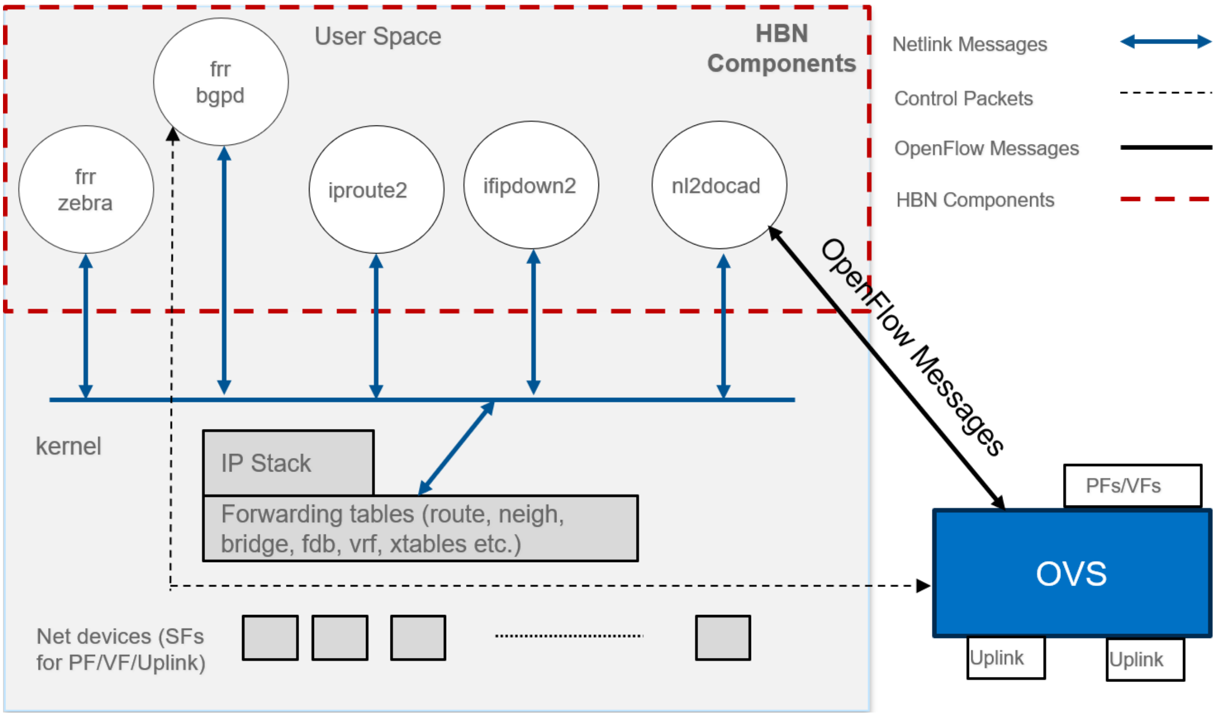
Host-based Networking (HBN) is a DOCA service that enables the network architect to design a network purely on L3 protocols, enabling routing to run on the server-side of the network by using the BlueField as a BGP router. The EVPN extension of BGP, supported by HBN, extends the L3 underlay network to multi-tenant environments with overlay L2 and L3 isolated networks.

The HBN solution packages a set of network functions inside a container which, itself, is packaged as a service pod to be run on BlueField Arm. At the core of HBN is the Linux networking BlueField acceleration driver Netlink-to-DOCA, or nl2docad. This daemon seamlessly accelerates Linux networking using DOCA APIs to program specific packet processing rules in BlueField hardware.

The driver mirrors the Linux kernel routing and bridging tables into the BlueField hardware tables by discovering the configured Linux networking objects using the Linux Netlink API. Dynamic network flows, as learned by the Linux kernel networking stack, are also programmed by the driver into BlueField hardware by listening to Linux kernel networking events.



The following diagram captures an overview of HBN and the interactions between various components of HBN.



- ifupdown2 is the interface manager which pushes all the interface related states to kernel
- The routing stack is implemented in FRR and pushes all the control states (EVPN MACs and routes) to kernel via netlink
- Kernel maintains the whole network state and relays the information using netlink. The kernel is also involved in the punt path and handling traffic that does not match any rules in the eSwitch.

- nl2docad listens for the network state via netlink and invokes the DOCA interface to accelerate the flows in BlueField hardware tables. nl2docad also offloads these flows to eSwitch.

17.8.1.1 Service Function Chaining

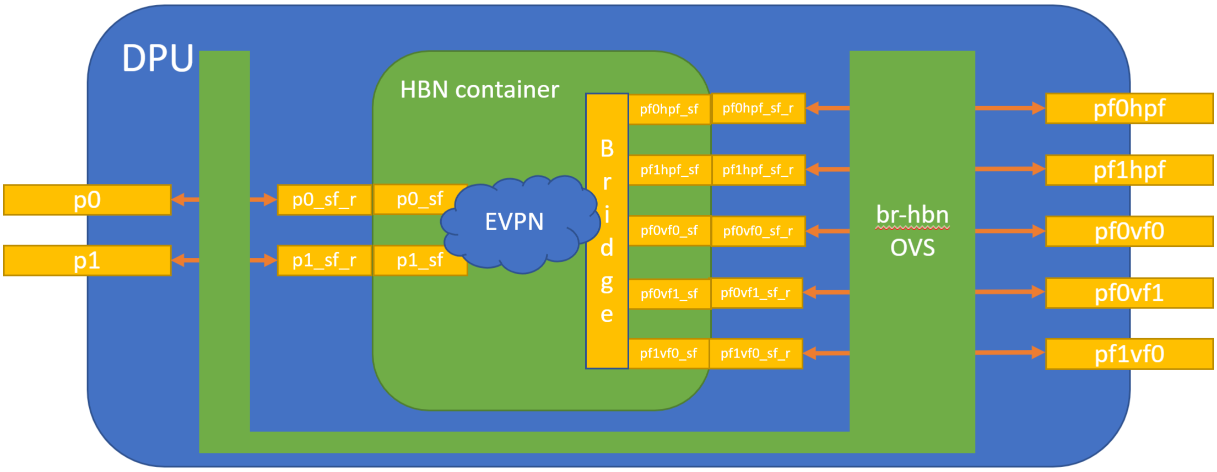
HBN is a "bump-in-the-wire" service and requires specific network configuration on BlueField called service function chaining (SFC). SFC configuration is used to redirect network traffic, which is originated from or forwarded to the host or BlueField itself via the HBN data plane.

The diagram below shows the fully detailed default configuration for HBN with SFC.

In this setup, the HBN container is configured to use sub-function ports (SFs) instead of the actual uplinks, PFs and VFs. To illustrate, for example:

- Uplinks - use `p0_if` instead of `p0`
- PF - use `pf0hpf_if` instead of `pf0hpf`
- VF - use `pf0vf0_if` instead of `pf0vf0`

The indirection layer between the SF and the actual ports is managed via a `br-hbn` OVS bridge automatically configured when the BFB image is installed on BlueField with HBN enabled. This indirection layer allows other services to be chained to existing SFs and provide additional functionality to transit traffic.



17.8.2 HBN Service Release Notes

The following subsections provide information on HBN service new features, interoperability, known issues, and bug fixes.

17.8.2.1 Changes and New Features

HBN 2.3.0 offers the following new features and updates:

- Added support for LLDP with HBN
- Added support to enforce security restrictions for NVUE API
- Added support for configurable PF/VF/SF mappings for HBN

- Added support for a 2-OVS bridge model to support custom steering flow, enabling user-defined modifications to forwarding pipeline
- Migrated stateful ACLs to OVS-DOCA CT to improve connection rate and better scalability
- Added IPv6 hash support for full IPv6 header
- Enabled async mode for bridge and route entry programing to improve flow programing rate
- Added support for 128K mega flow rules in OVS
- Modified HBN to reflect port states and indicate uplinks not based on names
- HBN NVUE config performance improvements

HBN 2.3.0 has the following user affecting changes from 2.2.0:

- HBN interface names changed from a suffix of `_sf` to `_if`. For example, `p0_sf` now becomes `p0_if`.
- Rest API access is now disabled by default

17.8.2.2 Supported Platforms and Interoperability

17.8.2.2.1 Supported BlueField Networking Platforms

HBN 2.3.0 has been validated on the following NVIDIA BlueField Networking Platforms:

- BlueField-2 DPUs:
  - BlueField-2 P-Series DPU 25GbE Dual-Port SFP56; PCIe Gen4 x8; Crypto Enabled; 16GB on-board DDR; 1GbE OOB management; HHHL
  - BlueField-2 P-Series DPU 25GbE Dual-Port SFP56; integrated BMC; PCIe Gen4 x8; Secure Boot Enabled; Crypto Enabled; 16GB on-board DDR; 1GbE OOB management; FHHL
  - BlueField-2 P-Series DPU 25GbE Dual-Port SFP56; integrated BMC; PCIe Gen4 x8; Secure Boot Enabled; Crypto Enabled; 32GB on-board DDR; 1GbE OOB management; FHHL
  - BlueField-2 P-Series DPU 100GbE Dual-Port QSFP56; integrated BMC; PCIe Gen4 x16; Secure Boot Enabled; Crypto Enabled; 32GB on-board DDR; 1GbE OOB management; FHHL
- BlueField-3 DPUs:
  - BlueField-3 B3210 P-Series FHHL DPU; 100GbE (default mode)/HDR100 IB; Dual-port QSFP112; PCIe Gen5.0 x16 with x16 PCIe extension option; 16 Arm cores; 32GB on-board DDR; integrated BMC; Crypto Enabled
  - BlueField-3 B3220 P-Series FHHL DPU; 200GbE (default mode)/NDR200 IB; Dual-port QSFP112; PCIe Gen5.0 x16 with x16 PCIe extension option; 16 Arm cores; 32GB on-board DDR; integrated BMC; Crypto Enabled
  - BlueField-3 B3240 P-Series Dual-slot FHHL DPU; 400GbE/NDR IB (default mode); Dual-port QSFP112; PCIe Gen5.0 x16 with x16 PCIe extension option; 16 Arm cores; 32GB on-board DDR; integrated BMC; Crypto Enabled
- BlueField-3 SuperNICs:
  - BlueField-3 B3210L E-series FHHL SuperNIC, 100GbE (default mode)/HDR100 IB, Dual port QSFP112, PCIe Gen4.0 x16, 8 Arm cores, 16GB on-board DDR, integrated BMC, Crypto Enabled
  - BlueField-3 B3220L E-Series FHHL SuperNIC, 200GbE (default mode)/NDR200 IB, Dual-port QSFP112, PCIe Gen5.0 x16, 8 Arm cores, 16GB on-board DDR, integrated BMC, Crypto Enabled

- BlueField-3 B3140L E-Series FHHL SuperNIC, 400GbE/ NDR IB (default mode), Single-port QSFP112, PCIe Gen5.0 x16, 8 Arm cores, 16GB on-board DDR, integrated BMC, Crypto Enabled
- BlueField-3 B3140H E-series HHHL SuperNIC, 400GbE (default mode)/NDR IB, Single-port QSFP112, PCIe Gen5.0 x16, 8 Arm cores, 16GB on board DDR, integrated BMC, Crypto Enabled

⚠ BlueField platforms with 8GB on-board DDR memory are currently not supported with HBN.

17.8.2.2.2 Supported BlueField OS

HBN 2.3.0 supports DOCA 2.8.0 (BSP 4.8.0) on Ubuntu 22.04 OS.

17.8.2.2.3 Verified Scalability Limits

HBN 2.8.0 has been tested to sustain the following maximum scalability limits:

Limit	BlueField-2	BlueField-3	Comments
VTEP peers (BlueFields per control plane) in the fabric	4k	4k	Number of BlueFields (VTEPs) within a single overlay fabric (reachable in the underlay)
L2 VNIs/Overlay networks per BlueField	20	20	Total number of L2 VNIs in the fabric for L2 VXLAN use-case assuming every interface is associated with its own VLAN + L2 VNI
L3 VNIs/Overlay networks per BlueField	20	20	Total number of L3 VNIs in the fabric for L3 VXLAN use-case assuming every interface is associated with its own VLAN + L2 VNI + L3 VNI + VRF
BlueFields per a single L2 VNI network	4k	4k	Total number of DPUs, configured with the same L2 VNI (3 real DPUs, 2000 emulated VTEPs)
BlueFields per a single L3 VNI network	4k	4k	Total number of DPUs, configured with the same L3 VNI (3 real DPUs, 2000 emulated VTEPs)
Maximum number of local MAC/ARP entries per BlueField	20	20	Max total number of MAC/ARP entries learned from the host on the DPU
Maximum number of local BGP routes per BlueField	200	200	Max total number of BGP routes advertised by the host to the BlueField (BGP peering with the host): 100 IPv4 + 100 IPv6
Maximum number of remote L3 LPM routes (underlay)	4k	4k	IPv4 or IPv6 underlay LPM routes per BlueField (default + host routes + LPM)
Maximum number of EVPN type-2 entries	16K	16k	Remote overlay MAC/IP entries for compute peers stored on a single BlueField (L2 EVPN use case)

Limit	BlueField-2	BlueField-3	Comments
Maximum number of EVPN type-5 entries	16K	16K	Remote overlay L3 LPM entries for compute peers stored on a single BlueField (L3 EVPN use case)
Maximum number of PFs on the Host side	2	2	Total number of PFs visible to the host
Maximum number of VFs on the Host side	16	16	Total number of VFs created on the host
Maximum number of SFs on BlueField side	2	2	Total number of SF devices created on BlueField Arm

17.8.2.3 Known Issues

The following table lists the known issues and limitations for this release of HBN.

3743942	Description: HBN container may hang in init-sfs during container restart when the HBN YAML file (i.e., `/etc/kubelet.d/doca_hbn.yaml`) is modified while container is running.
	Workaround: If the container hangs in init-sfs for more than 1 minute, reload the DPU.
	Keywords: Hang; container
	Reported in HBN version: 2.3.0
3961387	Description: The changing of the port number for NVUE REST API using nv CLI/API is not supported. The following command should not be used to change the port number: `nv set system api port <port-no>`
	Workaround: On HBN, NVUE is accessible through 8765 (i.e., default port number).
	Keywords: NVUE API; port number
	Reported in HBN version: 2.3.0
3965589	Description: When SR-IOV VFs are created or deleted and recreated, some ports may stay in ethX naming format and not be properly renamed to pfXvfY format. This results in the port remaining in error state as when running the command `ovs-vsctl show` due to the SFC and HBN not recognizing it.
	Workaround: Reboot the BlueField.
	Keywords: Port; nomenclature; convention
	Reported in HBN version: 2.3.0
3967748	Description: The command `nv show system api connections` does not return any data.
	Workaround: N/A
	Keywords: REST API; nginx
	Reported in HBN version: 2.3.0
4004191	Description: Due to security fixes on BlueField-2, the number of context switches increased by 20% which may result in user applications (e.g., nl2doca) running slower.
	Workaround: N/A
	Keyword: BlueField-2; performance
	Reported in HBN version: 2.3.0
3769309	Description: A ping or other IP connectivity from a locally connected host in vrf-X to an interface IP address on the DPU/HBN itself in vrf-Y will not work, even if VRF route-leaking is enabled between these two VRFs.
	Workaround: N/A
	Keyword: IP
	Reported in HBN version: 2.2.0
3886379	Description: Deleting and re-adding SR-IOV ports might result in some ports in br-hbn bridge going in error state.
	Workaround: If possible, an appropriate number of SR-IOV ports should be chosen at BFB install time. But if a change is made and if the system has this error, the host must undergo a power cycle to resolve the issue.
	Keyword: Bridge; SR-IOV
	Reported in HBN version: 2.2.0
3835295	Description: Traffic entering HBN service on a host PF/VF main-interface and exiting on a sub-interface of the same PF/VF (and vice versa) is not hardware offloaded. Similarly, traffic entering HBN service on one sub-interface and exiting on another sub-interface of the same host PF/VF is also not hardware offloaded.

3743942	Description: HBN container may hang in init-sfs during container restart when the HBN YAML file (i.e., `/etc/kubelet.d/doca_hbn.yaml`) is modified while container is running.
	Workaround: If the container hangs in init-sfs for more than 1 minute, reload the DPU.
	Keywords: Hang; container
	Reported in HBN version: 2.3.0
3961387	Description: The changing of the port number for NVUE REST API using nv CLI/API is not supported. The following command should not be used to change the port number: `nv set system api port <port-no>`
	Workaround: On HBN, NVUE is accessible through 8765 (i.e., default port number).
	Keywords: NVUE API; port number
	Reported in HBN version: 2.3.0
3965589	Description: When SR-IOV VFs are created or deleted and recreated, some ports may stay in ethX naming format and not be properly renamed to pfXvfY format. This results in the port remaining in error state as when running the command `ovs-vsctl show` due to the SFC and HBN not recognizing it.
	Workaround: Reboot the BlueField.
	Keywords: Port; nomenclature; convention
	Reported in HBN version: 2.3.0
3967748	Description: The command `nv show system api connections` does not return any data.
	Workaround: N/A
	Keywords: REST API; nginx
	Reported in HBN version: 2.3.0
4004191	Description: Due to security fixes on BlueField-2, the number of context switches increased by 20% which may result in user applications (e.g., nl2doca) running slower.
	Workaround: N/A
	Keyword: BlueField-2; performance
	Reported in HBN version: 2.3.0
3769309	Description: A ping or other IP connectivity from a locally connected host in vrf-X to an interface IP address on the DPU/HBN itself in vrf-Y will not work, even if VRF route-leaking is enabled between these two VRFs.
	Workaround: N/A
	Keyword: IP
	Reported in HBN version: 2.2.0
3886379	Description: Deleting and re-adding SR-IOV ports might result in some ports in br-hbn bridge going in error state.
	Workaround: If possible, an appropriate number of SR-IOV ports should be chosen at BFB install time. But if a change is made and if the system has this error, the host must undergo a power cycle to resolve the issue.
	Keyword: Bridge; SR-IOV
	Reported in HBN version: 2.2.0
3835295	Description: Traffic entering HBN service on a host PF/VF main-interface and exiting on a sub-interface of the same PF/VF (and vice versa) is not hardware offloaded. Similarly, traffic entering HBN service on one sub-interface and exiting on another sub-interface of the same host PF/VF is also not hardware offloaded.

3743942	Description: HBN container may hang in init-sfs during container restart when the HBN YAML file (i.e., `/etc/kubelet.d/doca_hbn.yaml`) is modified while container is running.
	Workaround: If the container hangs in init-sfs for more than 1 minute, reload the DPU.
	Keywords: Hang; container
	Reported in HBN version: 2.3.0
3961387	Description: The changing of the port number for NVUE REST API using nv CLI/API is not supported. The following command should not be used to change the port number: `nv set system api port <port-no>`
	Workaround: On HBN, NVUE is accessible through 8765 (i.e., default port number).
	Keywords: NVUE API; port number
	Reported in HBN version: 2.3.0
3965589	Description: When SR-IOV VFs are created or deleted and recreated, some ports may stay in ethX naming format and not be properly renamed to pfXvfY format. This results in the port remaining in error state as when running the command `ovs-vsctl show` due to the SFC and HBN not recognizing it.
	Workaround: Reboot the BlueField.
	Keywords: Port; nomenclature; convention
	Reported in HBN version: 2.3.0
3967748	Description: The command `nv show system api connections` does not return any data.
	Workaround: N/A
	Keywords: REST API; nginx
	Reported in HBN version: 2.3.0
4004191	Description: Due to security fixes on BlueField-2, the number of context switches increased by 20% which may result in user applications (e.g., nl2doca) running slower.
	Workaround: N/A
	Keyword: BlueField-2; performance
	Reported in HBN version: 2.3.0
3769309	Description: A ping or other IP connectivity from a locally connected host in vrf-X to an interface IP address on the DPU/HBN itself in vrf-Y will not work, even if VRF route-leaking is enabled between these two VRFs.
	Workaround: N/A
	Keyword: IP
	Reported in HBN version: 2.2.0
3886379	Description: Deleting and re-adding SR-IOV ports might result in some ports in br-hbn bridge going in error state.
	Workaround: If possible, an appropriate number of SR-IOV ports should be chosen at BFB install time. But if a change is made and if the system has this error, the host must undergo a power cycle to resolve the issue.
	Keyword: Bridge; SR-IOV
	Reported in HBN version: 2.2.0
3835295	Description: Traffic entering HBN service on a host PF/VF main-interface and exiting on a sub-interface of the same PF/VF (and vice versa) is not hardware offloaded. Similarly, traffic entering HBN service on one sub-interface and exiting on another sub-interface of the same host PF/VF is also not hardware offloaded.

3743942	Description: HBN container may hang in init-sfs during container restart when the HBN YAML file (i.e., `/etc/kubelet.d/doca_hbn.yaml`) is modified while container is running.
	Workaround: If the container hangs in init-sfs for more than 1 minute, reload the DPU.
	Keywords: Hang; container
	Reported in HBN version: 2.3.0
3961387	Description: The changing of the port number for NVUE REST API using nv CLI/API is not supported. The following command should not be used to change the port number: `nv set system api port <port-no>`
	Workaround: On HBN, NVUE is accessible through 8765 (i.e., default port number).
	Keywords: NVUE API; port number
	Reported in HBN version: 2.3.0
3965589	Description: When SR-IOV VFs are created or deleted and recreated, some ports may stay in ethX naming format and not be properly renamed to pfXvfY format. This results in the port remaining in error state as when running the command `ovs-vsctl show` due to the SFC and HBN not recognizing it.
	Workaround: Reboot the BlueField.
	Keywords: Port; nomenclature; convention
	Reported in HBN version: 2.3.0
3967748	Description: The command `nv show system api connections` does not return any data.
	Workaround: N/A
	Keywords: REST API; nginx
	Reported in HBN version: 2.3.0
4004191	Description: Due to security fixes on BlueField-2, the number of context switches increased by 20% which may result in user applications (e.g., nl2doca) running slower.
	Workaround: N/A
	Keyword: BlueField-2; performance
	Reported in HBN version: 2.3.0
3769309	Description: A ping or other IP connectivity from a locally connected host in vrf-X to an interface IP address on the DPU/HBN itself in vrf-Y will not work, even if VRF route-leaking is enabled between these two VRFs.
	Workaround: N/A
	Keyword: IP
	Reported in HBN version: 2.2.0
3886379	Description: Deleting and re-adding SR-IOV ports might result in some ports in br-hbn bridge going in error state.
	Workaround: If possible, an appropriate number of SR-IOV ports should be chosen at BFB install time. But if a change is made and if the system has this error, the host must undergo a power cycle to resolve the issue.
	Keyword: Bridge; SR-IOV
	Reported in HBN version: 2.2.0
3835295	Description: Traffic entering HBN service on a host PF/VF main-interface and exiting on a sub-interface of the same PF/VF (and vice versa) is not hardware offloaded. Similarly, traffic entering HBN service on one sub-interface and exiting on another sub-interface of the same host PF/VF is also not hardware offloaded.

3743942	Description: HBN container may hang in init-sfs during container restart when the HBN YAML file (i.e., `/etc/kubelet.d/doca_hbn.yaml`) is modified while container is running.
	Workaround: If the container hangs in init-sfs for more than 1 minute, reload the DPU.
	Keywords: Hang; container
	Reported in HBN version: 2.3.0
3961387	Description: The changing of the port number for NVUE REST API using nv CLI/API is not supported. The following command should not be used to change the port number: `nv set system api port <port-no>`
	Workaround: On HBN, NVUE is accessible through 8765 (i.e., default port number).
	Keywords: NVUE API; port number
	Reported in HBN version: 2.3.0
3965589	Description: When SR-IOV VFs are created or deleted and recreated, some ports may stay in ethX naming format and not be properly renamed to pfXvfY format. This results in the port remaining in error state as when running the command `ovs-vsctl show` due to the SFC and HBN not recognizing it.
	Workaround: Reboot the BlueField.
	Keywords: Port; nomenclature; convention
	Reported in HBN version: 2.3.0
3967748	Description: The command `nv show system api connections` does not return any data.
	Workaround: N/A
	Keywords: REST API; nginx
	Reported in HBN version: 2.3.0
4004191	Description: Due to security fixes on BlueField-2, the number of context switches increased by 20% which may result in user applications (e.g., nl2doca) running slower.
	Workaround: N/A
	Keyword: BlueField-2; performance
	Reported in HBN version: 2.3.0
3769309	Description: A ping or other IP connectivity from a locally connected host in vrf-X to an interface IP address on the DPU/HBN itself in vrf-Y will not work, even if VRF route-leaking is enabled between these two VRFs.
	Workaround: N/A
	Keyword: IP
	Reported in HBN version: 2.2.0
3886379	Description: Deleting and re-adding SR-IOV ports might result in some ports in br-hbn bridge going in error state.
	Workaround: If possible, an appropriate number of SR-IOV ports should be chosen at BFB install time. But if a change is made and if the system has this error, the host must undergo a power cycle to resolve the issue.
	Keyword: Bridge; SR-IOV
	Reported in HBN version: 2.2.0
3835295	Description: Traffic entering HBN service on a host PF/VF main-interface and exiting on a sub-interface of the same PF/VF (and vice versa) is not hardware offloaded. Similarly, traffic entering HBN service on one sub-interface and exiting on another sub-interface of the same host PF/VF is also not hardware offloaded.

3743942	Description: HBN container may hang in init-sfs during container restart when the HBN YAML file (i.e., `/etc/kubelet.d/doca_hbn.yaml`) is modified while container is running.
	Workaround: If the container hangs in init-sfs for more than 1 minute, reload the DPU.
	Keywords: Hang; container
	Reported in HBN version: 2.3.0
3961387	Description: The changing of the port number for NVUE REST API using nv CLI/ API is not supported. The following command should not be used to change the port number: `nv set system api port <port-no>`
	Workaround: On HBN, NVUE is accessible through 8765 (i.e., default port number).
	Keywords: NVUE API; port number
	Reported in HBN version: 2.3.0
3965589	Description: When SR-IOV VFs are created or deleted and recreated, some ports may stay in ethX naming format and not be properly renamed to pfXvfY format. This results in the port remaining in error state as when running the command `ovs-vsctl show` due to the SFC and HBN not recognizing it.
	Workaround: Reboot the BlueField.
	Keywords: Port; nomenclature; convention
	Reported in HBN version: 2.3.0
3967748	Description: The command `nv show system api connections` does not return any data.
	Workaround: N/A
	Keywords: REST API; nginx
	Reported in HBN version: 2.3.0
4004191	Description: Due to security fixes on BlueField-2, the number of context switches increased by 20% which may result in user applications (e.g., nl2doca) running slower.
	Workaround: N/A
	Keyword: BlueField-2; performance
	Reported in HBN version: 2.3.0
3769309	Description: A ping or other IP connectivity from a locally connected host in vrf-X to an interface IP address on the DPU/HBN itself in vrf-Y will not work, even if VRF route-leaking is enabled between these two VRFs.
	Workaround: N/A
	Keyword: IP
	Reported in HBN version: 2.2.0
3886379	Description: Deleting and re-adding SR-IOV ports might result in some ports in br-hbn bridge going in error state.
	Workaround: If possible, an appropriate number of SR-IOV ports should be chosen at BFB install time. But if a change is made and if the system has this error, the host must undergo a power cycle to resolve the issue.
	Keyword: Bridge; SR-IOV
	Reported in HBN version: 2.2.0
3835295	Description: Traffic entering HBN service on a host PF/VF main-interface and exiting on a sub-interface of the same PF/VF (and vice versa) is not hardware offloaded. Similarly, traffic entering HBN service on one sub-interface and exiting on another sub-interface of the same host PF/VF is also not hardware offloaded.

3743942	Description: HBN container may hang in init-sfs during container restart when the HBN YAML file (i.e., `/etc/kubelet.d/doca_hbn.yaml`) is modified while container is running.
	Workaround: If the container hangs in init-sfs for more than 1 minute, reload the DPU.
	Keywords: Hang; container
	Reported in HBN version: 2.3.0
3961387	Description: The changing of the port number for NVUE REST API using nv CLI/API is not supported. The following command should not be used to change the port number: `nv set system api port <port-no>`
	Workaround: On HBN, NVUE is accessible through 8765 (i.e., default port number).
	Keywords: NVUE API; port number
	Reported in HBN version: 2.3.0
3965589	Description: When SR-IOV VFs are created or deleted and recreated, some ports may stay in ethX naming format and not be properly renamed to pfXvfY format. This results in the port remaining in error state as when running the command `ovs-vsctl show` due to the SFC and HBN not recognizing it.
	Workaround: Reboot the BlueField.
	Keywords: Port; nomenclature; convention
	Reported in HBN version: 2.3.0
3967748	Description: The command `nv show system api connections` does not return any data.
	Workaround: N/A
	Keywords: REST API; nginx
	Reported in HBN version: 2.3.0
4004191	Description: Due to security fixes on BlueField-2, the number of context switches increased by 20% which may result in user applications (e.g., nl2doca) running slower.
	Workaround: N/A
	Keyword: BlueField-2; performance
	Reported in HBN version: 2.3.0
3769309	Description: A ping or other IP connectivity from a locally connected host in vrf-X to an interface IP address on the DPU/HBN itself in vrf-Y will not work, even if VRF route-leaking is enabled between these two VRFs.
	Workaround: N/A
	Keyword: IP
	Reported in HBN version: 2.2.0
3886379	Description: Deleting and re-adding SR-IOV ports might result in some ports in br-hbn bridge going in error state.
	Workaround: If possible, an appropriate number of SR-IOV ports should be chosen at BFB install time. But if a change is made and if the system has this error, the host must undergo a power cycle to resolve the issue.
	Keyword: Bridge; SR-IOV
	Reported in HBN version: 2.2.0
3835295	Description: Traffic entering HBN service on a host PF/VF main-interface and exiting on a sub-interface of the same PF/VF (and vice versa) is not hardware offloaded. Similarly, traffic entering HBN service on one sub-interface and exiting on another sub-interface of the same host PF/VF is also not hardware offloaded.

3743942	Description: HBN container may hang in init-sfs during container restart when the HBN YAML file (i.e., `/etc/kubelet.d/doca_hbn.yaml`) is modified while container is running.
	Workaround: If the container hangs in init-sfs for more than 1 minute, reload the DPU.
	Keywords: Hang; container
	Reported in HBN version: 2.3.0
3961387	Description: The changing of the port number for NVUE REST API using nv CLI/API is not supported. The following command should not be used to change the port number: `nv set system api port <port-no>`
	Workaround: On HBN, NVUE is accessible through 8765 (i.e., default port number).
	Keywords: NVUE API; port number
	Reported in HBN version: 2.3.0
3965589	Description: When SR-IOV VFs are created or deleted and recreated, some ports may stay in ethX naming format and not be properly renamed to pfXvfY format. This results in the port remaining in error state as when running the command `ovs-vsctl show` due to the SFC and HBN not recognizing it.
	Workaround: Reboot the BlueField.
	Keywords: Port; nomenclature; convention
	Reported in HBN version: 2.3.0
3967748	Description: The command `nv show system api connections` does not return any data.
	Workaround: N/A
	Keywords: REST API; nginx
	Reported in HBN version: 2.3.0
4004191	Description: Due to security fixes on BlueField-2, the number of context switches increased by 20% which may result in user applications (e.g., nl2doca) running slower.
	Workaround: N/A
	Keyword: BlueField-2; performance
	Reported in HBN version: 2.3.0
3769309	Description: A ping or other IP connectivity from a locally connected host in vrf-X to an interface IP address on the DPU/HBN itself in vrf-Y will not work, even if VRF route-leaking is enabled between these two VRFs.
	Workaround: N/A
	Keyword: IP
	Reported in HBN version: 2.2.0
3886379	Description: Deleting and re-adding SR-IOV ports might result in some ports in br-hbn bridge going in error state.
	Workaround: If possible, an appropriate number of SR-IOV ports should be chosen at BFB install time. But if a change is made and if the system has this error, the host must undergo a power cycle to resolve the issue.
	Keyword: Bridge; SR-IOV
	Reported in HBN version: 2.2.0
3835295	Description: Traffic entering HBN service on a host PF/VF main-interface and exiting on a sub-interface of the same PF/VF (and vice versa) is not hardware offloaded. Similarly, traffic entering HBN service on one sub-interface and exiting on another sub-interface of the same host PF/VF is also not hardware offloaded.

3743942	Description: HBN container may hang in init-sfs during container restart when the HBN YAML file (i.e., `/etc/kubelet.d/doca_hbn.yaml`) is modified while container is running.
	Workaround: If the container hangs in init-sfs for more than 1 minute, reload the DPU.
	Keywords: Hang; container
	Reported in HBN version: 2.3.0
3961387	Description: The changing of the port number for NVUE REST API using nv CLI/API is not supported. The following command should not be used to change the port number: `nv set system api port <port-no>`
	Workaround: On HBN, NVUE is accessible through 8765 (i.e., default port number).
	Keywords: NVUE API; port number
	Reported in HBN version: 2.3.0
3965589	Description: When SR-IOV VFs are created or deleted and recreated, some ports may stay in ethX naming format and not be properly renamed to pfXvfY format. This results in the port remaining in error state as when running the command `ovs-vsctl show` due to the SFC and HBN not recognizing it.
	Workaround: Reboot the BlueField.
	Keywords: Port; nomenclature; convention
	Reported in HBN version: 2.3.0
3967748	Description: The command `nv show system api connections` does not return any data.
	Workaround: N/A
	Keywords: REST API; nginx
	Reported in HBN version: 2.3.0
4004191	Description: Due to security fixes on BlueField-2, the number of context switches increased by 20% which may result in user applications (e.g., nl2doca) running slower.
	Workaround: N/A
	Keyword: BlueField-2; performance
	Reported in HBN version: 2.3.0
3769309	Description: A ping or other IP connectivity from a locally connected host in vrf-X to an interface IP address on the DPU/HBN itself in vrf-Y will not work, even if VRF route-leaking is enabled between these two VRFs.
	Workaround: N/A
	Keyword: IP
	Reported in HBN version: 2.2.0
3886379	Description: Deleting and re-adding SR-IOV ports might result in some ports in br-hbn bridge going in error state.
	Workaround: If possible, an appropriate number of SR-IOV ports should be chosen at BFB install time. But if a change is made and if the system has this error, the host must undergo a power cycle to resolve the issue.
	Keyword: Bridge; SR-IOV
	Reported in HBN version: 2.2.0
3835295	Description: Traffic entering HBN service on a host PF/VF main-interface and exiting on a sub-interface of the same PF/VF (and vice versa) is not hardware offloaded. Similarly, traffic entering HBN service on one sub-interface and exiting on another sub-interface of the same host PF/VF is also not hardware offloaded.

3743942	Description: HBN container may hang in init-sfs during container restart when the HBN YAML file (i.e., `/etc/kubelet.d/doca_hbn.yaml`) is modified while container is running.
	Workaround: If the container hangs in init-sfs for more than 1 minute, reload the DPU.
	Keywords: Hang; container
	Reported in HBN version: 2.3.0
3961387	Description: The changing of the port number for NVUE REST API using nv CLI/API is not supported. The following command should not be used to change the port number: `nv set system api port <port-no>`
	Workaround: On HBN, NVUE is accessible through 8765 (i.e., default port number).
	Keywords: NVUE API; port number
	Reported in HBN version: 2.3.0
3965589	Description: When SR-IOV VFs are created or deleted and recreated, some ports may stay in ethX naming format and not be properly renamed to pfXvfY format. This results in the port remaining in error state as when running the command `ovs-vsctl show` due to the SFC and HBN not recognizing it.
	Workaround: Reboot the BlueField.
	Keywords: Port; nomenclature; convention
	Reported in HBN version: 2.3.0
3967748	Description: The command `nv show system api connections` does not return any data.
	Workaround: N/A
	Keywords: REST API; nginx
	Reported in HBN version: 2.3.0
4004191	Description: Due to security fixes on BlueField-2, the number of context switches increased by 20% which may result in user applications (e.g., nl2doca) running slower.
	Workaround: N/A
	Keyword: BlueField-2; performance
	Reported in HBN version: 2.3.0
3769309	Description: A ping or other IP connectivity from a locally connected host in vrf-X to an interface IP address on the DPU/HBN itself in vrf-Y will not work, even if VRF route-leaking is enabled between these two VRFs.
	Workaround: N/A
	Keyword: IP
	Reported in HBN version: 2.2.0
3886379	Description: Deleting and re-adding SR-IOV ports might result in some ports in br-hbn bridge going in error state.
	Workaround: If possible, an appropriate number of SR-IOV ports should be chosen at BFB install time. But if a change is made and if the system has this error, the host must undergo a power cycle to resolve the issue.
	Keyword: Bridge; SR-IOV
	Reported in HBN version: 2.2.0
3835295	Description: Traffic entering HBN service on a host PF/VF main-interface and exiting on a sub-interface of the same PF/VF (and vice versa) is not hardware offloaded. Similarly, traffic entering HBN service on one sub-interface and exiting on another sub-interface of the same host PF/VF is also not hardware offloaded.

3743942	Description: HBN container may hang in init-sfs during container restart when the HBN YAML file (i.e., `/etc/kubelet.d/doca_hbn.yaml`) is modified while container is running.
	Workaround: If the container hangs in init-sfs for more than 1 minute, reload the DPU.
	Keywords: Hang; container
	Reported in HBN version: 2.3.0
3961387	Description: The changing of the port number for NVUE REST API using nv CLI/API is not supported. The following command should not be used to change the port number: `nv set system api port <port-no>`
	Workaround: On HBN, NVUE is accessible through 8765 (i.e., default port number).
	Keywords: NVUE API; port number
	Reported in HBN version: 2.3.0
3965589	Description: When SR-IOV VFs are created or deleted and recreated, some ports may stay in ethX naming format and not be properly renamed to pfXvfY format. This results in the port remaining in error state as when running the command `ovs-vsctl show` due to the SFC and HBN not recognizing it.
	Workaround: Reboot the BlueField.
	Keywords: Port; nomenclature; convention
	Reported in HBN version: 2.3.0
3967748	Description: The command `nv show system api connections` does not return any data.
	Workaround: N/A
	Keywords: REST API; nginx
	Reported in HBN version: 2.3.0
4004191	Description: Due to security fixes on BlueField-2, the number of context switches increased by 20% which may result in user applications (e.g., nl2doca) running slower.
	Workaround: N/A
	Keyword: BlueField-2; performance
	Reported in HBN version: 2.3.0
3769309	Description: A ping or other IP connectivity from a locally connected host in vrf-X to an interface IP address on the DPU/HBN itself in vrf-Y will not work, even if VRF route-leaking is enabled between these two VRFs.
	Workaround: N/A
	Keyword: IP
	Reported in HBN version: 2.2.0
3886379	Description: Deleting and re-adding SR-IOV ports might result in some ports in br-hbn bridge going in error state.
	Workaround: If possible, an appropriate number of SR-IOV ports should be chosen at BFB install time. But if a change is made and if the system has this error, the host must undergo a power cycle to resolve the issue.
	Keyword: Bridge; SR-IOV
	Reported in HBN version: 2.2.0
3835295	Description: Traffic entering HBN service on a host PF/VF main-interface and exiting on a sub-interface of the same PF/VF (and vice versa) is not hardware offloaded. Similarly, traffic entering HBN service on one sub-interface and exiting on another sub-interface of the same host PF/VF is also not hardware offloaded

3743942	Description: HBN container may hang in init-sfs during container restart when the HBN YAML file (i.e., `/etc/kubelet.d/doca_hbn.yaml`) is modified while container is running.
	Workaround: If the container hangs in init-sfs for more than 1 minute, reload the DPU.
	Keywords: Hang; container
	Reported in HBN version: 2.3.0
3961387	Description: The changing of the port number for NVUE REST API using nv CLI/API is not supported. The following command should not be used to change the port number: `nv set system api port <port-no>`
	Workaround: On HBN, NVUE is accessible through 8765 (i.e., default port number).
	Keywords: NVUE API; port number
	Reported in HBN version: 2.3.0
3965589	Description: When SR-IOV VFs are created or deleted and recreated, some ports may stay in ethX naming format and not be properly renamed to pfXvfY format. This results in the port remaining in error state as when running the command `ovs-vsctl show` due to the SFC and HBN not recognizing it.
	Workaround: Reboot the BlueField.
	Keywords: Port; nomenclature; convention
	Reported in HBN version: 2.3.0
3967748	Description: The command `nv show system api connections` does not return any data.
	Workaround: N/A
	Keywords: REST API; nginx
	Reported in HBN version: 2.3.0
4004191	Description: Due to security fixes on BlueField-2, the number of context switches increased by 20% which may result in user applications (e.g., nl2doca) running slower.
	Workaround: N/A
	Keyword: BlueField-2; performance
	Reported in HBN version: 2.3.0
3769309	Description: A ping or other IP connectivity from a locally connected host in vrf-X to an interface IP address on the DPU/HBN itself in vrf-Y will not work, even if VRF route-leaking is enabled between these two VRFs.
	Workaround: N/A
	Keyword: IP
	Reported in HBN version: 2.2.0
3886379	Description: Deleting and re-adding SR-IOV ports might result in some ports in br-hbn bridge going in error state.
	Workaround: If possible, an appropriate number of SR-IOV ports should be chosen at BFB install time. But if a change is made and if the system has this error, the host must undergo a power cycle to resolve the issue.
	Keyword: Bridge; SR-IOV
	Reported in HBN version: 2.2.0
3835295	Description: Traffic entering HBN service on a host PF/VF main-interface and exiting on a sub-interface of the same PF/VF (and vice versa) is not hardware offloaded. Similarly, traffic entering HBN service on one sub-interface and exiting on another sub-interface of the same host PF/VF is also not hardware offloaded

3743942	Description: HBN container may hang in init-sfs during container restart when the HBN YAML file (i.e., `/etc/kubelet.d/doca_hbn.yaml`) is modified while container is running.
	Workaround: If the container hangs in init-sfs for more than 1 minute, reload the DPU.
	Keywords: Hang; container
	Reported in HBN version: 2.3.0
3961387	Description: The changing of the port number for NVUE REST API using nv CLI/API is not supported. The following command should not be used to change the port number: `nv set system api port <port-no>`
	Workaround: On HBN, NVUE is accessible through 8765 (i.e., default port number).
	Keywords: NVUE API; port number
	Reported in HBN version: 2.3.0
3965589	Description: When SR-IOV VFs are created or deleted and recreated, some ports may stay in ethX naming format and not be properly renamed to pfXvfY format. This results in the port remaining in error state as when running the command `ovs-vsctl show` due to the SFC and HBN not recognizing it.
	Workaround: Reboot the BlueField.
	Keywords: Port; nomenclature; convention
	Reported in HBN version: 2.3.0
3967748	Description: The command `nv show system api connections` does not return any data.
	Workaround: N/A
	Keywords: REST API; nginx
	Reported in HBN version: 2.3.0
4004191	Description: Due to security fixes on BlueField-2, the number of context switches increased by 20% which may result in user applications (e.g., nl2doca) running slower.
	Workaround: N/A
	Keyword: BlueField-2; performance
	Reported in HBN version: 2.3.0
3769309	Description: A ping or other IP connectivity from a locally connected host in vrf-X to an interface IP address on the DPU/HBN itself in vrf-Y will not work, even if VRF route-leaking is enabled between these two VRFs.
	Workaround: N/A
	Keyword: IP
	Reported in HBN version: 2.2.0
3886379	Description: Deleting and re-adding SR-IOV ports might result in some ports in br-hbn bridge going in error state.
	Workaround: If possible, an appropriate number of SR-IOV ports should be chosen at BFB install time. But if a change is made and if the system has this error, the host must undergo a power cycle to resolve the issue.
	Keyword: Bridge; SR-IOV
	Reported in HBN version: 2.2.0
3835295	Description: Traffic entering HBN service on a host PF/VF main-interface and exiting on a sub-interface of the same PF/VF (and vice versa) is not hardware offloaded. Similarly, traffic entering HBN service on one sub-interface and exiting on another sub-interface of the same host PF/VF is also not hardware offloaded.

3743942	Description: HBN container may hang in init-sfs during container restart when the HBN YAML file (i.e., `/etc/kubelet.d/doca_hbn.yaml`) is modified while container is running.
	Workaround: If the container hangs in init-sfs for more than 1 minute, reload the DPU.
	Keywords: Hang; container
	Reported in HBN version: 2.3.0
3961387	Description: The changing of the port number for NVUE REST API using nv CLI/API is not supported. The following command should not be used to change the port number: `nv set system api port <port-no>`
	Workaround: On HBN, NVUE is accessible through 8765 (i.e., default port number).
	Keywords: NVUE API; port number
	Reported in HBN version: 2.3.0
3965589	Description: When SR-IOV VFs are created or deleted and recreated, some ports may stay in ethX naming format and not be properly renamed to pfXvfY format. This results in the port remaining in error state as when running the command `ovs-vsctl show` due to the SFC and HBN not recognizing it.
	Workaround: Reboot the BlueField.
	Keywords: Port; nomenclature; convention
	Reported in HBN version: 2.3.0
3967748	Description: The command `nv show system api connections` does not return any data.
	Workaround: N/A
	Keywords: REST API; nginx
	Reported in HBN version: 2.3.0
4004191	Description: Due to security fixes on BlueField-2, the number of context switches increased by 20% which may result in user applications (e.g., nl2doca) running slower.
	Workaround: N/A
	Keyword: BlueField-2; performance
	Reported in HBN version: 2.3.0
3769309	Description: A ping or other IP connectivity from a locally connected host in vrf-X to an interface IP address on the DPU/HBN itself in vrf-Y will not work, even if VRF route-leaking is enabled between these two VRFs.
	Workaround: N/A
	Keyword: IP
	Reported in HBN version: 2.2.0
3886379	Description: Deleting and re-adding SR-IOV ports might result in some ports in br-hbn bridge going in error state.
	Workaround: If possible, an appropriate number of SR-IOV ports should be chosen at BFB install time. But if a change is made and if the system has this error, the host must undergo a power cycle to resolve the issue.
	Keyword: Bridge; SR-IOV
	Reported in HBN version: 2.2.0
3835295	Description: Traffic entering HBN service on a host PF/VF main-interface and exiting on a sub-interface of the same PF/VF (and vice versa) is not hardware offloaded. Similarly, traffic entering HBN service on one sub-interface and exiting on another sub-interface of the same host PF/VF is also not hardware offloaded

3743942	Description: HBN container may hang in init-sfs during container restart when the HBN YAML file (i.e., `/etc/kubelet.d/doca_hbn.yaml`) is modified while container is running.
	Workaround: If the container hangs in init-sfs for more than 1 minute, reload the DPU.
	Keywords: Hang; container
	Reported in HBN version: 2.3.0
3961387	Description: The changing of the port number for NVUE REST API using nv CLI/ API is not supported. The following command should not be used to change the port number: `nv set system api port <port-no>`
	Workaround: On HBN, NVUE is accessible through 8765 (i.e., default port number).
	Keywords: NVUE API; port number
	Reported in HBN version: 2.3.0
3965589	Description: When SR-IOV VFs are created or deleted and recreated, some ports may stay in ethX naming format and not be properly renamed to pfXvfY format. This results in the port remaining in error state as when running the command `ovs-vsctl show` due to the SFC and HBN not recognizing it.
	Workaround: Reboot the BlueField.
	Keywords: Port; nomenclature; convention
	Reported in HBN version: 2.3.0
3967748	Description: The command `nv show system api connections` does not return any data.
	Workaround: N/A
	Keywords: REST API; nginx
	Reported in HBN version: 2.3.0
4004191	Description: Due to security fixes on BlueField-2, the number of context switches increased by 20% which may result in user applications (e.g., nl2doca) running slower.
	Workaround: N/A
	Keyword: BlueField-2; performance
	Reported in HBN version: 2.3.0
3769309	Description: A ping or other IP connectivity from a locally connected host in vrf-X to an interface IP address on the DPU/HBN itself in vrf-Y will not work, even if VRF route-leaking is enabled between these two VRFs.
	Workaround: N/A
	Keyword: IP
	Reported in HBN version: 2.2.0
3886379	Description: Deleting and re-adding SR-IOV ports might result in some ports in br-hbn bridge going in error state.
	Workaround: If possible, an appropriate number of SR-IOV ports should be chosen at BFB install time. But if a change is made and if the system has this error, the host must undergo a power cycle to resolve the issue.
	Keyword: Bridge; SR-IOV
	Reported in HBN version: 2.2.0
3835295	Description: Traffic entering HBN service on a host PF/VF main-interface and exiting on a sub-interface of the same PF/VF (and vice versa) is not hardware offloaded. Similarly, traffic entering HBN service on one sub-interface and exiting on another sub-interface of the same host PF/VF is also not hardware offloaded

3743942	Description: HBN container may hang in init-sfs during container restart when the HBN YAML file (i.e., `/etc/kubelet.d/doca_hbn.yaml`) is modified while container is running.
	Workaround: If the container hangs in init-sfs for more than 1 minute, reload the DPU.
	Keywords: Hang; container
	Reported in HBN version: 2.3.0
3961387	Description: The changing of the port number for NVUE REST API using nv CLI/API is not supported. The following command should not be used to change the port number: `nv set system api port <port-no>`
	Workaround: On HBN, NVUE is accessible through 8765 (i.e., default port number).
	Keywords: NVUE API; port number
	Reported in HBN version: 2.3.0
3965589	Description: When SR-IOV VFs are created or deleted and recreated, some ports may stay in ethX naming format and not be properly renamed to pfXvfY format. This results in the port remaining in error state as when running the command `ovs-vsctl show` due to the SFC and HBN not recognizing it.
	Workaround: Reboot the BlueField.
	Keywords: Port; nomenclature; convention
	Reported in HBN version: 2.3.0
3967748	Description: The command `nv show system api connections` does not return any data.
	Workaround: N/A
	Keywords: REST API; nginx
	Reported in HBN version: 2.3.0
4004191	Description: Due to security fixes on BlueField-2, the number of context switches increased by 20% which may result in user applications (e.g., nl2doca) running slower.
	Workaround: N/A
	Keyword: BlueField-2; performance
	Reported in HBN version: 2.3.0
3769309	Description: A ping or other IP connectivity from a locally connected host in vrf-X to an interface IP address on the DPU/HBN itself in vrf-Y will not work, even if VRF route-leaking is enabled between these two VRFs.
	Workaround: N/A
	Keyword: IP
	Reported in HBN version: 2.2.0
3886379	Description: Deleting and re-adding SR-IOV ports might result in some ports in br-hbn bridge going in error state.
	Workaround: If possible, an appropriate number of SR-IOV ports should be chosen at BFB install time. But if a change is made and if the system has this error, the host must undergo a power cycle to resolve the issue.
	Keyword: Bridge; SR-IOV
	Reported in HBN version: 2.2.0
3835295	Description: Traffic entering HBN service on a host PF/VF main-interface and exiting on a sub-interface of the same PF/VF (and vice versa) is not hardware offloaded. Similarly, traffic entering HBN service on one sub-interface and exiting on another sub-interface of the same host PF/VF is also not hardware offloaded.

3743942	Description: HBN container may hang in init-sfs during container restart when the HBN YAML file (i.e., `/etc/kubelet.d/doca_hbn.yaml`) is modified while container is running.
	Workaround: If the container hangs in init-sfs for more than 1 minute, reload the DPU.
	Keywords: Hang; container
	Reported in HBN version: 2.3.0
3961387	Description: The changing of the port number for NVUE REST API using nv CLI/API is not supported. The following command should not be used to change the port number: `nv set system api port <port-no>`
	Workaround: On HBN, NVUE is accessible through 8765 (i.e., default port number).
	Keywords: NVUE API; port number
	Reported in HBN version: 2.3.0
3965589	Description: When SR-IOV VFs are created or deleted and recreated, some ports may stay in ethX naming format and not be properly renamed to pfXvfY format. This results in the port remaining in error state as when running the command `ovs-vsctl show` due to the SFC and HBN not recognizing it.
	Workaround: Reboot the BlueField.
	Keywords: Port; nomenclature; convention
	Reported in HBN version: 2.3.0
3967748	Description: The command `nv show system api connections` does not return any data.
	Workaround: N/A
	Keywords: REST API; nginx
	Reported in HBN version: 2.3.0
4004191	Description: Due to security fixes on BlueField-2, the number of context switches increased by 20% which may result in user applications (e.g., nl2doca) running slower.
	Workaround: N/A
	Keyword: BlueField-2; performance
	Reported in HBN version: 2.3.0
3769309	Description: A ping or other IP connectivity from a locally connected host in vrf-X to an interface IP address on the DPU/HBN itself in vrf-Y will not work, even if VRF route-leaking is enabled between these two VRFs.
	Workaround: N/A
	Keyword: IP
	Reported in HBN version: 2.2.0
3886379	Description: Deleting and re-adding SR-IOV ports might result in some ports in br-hbn bridge going in error state.
	Workaround: If possible, an appropriate number of SR-IOV ports should be chosen at BFB install time. But if a change is made and if the system has this error, the host must undergo a power cycle to resolve the issue.
	Keyword: Bridge; SR-IOV
	Reported in HBN version: 2.2.0
3835295	Description: Traffic entering HBN service on a host PF/VF main-interface and exiting on a sub-interface of the same PF/VF (and vice versa) is not hardware offloaded. Similarly, traffic entering HBN service on one sub-interface and exiting on another sub-interface of the same host PF/VF is also not hardware offloaded.

3743942	Description: HBN container may hang in init-sfs during container restart when the HBN YAML file (i.e., `/etc/kubelet.d/doca_hbn.yaml`) is modified while container is running.
	Workaround: If the container hangs in init-sfs for more than 1 minute, reload the DPU.
	Keywords: Hang; container
	Reported in HBN version: 2.3.0
3961387	Description: The changing of the port number for NVUE REST API using nv CLI/API is not supported. The following command should not be used to change the port number: `nv set system api port <port-no>`
	Workaround: On HBN, NVUE is accessible through 8765 (i.e., default port number).
	Keywords: NVUE API; port number
	Reported in HBN version: 2.3.0
3965589	Description: When SR-IOV VFs are created or deleted and recreated, some ports may stay in ethX naming format and not be properly renamed to pfXvfY format. This results in the port remaining in error state as when running the command `ovs-vsctl show` due to the SFC and HBN not recognizing it.
	Workaround: Reboot the BlueField.
	Keywords: Port; nomenclature; convention
	Reported in HBN version: 2.3.0
3967748	Description: The command `nv show system api connections` does not return any data.
	Workaround: N/A
	Keywords: REST API; nginx
	Reported in HBN version: 2.3.0
4004191	Description: Due to security fixes on BlueField-2, the number of context switches increased by 20% which may result in user applications (e.g., nl2doca) running slower.
	Workaround: N/A
	Keyword: BlueField-2; performance
	Reported in HBN version: 2.3.0
3769309	Description: A ping or other IP connectivity from a locally connected host in vrf-X to an interface IP address on the DPU/HBN itself in vrf-Y will not work, even if VRF route-leaking is enabled between these two VRFs.
	Workaround: N/A
	Keyword: IP
	Reported in HBN version: 2.2.0
3886379	Description: Deleting and re-adding SR-IOV ports might result in some ports in br-hbn bridge going in error state.
	Workaround: If possible, an appropriate number of SR-IOV ports should be chosen at BFB install time. But if a change is made and if the system has this error, the host must undergo a power cycle to resolve the issue.
	Keyword: Bridge; SR-IOV
	Reported in HBN version: 2.2.0
3835295	Description: Traffic entering HBN service on a host PF/VF main-interface and exiting on a sub-interface of the same PF/VF (and vice versa) is not hardware offloaded. Similarly, traffic entering HBN service on one sub-interface and exiting on another sub-interface of the same host PF/VF is also not hardware offloaded.

17.8.2.4 Bug Fixes

The following table lists the known issues which have been fixed for this release of HBN.

17.8.3 HBN Service Deployment

17.8.3.1 HBN Service Requirements

ⓘ Refer to the "HBN Service Release Notes" page for information on the specific hardware and software requirements for HBN.

The following subsections describe specific prerequisites for the BlueField before deploying the DOCA HBN Service.

17.8.3.1.1 Enabling BlueField DPU Mode

HBN requires BlueField to work in either DPU mode or zero-trust mode of operation. Information about configuring BlueField modes of operation can be found under "NVIDIA BlueField Modes of Operation".

17.8.3.1.2 Enabling SFC

HBN requires SFC configuration to be activated on the BlueField before running the HBN service container. SFC allows for additional services/containers to be chained to HBN and provides additional data manipulation capabilities. SFC can be configured in 3 modes:

1. HBN-only mode - In this mode, one OVS bridge is created, `br-hbn`. All HBN-specific ports are added to this bridge. This is the default mode of operation. This mode is configured by setting `ENABLE_BR_HBN=yes` in `bf.cfg` and leaving `ENABLE_BR_SFC` to default.
2. Dual bridge mode - In this mode, 2 OVS bridges are created, `br-hbn` and `bf-sfc`. All HBN-specific ports are added to `bf-sfc` bridge and all these ports are patched into the `br-hbn` bridge. `bf-sfc` can be used to add various custom steering flows to direct traffic across different ports in the bridge. In this mode, both `ENABLE_BR_SFC` and `ENABLE_BR_HBN` are set as to `yes`. `BR_HBN_XXX` parameters are not set and all ports are under `BR_SFC_XXX` variables.
3. Mixed mode - this is similar to the dual bridge model, except that ports can be assigned to either of the bridges (i.e., some ports in `br-hbn` and some in `br-sfc` bridge). In this mode, ports are under `BR_SFC_XXX` and `BR_HBN_XXX`.
   The use of the bridge `br-sfc` allows defining deployment-specific rules before or after HBN pipeline. User can add OpenFlow rules directly to `bf-sfc` bridge. If `ENABLE_BR_SFC_DEFAULT_FLOWS` is set to `yes`, make sure user rules are inserted at higher priority to make it effective.

The following table describes various `bf.cfg` parameters used to configure these modes as well as other parameters which assign ports to various bridges:

Parameter	Description	Mandatory	Default Value	Example
`ENABLE_BR_HBN`	Setting this parameter to `yes` enables the `br-hbn` bridge ⚠ This setting is necessary to work with HBN.	Yes	`no`	`ENABLE_BR_HBN=yes`
`ENABLE_BR_SFC`	Setting this parameter to `yes` enables the `br-sfc` bridge ⓘ This is only needed when the second OVS bridge is required for custom steering flows.	No	`no`	`ENABLE_BR_SFC=no`
`BR_HBN_UPLINKS`	Uplinks added to `br-hbn` directly	No	`p0,p1`	`BR_HBN_UPLINKS="p0,p1"`
`BR_SFC_UPLINKS`	Uplinks added to `br-sfc` directly	No	`""`	`BR_SFC_UPLINKS=""`
`BR_HBN_REPS`	PFs and VFs added to `br-hbn` directly	No	`""`	`BR_HBN_REPS="pf0hpf,pf1hpf,pf0vf0-pf0vf12,pf1vf0-pf1vf4"`
`BR_SFC_REPS`	PFs and VFs added to `br-sfc` directly	No	`""`	`BR_SFC_REPS=""`
`BR_HBN_SFS`	DPU ports added to `br-hbn` directly. These ports are mostly service ports present on the DPU which require using HBN network offload services.	No	`""`	`BR_HBN_SFS=svc1,svc2`
`BR_SFC_SFS`	DPU ports added to `br-sfc` directly	No	`""`	`BR_SFC_SFS=svc1,svc2`
`BR_HBN_SFC_PATCH_PORTS`	Patch ports added to `br-sfc`. These are general purpose ports meant for muxing or demuxing of traffic across various PF/VF ports.	No		`BR_HBN_SFC_PATCH_PORTS=patch1`

Parameter	Description	Mandatory	Default Value	Example
`LINK_PROPAGATION`	Mapping of how link propagation should work. If nothing is provided, each uplink/PF/VF port reflects its status in its corresponding HBN port. For example, the status of p0 is reflected in `p0_if`.	No	Uplink/PF/VF to the corresponding HBN port	`LINK_PROPAGATION=""`
`ENABLE_BR_SFC_DEFAULT_FLOWS`	This parameter is used to provide default connectivity in the `br-sfc` bridge so that each port can send traffic to its corresponding output port	No	`no`	`ENABLE_BR_SFC_DEFAULT_FLOWS=yes`

> More detail about port connectivity in each mode is provided in section "HBN Deployment Configuration".

The following subsections provide additional information about SFC and instructions on enabling it during BlueField DOCA image installation.

17.8.3.1.2.1 Deploying BlueField DOCA Image with SFC from Host

For DOCA image installation on BlueField, the user should follow the instructions under NVIDIA DOCA Installation Guide for Linux with the following extra notes to enable BlueField for HBN setup:

1. Make sure link type is set to ETH under the "Installing Software on Host" section.
2. Add the following parameters to the `bf.cfg` configuration file:
    a. This configuration example is relevant for "HBN-only mode". Set the appropriate variables and values depending on your deployment model.
    b. Enable HBN specific OVS bridge on BlueField Arm by setting ENABLE_BR_HBN=yes.
    c. Define the uplink ports to be used by HBN `BR_HBN_UPLINKS='<port>'`.

    > Must include both ports (i.e., `p0,p1`) for dual-port BlueField devices and only `p0` for single-port BlueField devices.

    d. Include PF and VF ports to be used by HBN. The following example sets both PFs and 8 VFs on each uplink: `BR_HBN_REPS='pf0hpf,pf1hpf,pf0vf0-pf0vf7,pf1vf0-pf1vf7'`.
    e. (Optional) Include SF devices to be created and connected to HBN bridge on the BlueField Arm side by setting `BR_HBN_SFS='pf0dpu1,pf0dpu3'`.

    > If nothing is provided, `pf0dpu1` and `pf0dpu3` are created by default.

While older formats of `bf.cfg` still work in this release, they will be deprecated over the next 2 releases. So, its advisable to move to the new format to avoid any upgrade issues in future releases. The following is an example for the old `bf.cfg` format:

```
ENABLE_SFC_HBN=yes
NUM_VFs_PHYS_PORT0=12 # <num VFs supported by HBN on Physical Port 0> (valid range: 0-127) Default 14
NUM_VFs_PHYS_PORT1=2  # <num VFs supported by HBN on Physical Port 1> (valid range: 0-127) Default 0
```

3. Then run:

   ```
   bfb-install -c bf.cfg -r rshim0 -b <BFB-image>
   ```

4. Once SFC deployment is done, it creates 3 set of files:
   - `/etc/mellanox/hbn.conf` - this file can be used to redeploy SFC without the need to pass through `bf.cfg` again to modify interface mapping
   - `/etc/mellanox/sfc.conf` - this file provides a view of how various ports are connected in different bridges
   - `/etc/mellanox/mlnx-sf.conf` - this file includes all the HBN ports to be created and corresponding commands to create the port

17.8.3.1.2.2 Deploying BlueField DOCA Image with SFC Using PXE Boot

To enable HBN SFC using a PXE installation environment with BFB content, use the following configuration for PXE:

```
bfnet=<IFNAME>:<IPADDR>:<NETMASK> or <IFNAME>:dhcp
bfks=<URL of the kickstart script>
```

The kickstart script (bash) should include the following lines:

```
cat >> /etc/bf.cfg << EOF

ENABLE_BR_HBN=yes
BR_HBN_UPLINKS='p0,p1'
BR_HBN_REPS='pf0hpf,pf1hpf,pf0vf0-pf0vf7,pf1vf0-pf1vf7'
BR_HBN_SFS='pf0dpu1,pf0dpu3'
EOF
```

The `/etc/bf.cfg` generated above is sourced by the BFB `install.sh` script.

It is recommended to verify the accuracy of the BlueField's clock post-installation. This can be done using the following command:

```
$ date
```

Please refer to the known issues listed in the "NVIDIA DOCA Release Notes" for more information.

17.8.3.1.2.3 Redeploying SFC from BlueField

Redeploying SFC from BlueField can be done after the DPU has already been deployed using `bf.cfg` and either port mapping or bridge configuration needs to be change.

To redeploy SFC from BlueField:

1. Edit `/etc/mellanox/hbn.conf` by adding or removing entries in each segment as necessary.
2. Rerun the SFC install script:

   ```
   /opt/mellanox/sfc-hbn/install.sh -c -r
   ```

   This generates a new set of `sfc.conf` and `mlnx-sf.conf` and reloads the DPU.
   Configuration and reload can be split into 2 steps by removing the `-r` option and rebooting BlueField post configuration.

After the BlueField reloads, the command `ovs-vsctl show` should show all the new ports and bridges configured in OVS.

17.8.3.1.2.4 Deploying HBN with Other Services

When the HBN container is deployed by itself, BlueField Arm is configured with 3k huge pages. If it is deployed with other services, the actual number of huge-pages must be adjusted based on the requirements of those services. For example, SNAP or NVMesh may need approximately 1k to 5k huge pages. So, if HBN is running with either of these services on the same BlueField, the total number of hugepages must be set to the sum of the hugepage requirement of all the services.

For example, if NVMesh needs 3k hugepages, 6k total hugepages must be set when running with HBN. To do that, add the following parameters to the `bf.cfg` configuration file alongside other desired parameters.

```
HUGEPAGE_COUNT=6144
```

> This should be performed only on a BlueField-3 running with 32G of memory. Doing this on 16G system may cause memory issues for various applications on BlueField Arm.
> Also, HBN with other services is qualified only for 16 VFs.

17.8.3.2 Launching HBN Service

17.8.3.2.1 HBN Service Container Deployment

HBN service is available on NGC, NVIDIA's container catalog. For information about the deployment of DOCA containers on top of the BlueField, refer to NVIDIA DOCA Container Deployment Guide.

17.8.3.2.1.1 Downloading DOCA Container Resource File

Pull the latest DOCA container resource as a `*.zip` file from NGC and extract it to the `<resource>` folder (`doca_container_configs_2.7.0v1` in this example):

```
wget https://api.ngc.nvidia.com/v2/resources/nvidia/doca/doca_container_configs/versions/2.7.0v1/zip -O
doca_container_configs_2.7.0v1.zip
unzip -o doca_container_configs_2.7.0v1.zip -d doca_container_configs_2.7.0v1
```

17.8.3.2.1.2 Running HBN Preparation Script

The HBN script (`hbn-dpu-setup.sh`) performs the following steps on BlueField Arm which are required for HBN service to run:

1. Sets the BlueField to DPU mode if needed.
2. Enables IPv4/IPv6 kernel forwarding.
3. Sets up interface MTU if needed.
4. Sets up mount points between BlueField Arm and HBN container for logs and configuration persistency.
5. Sets up various paths as needed by supervisord and other services inside container.
6. Enables the REST API access if needed.
7. Creates or updates credentials

The script is located in `<resource>/scripts/doca_hbn/<hbn_version>/` folder, which is downloaded as part of the DOCA Container Resource.

> **Optional**
>
> To achieve the desired configuration on HBN's first boot, before running preparation script, users can update default NVUE or flat (network interfaces and FRR) configuration files, which are located in `<resource>/scripts/doca_hbn/<hbn_version>/` .
>
> - For NVUE-based configuration:
>   - `etc/nvue.d/startup.yaml`
> - For flat-files based configuration:
>   - `etc/network/interfaces`
>   - `etc/frr/frr.conf`
>   - `etc/frr/daemons`

Run the following commands to execute the `hbn-dpu-setup.sh` script:

```
cd <resource>/scripts/doca_hbn/2.3.0/
chmod +x hbn-dpu-setup.sh
sudo ./hbn-dpu-setup.sh
```

The following is the help menu for the `hbn-dpu-setup.sh` script:

```
./hbn-dpu-setup.sh -h
usage: hbn-dpu-setup.sh
hbn-dpu-setup.sh -m|--mtu <MTU> Use <MTU> bytes for all HBN interfaces (default 9216)
hbn-dpu-setup.sh -u|--username <username> User creation
hbn-dpu-setup.sh -p|--password <password> Password for --username <username>
hbn-dpu-setup.sh -e|--enable-rest-api-access Enable REST API from external IPs
hbn-dpu-setup.sh -h|--help
```

Enabling REST API Access

To enable the REST API access:

1. Change the default password for the `nvidia` username:

```
./hbn-dpu-setup.sh -u nvidia -p <new-password>
```

2. Enable REST API:

   ```
   ./hbn-dpu-setup.sh --enable-rest-api-access
   ```

3. Perform BlueField system-level reset.

17.8.3.2.1.3 Spawning HBN Container

HBN container `.yaml` configuration is called `doca_hbn.yaml` and it is located in `<resource>/configs/<doca_version>/` directory. To spawn the HBN container, simply copy the `doca_hbn.yaml` file to the `/etc/kubelet.d` directory:

```
cd <resource>/configs/2.8.0/
sudo cp doca_hbn.yaml /etc/kubelet.d/
```

Kubelet automatically pulls the container image from NGC and spawns a pod executing the container. The DOCA HBN Service starts executing right away.

17.8.3.2.1.4 Verifying HBN Container is Running

To inspect the HBN container and verify if it is running correctly:

1. Check HBN pod and container status and logs:
   a. Examine the currently active pods and their IDs (it may take up to 20 seconds for the pod to start):

      ```
      sudo crictl pods
      ```

   b. View currently active containers and their IDs:

      ```
      sudo crictl ps
      ```

   c. Examine logs of a given container:

      ```
      sudo crictl logs
      ```

   d. Examine kubelet logs if something did not work as expected:

      ```
      sudo journalctl -u kubelet@mgmt
      ```

2. Log into the HBN container:

   ```
   sudo crictl exec -it $(crictl ps | grep hbn | awk '{print $1;}') bash
   ```

3. While logged into HBN container, verify that the `frr`, `nl2doca`, and `neighmgr` services are running:

   ```
   (hbn-container)$ supervisorctl status frr
   (hbn-container)$ supervisorctl status nl2doca
   (hbn-container)$ supervisorctl status neighmgr
   ```

4. Users may also examine various logs under `/var/log` inside the HBN container.

17.8.3.2.2 HBN Deployment Configuration

The HBN service comes with four types of configurable interfaces:

- Two uplinks (`p0_if` , `p1_if`)
- Two PF port representors (`pf0hpf_if` , `pf1hpf_if`)
- User-defined number of VFs (i.e., `pf0vf0_if` , `pf0vf1_if` , …, `pf1vf0_if` , `pf1vf1_if` , …)
- DPU interfaces to connect to services running on BlueField, outside of the HBN container (`pf0dpu1_if` and `pf0dpu3_if`)

The `*_if` suffix indicates that these are sub-functions and are different from the physical uplinks (i.e., PFs, VFs). They can be viewed as virtual interfaces from a virtualized BlueField.

Each of these interfaces is connected outside the HBN container to the corresponding physical interface, see section "Service Function Chaining" (SFC) for more details.

The HBN container runs as an isolated namespace and does not see any interfaces outside the container (`oob_net0` , real uplinks and PFs, `*_if_r` representors).

17.8.3.2.2.1 HBN-only Deployment Configuration

This is the default deployment model of HBN. In this model, only one OVS bridge is created.

The following is a sample `bf.cfg` and the resulting OVS and port configurations:

- Sample `bf.cfg` :

bf.cfg

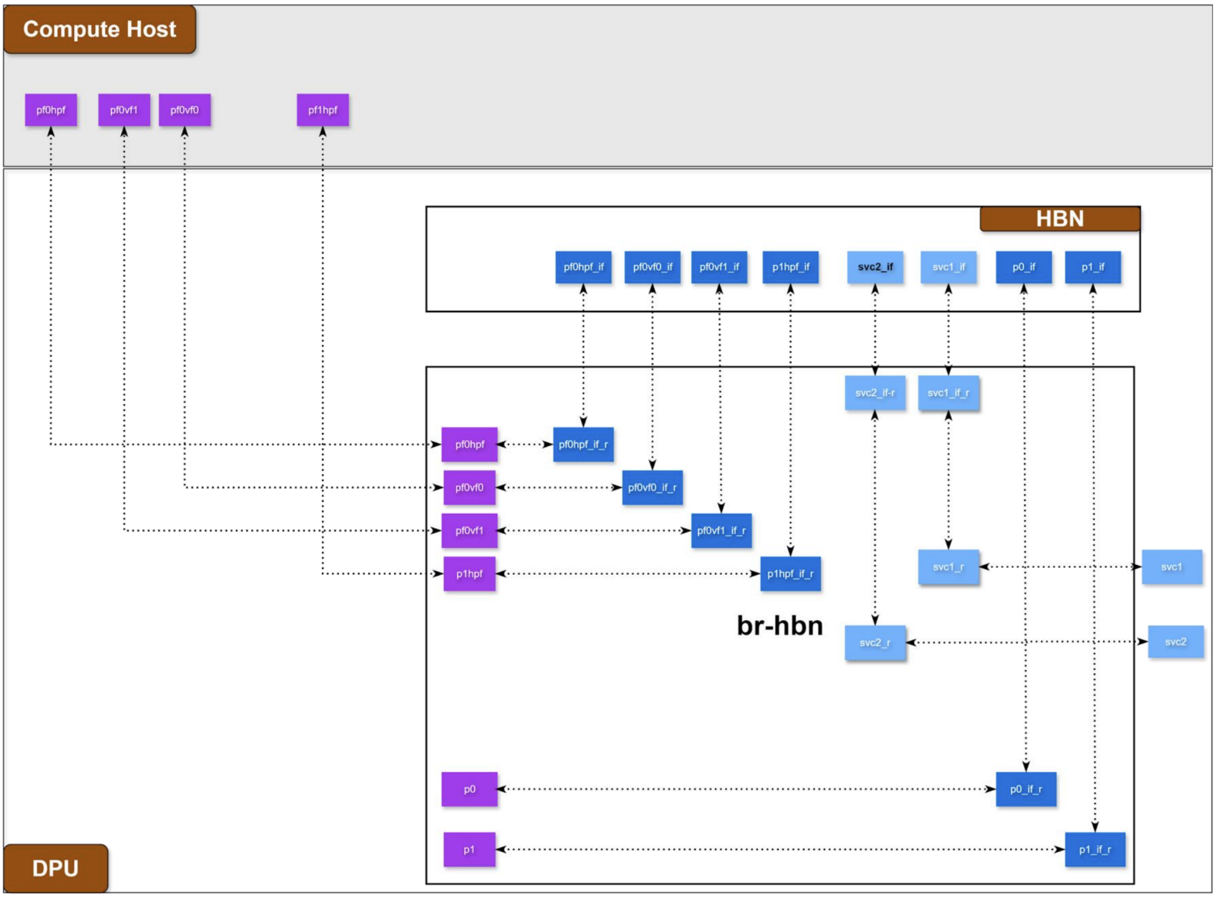
```
BR_HBN_UPLINKS="p0,p1"
BR_HBN_REPS="pf0hpf,pf1hpf,pf0vf0-pf0vf12,pf1vf0-pf1vf1"
BR_HBN_SFS="svc1,svc2"
```

- Generated `hbn.conf` :

Generated hbn.conf

```
[BR_HBN_UPLINKS]
p0
p1
[BR_HBN_REPS]
pf0hpf
pf0vf0
pf0vf1
pf0vf2
pf0vf3
pf0vf4
pf0vf5
pf0vf6
pf0vf7
pf0vf8
pf0vf9
pf0vf10
pf0vf11
pf0vf12
pf1hpf
pf1vf0
pf1vf1

[BR_HBN_SFS]
svc1
```

```
svc2

[BR_SFC_UPLINKS]


[BR_SFC_REPS]

[BR_SFC_SFS]

[BR_HBN_SFC_PATCH_PORTS]


[LINK_PROPAGATION]
p0:p0_if_r
p1:p1_if_r
pf0hpf:pf0hpf_if_r
pf0vf0:pf0vf0_if_r
pf0vf1:pf0vf1_if_r
pf0vf2:pf0vf2_if_r
pf0vf3:pf0vf3_if_r
pf0vf4:pf0vf4_if_r
pf0vf5:pf0vf5_if_r
pf0vf6:pf0vf6_if_r
pf0vf7:pf0vf7_if_r
pf0vf8:pf0vf8_if_r
pf0vf9:pf0vf9_if_r
pf0vf10:pf0vf10_if_r
pf0vf11:pf0vf11_if_r
pf0vf12:pf0vf12_if_r
pf1hpf:pf1hpf_if_r
pf1vf0:pf1vf0_if_r
pf1vf1:pf1vf1_if_r
svc1_r:svc1_if_r
svc2_r:svc2_if_r

[ENABLE_BR_SFC]

[ENABLE_BR_SFC_DEFAULT_FLOWS]
```



17.8.3.2.2.2 Dual Bridge HBN Deployment Configuration

The following is a sample `bf.cfg` and the resulting OVS and port configurations:

- Sample `bf.cfg`:

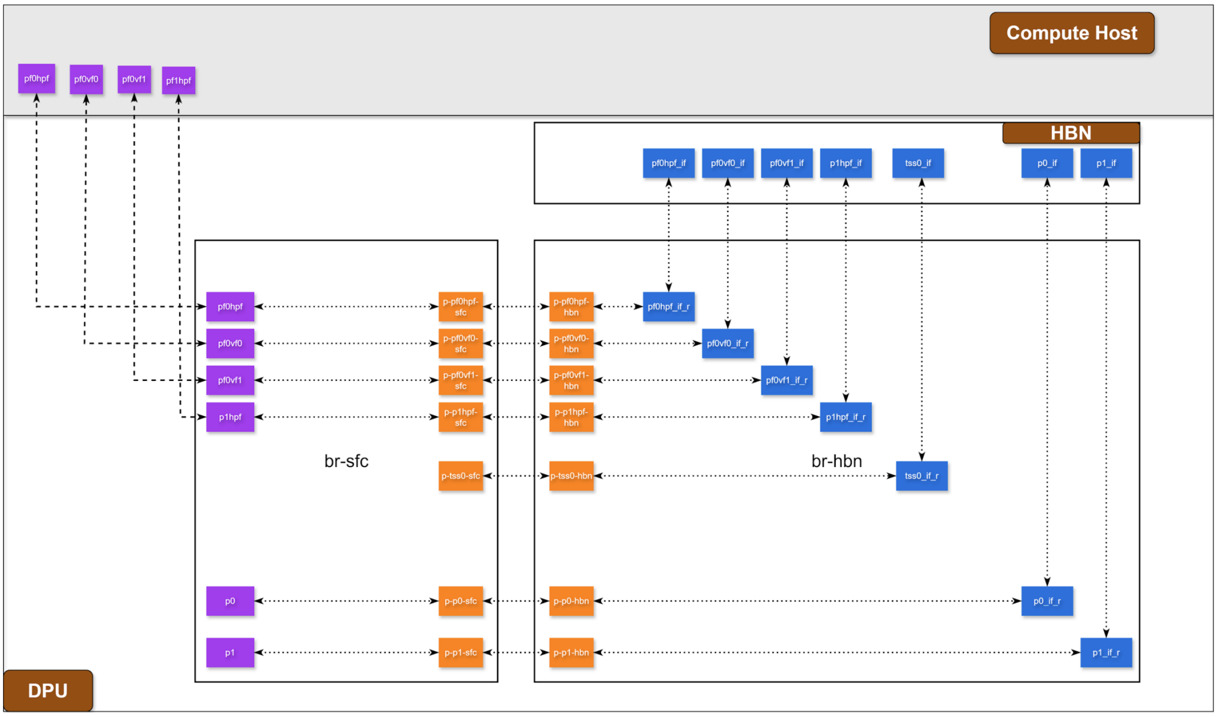
bf.cfg

```
BR_HBN_UPLINKS=""
BR_SFC_UPLINKS="p0,p1"
BR_HBN_REPS=""
BR_SFC_REPS="pf0hpf,pf1hpf,pf0vf0-pf0vf1,pf1vf0-pf1vf1"
BR_HBN_SFS=""
BR_SFC_SFS=""
BR_HBN_SFC_PATCH_PORTS="tss0"
LINK_PROPAGATION="pf0hpf:tss0"
ENABLE_BR_SFC=yes
ENABLE_BR_SFC_DEFAULT_FLOWS=yes
```

- Generated `hbn.conf`:

Generated hbn.conf

```
[BR_HBN_UPLINKS]


[BR_HBN_REPS]


[BR_HBN_SFS]


[BR_SFC_UPLINKS]
p0
p1

[BR_SFC_REPS]
pf0hpf
pf0vf0
pf0vf1
pf1hpf
pf1vf0
pf1vf1

[BR_SFC_SFS]


[BR_HBN_SFC_PATCH_PORTS]
tss0

[LINK_PROPAGATION]
pf0hpf:tss0
p0:p0_if_r
p1:p1_if_r
pf0vf0:pf0vf0_if_r
pf0vf1:pf0vf1_if_r
pf1hpf:pf1hpf_if_r
pf1vf0:pf1vf0_if_r
pf1vf1:pf1vf1_if_r

[ENABLE_BR_SFC]
yes

[ENABLE_BR_SFC_DEFAULT_FLOWS]
yes
```



17.8.3.2.2.3 Mixed Mode HBN Deployment Configuration

The following is a sample `bf.cfg` and the resulting OVS and port configurations:

- Sample `bf.cfg`:

bf.cfg

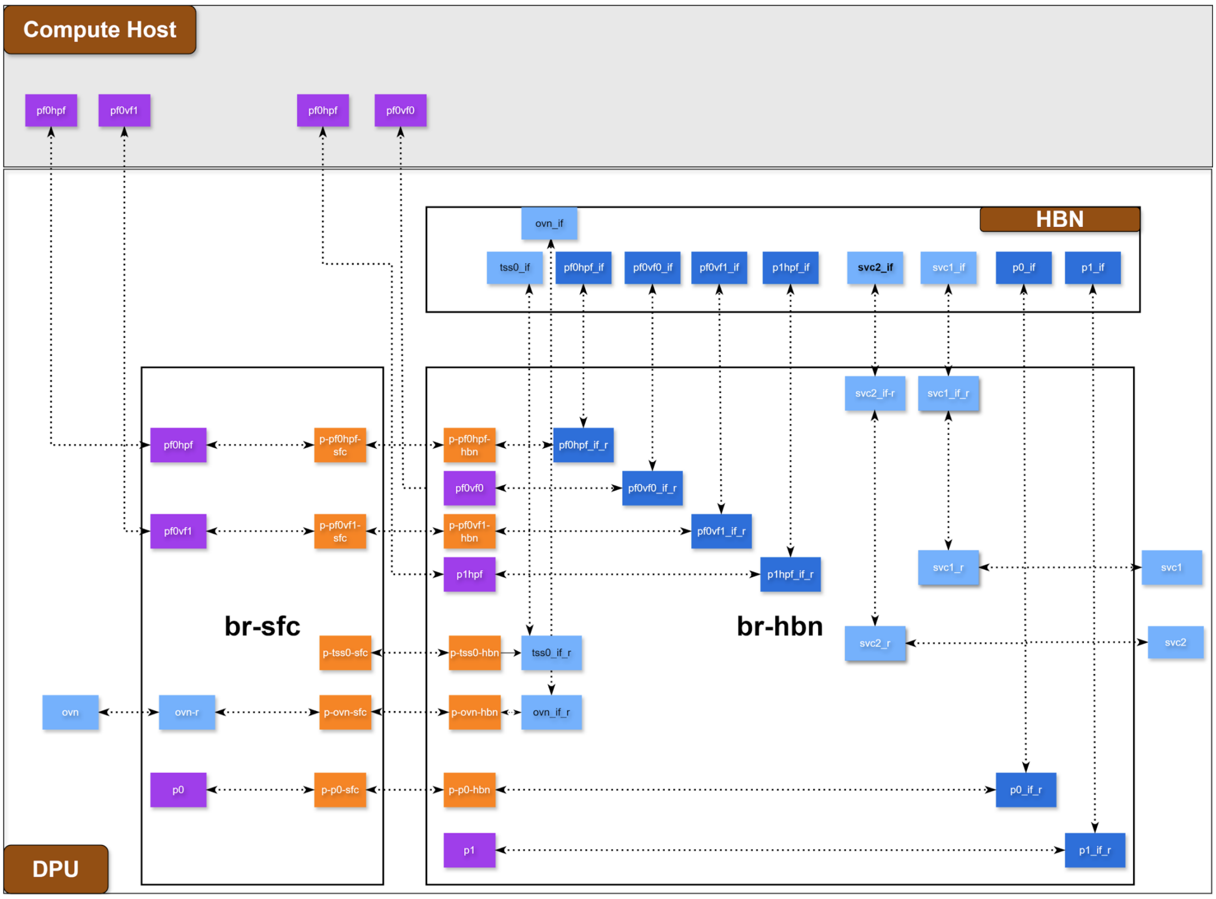
```
BR_HBN_UPLINKS="p1"
BR_SFC_UPLINKS="p0"
BR_HBN_REPS="pf1hpf,pf0vf0"
BR_SFC_REPS="pf0hpf,pf0vf1"
BR_HBN_SFS="svc1,svc2"
BR_SFC_SFS="ovn"
BR_HBN_SFC_PATCH_PORTS="tss0"
LINK_PROPAGATION="pf0hpf:tss0"
ENABLE_BR_SFC=yes
ENABLE_BR_SFC_DEFAULT_FLOWS=yes
```

- Generated `hbn.conf`:

Generated hbn.conf

```
[BR_HBN_UPLINKS]
p1

[BR_HBN_REPS]
pf0vf0
pf1hpf

[BR_HBN_SFS]
svc1
svc2

[BR_SFC_UPLINKS]
p0

[BR_SFC_REPS]
pf0hpf
pf0vf1

[BR_SFC_SFS]
```

```
ovn

[BR_HBN_SFC_PATCH_PORTS]
tss0

[LINK_PROPAGATION]
pf0hpf:tss0
p1:p1_if_r
p0:p0_if_r
pf0vf0:pf0vf0_if_r
pf0hpf:pf0hpf_if_r
pf0vf1:pf0vf1_if_r
pf1hpf:pf1hpf_if_r
svc1_r:svc1_if_r
svc2_r:svc2_if_r
ovn_r:ovn_if_r

[ENABLE_BR_SFC]
yes

[ENABLE_BR_SFC_DEFAULT_FLOWS]
yes
```



17.8.3.2.3 HBN Deployment Considerations

17.8.3.2.3.1 SF Interface State Tracking

When HBN is deployed with SFC, the interface state of the following network devices is propagated to their corresponding SFs:

- Uplinks - `p0`, `p1`
- PFs - `pf0hpf`, `pf1hpf`
- VFs - `pf0vfX`, `pf1vfX` where `X` is the VF number

For example, if the p0 uplink cable gets disconnected:

- `p0` transitions to DOWN state with `NO-CARRIER` (default behavior on Linux); and
- `p0` state is propagated to `p0_if` whose state also becomes DOWN with NO-CARRIER

After `p0` connection is reestablished:

- `p0` transitions to UP state; and
- `p0` state is propagated to `p0_if` whose state becomes UP

Interface state propagation only happens in the uplink/PF/VF-to-SF direction.

A daemon called `sfc-state-propagation` runs on BlueField, outside of the HBN container, to sync the state. The daemon listens to netlink notifications for interfaces and transfers the state to SFs.

17.8.3.2.3.2 SF Interface MTU

In the HBN container, all the interfaces MTU are set to 9216 by default. MTU of specific interfaces can be overwritten using flat-files configuration or NVUE.

On BlueField side (i.e., outside of the HBN container), the MTU of the uplinks, PFs and VFs interfaces are also set to 9216. This can be changed by modifying `/etc/systemd/network/30-hbn-mtu.network` or by adding a new configuration file in the `/etc/systemd/network` for specific directories.

To reload this configuration, run:

```
systemctl restart systemd-networkd
```

17.8.3.2.3.3 Connecting to DOCA Services to HBN on BlueField Arm

There are various SF ports (named `pf0dpuX_if`, where X is [0..n]) on BlueField Arm, which can be used to run any services on BlueField and use HBN to provide network connectivity. These ports can have a flexible naming convention based on the service name. For example, to support OVN service, it can create an interface named `ovn` which can be used by the OVN service running on the BlueField Arm, and it will get a corresponding HBN port named `ovn_if`. These interfaces are created using either `BR_SFC_SFS` or `BR_HBN_SFS` based on which the bridge needs the service interface and mode of service deployment.

Traffic between BlueField and the outside world is hardware-accelerated when the HBN side port is an L3 interface or access-port using switch virtual interface (SVI). So, it is treated the same way as PF or VF ports from a traffic handling standpoint.

> ⓘ There are 2 SF port pairs created by default on BlueField Arm side so there can be 2 separate DOCA services running at same time.

17.8.3.2.3.4 Disabling BlueField Uplinks

The uplink ports must be always kept administratively up for proper operation of HBN. Otherwise, the NVIDIA® ConnectX® firmware would bring down the corresponding representor port which would cause data forwarding to stop.

> Change in operational status of uplink (e.g., carrier down) would result in traffic being switched to the other uplink.

When using ECMP failover on the two uplink SFs, locally disabling one uplink does not result in traffic switching to the second uplink. Disabling local link in this case means to set one uplink admin DOWN directly on BlueField.

To test ECMP failover scenarios correctly, the uplink must be disabled from its remote counterpart (i.e., execute admin DOWN on the remote system's link which is connected to the uplink).

17.8.3.2.3.5 HBN NVUE User Credentials

The preconfigured default user credentials are as follows:

Username	`nvidia`
Password	`nvidia`

NVUE user credentials can be added post installation:

1. This can be done by specifying additional `--username` and `--password` to the HBN startup script (refer to "Running HBN Preparation Script"). For example:

```
sudo ./hbn-dpu-setup.sh -u newuser -p newpassword
```

2. After executing this script, respawn the container or start the `decrypt-user-add` script inside running HBN container:

```
supervisorctl start decrypt-user-add
decrypt-user-add: started
```

The script creates a new user in the HBN container:

```
cat /etc/passwd | grep newuser
newuser:x:1001:1001::/home/newuser:/bin/bash
```

17.8.3.2.3.6 HBN NVUE Interface Classification

Interface	Interface Type	NVUE Type
`p0_if`	Uplink representor	swp
`p1_if`	Uplink representor	swp
`lo`	Loopback	loopback
`pf0hpf_if`	Host representor	swp
`pf1hpf_if`	Host representor	swp
`pf0vfx_if` (where `x` is 0-255)	VF representor	swp
`pf1vfx_if` (where `x` is 0-255)	VF representor	swp

17.8.3.2.3.7 HBN Files Persistence

The following directories are mounted from BlueField Arm to the HBN container namespace and are persistent across HBN service restarts and BlueField reboots:

	BlueField Arm Mount Point	HBN Container Mount Point
Configuration file mount points	`/var/lib/hbn/etc/network/`	`/etc/network/`
	`/var/lib/hbn/etc/frr/`	`/etc/frr/`
	`/var/lib/hbn/etc/nvue.d/`	`/etc/nvue.d/`
	`/var/lib/hbn/etc/supervisor/conf.d/`	`/etc/supervisor/conf.d/`
	`/var/lib/hbn/var/lib/nvue/`	`/var/lib/nvue/`
Support and log file mount points	`/var/lib/hbn/var/support/`	`/var/support/`
	`/var/log/doca/hbn/`	`/var/log/hbn/`

17.8.3.2.3.8 SR-IOV Support in HBN

Creating SR-IOV VFs on Host

The first step to use SR-IOV is to create Virtual Functions (VFs) on the host server.

VFs can be created using the following command:

```
sudo echo N > /sys/class/net/<host-rep>/device/sriov_numvfs
```

Where:

- `<host-rep>` is one of the two host representors (e.g., `ens1f0` or `ens1f1`)
- 0≤ `N` ≤16 is the desired total number of VFs
    - Set `N` =0 to delete all the VFs on 0≤N≤16
    - `N` =16 is the maximum number of VFs supported on HBN across all representors

Automatic Creation of VF Representors and SF Devices on BlueField

VFs created on the host must have corresponding VF representor devices and SF devices for HBN on BlueField side. For example:

- `ens1f0vf0` is the first SR-IOV VF device from the first host representor; this interface is created on the host server
- `pf0vf0` is the corresponding VF representor device to `ens1f0vf0` ; this device is present on the BlueField Arm side and automatically created at the same time as `ens1f0vf0` is created by the user on the host side
- `pf0vf0_if` is the corresponding SF device for `pf0vf0` which is used to connect the VF to HBN pipeline

The creation of the SF device for VFs is done ahead of time when provisioning the BlueField and installing the DOCA image on it, see section "Enabling SFC" to see how to select how many SFs to create ahead of time.

The SF devices for VFs (i.e., `pfXvfY`) are pre-mapped to work with the corresponding VF representors when these are created with the command from the previous step.

17.8.3.2.3.9 Management VRF

Two management VRFs are automatically configured for HBN when BlueField is deployed with SFC:

- The first management VRF is outside the HBN container on BlueField. This VRF provides separation between out-of-band (OOB) traffic (via `oob_net0` or `tmfifo_net0`) and data-plane traffic via uplinks and PFs.
- The second management VRF is inside the HBN container and provides similar separation. The OOB traffic (via `eth0`) is isolated from the traffic via the `*_if` interfaces.

MGMT VRF on BlueField Arm

The management (mgmt) VRF is enabled by default when the BlueField is deployed with SFC (see section "Enabling SFC"). The mgmt VRF provides separation between the OOB management network and the in-band data plane network.

The uplinks and PFs/VFs use the default routing table while the `oob_net0` (OOB Ethernet port) and the `tmifo_net0` netdevices use the mgmt VRF to route their packets.

When logging in either via SSH or the console, the shell is by default in mgmt VRF context. This is indicated by a mgmt added to the shell prompt:

```
root@bf2:mgmt:/home/ubuntu#
root@bf2:mgmt:/home/ubuntu# ip vrf identify
mgmt.
```

When logging into the HBN container with `crictl`, the HBN shell will be in the default VRF. Users must switch to MGMT VRF manually if OOB access is required. Use `ip vrf exec` to do so.

```
root@bf2:mgmt:/home/ubuntu# ip vrf exec mgmt bash
```

The user must run `ip vrf exec mgmt` to perform operations requiring OOB access (e.g., apt-get update).

Network devices belonging to the mgmt VRF can be listed with the `vrf` utility:

```
root@bf2:mgmt:/home/ubuntu# vrf link list

VRF: mgmt
--------------------
tmfifo_net0      UP             00:1a:ca:ff:ff:03 <BROADCAST,MULTICAST,UP,LOWER_UP>
oob_net0         UP             08:c0:eb:c0:5a:32 <BROADCAST,MULTICAST,UP,LOWER_UP>

root@bf2:mgmt:/home/ubuntu# vrf help
vrf <OPTS>

VRF domains:
    vrf list

Links associated with VRF domains:
    vrf link list [<vrf-name>]
```

```
Tasks and VRF domain asociation:
    vrf task exec <vrf-name> <command>
    vrf task list [<vrf-name>]
    vrf task identify <pid>

    NOTE: This command affects only AF_INET and AF_INET6 sockets opened by the
          command that gets exec'ed. Specifically, it has *no* impact on netlink
          sockets (e.g., ip command).
```

To show the routing table for the default VRF, run:

```
root@bf2:mgmt:/home/ubuntu# ip route show
```

To show the routing table for the mgmt VRF, run:

```
root@bf2:mgmt:/home/ubuntu# ip route show vrf mgmt
```

MGMT VRF Inside HBN Container

Inside the HBN container, a separate mgmt VRF is present. Similar commands as those listed under section "MGMT VRF on BlueField Arm" can be used to query management routes.

The `*_if` interfaces use the default routing table while the `eth0` (OOB) uses the mgmt VRF to route out-of-band packets out of the container. The OOB traffic gets NATed through the `oob_net0` interface on BlueField Arm, ultimately using the BlueField OOB's IP address.

When logging into the HBN container via `crictl`, the shell enters the default VRF context by default. Switching to the mgmt VRF can be done using the command `ip vrf exec mgmt <cmd>`.

Existing Services in MGMT VRF on BlueField Arm

On the BlueField Arm, outside the HBN container, a set of existing services run in the mgmt VRF context as they need OOB network access:

- containerd
- kubelet
- ssh
- docker

These services can be restarted and queried for their status using the command `systemctl` while adding `@mgmt` to the original service name. For example:

- To restart containerd:

  ```
  root@bf2:mgmt:/home/ubuntu# systemctl restart containerd@mgmt
  ```

- To query containerd status:

  ```
  root@bf2:mgmt:/home/ubuntu# systemctl status containerd@mgmt
  ```

> ⚠ The original version of these services (without `@mgmt`) are not used and must not be started.

Running New Service in MGMT VRF on BlueField Arm

If a service needs OOB access to run, it can be added to the set of services running in mgmt VRF context. Adding such a service is only possible on the BlueField Arm (i.e., outside the HBN container).

To add a service to the set of mgmt VRF services:

1. Add it to `/etc/vrf/systemd.conf` (if it is not present already). For example, NTP is already listed in this file.
2. Run the following:

   ```
   root@bf2:mgmt:/home/ubuntu# systemctl daemon-reload
   ```

3. Stop and disable to the non-VRF version of the service to be able to start the mgmt VRF one:

   ```
   root@bf2:mgmt:/home/ubuntu# systemctl stop ntp
   root@bf2:mgmt:/home/ubuntu# systemctl disable ntp
   root@bf2:mgmt:/home/ubuntu# systemctl enable ntp@mgmt
   root@bf2:mgmt:/home/ubuntu# systemctl start ntp@mgmt
   ```

17.8.4 HBN Service Configuration

To start configuring HBN, log into the HBN container:

```
sudo crictl exec -it $(crictl ps | grep hbn | awk '{print $1;}') bash
```

17.8.4.1 General Network Configuration

17.8.4.1.1 Flat Files Configuration

Add network interfaces and FRR configuration files to HBN to achieve the desired configuration:

- `/etc/network/interfaces`

  > ⚠ Refer to NVIDIA® Cumulus® Linux documentation for more information.

- `/etc/frr/frr.conf`; `/etc/frr/daemons`

  > ⚠ Refer to NVIDIA® Cumulus® Linux documentation for more information.

17.8.4.2 NVUE Configuration

This section assumes familiarity with NVIDIA user experience (NVUE) Cumulus Linux documentation. The following subsections, only expand on HBN-specific aspects of NVUE.

17.8.4.2.1 NVUE Service

HBN installs NVUE by default and enables NVUE service at boot.

17.8.4.2.2 NVUE REST API

HBN enables the REST API by default but with localhost access. The user cannot access REST API from the outside by default.

To enable REST API access, please refer to section "Enable REST API Access".

Users may run the cURL commands from the command line. Use the default HBN username, `nvidia`, and password which must be updated when enabling the REST API using the HBN preparation script.

To change the default password of the `nvidia` user or add additional users for NVUE access, refer to section "HBN NVUE User Credentials".

REST API example:

```
curl -u 'nvidia:nvidia' --insecure https://<mgmt_ip>:8765/nvue_v1/vrf/default/router/bgp
{
  "configured-neighbors": 2,
  "established-neighbors": 2,
  "router-id": "10.10.10.201"
}
```

17.8.4.2.2.1 NVUE REST API Management Through CLI

- To enable the REST API service, run:

  ```
  nv set system api state enabled
  ```

- To disable the REST API service:

  ```
  nv set system api state disabled
  ```

- To bind the REST API service to a specific address:

  ```
  nv set system api listening-address <localhost|ipv4|ipv6|0.0.0.0>
  ```

For information about using the NVUE REST API, refer to the NVUE API documentation.

17.8.4.2.3 NVUE CLI

For information about using the NVUE CLI, refer to the NVUE CLI documentation

17.8.4.2.4 NVUE Startup Configuration File

When the network configuration is saved using NVUE, HBN writes the configuration to the `/etc/nvue.d/startup.yaml` file.

Startup configuration is applied by following the supervisor daemon at boot time. `nvued-startup` will appear in `EXITED` state after applying the startup configuration.

```
# supervisorctl status nvued-startup
nvued-startup                    EXITED    Apr 17 10:04 AM
```

`nv config apply startup` applies the yaml configuration saved at `/etc/nvue.d/`.

`nv config save` saves the running configuration to `/etc/nvue.d/startup.yaml`.

17.8.4.3 HBN Configuration Examples

17.8.4.3.1 HBN Default Configuration

After a fresh HBN installation, the default `/etc/network/interfaces` file would contain only the declaration of the two uplink SFs and a loopback interface.

```
source /etc/network/interfaces.d/*.intf

auto lo
iface lo inet loopback

auto p0_if
iface p0_if

auto p1_if
iface p1_if
```

FRR configuration files would also be present under `/etc/frr/` but no configuration would be enabled.

17.8.4.3.2 Layer-3 Routing

17.8.4.3.2.1 Native Routing with BGP and ECMP

HBN supports unicast routing with BGP and ECMP for IPv4 and IPv6 traffic. ECMP is achieved by distributing traffic using hash calculation based on the source IP, destination IP, and protocol type of the IP header.

For TCP and UDP packets, it also includes source port and destination port.

ECMP Example

ECMP is implemented any time routes have multiple paths over uplinks or host ports. For example, 20.20.20.0/24 has 2 paths using both uplinks, so a path is selected based on a hash of the IP headers.

```
20.20.20.0/24 proto bgp metric 20
    nexthop via 169.254.0.1 dev p0_if weight 1 onlink <<<<< via uplink p0_if
    nexthop via 169.254.0.1 dev p1_if weight 1 onlink <<<<< via uplink p1_if
```

HBN supports up to 16 paths for ECMP.

Sample NVUE Configuration for Native Routing

```
nv set interface lo ip address 10.10.10.1/32
nv set interface lo ip address 2010:10:10::1/128
nv set interface vlan100 type svi
nv set interface vlan100 vlan 100
nv set interface vlan100 base-interface br_default
nv set interface vlan100 ip address 2030:30:30::1/64
nv set interface vlan100 ip address 30.30.30.1/24
nv set bridge domain br_default vlan 100
nv set interface pf0hpf_if,pf1hpf_if bridge domain br_default access 100
nv set vrf default router bgp router-id 10.10.10.1
nv set vrf default router bgp autonomous-system 65501
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf default router bgp neighbor p0_if remote-as external
nv set vrf default router bgp neighbor p0_if type unnumbered
nv set vrf default router bgp neighbor p0_if address-family ipv4-unicast enable on
nv set vrf default router bgp neighbor p0_if address-family ipv6-unicast enable on
nv set vrf default router bgp neighbor p1_if remote-as external
nv set vrf default router bgp neighbor p1_if type unnumbered
nv set vrf default router bgp neighbor p1_if address-family ipv4-unicast enable on
nv set vrf default router bgp neighbor p1_if address-family ipv6-unicast enable on
```

Sample Flat Files Configuration for Native Routing

Example `/etc/network/interfaces` configuration:

```
auto lo
iface lo inet loopback
    address 10.10.10.1/32
    address 2010:10:10::1/128

auto p0_if
iface p0_if

auto p1_if
iface p1_if

auto pf0hpf_if
iface pf0hpf_if
    bridge-access 100

auto pf1hpf_if
iface pf1hpf_if
    bridge-access 100

auto vlan100
iface vlan100
    address 2030:30:30::1/64
    address 30.30.30.1/24
    vlan-raw-device br_default
    vlan-id 100

auto br_default
iface br_default
    bridge-ports pf0hpf_if pf1hpf_if
    bridge-vlan-aware yes
    bridge-vids 100
    bridge-pvid 1
```

Example `/etc/frr/daemons` configuration:

```
bgpd=yes
vtysh_enable=yes


FRR Config file @ /etc/frr/frr.conf -
!
frr version 7.5+cl5.3.0u0
frr defaults datacenter
hostname BLUEFIELD2
log syslog informational
no zebra nexthop kernel enable
!
router bgp 65501
 bgp router-id 10.10.10.1
 bgp bestpath as-path multipath-relax
 neighbor p0_if interface remote-as external
 neighbor p0_if advertisement-interval 0
 neighbor p0_if timers 3 9
 neighbor p0_if timers connect 10
 neighbor p1_if interface remote-as external
 neighbor p1_if advertisement-interval 0
 neighbor p1_if timers 3 9
 neighbor p1_if timers connect 10
 !
 address-family ipv4 unicast
  redistribute connected
  maximum-paths 64
  maximum-paths ibgp 64
 exit-address-family
 !
 address-family ipv6 unicast
```

```
  redistribute connected
  neighbor p0_if activate
  neighbor p1_if activate
  maximum-paths 64
  maximum-paths ibgp 64
 exit-address-family
!
line vty
!
end
```

Direct Routing on Host-facing Interfaces

Host-facing interfaces (PFs and VFs) are not restricted to be part of the bridge for routing. HBN supports L3-only configuration with direct routing on host-facing PFs and VFs.

Sample NVUE Configuration

```
nv set interface pf0hpf_if ip address 30.30.11.1/24
nv set interface pf0hpf_if ip address 2030:30:11::1/64
nv set interface pf0vf0_if ip address 30.30.13.1/24
nv set interface pf0vf0_if ip address 2030:30:13::1/64
```

Sample Flat File Configuration

```
auto pf0hpf_if
iface pf0hpf_if
    address 2030:30:11::1/64
    address 30.30.11.1/24

auto pf0vf0_if
iface pf0vf0_if
    address 2030:30:13::1/64
    address 30.30.13.1/24
```

17.8.4.3.2.2 BGP Peering with the Host

HBN supports the ability to establish a BGP session between the host and the HBN service running on BlueField Arm and allow the host to announce arbitrary route prefixes through the BlueField into the underlay fabric. The host can use any standard BGP protocol stack implementation to establish BGP peering with HBN.

Traffic to and from endpoints on the host gets offloaded.

> Both IPv4 and IPv6 unicast AFI/SAFI are supported.

It is possible to apply route filtering for these prefixes to limit the potential security impact in this configuration.

Sample NVUE Configuration for Host BGP Peering

The following code block shows configuration to peer to host at `45.3.0.4` and `2001:cafe:1ead::4`. The BGP session can be established using IPv4 or IPv6 address.

> Either of these sessions can support IPv4 unicast and IPv6 unicast AFI/SAFI.

NVUE configuration for peering with host:

```
nv set vrf default router bgp autonomous-system 63642
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor 45.3.0.4 nexthop-connected-check off
nv set vrf default router bgp neighbor 45.3.0.4 peer-group dpu_host
nv set vrf default router bgp neighbor 45.3.0.4 type numbered
nv set vrf default router bgp neighbor 2001:cafe:1ead::4 nexthop-connected-check off
nv set vrf default router bgp neighbor 2001:cafe:1ead::4 peer-group dpu_host
nv set vrf default router bgp neighbor 2001:cafe:1ead::4 type numbered
nv set vrf default router bgp peer-group dpu_host address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group dpu_host address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group dpu_host remote-as external
```

Sample Flat Files Configuration for Host BGP peering

The following block shows configuration to peer to host at `45.3.0.4` and `2001:cafe:1ead::4`. The BGP session can be established using IPv4 or IPv6 address.

`frr.conf` file:

```
router bgp 63642
 bgp router-id 27.0.0.4
 bgp bestpath as-path multipath-relax
 neighbor dpu_host peer-group
 neighbor dpu_host remote-as external
 neighbor dpu_host advertisement-interval 0
 neighbor dpu_host timers 3 9
 neighbor dpu_host timers connect 10
 neighbor dpu_host disable-connected-check
 neighbor fabric peer-group
 neighbor fabric remote-as external
 neighbor fabric advertisement-interval 0
 neighbor fabric timers 3 9
 neighbor fabric timers connect 10
 neighbor 45.3.0.4 peer-group dpu_host
 neighbor 2001:cafe:1ead::4 peer-group dpu_host
 neighbor p0_if interface peer-group fabric
 neighbor p1_if interface peer-group fabric
 !
 address-family ipv4 unicast
    neighbor dpu_host activate
 !
 address-family ipv6 unicast
    neighbor dpu_host activate
```

Sample FRR configuration on the Host

Any BGP implementation can be used on the host to peer to HBN and advertise endpoints. The following is an example using FRR BGP:

Sample FRR configuration on the host:

```
bf2-s12# sh run
Building configuration...

Current configuration:
!
frr version 7.2.1
frr defaults traditional
hostname bf2-s12
no ip forwarding
no ipv6 forwarding
!
router bgp 1000008
!
router bgp 1000008 vrf v_200_2000
 neighbor 45.3.0.2 remote-as external
 neighbor 2001:cafe:1ead::2 remote-as external
 !
 address-family ipv4 unicast
  redistribute connected
 exit-address-family
 !
 address-family ipv6 unicast
  redistribute connected
  neighbor 45.3.0.2 activate
  neighbor 2001:cafe:1ead::2 activate
 exit-address-family
!
line vty
!
end
```

Sample interfaces configuration on the host:

```
root@bf2-s12:/home/cumulus# ifquery -a
auto lo
iface lo inet loopback
```

```
    address 27.0.0.7/32
    address 2001:c000:10ff:f00d::7/128

auto v_200_2000
iface v_200_2000
    address 60.1.0.1
    address 60.1.0.2
    address 60.1.0.3
    address 2001:60:1::1
    address 2001:60:1::2
    address 2001:60:1::3
    vrf-table auto
auto ens1f0np0
iface ens1f0np0
    address 45.3.0.4/24
    address 2001:cafe:1ead::4/64
    gateway 45.3.0.1
    gateway 2001:cafe:1ead::1
    vrf v_200_2000
    hwaddress 00:03:00:08:00:12
    mtu 9162
```

17.8.4.3.2.3 VRF Route Leaking

VRFs are typically used when multiple independent routing and forwarding tables are desirable. However, users may want to reach destinations in one VRF from another VRF, as in the following cases:

- To make a service, such as a firewall available to multiple VRFs
- To enable routing to external networks or the Internet for multiple VRFs, where the external network itself is reachable through a specific VRF

Route leaking can be used to reach remote destinations as well as directly connected destinations in another VRF. Multiple VRFs can import routes from a single source VRF, and a VRF can import routes from multiple source VRFs. This can be used when a single VRF provides connectivity to external networks or a shared service for other VRFs. It is possible to control the routes leaked dynamically across VRFs with a route map.

When route leaking is used:

- The `redistribute` command (not `network` command) must be used in BGP to leak non-BGP routes (connected or static routes)
- It is not possible to leak routes between the default and non-default VRF

Kernel limitation

Ping or other IP traffic from a locally connected host in vrfX to a local interface IP address on the BlueField/HBN in vrfY does not work, even if VRF route-leaking is enabled between these two VRFs.

In the following example commands, routes in the BGP routing table of VRF `BLUE` dynamically leak into VRF `RED`:

```
nv set vrf RED router bgp address-family ipv4-unicast route-import from-vrf list BLUE
nv config apply
```

The following example commands delete leaked routes from VRF `BLUE` to VRF `RED`:

```
nv unset vrf RED router bgp address-family ipv4-unicast route-import from-vrf list BLUE
nv config apply
```

To exclude certain prefixes from the import process, configure the prefixes in a route map.

The following example configures a route map to match the source protocol BGP and imports the routes from VRF `BLUE` to VRF `RED`. For the imported routes, the community is 11:11 in VRF `RED`.

```
nv set vrf RED router bgp address-family ipv4-unicast route-import from-vrf list BLUE
nv set router policy route-map BLUEtoRED rule 10 match type ipv4
nv set router policy route-map BLUEtoRED rule 10 match source-protocol bgp
nv set router policy route-map BLUEtoRED rule 10 action permit
nv set router policy route-map BLUEtoRED rule 10 set community 11:11
nv set vrf RED router bgp address-family ipv4-unicast route-import from-vrf route-map BLUEtoRED
nv config
```

To check the status of the VRF route leaking, run:

- NVUE command:

  ```
  nv show vrf <vrf-name> router bgp address-family ipv4-unicast route-import
  ```

- Vtysh command:

  ```
  show ip bgp vrf <vrf-name> ipv4|ipv6 unicast route-leak command.
  ```

- For example:

  ```
  nv show vrf RED router bgp address-family ipv4-unicast route-import
                  operational   applied
  --------------  ------------  ---------
  from-vrf
    enable                      on
    route-map                   BLUEtoRED
    [list]        BLUE          BLUE
  [route-target]  10.10.10.1:3
  ```

To show more detailed status information, the following NVUE commands are available:

- `nv show vrf <vrf-name> router bgp address-family ipv4-unicast route-import from-vrf`
- `nv show vrf <vrf-name> router bgp address-family ipv4-unicast route-import from-vrf list`
- `nv show vrf <vrf-name> router bgp address-family ipv4-unicast route-import from-vrf list <leak-vrf-id>`

To view the BGP routing table, run:

- NVUE command:

  ```
  nv show vrf <vrf-name> router bgp address-family ipv4-unicast
  ```

- Vtysh command:

  ```
  show ip bgp vrf <vrf-name> ipv4|ipv6 unicast
  ```

To view the FRR IP routing table, run:

- Vtysh command:

  ```
  show ip route vrf <vrf-name>
  ```

- Or:

```
net show route vrf <vrf-name>
```

These commands show all routes, including routes leaked from other VRFs.

17.8.4.3.2.4 VLAN Subinterfaces

A VLAN subinterface is a VLAN device on an interface. The VLAN ID appends to the parent interface using dot (`.`) VLAN notation which is a standard way to specify a VLAN device in Linux.

For example:

- A VLAN with ID 100 which is a subinterface of `p0_if` is annotated as `p0_if.100`
- The subinterface `p0_if.100` only receives packets that have a VLAN 100 tag on port `p0_if`
- Any packets transmitted from `p0_if.100` would have VLAN tag 100

In HBN, VLAN subinterfaces can be created on uplink ports as well as on the host-facing PF and VF ports. A VLAN subinterface only receives traffic tagged for that VLAN.

VLAN subinterfaces are L3 interfaces and should not be added to a bridge.

In the following example, uplink subinterface on `p0_if` with VLAN ID 10 and a host facing subinterface on VF ports `pf1vf0_if` with VLAN ID 999 are created. The host-facing subinterface is also assigned with IPv4 and IPv6 addresses.

Subinterface configuration using NVUE commands:

```
nv set interface p0_if.10 base-interface p0_if
nv set interface p0_if.10 type sub
nv set interface p0_if.10 vlan 10

nv set interface pf1vf0_if type swp
nv set interface pf1vf0_if.999 base-interface pf1vf0_if
nv set interface pf1vf0_if.999 type sub
nv set interface pf1vf0_if.999 vlan 999
nv set interface pf1vf0_if ip address 30.30.14.1/24
nv set interface pf1vf0_if ip address 2030:30:14::1/64
```

Same configuration using sample flat file in `/etc/network/interfaces`:

subinterface configuration e/n/i file

```
auto p0_if.10
iface p0_if.10

auto pf1vf0_if.999
iface pf1vf0_if.999
    address 2030:30:40::1/64
    address 30.30.40.1/24
```

17.8.4.3.3 Ethernet Virtual Private Network - EVPN

HBN supports VXLAN with EVPN control plane for intra-subnet bridging (L2) services for IPv4 and IPv6 traffic in the overlay.

For the underlay, only IPv4 or BGP unnumbered configuration is supported.

HBN supports VXLAN encapsulation only over uplink parent interfaces.

17.8.4.3.3.1 Single VXLAN Device

With a single VXLAN device, a set of VXLAN network identifiers (VNIs) represents a single device model. The single VXLAN device has a set of attributes that belong to the VXLAN construct. Individual VNIs include VLAN-to-VNI mapping which allows users to specify which VLANs are associated with which VNIs. A single VXLAN device simplifies the configuration and reduces the overhead by replacing multiple traditional VXLAN devices with a single VXLAN device.

Users may configure a single VXLAN device automatically with NVUE, or manually by editing the `/etc/network/interfaces` file. When users configure a single VXLAN device with NVUE, NVUE creates a unique name for the device in the following format using the bridge name as the hash key: `vxlan<id>`.
This example configuration performs the following steps:

1. Creates a single VXLAN device (vxlan21).
2. Maps VLAN 10 to VNI 10 and VLAN 20 to VNI 20.
3. Adds the VXLAN device to the default bridge.

```
cumulus@leaf01:~$ nv set bridge domain bridge vlan 10 vni 10
cumulus@leaf01:~$ nv set bridge domain bridge vlan 20 vni 20
cumulus@leaf01:~$ nv set nve vxlan source address 10.10.10.1
cumulus@leaf01:~$ nv config apply
```

Alternately, users may edit the file `/etc/network/interfaces` as follows, then run the `ifreload -a` command to apply the SVD configuration.

```
auto lo
iface lo inet loopback
    vxlan-local-tunnelip 10.10.10.1

auto vxlan21
iface vxlan21
    bridge-vlan-vni-map 10=10 20=20
    bridge-learning off

auto bridge
iface bridge
    bridge-vlan-aware yes
    bridge-ports vxlan21 pf0hpf_if pf1hpf_if
    bridge-vids 10 20
    bridge-pvid 1
```

Users may not use a combination of single and traditional VXLAN devices.

17.8.4.3.3.2 Sample Switch Configuration for EVPN

The following is a sample NVUE config for underlay switches (NVIDIA® Spectrum® with Cumulus Linux) to enable EVPN deployments with HBN.

It assumes that the uplinks on all BlueField devices are connected to ports `swp1-4` on the switch.

```
nv set evpn enable on
nv set router bgp enable on

nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp autonomous-system 63640
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor swp1 peer-group fabric
nv set vrf default router bgp neighbor swp1 type unnumbered
nv set vrf default router bgp neighbor swp2 peer-group fabric
nv set vrf default router bgp neighbor swp2 type unnumbered
nv set vrf default router bgp neighbor swp3 peer-group fabric
nv set vrf default router bgp neighbor swp3 type unnumbered
nv set vrf default router bgp neighbor swp4 peer-group fabric
nv set vrf default router bgp neighbor swp4 type unnumbered
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp router-id 27.0.0.10

nv set interface lo ip address 2001:c000:10ff:f00d::10/128
nv set interface lo ip address 27.0.0.10/32
nv set interface lo type loopback
nv set interface swp1,swp2,swp3,swp4 type swp
```

17.8.4.3.3.3 Layer-2 EVPN

Sample NVUE Configuration for L2 EVPN

The following is a sample NVUE configuration which has L2-VNIs (`2000` , `2001`) for EVPN bridging on BlueField.

```
nv set bridge domain br_default encap 802.1Q
nv set bridge domain br_default type vlan-aware
nv set bridge domain br_default vlan 200 vni 2000 flooding enable auto
nv set bridge domain br_default vlan 200 vni 2000 mac-learning off
nv set bridge domain br_default vlan 201 vni 2001 flooding enable auto
nv set bridge domain br_default vlan 201 vni 2001 mac-learning off

nv set evpn enable on
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan mac-learning off
nv set nve vxlan source address 27.0.0.4
nv set router bgp enable on
nv set system global anycast-mac 44:38:39:42:42:07
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on

nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp autonomous-system 63642
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor p0_if peer-group fabric
nv set vrf default router bgp neighbor p0_if type unnumbered
nv set vrf default router bgp neighbor p1_if peer-group fabric
nv set vrf default router bgp neighbor p1_if type unnumbered
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast policy outbound route-map
MY_ORIGIN_ASPATH_ONLY
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast policy outbound route-map
MY_ORIGIN_ASPATH_ONLY
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp router-id 27.0.0.4

nv set interface lo ip address 2001:c000:10ff:f00d::4/128
nv set interface lo ip address 27.0.0.4/32
nv set interface lo type loopback
nv set interface p0_if,p1_if,pf0hpf_if,pf1hpf_if type swp
nv set interface pf0hpf_if bridge domain br_default access 200
nv set interface pf1hpf_if bridge domain br_default access 201

nv set interface vlan200-201 base-interface br_default
nv set interface vlan200-201 ip ipv4 forward on
nv set interface vlan200-201 ip ipv6 forward on
nv set interface vlan200-201 ip vrr enable on
nv set interface vlan200-201 ip vrr state up
nv set interface vlan200-201 link mtu 9050
nv set interface vlan200-201 type svi
nv set interface vlan200 ip address 2001:cafe:1ead::3/64
nv set interface vlan200 ip address 45.3.0.2/24
nv set interface vlan200 ip vrr address 2001:cafe:1ead::1/64
nv set interface vlan200 ip vrr address 45.3.0.1/24
nv set interface vlan200 vlan 200
nv set interface vlan201 ip address 2001:cafe:1ead:1::3/64
nv set interface vlan201 ip address 45.3.1.2/24
nv set interface vlan201 ip vrr address 2001:cafe:1ead:1::1/64
nv set interface vlan201 ip vrr address 45.3.1.1/24
nv set interface vlan201 vlan 201
```

Sample Flat Files Configuration for L2 EVPN

The following is a sample flat files configuration which has L2-VNIs (`vx-2000` , `vx-2001`) for EVPN bridging on BlueField.

This file is located at `/etc/network/interfaces` :

```
auto lo
iface lo inet loopback
    address 2001:c000:10ff:f00d::4/128
    address 27.0.0.4/32
    vxlan-local-tunnelip 27.0.0.4

auto p0_if
iface p0_if

auto p1_if
iface p1_if

auto pf0hpf_if
iface pf0hpf_if
    bridge-access 200

auto pf1hpf_if
iface pf1hpf_if
    bridge-access 201

auto vlan200
iface vlan200
    address 2001:cafe:1ead::3/64
    address 45.3.0.2/24
    mtu 9050
    address-virtual 00:00:5e:00:01:01 2001:cafe:1ead::1/64 45.3.0.1/24
    vlan-raw-device br_default
    vlan-id 200

auto vlan201
iface vlan201
    address 2001:cafe:1ead:1::3/64
    address 45.3.1.2/24
    mtu 9050
    address-virtual 00:00:5e:00:01:01 2001:cafe:1ead:1::1/64 45.3.1.1/24
    vlan-raw-device br_default
    vlan-id 201

auto vxlan48
iface vxlan48
    bridge-vlan-vni-map 200=2000 201=2001
217=2017
    bridge-learning off

auto br_default
iface br_default
    bridge-ports pf0hpf_if pf1hpf_if vxlan48
    bridge-vlan-aware yes
    bridge-vids 200 201
    bridge-pvid 1
```

This file tells the `frr` package which daemon to start and is located at `/etc/frr/daemons` :

```
bgpd=yes
ospfd=no
ospf6d=no
isisd=no
pimd=no
ldpd=no
pbrd=no
vrrpd=no
fabricd=no
nhrpd=no
eigrpd=no
babeld=no
sharpd=no
fabricd=no
ripngd=no
ripd=no

vtysh_enable=yes
zebra_options="  -M cumulus_mlag -M snmp -A 127.0.0.1 -s 90000000"
bgpd_options="   -M snmp -A 127.0.0.1"
ospfd_options="  -M snmp -A 127.0.0.1"
ospf6d_options=" -M snmp -A ::1"
ripd_options="   -A 127.0.0.1"
ripngd_options=" -A ::1"
isisd_options="  -A 127.0.0.1"
pimd_options="   -A 127.0.0.1"
ldpd_options="   -A 127.0.0.1"
nhrpd_options="  -A 127.0.0.1"
eigrpd_options=" -A 127.0.0.1"
babeld_options=" -A 127.0.0.1"
sharpd_options=" -A 127.0.0.1"
pbrd_options="   -A 127.0.0.1"
staticd_options="-A 127.0.0.1"
fabricd_options="-A 127.0.0.1"
```

```
vrrpd_options="  -A 127.0.0.1"

frr_profile="datacenter"
```

FRR configuration file is located at `/etc/frr/frr.conf`:

```
!---- Cumulus Defaults ----
frr defaults datacenter
log syslog informational
no zebra nexthop kernel enable
vrf default
outer bgp 63642 vrf default
bgp router-id 27.0.0.4
bgp bestpath as-path multipath-relax
timers bgp 3 9
bgp deterministic-med
! Neighbors
neighbor fabric peer-group
neighbor fabric remote-as external
neighbor fabric timers 3 9
neighbor fabric timers connect 10
neighbor fabric advertisement-interval 0
neighbor p0_if interface peer-group fabric
neighbor p1_if interface peer-group fabric
address-family ipv4 unicast
maximum-paths ibgp 64
maximum-paths 64
distance bgp 20 200 200
neighbor fabric activate
exit-address-family
address-family ipv6 unicast
maximum-paths ibgp 64
maximum-paths 64
distance bgp 20 200 200
neighbor fabric activate
exit-address-family
address-family l2vpn evpn
advertise-all-vni
neighbor fabric activate
exit-address-family
```

17.8.4.3.3.4 Layer-3 EVPN with Symmetric Routing

In distributed symmetric routing, each VXLAN endpoint (VTEP) acts as a layer-3 gateway, performing routing for its attached hosts. However, both the ingress VTEP and egress VTEP route the packets (similar to traditional routing behavior of routing to a next-hop router). In a VXLAN encapsulated packet, the inner destination MAC address is the router MAC address of the egress VTEP to indicate that the egress VTEP is the next hop and that it must also perform the routing.

All routing happens in the context of a tenant (VRF). For a packet that the ingress VTEP receives from a locally attached host, the SVI interface corresponding to the VLAN determines the VRF. For a packet that the egress VTEP receives over the VXLAN tunnel, the VNI in the packet has to specify the VRF. For symmetric routing, this is a VNI corresponding to the tenant and is different from either the source VNI or the destination VNI. This VNI is a layer-3 VNI or interconnecting VNI. The regular VNI, which maps a VLAN, is the layer-2 VNI.

For more details about this, refer to the Cumulus Linux User Manual.

HBN uses a one-to-one mapping between an L3 VNI and a tenant (VRF).

The VRF to L3 VNI mapping has to be consistent across all VTEPs.

An L3 VNI and an L2 VNI cannot have the same ID.

In an EVPN symmetric routing configuration, when the switch announces a type-2 (MAC/IP) route, in addition to containing two VNIs (L2 and L3 VNIs), the route also contains separate route targets (RTs) for L2 and L3. The L3 RT associates the route with the tenant VRF. By default, this is auto-derived using the L3 VNI instead of the L2 VNI. However, this is configurable.

For EVPN symmetric routing, users must perform the configuration listed in the following subsections. Optional configuration includes configuring a route distinguisher (RD) and RTs for the tenant VRF, and advertising the locally-attached subnets.

Sample NVUE Configuration for L3 EVPN

If using NVUE to configure EVPN symmetric routing, the following is a sample configuration using NVUE commands:

```
nv set bridge domain br_default vlan 111 vni 1000111
nv set bridge domain br_default vlan 112 vni 1000112
nv set bridge domain br_default vlan 213 vni 1000213
nv set bridge domain br_default vlan 214 vni 1000214
nv set evpn enable on
nv set interface lo ip address 6.0.0.19/32
nv set interface lo type loopback
nv set interface p0_if description 'alias p0_if to leaf-21 swp3'
nv set interface p0_if,p1_if,pf0hpf_if,pf0vf0_if,pf1hpf_if,pf1vf0_if type swp
nv set interface p1_if description 'alias p1_if to leaf-22 swp3'
nv set interface pf0hpf_if bridge domain br_default access 111
nv set interface pf0hpf_if description 'alias pf0hpf_if to host-211 ens2f0np0'
nv set interface pf0vf0_if bridge domain br_default access 112
nv set interface pf0vf0_if description 'alias pf0vf0_if to host-211 ens2f0np0v0'
nv set interface pf1hpf_if bridge domain br_default access 213
nv set interface pf1hpf_if description 'alias pf1hpf_if to host-211 ens2f1np1'
nv set interface pf1vf0_if bridge domain br_default access 214
nv set interface pf1vf0_if description 'alias pf1vf0_if to host-211 ens2f1np0v0'
nv set interface vlan111 ip address 60.1.1.21/24
nv set interface vlan111 ip address 2060:1:1:1::21/64
nv set interface vlan111 ip vrr address 60.1.1.250/24
nv set interface vlan111 ip vrr address 2060:1:1:1::250/64
nv set interface vlan111 vlan 111
nv set interface vlan111,213 ip vrf vrf2
nv set interface vlan111-112,213-214 ip vrr enable on
nv set interface vlan111-112,213-214 ip vrr mac-address 00:00:5e:00:01:01
nv set interface vlan111-112,213-214 ip ipv4 forward on
nv set interface vlan111-112,213-214 ip ipv6 forward on
nv set interface vlan111-112,213-214 type svi
nv set interface vlan112 ip address 50.1.1.21/24
nv set interface vlan112 ip address 2050:1:1:1::21/64
nv set interface vlan112 ip vrr address 50.1.1.250/24
nv set interface vlan112 ip vrr address 2050:1:1:1::250/64
nv set interface vlan112 vlan 112
nv set interface vlan112,214 ip vrf vrf1
nv set interface vlan213 ip address 60.1.210.21/24
nv set interface vlan213 ip address 2060:1:1:210::21/64
nv set interface vlan213 ip vrr address 60.1.210.250/24
nv set interface vlan213 ip vrr address 2060:1:1:210::250/64
nv set interface vlan213 vlan 213
nv set interface vlan214 ip address 50.1.210.21/24
nv set interface vlan214 ip address 2050:1:1:210::21/64
nv set interface vlan214 ip vrr address 50.1.210.250/24
nv set interface vlan214 ip vrr address 2050:1:1:210::250/64
nv set interface vlan214 vlan 214
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan source address 6.0.0.19
nv set platform
nv set router bgp enable on
nv set router policy route-map ALLOW_LOBR rule 10 action permit
nv set router policy route-map ALLOW_LOBR rule 10 match interface lo
nv set router policy route-map ALLOW_LOBR rule 20 action permit
nv set router policy route-map ALLOW_LOBR rule 20 match interface br_default
nv set router policy route-map ALLOW_VRF1 rule 10 action permit
nv set router policy route-map ALLOW_VRF1 rule 10 match interface vrf1
nv set router policy route-map ALLOW_VRF2 rule 10 action permit
nv set router policy route-map ALLOW_VRF2 rule 10 match interface vrf2
nv set router vrr enable on
nv set system global system-mac 00:01:00:00:1e:03
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast multipaths ebgp 16
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected route-map ALLOW_LOBR
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp autonomous-system 650019
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor p0_if address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp neighbor p0_if address-family l2vpn-evpn enable on
nv set vrf default router bgp neighbor p0_if peer-group TOR_LEAF_SPINE
nv set vrf default router bgp neighbor p0_if remote-as external
nv set vrf default router bgp neighbor p0_if type unnumbered
nv set vrf default router bgp neighbor p1_if address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp neighbor p1_if address-family l2vpn-evpn enable on
nv set vrf default router bgp neighbor p1_if peer-group TOR_LEAF_SPINE
nv set vrf default router bgp neighbor p1_if remote-as external
nv set vrf default router bgp neighbor p1_if type unnumbered
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp path-selection routerid-compare on
nv set vrf default router bgp peer-group TOR_LEAF_SPINE address-family ipv4-unicast enable on
nv set vrf default router bgp router-id 6.0.0.19
nv set vrf vrf1 evpn enable on
nv set vrf vrf1 evpn vni 104001
nv set vrf vrf1 loopback ip address 50.1.21.21/32
nv set vrf vrf1 loopback ip address 2050:50:50:21::21/128
nv set vrf vrf1 router bgp address-family ipv4-unicast enable on
nv set vrf vrf1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf vrf1 router bgp address-family ipv4-unicast redistribute connected route-map ALLOW_VRF1
nv set vrf vrf1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf vrf1 router bgp address-family ipv6-unicast enable on
nv set vrf vrf1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf vrf1 router bgp address-family ipv6-unicast redistribute connected route-map ALLOW_VRF1
nv set vrf vrf1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf vrf1 router bgp autonomous-system 650019
nv set vrf vrf1 router bgp enable on
nv set vrf vrf1 router bgp router-id 50.1.21.21
nv set vrf vrf2 evpn enable on
nv set vrf vrf2 evpn vni 104002
nv set vrf vrf2 loopback ip address 60.1.21.21/32
nv set vrf vrf2 loopback ip address 2060:60:60:21::21/128
nv set vrf vrf2 router bgp address-family ipv4-unicast enable on
nv set vrf vrf2 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf vrf2 router bgp address-family ipv4-unicast redistribute connected route-map ALLOW_VRF2
nv set vrf vrf2 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf vrf2 router bgp address-family ipv6-unicast enable on
nv set vrf vrf2 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf vrf2 router bgp address-family ipv6-unicast redistribute connected route-map ALLOW_VRF2
nv set vrf vrf2 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf vrf2 router bgp autonomous-system 650019
nv set vrf vrf2 router bgp enable on
nv set vrf vrf2 router bgp router-id 60.1.21.21
```

Sample Flat Files Configuration for L3 EVPN

The following is a sample flat files configuration which has L2 VNIs and L3 VNIs for EVPN bridging and symmetric routing on BlueField.

This file is located at `/etc/network/interfaces`:

```
auto lo
iface lo inet loopback
    address 6.0.0.19/32
    vxlan-local-tunnelip 6.0.0.19

auto vrf1
iface vrf1
    address 2050:50:50:21::21/128
    address 50.1.21.21/32
    vrf-table auto

auto vrf2
iface vrf2
    address 2060:60:60:21::21/128
    address 60.1.21.21/32
    vrf-table auto

auto p0_if
iface p0_if
    alias alias p0_if to leaf-21 swp3

auto p1_if
iface p1_if
    alias alias p1_if to leaf-22 swp3

auto pf0hpf_if
iface pf0hpf_if
    alias alias pf0hpf_if to host-211 ens2f0np0
    bridge-access 111

auto pf0vf0_if
iface pf0vf0_if
    alias alias pf0vf0_if to host-211 ens2f0np0v0
    bridge-access 112

auto pf1hpf_if
iface pf1hpf_if
    alias alias pf1hpf_if to host-211 ens2f1np1
    bridge-access 213

auto pf1vf0_if
iface pf1vf0_if
    alias alias pf1vf0_if to host-211 ens2f1np0v0
    bridge-access 214

auto vlan111
iface vlan111
    address 2060:1:1:1::21/64
    address 60.1.1.21/24
    address-virtual 00:00:5e:00:01:01 2060:1:1:1::250/64 60.1.1.250/24
    hwaddress 00:01:00:00:1e:03
    vrf vrf2
    vlan-raw-device br_default
    vlan-id 111

auto vlan112
iface vlan112
    address 2050:1:1:1::21/64
    address 50.1.1.21/24
    address-virtual 00:00:5e:00:01:01 2050:1:1:1::250/64 50.1.1.250/24
    hwaddress 00:01:00:00:1e:03
    vrf vrf1
    vlan-raw-device br_default
    vlan-id 112

auto vlan213
iface vlan213
    address 2060:1:1:210::21/64
    address 60.1.210.21/24
    address-virtual 00:00:5e:00:01:01 2060:1:1:210::250/64 60.1.210.250/24
    hwaddress 00:01:00:00:1e:03
    vrf vrf2
    vlan-raw-device br_default
    vlan-id 213

auto vlan214
iface vlan214
    address 2050:1:1:210::21/64
    address 50.1.210.21/24
    address-virtual 00:00:5e:00:01:01 2050:1:1:210::250/64 50.1.210.250/24
    hwaddress 00:01:00:00:1e:03
    vrf vrf1
    vlan-raw-device br_default
    vlan-id 214

auto vlan4058_l3
iface vlan4058_l3
    vrf vrf1
    vlan-raw-device br_default
    address-virtual none
    vlan-id 4058

auto vlan4059_l3
iface vlan4059_l3
    vrf vrf2
    vlan-raw-device br_default
    address-virtual none
    vlan-id 4059

auto vxlan48
iface vxlan48
    bridge-vlan-vni-map 111=1000111 112=1000112 213=1000213 214=1000214 4058=104001 4059=104002
    bridge-learning off

auto br_default
iface br_default
    bridge-ports pf0hpf_if pf0vf0_if pf1hpf_if pf1vf0_if vxlan48
    hwaddress 00:01:00:00:1e:03
    bridge-vlan-aware yes
    bridge-vids 111 112 213 214
    bridge-pvid 1
```

FRR configuration is located at `/etc/frr/frr.conf`:

```
frr version 8.4.3
frr defaults datacenter
hostname doca-hbn-service-bf3-s05-1-ipmi
log syslog informational
no zebra nexthop kernel enable
service integrated-vtysh-config
!
vrf vrf1
 vni 104001
exit-vrf
!
vrf vrf2
 vni 104002
exit-vrf
!
router bgp 650019
 bgp router-id 6.0.0.19
 bgp bestpath as-path multipath-relax
 bgp bestpath compare-routerid
 neighbor TOR_LEAF_SPINE peer-group
 neighbor TOR_LEAF_SPINE advertisement-interval 0
 neighbor TOR_LEAF_SPINE timers 3 9
 neighbor TOR_LEAF_SPINE timers connect 10
 neighbor p0_if interface peer-group TOR_LEAF_SPINE
 neighbor p0_if remote-as external
 neighbor p0_if advertisement-interval 0
 neighbor p0_if timers 3 9
 neighbor p0_if timers connect 10
 neighbor p1_if interface peer-group TOR_LEAF_SPINE
 neighbor p1_if remote-as external
 neighbor p1_if advertisement-interval 0
 neighbor p1_if timers 3 9
 neighbor p1_if timers connect 10
 !
 address-family ipv4 unicast
  redistribute connected route-map ALLOW_LOBR
  maximum-paths 16
  maximum-paths ibgp 64
 exit-address-family
 !
 address-family l2vpn evpn
  neighbor p0_if activate
  neighbor p1_if activate
  advertise-all-vni
 exit-address-family
```

```
exit
!
router bgp 650019 vrf vrf1
 bgp router-id 50.1.21.21
 !
 address-family ipv4 unicast
  redistribute connected route-map ALLOW_VRF1
  maximum-paths 64
  maximum-paths ibgp 64
 exit-address-family
 !
 address-family ipv6 unicast
  redistribute connected route-map ALLOW_VRF1
  maximum-paths 64
  maximum-paths ibgp 64
 exit-address-family
 !
 address-family l2vpn evpn
  advertise ipv4 unicast
  advertise ipv6 unicast
 exit-address-family
exit
!
router bgp 650019 vrf vrf2
 bgp router-id 60.1.21.21
 !
 address-family ipv4 unicast
  redistribute connected route-map ALLOW_VRF2
  maximum-paths 64
  maximum-paths ibgp 64
 exit-address-family
 !
 address-family ipv6 unicast
  redistribute connected route-map ALLOW_VRF2
  maximum-paths 64
  maximum-paths ibgp 64
 exit-address-family
 !
 address-family l2vpn evpn
  advertise ipv4 unicast
  advertise ipv6 unicast
 exit-address-family
exit
!
route-map ALLOW_LOBR permit 10
 match interface lo
exit
!
route-map ALLOW_LOBR permit 20
 match interface br_default
exit
!
route-map ALLOW_VRF1 permit 10
 match interface vrf1
exit
!
route-map ALLOW_VRF2 permit 10
 match interface vrf2
exit
```

17.8.4.3.3.5 Multi-hop eBGP Peering for EVPN (Route Server in Symmetric EVPN Routing)

eBGP multi-hop peering for EVPN support in a route server-like role in EVPN topology, allows the deployment of EVPN on any cloud that supports IP transport.

Route servers and BF/HBN VTEPs are connected via the IP cloud. That is:

- Switches in the cloud provider need not be EVPN-aware
- Switches in the provider fabric provide IPv4 and IPv6 transport and do not have to support EVPN

Sample Route Server Configuration for EVPN

The following is a sample configuration of an Ubuntu server running FRR 9.0 stable, configured as EVPN route server and an HBN VTEP that is peering to two spine switches for IP connectivity and 3 Route servers for EVPN overlay control.

```
root@sn1:/home/cumulus# uname -a
Linux sn1 5.15.0-88-generic #98-Ubuntu SMP Mon Oct 2 15:18:56 UTC 2023 x86_64 x86_64 x86_64 GNU/Linux
root@sn1:/home/cumulus# dpkg -l frr
Desired=Unknown/Install/Remove/Purge/Hold
| Status=Not/Inst/Conf-files/Unpacked/halF-conf/Half-inst/trig-aWait/Trig-pend
|/ Err?=(none)/Reinst-required (Status,Err: uppercase=bad)
||/ Name           Version               Architecture Description
+++-==============-=====================-============-=============================================================
ii  frr            9.0.1-0~ubuntu22.04.1 amd64        FRRouting suite of internet protocols (BGP, OSPF, IS-IS, ...)
root@sn1:/home/cumulus#
```

FRR configuration (`frr.conf`):

```
sn1# sh run
Building configuration...

Current configuration:
!
frr version 9.0.1
frr defaults datacenter
hostname sn1
no ip forwarding
no ipv6 forwarding
service integrated-vtysh-config
!
router bgp 4200065507
 bgp router-id 6.0.0.7
 timers bgp 60 180
 neighbor rclients peer-group
 neighbor rclients remote-as external
 neighbor rclients ebgp-multihop 10
 neighbor rclients update-source lo
 neighbor rclients advertisement-interval 0
 neighbor rclients timers 3 9
 neighbor rclients timers connect 10
 neighbor rcsuper peer-group
 neighbor rcsuper remote-as external
 neighbor rcsuper advertisement-interval 0
 neighbor rcsuper timers 3 9
 neighbor rcsuper timers connect 10
 neighbor swp1 interface peer-group rcsuper
 bgp listen range 6.0.0.0/24 peer-group rclients
 !
 address-family ipv4 unicast
  redistribute connected
  neighbor fabric route-map pass in
  neighbor fabric route-map pass out
  no neighbor rclients activate
  maximum-paths 64
  maximum-paths ibgp 64
 exit-address-family
 !
 address-family l2vpn evpn
  neighbor rclients activate
  neighbor rcsuper activate
 exit-address-family
exit
!
route-map pass permit 10
 set community 11:11 additive
exit
!
end
sn1#
```

Interfaces configuration (`/etc/network/interfaces`):

```
root@sn1:/home/cumulus# ifquery -a
auto lo
iface lo inet loopback
    address 6.0.0.7/32

auto lo
iface lo inet loopback

auto swp1
iface swp1

auto eth0
iface eth0
    address 192.168.0.15/24
    gateway 192.168.0.2

root@sn1:/home/cumulus#
```

Sample HBN Configuration for Deployments with EVPN Route Server

```
root@doca-hbn-service-bf2-s12-1-ipmi:/tmp# nv config show -o commands
nv set bridge domain br_default vlan 101 vni 10101
nv set bridge domain br_default vlan 102 vni 10102
nv set bridge domain br_default vlan 201 vni 10201
nv set bridge domain br_default vlan 202 vni 10202
nv set evpn enable on
nv set evpn route-advertise svi-ip off
nv set interface ilan3200 ip vrf internet1
nv set interface ilan3200 vlan 3200
nv set interface ilan3200,slan3201,vlan101-102,201-202,3001-3002 base-interface br_default
nv set interface ilan3200,slan3201,vlan101-102,201-202,3001-3002 type svi
nv set interface lo ip address 6.0.0.13/32
nv set interface lo ip address 2001::13/128
nv set interface lo type loopback
nv set interface p0_if,p1_if,pf0hpf_if,pf0vf0_if,pf0vf1_if,pf0vf2_if,pf0vf3_if,pf1hpf_if type swp
nv set interface pf0vf0_if bridge domain br_default access 101
nv set interface pf0vf1_if bridge domain br_default access 102
nv set interface pf0vf2_if bridge domain br_default access 201
nv set interface pf0vf3_if bridge domain br_default access 202
nv set interface slan3201 ip vrf special1
nv set interface slan3201 vlan 3201
nv set interface vlan101 ip address 21.1.0.13/16
nv set interface vlan101 ip address 2020:0:1:1::13/64
nv set interface vlan101 ip vrr address 21.1.0.250/16
nv set interface vlan101 ip vrr address 2020:0:1:1::250/64
nv set interface vlan101 ip vrr mac-address 00:00:01:00:00:65
nv set interface vlan101 vlan 101
nv set interface vlan101-102,201-202 ip vrr enable on
nv set interface vlan101-102,3001 ip vrf tenant1
nv set interface vlan102 ip address 21.2.0.13/16
nv set interface vlan102 ip address 2020:0:1:2::13/64
nv set interface vlan102 ip vrr address 21.2.0.250/16
nv set interface vlan102 ip vrr address 2020:0:1:2::250/64
nv set interface vlan102 ip vrr mac-address 00:00:01:00:00:66
nv set interface vlan102 vlan 102
nv set interface vlan201 ip address 22.1.0.13/16
nv set interface vlan201 ip address 2020:0:2:1::13/64
nv set interface vlan201 ip vrr address 22.1.0.250/16
nv set interface vlan201 ip vrr address 2020:0:2:1::250/64
nv set interface vlan201 ip vrr mac-address 00:00:02:00:00:c9
nv set interface vlan201 vlan 201
nv set interface vlan201-202,3002 ip vrf tenant2
nv set interface vlan202 ip address 22.2.0.13/16
nv set interface vlan202 ip address 2020:0:2:2::13/64
nv set interface vlan202 ip vrr address 22.2.0.250/16
nv set interface vlan202 ip vrr address 2020:0:2:2::250/64
nv set interface vlan202 ip vrr mac-address 00:00:02:00:00:ca
nv set interface vlan202 vlan 202
nv set interface vlan3001 vlan 3001
nv set interface vlan3002 vlan 3002
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan source address 6.0.0.13
nv set platform
nv set router bgp autonomous-system 4200065011
nv set router bgp enable on
nv set router bgp router-id 6.0.0.13
nv set router vrr enable on
nv set system config snippet
nv set system global
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor 6.0.0.7 peer-group rservers
nv set vrf default router bgp neighbor 6.0.0.7 type numbered
nv set vrf default router bgp neighbor 6.0.0.8 peer-group rservers
nv set vrf default router bgp neighbor 6.0.0.8 type numbered
nv set vrf default router bgp neighbor 6.0.0.9 peer-group rservers
nv set vrf default router bgp neighbor 6.0.0.9 type numbered
nv set vrf default router bgp neighbor p0_if peer-group fabric
nv set vrf default router bgp neighbor p0_if type unnumbered
nv set vrf default router bgp neighbor p1_if peer-group fabric
nv set vrf default router bgp neighbor p1_if type unnumbered
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on

nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp peer-group rservers address-family ipv4-unicast enable off
nv set vrf default router bgp peer-group rservers address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group rservers address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group rservers multihop-ttl 3
nv set vrf default router bgp peer-group rservers remote-as external
nv set vrf default router bgp peer-group rservers update-source lo
nv set vrf internet1 evpn enable on
nv set vrf internet1 evpn vni 42000
nv set vrf internet1 loopback ip address 8.1.0.13/32
nv set vrf internet1 loopback ip address 2008:0:1::13/64
nv set vrf internet1 router bgp address-family ipv4-unicast enable on
nv set vrf internet1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf internet1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf internet1 router bgp enable on
nv set vrf special1 evpn enable on
nv set vrf special1 evpn vni 42001
nv set vrf special1 loopback ip address 9.1.0.13/32
nv set vrf special1 loopback ip address 2009:0:1::13/64
nv set vrf special1 router bgp address-family ipv4-unicast enable on
nv set vrf special1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf special1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf special1 router bgp enable on
nv set vrf tenant1 evpn enable on
nv set vrf tenant1 evpn vni 30001
nv set vrf tenant1 router bgp address-family ipv4-unicast enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp enable on
nv set vrf tenant1 router bgp router-id 6.0.0.13
nv set vrf tenant2 evpn enable on
nv set vrf tenant2 evpn vni 30002
nv set vrf tenant2 router bgp address-family ipv4-unicast enable on
nv set vrf tenant2 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant2 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant2 router bgp enable on
nv set vrf tenant2 router bgp router-id 6.0.0.13
root@doca-hbn-service-bf2-s12-1-ipmi:/tmp#
```

Verifying BGP sessions in HBN:

```
doca-hbn-service-bf2-s12-1-ipmi# sh bgp sum

IPv4 Unicast Summary (VRF default):
BGP router identifier 6.0.0.13, local AS number 4200065011 vrf-id 0
BGP table version 20
RIB entries 21, using 4032 bytes of memory
Peers 2, using 40 KiB of memory
Peer groups 2, using 128 bytes of memory

Neighbor        V         AS   MsgRcvd   MsgSent   TblVer  InQ OutQ  Up/Down State/PfxRcd   PfxSnt Desc
spine11(p0_if)  4      65201     30617     30620        0    0    0 1d01h30m            9       11 N/A
spine12(p1_if)  4      65201     30620     30623        0    0    0 1d01h30m            9       11 N/A

Total number of neighbors 2

IPv6 Unicast Summary (VRF default):
BGP router identifier 6.0.0.13, local AS number 4200065011 vrf-id 0
BGP table version 0
RIB entries 0, using 0 bytes of memory
Peers 2, using 40 KiB of memory
Peer groups 2, using 128 bytes of memory

Neighbor        V         AS   MsgRcvd   MsgSent   TblVer  InQ OutQ  Up/Down State/PfxRcd   PfxSnt Desc
spine11(p0_if)  4      65201     30617     30620        0    0    0 1d01h30m            0        0 N/A
spine12(p1_if)  4      65201     30620     30623        0    0    0 1d01h30m            0        0 N/A

Total number of neighbors 2

L2VPN EVPN Summary (VRF default):
BGP router identifier 6.0.0.13, local AS number 4200065011 vrf-id 0
BGP table version 0
RIB entries 79, using 15 KiB of memory
Peers 3, using 60 KiB of memory
Peer groups 2, using 128 bytes of memory

Neighbor        V         AS   MsgRcvd   MsgSent   TblVer  InQ OutQ  Up/Down State/PfxRcd   PfxSnt Desc
sn1(6.0.0.7)    4 4200065507     31410     31231        0    0    0 00:27:51           69       95 N/A
sn2(6.0.0.8)    4 4200065508     31169     31062        0    0    0 02:34:47           69       95 N/A
sn3(6.0.0.9)    4 4200065509     31285     31059        0    0    0 02:34:47           69       95 N/A

Total number of neighbors 3
doca-hbn-service-bf2-s12-1-ipmi#
```

The command output shows that the HBN has BGP sessions with spine switches exchanging IPv4/IPv6 unicast. BGP sessions with route servers `sn1`, `sn2`, and `sn3` only exchanging L2VPN EVPN AFI/SAFI.

17.8.4.3.3.6 Downstream VNI (DVNI)

Downstream VNI (symmetric EVPN route leaking) allows users to leak remote EVPN routes without having the source tenant VRF locally configured. A common use case is where upstream switches learn the L3VNI from downstream leaf switches and impose the learned L3VNI to the traffic VXLAN routed to the associated VRF. This eliminates the need to configure L3VNI-SVI interfaces on all leaf switches and enables shared service and hub-and-spoke scenarios.

To configure access to a shared service in a specific VRF, users must:

1. Configure route-target import statements, effectively leaking routes from remote tenants to the shared VRF.
2. Import shared VRF's route-target at the remote nodes.

The route target import or export statement takes the following format:

```
route-target import|export <asn>:<vni>
```

For example:

```
route-target import 65101:6000
```

For route target import statements, users can use `route-target import ANY:<vni>` for NVUE commands or `route-target import *:<vni>` in the `/etc/frr/frr.conf` file. `ANY` in NVUE commands or the asterisk (`*`) in the `/etc/frr/frr.conf` file use any ASN (autonomous system number) as a wildcard.

The NVUE commands are as follows:

1. To configure a route import statement:

   ```
   nv set vrf <vrf> router bgp route-import from-evpn route-target <asn>:<vni>
   ```

2. To configure a route export statement:

   ```
   nv set vrf <vrf> router bgp route-export from-evpn route-target <asn>:<vni>
   ```

Important considerations when implementing DVNI configuration:

- EVPN symmetric mode supports downstream VNI with L3 VNIs and single VXLAN devices only
- You can configure multiple import and export route targets in a VRF
- You cannot leak (import) overlapping tenant prefixes into the same destination VRF

> If symmetric EVPN configuration is using automatic import/export (which is often the case), when DVNI is configured, automatic import of a tenant's VNI is disabled which isolates the VRF from the tenant. To avoid this issue, add `route-import from-evpn route-target auto` to the command line.

DVNI Configurations for Shared Internet Service

Configuration example here considers a scenario where External/Internet connectivity is available via a firewall (FW), which is connected to a shared VRF (`vrf external` in this example).

The routes on super spine switches have `external` VRF configured in which the route-targets from remote tenants are imported.

On BlueField devices with HBN, a local tenant VRF imports route-target corresponding to the shared `external` VRF.

L3VNI:

Tenant	L3VNI	
tenant1	30001	On HBN VTEPs
tenant2	30002	On HBN VTEPs
tenant3	30003	On HBN VTEPs
tenant4	30004	On HBN VTEPs
tenant5	30005	On HBN VTEPs
tenant6	30006	On HBN VTEPs

Tenant	L3VNI	
external	60000	Configured on superspines and connects to external world

On BlueField devices with HBN, every tenant VRF on HBN one must import VNI of shared `external` VRF:

```
nv set vrf tenant1 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant1 router bgp route-import from-evpn route-target auto
nv set vrf tenant2 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant2 router bgp route-import from-evpn route-target auto
nv set vrf tenant3 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant3 router bgp route-import from-evpn route-target auto
nv set vrf tenant4 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant4 router bgp route-import from-evpn route-target auto
nv set vrf tenant5 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant5 router bgp route-import from-evpn route-target auto
nv set vrf tenant6 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant6 router bgp route-import from-evpn route-target auto
root@doca-hbn-service-bf3-s06-1-ipmi:/tmp#
```

On super spine switches (SS1 in this example), every remote tenant VRF that needs access to shared services has to be leaked to the shared `external` VRF.

```
nv set vrf external router bgp route-import from-evpn route-target ANY:30001
nv set vrf external router bgp route-import from-evpn route-target ANY:30002
nv set vrf external router bgp route-import from-evpn route-target ANY:30003
nv set vrf external router bgp route-import from-evpn route-target ANY:30004
nv set vrf external router bgp route-import from-evpn route-target ANY:30005
nv set vrf external router bgp route-import from-evpn route-target ANY:30006
nv set vrf external router bgp route-import from-evpn route-target auto
root@superspine1:mgmt:/home/cumulus#
```

All super spines in this case need this configuration.

DVNI Leaked Routes in VRF Table of HBN

ⓘ Each super spine here is advertising reachability providing 4-way overlay ECMP.

Kernel table for all tenant VRFs, showing the imported shared service:

```
root@doca-hbn-service-bf3-s06-1-ipmi:/tmp# ip -4 route show table all  6.0.0.4/32
6.0.0.4 table tenant1 proto bgp metric 20
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.12 ttl 0 tos 0 via 6.0.0.12 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.13 ttl 0 tos 0 via 6.0.0.13 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.14 ttl 0 tos 0 via 6.0.0.14 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.15 ttl 0 tos 0 via 6.0.0.15 dev vxlan48 weight 1 onlink
6.0.0.4 table tenant2 proto bgp metric 20
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.12 ttl 0 tos 0 via 6.0.0.12 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.13 ttl 0 tos 0 via 6.0.0.13 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.14 ttl 0 tos 0 via 6.0.0.14 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.15 ttl 0 tos 0 via 6.0.0.15 dev vxlan48 weight 1 onlink
6.0.0.4 table tenant3 proto bgp metric 20
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.12 ttl 0 tos 0 via 6.0.0.12 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.13 ttl 0 tos 0 via 6.0.0.13 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.14 ttl 0 tos 0 via 6.0.0.14 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.15 ttl 0 tos 0 via 6.0.0.15 dev vxlan48 weight 1 onlink
6.0.0.4 table tenant4 proto bgp metric 20
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.12 ttl 0 tos 0 via 6.0.0.12 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.13 ttl 0 tos 0 via 6.0.0.13 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.14 ttl 0 tos 0 via 6.0.0.14 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.15 ttl 0 tos 0 via 6.0.0.15 dev vxlan48 weight 1 onlink
6.0.0.4 table tenant5 proto bgp metric 20
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.12 ttl 0 tos 0 via 6.0.0.12 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.13 ttl 0 tos 0 via 6.0.0.13 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.14 ttl 0 tos 0 via 6.0.0.14 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.15 ttl 0 tos 0 via 6.0.0.15 dev vxlan48 weight 1 onlink
6.0.0.4 table tenant6 proto bgp metric 20
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.12 ttl 0 tos 0 via 6.0.0.12 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.13 ttl 0 tos 0 via 6.0.0.13 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.14 ttl 0 tos 0 via 6.0.0.14 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.15 ttl 0 tos 0 via 6.0.0.15 dev vxlan48 weight 1 onlink
root@doca-hbn-service-bf3-s06-1-ipmi:/tmp#
```

FRR RIB table:

```
root@doca-hbn-service-bf3-s06-1-ipmi:/tmp# vtysh

Hello, this is FRRouting (version 8.4.3).
Copyright 1996-2005 Kunihiro Ishiguro, et al.

doca-hbn-service-bf3-s06-1-ipmi# sh ip route vrf tenant1
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
       T - Table, A - Babel, D - SHARP, F - PBR, f - OpenFabric,
       Z - FRR,
       > - selected route, * - FIB route, q - queued, r - rejected, b - backup
       t - trapped, o - offload failure

VRF tenant1:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:36
B>* 6.0.0.4/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
  *                   via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
  *                   via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
  *                   via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
B>* 6.6.0.12/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
B>* 6.6.0.13/32 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
B>* 6.6.0.14/32 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
B>* 6.6.0.15/32 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
B>* 7.1.0.6/32 [20/0] via 6.0.0.6, vlan4052_l3 onlink, weight 1, 00:05:37
C>* 7.1.0.16/32 is directly connected, tenant1, 00:10:36
B>* 7.1.0.18/32 [20/0] via 6.0.0.18, vlan4052_l3 onlink, weight 1, 00:05:37
B>* 7.1.0.20/32 [20/0] via 6.0.0.20, vlan4052_l3 onlink, weight 1, 00:05:37
C>* 21.1.0.0/16 is directly connected, vlan101, 00:10:36
C * 21.1.0.0/16 [0/1024] is directly connected, vlan101-v0, 00:10:36
C * 21.2.0.0/16 [0/1024] is directly connected, vlan102-v0, 00:10:36
C>* 21.2.0.0/16 is directly connected, vlan102, 00:10:36
B>* 101.12.4.0/24 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
B>* 101.13.4.0/24 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
B>* 101.14.4.0/24 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
B>* 101.15.4.0/24 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
doca-hbn-service-bf3-s06-1-ipmi# sh ip route vrf all
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
       T - Table, A - Babel, D - SHARP, F - PBR, f - OpenFabric,
       Z - FRR,
       > - selected route, * - FIB route, q - queued, r - rejected, b - backup
       t - trapped, o - offload failure

VRF default:
B>* 6.0.0.6/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:06:47
  *                   via fe80::202:ff:fe00:27, p1_if, weight 1, 00:06:47
B>* 6.0.0.7/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:05:48
  *                   via fe80::202:ff:fe00:27, p1_if, weight 1, 00:05:48
B>* 6.0.0.8/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:05:38
  *                   via fe80::202:ff:fe00:27, p1_if, weight 1, 00:05:38
B>* 6.0.0.9/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:05:28
  *                   via fe80::202:ff:fe00:27, p1_if, weight 1, 00:05:28
B>* 6.0.0.10/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:06:49
B>* 6.0.0.11/32 [20/0] via fe80::202:ff:fe00:27, p1_if, weight 1, 00:06:47
B>* 6.0.0.12/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:06:47
  *                    via fe80::202:ff:fe00:27, p1_if, weight 1, 00:06:47
B>* 6.0.0.13/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:06:47
  *                    via fe80::202:ff:fe00:27, p1_if, weight 1, 00:06:47
B>* 6.0.0.14/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:06:47
  *                    via fe80::202:ff:fe00:27, p1_if, weight 1, 00:06:47
B>* 6.0.0.15/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:06:47
  *                    via fe80::202:ff:fe00:27, p1_if, weight 1, 00:06:47
C>* 6.0.0.16/32 is directly connected, lo, 00:10:42
B>* 6.0.0.18/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:06:47
  *                    via fe80::202:ff:fe00:27, p1_if, weight 1, 00:06:47
B>* 6.0.0.20/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:06:47
  *                    via fe80::202:ff:fe00:27, p1_if, weight 1, 00:06:47
B>* 192.168.0.0/24 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:05:48
  *                       via fe80::202:ff:fe00:27, p1_if, weight 1, 00:05:48

VRF internet1:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:42
B>* 8.1.0.6/32 [20/0] via 6.0.0.6, vlan4004_l3 onlink, weight 1, 00:05:43
C>* 8.1.0.16/32 is directly connected, internet1, 00:10:42
B>* 8.1.0.18/32 [20/0] via 6.0.0.18, vlan4004_l3 onlink, weight 1, 00:05:43
B>* 8.1.0.20/32 [20/0] via 6.0.0.20, vlan4004_l3 onlink, weight 1, 00:05:43

VRF mgmt:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:42
C>* 10.88.0.0/16 is directly connected, eth0, 00:10:42

VRF special1:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:42
B>* 9.1.0.6/32 [20/0] via 6.0.0.6, vlan4033_l3 onlink, weight 1, 00:05:43
C>* 9.1.0.16/32 is directly connected, special1, 00:10:42
B>* 9.1.0.18/32 [20/0] via 6.0.0.18, vlan4033_l3 onlink, weight 1, 00:05:43
B>* 9.1.0.20/32 [20/0] via 6.0.0.20, vlan4033_l3 onlink, weight 1, 00:05:43

VRF tenant1:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:42
B>* 6.0.0.4/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.12/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.13/32 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.14/32 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.15/32 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 7.1.0.6/32 [20/0] via 6.0.0.6, vlan4052_l3 onlink, weight 1, 00:05:43
C>* 7.1.0.16/32 is directly connected, tenant1, 00:10:42
```

```
B>* 7.1.0.18/32 [20/0] via 6.0.0.18, vlan4052_l3 onlink, weight 1, 00:05:43
B>* 7.1.0.20/32 [20/0] via 6.0.0.20, vlan4052_l3 onlink, weight 1, 00:05:43
C>* 21.1.0.0/16 is directly connected, vlan101, 00:10:42
C * 21.1.0.0/16 [0/1024] is directly connected, vlan101-v0, 00:10:42
C * 21.2.0.0/16 [0/1024] is directly connected, vlan102-v0, 00:10:42
C>* 21.2.0.0/16 is directly connected, vlan102, 00:10:42
B>* 101.12.4.0/24 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.13.4.0/24 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.14.4.0/24 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.15.4.0/24 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44

VRF tenant2:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:42
B>* 6.0.0.4/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.12/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.13/32 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.14/32 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.15/32 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 7.2.0.6/32 [20/0] via 6.0.0.6, vlan4037_l3 onlink, weight 1, 00:05:43
C>* 7.2.0.16/32 is directly connected, tenant2, 00:10:42
B>* 7.2.0.18/32 [20/0] via 6.0.0.18, vlan4037_l3 onlink, weight 1, 00:05:43
B>* 7.2.0.20/32 [20/0] via 6.0.0.20, vlan4037_l3 onlink, weight 1, 00:05:43
C * 22.1.0.0/16 [0/1024] is directly connected, vlan201-v0, 00:10:42
C>* 22.1.0.0/16 is directly connected, vlan201, 00:10:42
C * 22.2.0.0/16 [0/1024] is directly connected, vlan202-v0, 00:10:42
C>* 22.2.0.0/16 is directly connected, vlan202, 00:10:42
B>* 101.12.4.0/24 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.13.4.0/24 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.14.4.0/24 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.15.4.0/24 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44

VRF tenant3:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:42
B>* 6.0.0.4/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.12/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.13/32 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.14/32 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.15/32 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 7.3.0.6/32 [20/0] via 6.0.0.6, vlan4022_l3 onlink, weight 1, 00:05:43
C>* 7.3.0.16/32 is directly connected, tenant3, 00:10:42
B>* 7.3.0.18/32 [20/0] via 6.0.0.18, vlan4022_l3 onlink, weight 1, 00:05:43
B>* 7.3.0.20/32 [20/0] via 6.0.0.20, vlan4022_l3 onlink, weight 1, 00:05:43
C>* 23.17.0.0/16 is directly connected, pf0vf4_if.3, 00:10:42
B>* 23.19.0.0/16 [20/0] via 6.0.0.18, vlan4022_l3 onlink, weight 1, 00:05:43
B>* 23.21.0.0/16 [20/0] via 6.0.0.20, vlan4022_l3 onlink, weight 1, 00:05:43
B>* 101.12.4.0/24 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.13.4.0/24 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.14.4.0/24 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.15.4.0/24 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44

VRF tenant4:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:42
B>* 6.0.0.4/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.12/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.13/32 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.14/32 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.15/32 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 7.4.0.6/32 [20/0] via 6.0.0.6, vlan4017_l3 onlink, weight 1, 00:05:43
C>* 7.4.0.16/32 is directly connected, tenant4, 00:10:42
B>* 7.4.0.18/32 [20/0] via 6.0.0.18, vlan4017_l3 onlink, weight 1, 00:05:43
B>* 7.4.0.20/32 [20/0] via 6.0.0.20, vlan4017_l3 onlink, weight 1, 00:05:43
C>* 24.17.0.0/16 is directly connected, pf0vf4_if.4, 00:10:42
B>* 24.19.0.0/16 [20/0] via 6.0.0.18, vlan4017_l3 onlink, weight 1, 00:05:43
B>* 24.21.0.0/16 [20/0] via 6.0.0.20, vlan4017_l3 onlink, weight 1, 00:05:43
B>* 101.12.4.0/24 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.13.4.0/24 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.14.4.0/24 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.15.4.0/24 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44

VRF tenant5:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:42
B>* 6.0.0.4/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.12/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.13/32 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.14/32 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.15/32 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 7.5.0.6/32 [20/0] via 6.0.0.6, vlan4046_l3 onlink, weight 1, 00:05:43
C>* 7.5.0.16/32 is directly connected, tenant5, 00:10:42
B>* 7.5.0.18/32 [20/0] via 6.0.0.18, vlan4046_l3 onlink, weight 1, 00:05:43
B>* 7.5.0.20/32 [20/0] via 6.0.0.20, vlan4046_l3 onlink, weight 1, 00:05:43
C>* 25.17.0.0/16 is directly connected, pf0vf4_if.5, 00:10:42
B>* 25.19.0.0/16 [20/0] via 6.0.0.18, vlan4046_l3 onlink, weight 1, 00:05:43
B>* 25.21.0.0/16 [20/0] via 6.0.0.20, vlan4046_l3 onlink, weight 1, 00:05:43
B>* 101.12.4.0/24 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.13.4.0/24 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.14.4.0/24 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.15.4.0/24 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44

VRF tenant6:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:42
B>* 6.0.0.4/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
```

```
B>* 6.6.0.12/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.13/32 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.14/32 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.15/32 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 7.6.0.6/32 [20/0] via 6.0.0.6, vlan4041_l3 onlink, weight 1, 00:05:43
C>* 7.6.0.16/32 is directly connected, tenant6, 00:10:42
B>* 7.6.0.18/32 [20/0] via 6.0.0.18, vlan4041_l3 onlink, weight 1, 00:05:43
B>* 7.6.0.20/32 [20/0] via 6.0.0.20, vlan4041_l3 onlink, weight 1, 00:05:43
C>* 26.17.0.0/16 is directly connected, pf0vf4_if.6, 00:10:42
B>* 26.19.0.0/16 [20/0] via 6.0.0.18, vlan4041_l3 onlink, weight 1, 00:05:43
B>* 26.21.0.0/16 [20/0] via 6.0.0.20, vlan4041_l3 onlink, weight 1, 00:05:43
B>* 101.12.4.0/24 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.13.4.0/24 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.14.4.0/24 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.15.4.0/24 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
doca-hbn-service-bf3-s06-1-ipmi#
```

DVNI Debugging

BGP/Zebra debug:

```
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [GKC5Y-XBAX9] vrf tenant1: import evpn prefix [5]:[0]
:[32]:[6.0.0.4] parent 0xaaaafda63a90 flags 0x410
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [KZNVF-SX7KT] ... new pi dest 0xaaaafe524650 (l 2)
pi 0xaaaafe5ae400 (l 1, f 0x4010)
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [GKC5Y-XBAX9] vrf tenant2: import evpn prefix [5]:[0]
:[32]:[6.0.0.4] parent 0xaaaafda63a90 flags 0x410
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [KZNVF-SX7KT] ... new pi dest 0xaaaafe51c420 (l 2)
pi 0xaaaafe55d230 (l 1, f 0x4010)
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [GKC5Y-XBAX9] vrf tenant3: import evpn prefix [5]:[0]
:[32]:[6.0.0.4] parent 0xaaaafda63a90 flags 0x410
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [KZNVF-SX7KT] ... new pi dest 0xaaaafe51a670 (l 2)
pi 0xaaaafe674820 (l 1, f 0x4010)
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [GKC5Y-XBAX9] vrf tenant4: import evpn prefix [5]:[0]
:[32]:[6.0.0.4] parent 0xaaaafda63a90 flags 0x410
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [KZNVF-SX7KT] ... new pi dest 0xaaaafe519fb0 (l 2)
pi 0xaaaafe675e40 (l 1, f 0x4010)
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [GKC5Y-XBAX9] vrf tenant5: import evpn prefix [5]:[0]
:[32]:[6.0.0.4] parent 0xaaaafda63a90 flags 0x410
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [KZNVF-SX7KT] ... new pi dest 0xaaaafe55ae50 (l 2)
pi 0xaaaafe5482f0 (l 1, f 0x4010)
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [GKC5Y-XBAX9] vrf tenant6: import evpn prefix [5]:[0]
:[32]:[6.0.0.4] parent 0xaaaafda63a90 flags 0x410
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [KZNVF-SX7KT] ... new pi dest 0xaaaafdaf3590 (l 2)
pi 0xaaaafe48fbf0 (l 1, f 0x4010)
```

DVNI table:

```
root@doca-hbn-service-bf3-s06-1-ipmi:/tmp# cat /cumulus/nl2docad/run/software-tables/15
{
  "table": {
    "id": 15,
    "name": "HAL Downstream-VNI Table ",
    "count": 1,
    "records": [
    {
      "vni": 60000,
      "fid": 4098,
      "mark-for-del": 0,
      "vtep-users":
      {
        "count": 4,
        "vtep-user-list": [
          {
            "dest-vtep": "6.0.0.12",
            "dest-mac": "44:38:39:f0:00:12",
            "is-dmac-null": 0,
            "ref-cnt": 36
          },
          {
            "dest-vtep": "6.0.0.14",
            "dest-mac": "44:38:39:f0:00:14",
            "is-dmac-null": 0,
            "ref-cnt": 36
          },
          {
            "dest-vtep": "6.0.0.13",
            "dest-mac": "44:38:39:f0:00:13",
            "is-dmac-null": 0,
            "ref-cnt": 36
          },
          {
            "dest-vtep": "6.0.0.15",
            "dest-mac": "44:38:39:f0:00:15",
            "is-dmac-null": 0,
            "ref-cnt": 36
          }
        ]
      }
    }
   ]
  }
}root@doca-hbn-service-bf3-s06-1-ipmi:/tmp#
```

Sample DVNI Configuration

HBN configuration example for BlueField devices:

```
root@doca-hbn-service-bf3-s06-1-ipmi:/tmp# nv config show -o commands
nv set bridge domain br_default vlan 101 vni 10101
nv set bridge domain br_default vlan 102 vni 10102
nv set bridge domain br_default vlan 201 vni 10201
nv set bridge domain br_default vlan 202 vni 10202
nv set evpn enable on
nv set evpn route-advertise svi-ip off
nv set interface ilan3200 ip vrf internet1
nv set interface ilan3200 vlan 3200
nv set interface ilan3200,slan3201,vlan101-102,201-202,3001-3006 base-interface br_default
nv set interface ilan3200,slan3201,vlan101-102,201-202,3001-3006 type svi
nv set interface lo ip address 6.0.0.16/32
nv set interface lo ip address 2001::16/128
nv set interface lo type loopback
nv set interface p0_if,p1_if,pf0hpf_if,pf0vf0_if,pf0vf1_if,pf0vf2_if,pf0vf3_if,pf0vf4_if,pf1hpf_if type swp
nv set interface pf0vf0_if bridge domain br_default access 101
nv set interface pf0vf1_if bridge domain br_default access 102
nv set interface pf0vf2_if bridge domain br_default access 201
nv set interface pf0vf3_if bridge domain br_default access 202
nv set interface pf0vf4_if.3 ip address 23.17.0.16/16
nv set interface pf0vf4_if.3 ip address 2020:0:3:17::16/64
nv set interface pf0vf4_if.3 vlan 3
nv set interface pf0vf4_if.3,vlan3003 ip vrf tenant3
nv set interface pf0vf4_if.3-6 base-interface pf0vf4_if
nv set interface pf0vf4_if.3-6 type sub
nv set interface pf0vf4_if.4 ip address 24.17.0.16/16
nv set interface pf0vf4_if.4 ip address 2020:0:4:17::16/64
nv set interface pf0vf4_if.4 vlan 4
nv set interface pf0vf4_if.4,vlan3004 ip vrf tenant4
nv set interface pf0vf4_if.5 ip address 25.17.0.16/16
nv set interface pf0vf4_if.5 ip address 2020:0:5:17::16/64
nv set interface pf0vf4_if.5 vlan 5
nv set interface pf0vf4_if.5,vlan3005 ip vrf tenant5
nv set interface pf0vf4_if.6 ip address 26.17.0.16/16
nv set interface pf0vf4_if.6 ip address 2020:0:6:17::16/64
nv set interface pf0vf4_if.6 vlan 6
nv set interface pf0vf4_if.6,vlan3006 ip vrf tenant6
nv set interface slan3201 ip vrf special1
nv set interface slan3201 vlan 3201
nv set interface vlan101 ip address 21.1.0.16/16
nv set interface vlan101 ip address 2020:0:1:1::16/64
nv set interface vlan101 ip vrr address 21.1.0.250/16
nv set interface vlan101 ip vrr address 2020:0:1:1::250/64
nv set interface vlan101 ip vrr mac-address 00:00:01:00:00:65
nv set interface vlan101 vlan 101
nv set interface vlan101-102,201-202 ip vrr enable on
nv set interface vlan101-102,3001 ip vrf tenant1
nv set interface vlan102 ip address 21.2.0.16/16
nv set interface vlan102 ip address 2020:0:1:2::16/64
nv set interface vlan102 ip vrr address 21.2.0.250/16
nv set interface vlan102 ip vrr address 2020:0:1:2::250/64
nv set interface vlan102 ip vrr mac-address 00:00:01:00:00:66
nv set interface vlan102 vlan 102
nv set interface vlan201 ip address 22.1.0.16/16
nv set interface vlan201 ip address 2020:0:2:1::16/64
nv set interface vlan201 ip vrr address 22.1.0.250/16
nv set interface vlan201 ip vrr address 2020:0:2:1::250/64
nv set interface vlan201 ip vrr mac-address 00:00:02:00:00:c9
nv set interface vlan201 vlan 201
nv set interface vlan201-202,3002 ip vrf tenant2
nv set interface vlan202 ip address 22.2.0.16/16
nv set interface vlan202 ip address 2020:0:2:2::16/64
nv set interface vlan202 ip vrr address 22.2.0.250/16
nv set interface vlan202 ip vrr address 2020:0:2:2::250/64
nv set interface vlan202 ip vrr mac-address 00:00:02:00:00:ca
nv set interface vlan202 vlan 202
nv set interface vlan3001 vlan 3001
nv set interface vlan3002 vlan 3002
nv set interface vlan3003 vlan 3003
nv set interface vlan3004 vlan 3004
nv set interface vlan3005 vlan 3005
nv set interface vlan3006 vlan 3006
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan source address 6.0.0.16
nv set platform
nv set router bgp autonomous-system 65011
nv set router bgp enable on
nv set router bgp router-id 6.0.0.16
nv set router vrr enable on
nv set system config snippet
nv set system global
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor 6.0.0.7 peer-group rservers
nv set vrf default router bgp neighbor 6.0.0.7 type numbered
nv set vrf default router bgp neighbor 6.0.0.8 peer-group rservers
nv set vrf default router bgp neighbor 6.0.0.8 type numbered
nv set vrf default router bgp neighbor 6.0.0.9 peer-group rservers
nv set vrf default router bgp neighbor 6.0.0.9 type numbered
nv set vrf default router bgp neighbor p0_if peer-group fabric
nv set vrf default router bgp neighbor p0_if type unnumbered
nv set vrf default router bgp neighbor p1_if peer-group fabric
nv set vrf default router bgp neighbor p1_if type unnumbered
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group fabric bfd detect-multiplier 3
nv set vrf default router bgp peer-group fabric bfd enable on
nv set vrf default router bgp peer-group fabric bfd min-rx-interval 1000
nv set vrf default router bgp peer-group fabric bfd min-tx-interval 1000
nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp peer-group rservers address-family ipv4-unicast enable off
nv set vrf default router bgp peer-group rservers address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group rservers address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group rservers multihop-ttl 10
nv set vrf default router bgp peer-group rservers remote-as external
nv set vrf default router bgp peer-group rservers update-source lo
nv set vrf internet1 evpn enable on
nv set vrf internet1 evpn vni 42000
nv set vrf internet1 loopback ip address 8.1.0.16/32
nv set vrf internet1 loopback ip address 2008:0:1::16/64
nv set vrf internet1 router bgp address-family ipv4-unicast enable on
nv set vrf internet1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf internet1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf internet1 router bgp address-family ipv6-unicast enable on
nv set vrf internet1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf internet1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf internet1 router bgp enable on
nv set vrf special1 evpn enable on
nv set vrf special1 evpn vni 42001
nv set vrf special1 loopback ip address 9.1.0.16/32
nv set vrf special1 loopback ip address 2009:0:1::16/64
nv set vrf special1 router bgp address-family ipv4-unicast enable on
nv set vrf special1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf special1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf special1 router bgp address-family ipv6-unicast enable on
nv set vrf special1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf special1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf special1 router bgp enable on
nv set vrf tenant1 evpn enable on
nv set vrf tenant1 evpn vni 30001
nv set vrf tenant1 loopback ip address 7.1.0.16/32
nv set vrf tenant1 loopback ip address 2007:0:1::16/64
nv set vrf tenant1 router bgp address-family ipv4-unicast enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp enable on
nv set vrf tenant1 router bgp neighbor 21.1.0.17 peer-group hostgroup
nv set vrf tenant1 router bgp neighbor 21.1.0.17 type numbered
nv set vrf tenant1 router bgp peer-group hostgroup address-family ipv4-unicast enable on
nv set vrf tenant1 router bgp peer-group hostgroup address-family ipv6-unicast enable on
nv set vrf tenant1 router bgp peer-group hostgroup remote-as external
nv set vrf tenant1 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant1 router bgp route-import from-evpn route-target auto
nv set vrf tenant1 router bgp router-id 6.0.0.16
nv set vrf tenant2 evpn enable on
nv set vrf tenant2 evpn vni 30002
nv set vrf tenant2 loopback ip address 7.2.0.16/32
nv set vrf tenant2 loopback ip address 2007:0:2::16/64
nv set vrf tenant2 router bgp address-family ipv4-unicast enable on
nv set vrf tenant2 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant2 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant2 router bgp address-family ipv6-unicast enable on
nv set vrf tenant2 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant2 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant2 router bgp enable on
nv set vrf tenant2 router bgp neighbor 22.1.0.17 peer-group hostgroup
nv set vrf tenant2 router bgp neighbor 22.1.0.17 type numbered
nv set vrf tenant2 router bgp peer-group hostgroup address-family ipv4-unicast enable on
nv set vrf tenant2 router bgp peer-group hostgroup address-family ipv6-unicast enable on
nv set vrf tenant2 router bgp peer-group hostgroup remote-as external
nv set vrf tenant2 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant2 router bgp route-import from-evpn route-target auto
nv set vrf tenant2 router bgp router-id 6.0.0.16
nv set vrf tenant3 evpn enable on
nv set vrf tenant3 evpn vni 30003
nv set vrf tenant3 loopback ip address 7.3.0.16/32
nv set vrf tenant3 loopback ip address 2007:0:3::16/64
nv set vrf tenant3 router bgp address-family ipv4-unicast enable on
nv set vrf tenant3 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant3 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant3 router bgp address-family ipv6-unicast enable on
nv set vrf tenant3 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant3 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant3 router bgp enable on
nv set vrf tenant3 router bgp neighbor 23.17.0.17 peer-group hostgroup
nv set vrf tenant3 router bgp neighbor 23.17.0.17 type numbered
nv set vrf tenant3 router bgp peer-group hostgroup address-family ipv4-unicast enable on
nv set vrf tenant3 router bgp peer-group hostgroup address-family ipv6-unicast enable on
nv set vrf tenant3 router bgp peer-group hostgroup remote-as external
nv set vrf tenant3 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant3 router bgp route-import from-evpn route-target auto
nv set vrf tenant3 router bgp router-id 6.0.0.16
nv set vrf tenant3 table auto
nv set vrf tenant4 evpn enable on
nv set vrf tenant4 evpn vni 30004
nv set vrf tenant4 loopback ip address 7.4.0.16/32
nv set vrf tenant4 loopback ip address 2007:0:4::16/64
nv set vrf tenant4 router bgp address-family ipv4-unicast enable on
nv set vrf tenant4 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant4 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant4 router bgp address-family ipv6-unicast enable on
nv set vrf tenant4 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant4 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant4 router bgp enable on
nv set vrf tenant4 router bgp neighbor 24.17.0.17 peer-group hostgroup
nv set vrf tenant4 router bgp neighbor 24.17.0.17 type numbered
nv set vrf tenant4 router bgp peer-group hostgroup address-family ipv4-unicast enable on
nv set vrf tenant4 router bgp peer-group hostgroup address-family ipv6-unicast enable on
nv set vrf tenant4 router bgp peer-group hostgroup remote-as external
nv set vrf tenant4 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant4 router bgp route-import from-evpn route-target auto
nv set vrf tenant4 router bgp router-id 6.0.0.16
nv set vrf tenant4 table auto
nv set vrf tenant5 evpn enable on
nv set vrf tenant5 evpn vni 30005
nv set vrf tenant5 loopback ip address 7.5.0.16/32
nv set vrf tenant5 loopback ip address 2007:0:5::16/64
nv set vrf tenant5 router bgp address-family ipv4-unicast enable on
nv set vrf tenant5 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant5 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant5 router bgp address-family ipv6-unicast enable on
nv set vrf tenant5 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant5 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant5 router bgp enable on
nv set vrf tenant5 router bgp neighbor 25.17.0.17 peer-group hostgroup
nv set vrf tenant5 router bgp neighbor 25.17.0.17 type numbered
nv set vrf tenant5 router bgp peer-group hostgroup address-family ipv4-unicast enable on
nv set vrf tenant5 router bgp peer-group hostgroup address-family ipv6-unicast enable on
nv set vrf tenant5 router bgp peer-group hostgroup remote-as external
nv set vrf tenant5 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant5 router bgp route-import from-evpn route-target auto
nv set vrf tenant5 router bgp router-id 6.0.0.16
nv set vrf tenant5 table auto
nv set vrf tenant6 evpn enable on
nv set vrf tenant6 evpn vni 30006
nv set vrf tenant6 loopback ip address 7.6.0.16/32
nv set vrf tenant6 loopback ip address 2007:0:6::16/64
nv set vrf tenant6 router bgp address-family ipv4-unicast enable on
nv set vrf tenant6 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant6 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant6 router bgp address-family ipv6-unicast enable on
nv set vrf tenant6 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant6 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant6 router bgp enable on
nv set vrf tenant6 router bgp neighbor 26.17.0.17 peer-group hostgroup
nv set vrf tenant6 router bgp neighbor 26.17.0.17 type numbered
nv set vrf tenant6 router bgp peer-group hostgroup address-family ipv4-unicast enable on
nv set vrf tenant6 router bgp peer-group hostgroup address-family ipv6-unicast enable on
nv set vrf tenant6 router bgp peer-group hostgroup remote-as external
nv set vrf tenant6 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant6 router bgp route-import from-evpn route-target auto
nv set vrf tenant6 router bgp router-id 6.0.0.16
nv set vrf tenant6 table auto
root@doca-hbn-service-bf3-s06-1-ipmi:/tmp#
```

SS1 switch configuration example:

```
root@superspine1:mgmt:/home/cumulus# nv config show -o commands
nv set bridge domain br_default vlan 101 vni 10101
nv set bridge domain br_default vlan 102 vni 10102
nv set bridge domain br_default vlan 201 vni 10201
nv set bridge domain br_default vlan 202 vni 10202
nv set evpn enable on
nv set interface eth0 ip address 192.168.0.15/24
nv set interface eth0 ip gateway 192.168.0.2
nv set interface eth0 type eth
nv set interface lo ip address 6.0.0.12/32
nv set interface lo ip address 2001::12/128
nv set interface lo type loopback
nv set interface swp1-6 type swp
nv set interface swp6 ip address 101.12.4.12/24
nv set interface swp6 ip address 2101:12::4:12/112
nv set interface swp6 ip vrf external
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan source address 6.0.0.12
nv set platform
nv set router bgp autonomous-system 65300
nv set router bgp enable on
nv set router bgp router-id 6.0.0.12
nv set system config snippet
nv set system global system-mac 44:38:39:f0:00:12
nv set system hostname superspine1
nv set system ssh-server permit-root-login enabled
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor swp1 peer-group fabric
nv set vrf default router bgp neighbor swp1 type unnumbered
nv set vrf default router bgp neighbor swp2 peer-group fabric
nv set vrf default router bgp neighbor swp2 type unnumbered
nv set vrf default router bgp neighbor swp3 peer-group rservers
nv set vrf default router bgp neighbor swp3 type unnumbered
nv set vrf default router bgp neighbor swp4 peer-group rservers
nv set vrf default router bgp neighbor swp4 type unnumbered
nv set vrf default router bgp neighbor swp5 peer-group rservers
nv set vrf default router bgp neighbor swp5 type unnumbered
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group fabric bfd detect-multiplier 3
nv set vrf default router bgp peer-group fabric bfd enable on
nv set vrf default router bgp peer-group fabric bfd min-rx-interval 1000
nv set vrf default router bgp peer-group fabric bfd min-tx-interval 1000
nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp peer-group rservers address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group rservers address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group rservers address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group rservers remote-as external
nv set vrf external evpn enable on
nv set vrf external evpn vni 60000
nv set vrf external loopback ip address 6.6.0.12/32
nv set vrf external loopback ip address 2006:0:6::12/64
nv set vrf external router bgp address-family ipv4-unicast enable on
nv set vrf external router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf external router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf external router bgp address-family ipv6-unicast enable on
nv set vrf external router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf external router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf external router bgp address-family l2vpn-evpn enable on
nv set vrf external router bgp enable on
nv set vrf external router bgp neighbor swp6 peer-group peer-group-fw
nv set vrf external router bgp neighbor swp6 type unnumbered
nv set vrf external router bgp peer-group peer-group-fw address-family ipv4-unicast enable on
nv set vrf external router bgp peer-group peer-group-fw address-family ipv6-unicast enable on
nv set vrf external router bgp peer-group peer-group-fw remote-as external
nv set vrf external router bgp route-import from-evpn route-target ANY:30001
nv set vrf external router bgp route-import from-evpn route-target ANY:30002
nv set vrf external router bgp route-import from-evpn route-target ANY:30003
nv set vrf external router bgp route-import from-evpn route-target ANY:30004
nv set vrf external router bgp route-import from-evpn route-target ANY:30005
nv set vrf external router bgp route-import from-evpn route-target ANY:30006
nv set vrf external router bgp route-import from-evpn route-target auto
root@superspine1:mgmt:/home/cumulus#
```

17.8.4.3.3.7 Gateway Application Using Downstream VNI and Subinterface

A DPU running the HBN service can be deployed in the role of a border gateway using a combination of HBN features, specifically, EVPN symmetric routing, downstream VNI, VRF route-leaking, and VLAN sub-interfaces. Such a border gateway can do the northbound traffic handoff (to external networks or the Internet) for one or more tenants. In this gateway configuration, the BlueField's uplinks must carry both the tenant traffic which would be in the "overlay" and VXLAN-encapsulated, as well as traffic to and from the external network or Internet, which would be direct-routed in the "underlay". This is accomplished by configuring and running VXLAN-EVPN on the uplink interfaces while configuring and using additional VLAN sub-interfaces on those same uplinks for the traffic to and from external networks. These VLAN sub-interfaces would be configured into an Internet or external VRF for separation from the VXLAN-encapsulated traffic which is carried over the default VRF.

With a BlueField running HBN able to act as a border gateway, there is no longer a dependence on physical switches and routers to terminate VXLAN traffic and perform this role, hence the requirements on the underlying network is simply to provide end-to-end IP/UDP connectivity and facilitate the setup of overlay networks on top. Additionally, multiple border gateways can be easily deployed in the network, including dedicated gateways per tenant or shared gateways for groups of tenants.

> Since HBN currently does not support network address translation (NAT), a dedicated border gateway must be deployed per tenant, for those tenants that have overlapping IP addresses.

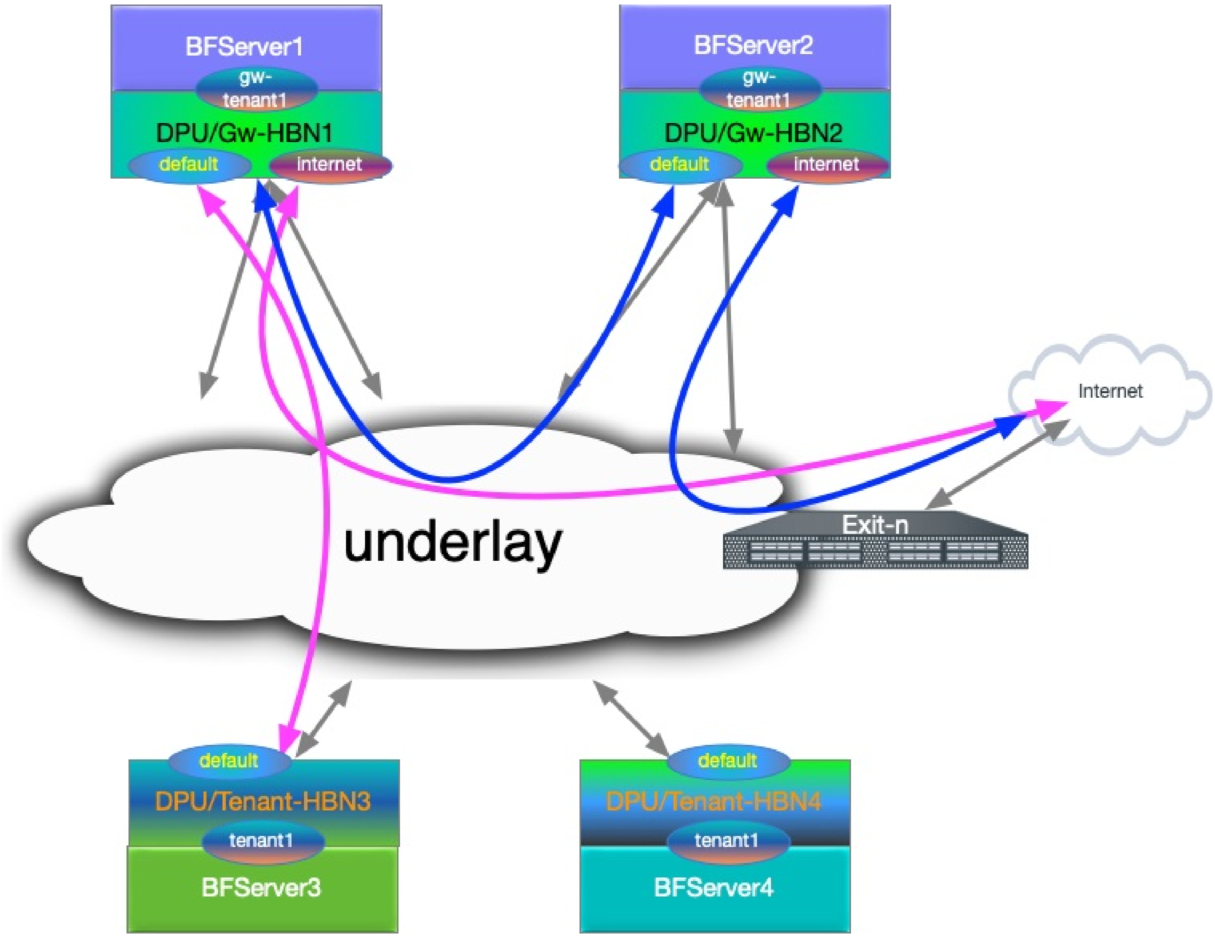
For more details and configuration of some of the key features that together enable the border gateway functionality, refer to sections on Downstream VNIs and VLAN Subinterfaces.

Gateway Application Example

The following topology diagram and associated configuration snippets show two different use cases of border gateway deployment:

- `tenant1` is an example of a tenant hosted on a server(s) with a non-gateway BlueField, using a dedicated border gateway on BlueField Gw-HBN1 for Internet connectivity. Traffic flow to and from the Internet for this tenant is marked in pink.

- `gw_tenant1` is an example of a tenant hosted on a server(s) with a gateway BlueField. In this case, the border gateway for this tenant is provided by BlueField Gw-HBN2. Traffic flow to and from the Internet for this tenant is depicted in blue.



L3 VNI Origin Map

HBN	VRF	L3 VNI
`gw-hbn1` and `gw-hbn2`	`internet1`	`10000`
`gw-hbn1` and `gw-hbn2`	`gw_tenant1`	`30000`
`tenant-hbn3` and `tenant-hbn4`	`tenant1`	`20000`

Configuration Snippet for Internet VRF

- Internet VRF is established in BGP sessions using sub-interface features with underlay switches (i.e., `p0_if.60` and `p1_if.60`)
- The Internet VRF also imports all the tenant VRFs (local and remote) using the downstream VNI feature with from-EVPN syntax

```
nv set interface p0_if.60,p1_if.60,vlan10 ip vrf internet1
nv set vrf internet1 evpn enable on
nv set vrf internet1 evpn vni 10000
     nv set vrf internet1 loopback ip address 6.2.0.1/32
nv set vrf internet1 loopback ip address 2001:cafe:feed::1/128
nv set vrf internet1 router bgp address-family ipv4-unicast enable on
nv set vrf internet1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf internet1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf internet1 router bgp address-family ipv6-unicast enable on
nv set vrf internet1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf internet1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf internet1 router bgp address-family l2vpn-evpn enable on
nv set vrf internet1 router bgp autonomous-system 65552
nv set vrf internet1 router bgp enable on
nv set vrf internet1 router bgp neighbor p0_if.60 capabilities source-address internet1
nv set vrf internet1 router bgp neighbor p0_if.60 peer-group l3_pg1
nv set vrf internet1 router bgp neighbor p0_if.60 type unnumbered
nv set vrf internet1 router bgp neighbor p1_if.60 capabilities source-address internet1
nv set vrf internet1 router bgp neighbor p1_if.60 peer-group l3_pg1
nv set vrf internet1 router bgp neighbor p1_if.60 type unnumbered
nv set vrf internet1 router bgp peer-group l3_pg1 address-family ipv4-unicast enable on
nv set vrf internet1 router bgp peer-group l3_pg1 address-family ipv6-unicast enable on
nv set vrf internet1 router bgp peer-group l3_pg1 remote-as external
nv set vrf internet1 router bgp route-export to-evpn route-target 65552:10000
nv set vrf internet1 router bgp route-import from-evpn route-target ANY:20000
nv set vrf internet1 router bgp route-import from-evpn route-target ANY:30000
nv set vrf internet1 router bgp route-import from-evpn route-target auto
nv set vrf internet1 router bgp router-id 27.0.0.5
```

Configuration Snippet for Gateway Local Tenant

- `gw_tenant` is stretched across 2 gateway and connected using L3 VNI
- `gw_tenant` has multiple SVIs, which are represented as `vlan30` and `vlan31` SVIs
- Internet L3 VNI is imported using DVNI. The example also explicitly adds route targets using auto.

`gw_tenant` VRF:

```
nv set interface vlan30-31 ip vrf gw_tenant1
nv set vrf gw_tenant1 evpn enable on
nv set vrf gw_tenant1 evpn vni 30000
nv set vrf gw_tenant1 loopback ip address 15.3.0.1/32
nv set vrf gw_tenant1 loopback ip address 2001:bad:c0de::1/128
nv set vrf gw_tenant1 router bgp address-family ipv4-unicast enable on
nv set vrf gw_tenant1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf gw_tenant1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf gw_tenant1 router bgp address-family ipv6-unicast enable on
nv set vrf gw_tenant1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf gw_tenant1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf gw_tenant1 router bgp address-family l2vpn-evpn enable on
nv set vrf gw_tenant1 router bgp autonomous-system 65552
nv set vrf gw_tenant1 router bgp enable on
nv set vrf gw_tenant1 router bgp route-export to-evpn route-target 65552:30000
nv set vrf gw_tenant1 router bgp route-import from-evpn route-target ANY:10000
nv set vrf gw_tenant1 router bgp route-import from-evpn route-target auto
nv set vrf gw_tenant1 router bgp router-id 27.0.0.5
```

Configuration Snippet for Remote Tenant

- `tenant1` is stretched across 2 remote HBN VTEP and connected using L3 VNI
- `tenant1` is importing Internet L3 VNI routes in `tenant1` and adding its own using route-target auto

Tenant VRF:

```
nv set interface vlan20-21 ip vrf tenant1
nv set vrf tenant1 evpn enable on
nv set vrf tenant1 evpn vni 20000
nv set vrf tenant1 loopback ip address 15.1.0.1/32
nv set vrf tenant1 loopback ip address 2001:c001:c0de::1/128
nv set vrf tenant1 router bgp address-family ipv4-unicast enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp address-family l2vpn-evpn enable on
nv set vrf tenant1 router bgp autonomous-system 6300656
nv set vrf tenant1 router bgp enable on
nv set vrf tenant1 router bgp route-export to-evpn route-target 6300656:20000
nv set vrf tenant1 router bgp route-import from-evpn route-target ANY:10000
nv set vrf tenant1 router bgp route-import from-evpn route-target auto
nv set vrf tenant1 router bgp router-id 27.0.0.17
```

HBN Accelerated Routing Plan

The following subsections pick a few IP endpoints from the code snippets above and examine their route distribution.

- The gateway devices have a remote tenant
- Internet route is injected using the default originator from the exit node

Gateway-1 Route Info

- BGP sharing the uplink via a sub-interface feature in the Internet VRF.

  gateway1 - External Routes Internet VRF

  ```
  root@hbn:/# ip -4 route show vrf internet1 default
  default proto bgp metric 20
      nexthop via 169.254.0.1 dev p0_if.60 weight 1 onlink
      nexthop via 169.254.0.1 dev p1_if.60 weight 1 onlink

  root@hbn:/# ip -6 route show vrf internet1 default
  default proto bgp metric 20 pref medium
      nexthop via fe80::202:ff:fe00:1b dev p0_if.60 weight 1
      nexthop via fe80::202:ff:fe00:23 dev p1_if.60 weight 1
  ```

- Local Tenant routing information: The Internet is reached using L3 VNI via a peer gateway.

  gateway1 - External Routes gw_tenant VRF

  ```
  root@hbn:/# ip -4 route show vrf gw_tenant1  default
  default  encap ip id 10000 src 0.0.0.0 dst 27.0.0.7 ttl 0 tos 0 via 27.0.0.7 dev vxlan48 proto bgp metric
  20 onlink

  root@hbn:/# ip -6 route show vrf gw_tenant1  default
  default  encap ip id 10000 src 0.0.0.0 dst 27.0.0.7 ttl 0 tos 0 via ::ffff:27.0.0.7 dev vxlan48 proto bgp
  metric 20 onlink pref medium
  ```

- Remote tenant routing reachability via `gateway1` using DVNI CFG.
- Considering an IP endpoint from the remote `tenant1` VRF on Tenant-HBN3.

  gateway1 - Routes Internet VRF

  ```
  root@hbn:/# ip -4 route show vrf internet1  15.1.0.1/32
  15.1.0.1  encap ip id 20000 src 0.0.0.0 dst 27.0.0.17 ttl 0 tos 0 via 27.0.0.17 dev vxlan48 proto bgp
  metric 20 onlink

  root@hbn:/# ip -6 route show vrf internet1  2001:c001:c0de::1/128
  2001:c001:c0de::1  encap ip id 20000 src 0.0.0.0 dst 27.0.0.17 ttl 0 tos 0 via ::ffff:27.0.0.17 dev vxlan48
  proto bgp metric 20 onlink pref medium
  ```

Tenant-HBN3 Route Info

- IP endpoint as `gateway1` VRF loopback and DVNI handoff for the VNI is reaching the `gateway1` node.

  tenant-hbn3 - Routes tenant VRF

  ```
  root@hbn:/# ip -4 route show vrf tenant1 6.2.0.1/32
  6.2.0.1  encap ip id 10000 src 0.0.0.0 dst 27.0.0.5 ttl 0 tos 0 via 27.0.0.5 dev vxlan48 proto bgp metric
  20 onlink

  root@hbn:/# ip -6 route show vrf tenant1 2001:cafe:feed::1/128
  ```

```
2001:cafe:feed::1  encap ip id 10000 src 0.0.0.0 dst 27.0.0.5 ttl 0 tos 0 via ::ffff:27.0.0.5 dev vxlan48
proto bgp metric 20 onlink pref medium
```

- Internet VRF default route is reaching the remote tenant VRF.

  tenant-hbn3 external - Routes tenant VRF

  ```
  root@hbn:/# ip -4 route show vrf tenant1 default
  default proto bgp metric 20
      nexthop  encap ip id 10000 src 0.0.0.0 dst 27.0.0.5 ttl 0 tos 0 via 27.0.0.5 dev vxlan48 weight 1
   onlink
      nexthop  encap ip id 10000 src 0.0.0.0 dst 27.0.0.7 ttl 0 tos 0 via 27.0.0.7 dev vxlan48 weight 1
   onlink

  root@hbn:/# ip -6 route show vrf tenant1 default
  default proto bgp metric 20 pref medium
      nexthop  encap ip id 10000 src 0.0.0.0 dst 27.0.0.5 ttl 0 tos 0 via ::ffff:27.0.0.5 dev vxlan48 weight
  1 onlink
      nexthop  encap ip id 10000 src 0.0.0.0 dst 27.0.0.7 ttl 0 tos 0 via ::ffff:27.0.0.7 dev vxlan48 weight
  1 onlink
  ```

Gateway and Tenant Complete Configuration Example

Gateway-1 Full Configuration

Gateway-HBN-1

```
nv set bridge domain br_default encap 802.1Q
nv set bridge domain br_default type vlan-aware
nv set bridge domain br_default untagged 1
nv set bridge domain br_default vlan 10,30-31
nv set evpn enable on
nv set interface lo ip address 27.0.0.5/32
nv set interface lo ip address 2001:c001:ff:f00d::5/128
nv set interface lo type loopback
nv set interface p0_if,p1_if,pf0hpf_if,pf0vf0_if,pf0vf1_if,pf0vf2_if,pf0vf3_if,pf0vf4_if,pf1hpf_if type swp
nv set interface p0_if.60 base-interface p0_if
nv set interface p0_if.60,p1_if.60 type sub
nv set interface p0_if.60,p1_if.60 vlan 60
nv set interface p0_if.60,p1_if.60,vlan10 ip vrf internet1
nv set interface p1_if.60 base-interface p1_if
nv set interface pf0hpf_if bridge domain br_default access 30
nv set interface pf0vf0_if bridge domain br_default access 31
nv set interface vlan10 ip address 12.2.0.1/24
nv set interface vlan10 ip address 2001:c001:d00d::1/96
nv set interface vlan10 vlan 10
nv set interface vlan10,30-31 ip ipv4 forward on
nv set interface vlan10,30-31 ip ipv6 forward on
nv set interface vlan10,30-31 type svi
nv set interface vlan30 ip address 45.3.0.1/24
nv set interface vlan30 ip address 2001:b055:b00c::1/96
nv set interface vlan30 vlan 30
nv set interface vlan30-31 ip vrf gw_tenant1
nv set interface vlan31 ip address 45.3.1.1/24
nv set interface vlan31 ip address 2001:b055:b00c::1:0:1/96
nv set interface vlan31 vlan 31
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan mac-learning off
nv set nve vxlan source address 27.0.0.5
nv set platform
nv set router bgp enable on
nv set system config snippet
nv set system global anycast-mac 44:38:39:42:42:17
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp autonomous-system 65552
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor 27.0.0.11 peer-group rs_client
nv set vrf default router bgp neighbor 27.0.0.11 type numbered
nv set vrf default router bgp neighbor 27.0.0.12 peer-group rs_client
nv set vrf default router bgp neighbor 27.0.0.12 type numbered
nv set vrf default router bgp neighbor p0_if capabilities source-address lo
nv set vrf default router bgp neighbor p0_if peer-group fabric
nv set vrf default router bgp neighbor p0_if type unnumbered
nv set vrf default router bgp neighbor p1_if capabilities source-address lo
nv set vrf default router bgp neighbor p1_if peer-group fabric
nv set vrf default router bgp neighbor p1_if type unnumbered
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn enable off
```

```
nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp peer-group fabric timers connection-retry 5
nv set vrf default router bgp peer-group fabric timers hold 30
nv set vrf default router bgp peer-group fabric timers keepalive 10
nv set vrf default router bgp peer-group rs_client address-family ipv4-unicast enable off
nv set vrf default router bgp peer-group rs_client address-family ipv6-unicast enable off
nv set vrf default router bgp peer-group rs_client address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group rs_client address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group rs_client multihop-ttl 5
nv set vrf default router bgp peer-group rs_client remote-as external
nv set vrf default router bgp peer-group rs_client timers connection-retry 5
nv set vrf default router bgp peer-group rs_client timers hold 30
nv set vrf default router bgp peer-group rs_client timers keepalive 10
nv set vrf default router bgp router-id 27.0.0.5
nv set vrf gw_tenant1 evpn enable on
nv set vrf gw_tenant1 evpn vni 30000
nv set vrf gw_tenant1 loopback ip address 15.3.0.1/32
nv set vrf gw_tenant1 loopback ip address 2001:bad:c0de::1/128
nv set vrf gw_tenant1 router bgp address-family ipv4-unicast enable on
nv set vrf gw_tenant1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf gw_tenant1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf gw_tenant1 router bgp address-family ipv6-unicast enable on
nv set vrf gw_tenant1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf gw_tenant1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf gw_tenant1 router bgp address-family l2vpn-evpn enable on
nv set vrf gw_tenant1 router bgp autonomous-system 65552
nv set vrf gw_tenant1 router bgp enable on
nv set vrf gw_tenant1 router bgp route-export to-evpn route-target 65552:30000
nv set vrf gw_tenant1 router bgp route-import from-evpn route-target ANY:10000
nv set vrf gw_tenant1 router bgp route-import from-evpn route-target auto
nv set vrf gw_tenant1 router bgp router-id 27.0.0.5
nv set vrf internet1 evpn enable on
nv set vrf internet1 evpn vni 10000
nv set vrf internet1 loopback ip address 6.2.0.1/32
nv set vrf internet1 loopback ip address 2001:cafe:feed::1/128
nv set vrf internet1 router bgp address-family ipv4-unicast enable on
nv set vrf internet1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf internet1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf internet1 router bgp address-family ipv6-unicast enable on
nv set vrf internet1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf internet1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf internet1 router bgp address-family l2vpn-evpn enable on
nv set vrf internet1 router bgp autonomous-system 65552
nv set vrf internet1 router bgp enable on
nv set vrf internet1 router bgp neighbor p0_if.60 capabilities source-address internet1
nv set vrf internet1 router bgp neighbor p0_if.60 peer-group l3_pg1
nv set vrf internet1 router bgp neighbor p0_if.60 type unnumbered
nv set vrf internet1 router bgp neighbor p1_if.60 capabilities source-address internet1
nv set vrf internet1 router bgp neighbor p1_if.60 peer-group l3_pg1
nv set vrf internet1 router bgp neighbor p1_if.60 type unnumbered
nv set vrf internet1 router bgp peer-group l3_pg1 address-family ipv4-unicast enable on
nv set vrf internet1 router bgp peer-group l3_pg1 address-family ipv6-unicast enable on
nv set vrf internet1 router bgp peer-group l3_pg1 remote-as external
nv set vrf internet1 router bgp route-export to-evpn route-target 65552:10000
nv set vrf internet1 router bgp route-import from-evpn route-target ANY:20000
nv set vrf internet1 router bgp route-import from-evpn route-target ANY:30000
nv set vrf internet1 router bgp route-import from-evpn route-target auto
nv set vrf internet1 router bgp router-id 27.0.0.5
```

Gateway-2 Full Configuration

Gateway-HBN-2

```
nv set bridge domain br_default encap 802.1Q
nv set bridge domain br_default type vlan-aware
nv set bridge domain br_default untagged 1
nv set bridge domain br_default vlan 10,30-31
nv set evpn enable on
nv set interface lo ip address 27.0.0.7/32
nv set interface lo ip address 2001:c001:ff:f00d::7/128
nv set interface lo type loopback
nv set interface p0_if,p1_if,pf0hpf_if,pf0vf0_if,pf0vf1_if,pf0vf2_if,pf0vf3_if,pf0vf4_if,pf1hpf_if type swp
nv set interface p0_if.60 base-interface p0_if
nv set interface p0_if.60,p1_if.60 type sub
nv set interface p0_if.60,p1_if.60 vlan 60
nv set interface p0_if.60,p1_if.60,vlan10 ip vrf internet1
nv set interface p1_if.60 base-interface p1_if
nv set interface pf0hpf_if bridge domain br_default access 30
nv set interface pf0vf0_if bridge domain br_default access 31
nv set interface vlan10 ip address 12.2.1.1/24
nv set interface vlan10 ip address 2001:c001:d00d::1:0:1/96
nv set interface vlan10 vlan 10
nv set interface vlan10,30-31 ip ipv4 forward on
nv set interface vlan10,30-31 ip ipv6 forward on
nv set interface vlan10,30-31 type svi
nv set interface vlan30 ip address 45.3.2.1/24
nv set interface vlan30 ip address 2001:b055:b00c::2:0:1/96
nv set interface vlan30 vlan 30
nv set interface vlan30-31 ip vrf gw_tenant1
nv set interface vlan31 ip address 45.3.3.1/24
nv set interface vlan31 ip address 2001:b055:b00c::3:0:1/96
nv set interface vlan31 vlan 31
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan mac-learning off
nv set nve vxlan source address 27.0.0.7
nv set platform
nv set router bgp enable on
nv set system config snippet
nv set system global anycast-mac 44:38:39:42:42:19
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp autonomous-system 65554
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor 27.0.0.11 peer-group rs_client
nv set vrf default router bgp neighbor 27.0.0.11 type numbered
nv set vrf default router bgp neighbor 27.0.0.12 peer-group rs_client
nv set vrf default router bgp neighbor 27.0.0.12 type numbered
nv set vrf default router bgp neighbor p0_if capabilities source-address lo
nv set vrf default router bgp neighbor p0_if peer-group fabric
nv set vrf default router bgp neighbor p0_if type unnumbered
nv set vrf default router bgp neighbor p1_if capabilities source-address lo
nv set vrf default router bgp neighbor p1_if peer-group fabric
nv set vrf default router bgp neighbor p1_if type unnumbered
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn enable off
nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp peer-group fabric timers connection-retry 5
nv set vrf default router bgp peer-group fabric timers hold 30
nv set vrf default router bgp peer-group fabric timers keepalive 10
nv set vrf default router bgp peer-group rs_client address-family ipv4-unicast enable off
nv set vrf default router bgp peer-group rs_client address-family ipv6-unicast enable off
nv set vrf default router bgp peer-group rs_client address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group rs_client address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group rs_client multihop-ttl 5
nv set vrf default router bgp peer-group rs_client remote-as external
nv set vrf default router bgp peer-group rs_client timers connection-retry 5
nv set vrf default router bgp peer-group rs_client timers hold 30
nv set vrf default router bgp peer-group rs_client timers keepalive 10
nv set vrf default router bgp router-id 27.0.0.7
nv set vrf gw_tenant1 evpn enable on
nv set vrf gw_tenant1 evpn vni 30000
nv set vrf gw_tenant1 loopback ip address 15.3.0.2/32
nv set vrf gw_tenant1 loopback ip address 2001:bad:c0de::2/128
nv set vrf gw_tenant1 router bgp address-family ipv4-unicast enable on
nv set vrf gw_tenant1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf gw_tenant1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf gw_tenant1 router bgp address-family ipv6-unicast enable on
nv set vrf gw_tenant1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf gw_tenant1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf gw_tenant1 router bgp address-family l2vpn-evpn enable on
nv set vrf gw_tenant1 router bgp autonomous-system 65554
nv set vrf gw_tenant1 router bgp enable on
nv set vrf gw_tenant1 router bgp route-export to-evpn route-target 65554:30000
nv set vrf gw_tenant1 router bgp route-import from-evpn route-target ANY:10000
nv set vrf gw_tenant1 router bgp route-import from-evpn route-target auto
nv set vrf gw_tenant1 router bgp router-id 27.0.0.7
nv set vrf internet1 evpn enable on
nv set vrf internet1 evpn vni 10000
nv set vrf internet1 loopback ip address 6.2.0.2/32
nv set vrf internet1 loopback ip address 2001:cafe:feed::2/128
nv set vrf internet1 router bgp address-family ipv4-unicast enable on
nv set vrf internet1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf internet1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf internet1 router bgp address-family ipv6-unicast enable on
nv set vrf internet1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf internet1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf internet1 router bgp address-family l2vpn-evpn enable on
nv set vrf internet1 router bgp autonomous-system 65554
nv set vrf internet1 router bgp enable on
nv set vrf internet1 router bgp neighbor p0_if.60 capabilities source-address internet1
nv set vrf internet1 router bgp neighbor p0_if.60 peer-group l3_pg1
nv set vrf internet1 router bgp neighbor p0_if.60 type unnumbered
nv set vrf internet1 router bgp neighbor p1_if.60 capabilities source-address internet1
nv set vrf internet1 router bgp neighbor p1_if.60 peer-group l3_pg1
nv set vrf internet1 router bgp neighbor p1_if.60 type unnumbered
nv set vrf internet1 router bgp peer-group l3_pg1 address-family ipv4-unicast enable on
nv set vrf internet1 router bgp peer-group l3_pg1 address-family ipv6-unicast enable on
nv set vrf internet1 router bgp peer-group l3_pg1 remote-as external
nv set vrf internet1 router bgp route-export to-evpn route-target 65554:10000
nv set vrf internet1 router bgp route-import from-evpn route-target ANY:20000
nv set vrf internet1 router bgp route-import from-evpn route-target ANY:30000
nv set vrf internet1 router bgp route-import from-evpn route-target auto
nv set vrf internet1 router bgp router-id 27.0.0.7
```

Tenant-HBN-3 Full Configuration

Tenant-HBN-3

```
nv set bridge domain br_default encap 802.1Q
nv set bridge domain br_default type vlan-aware
nv set bridge domain br_default untagged 1
nv set bridge domain br_default vlan 20-21
nv set evpn enable on
nv set interface lo ip address 27.0.0.17/32
nv set interface lo ip address 2001:c001:ff:f00d::11/128
nv set interface lo type loopback
nv set interface p0-1,pf0hpf,pf0vf0-12,pf1hpf,pf1vf0-4 type swp
nv set interface pf0hpf bridge domain br_default access 20
nv set interface pf0vf0 bridge domain br_default access 21
nv set interface vlan20 ip address 45.1.0.1/24
nv set interface vlan20 ip address 2001:c001:b00c::1/96
nv set interface vlan20 vlan 20
nv set interface vlan20-21 ip ipv4 forward on
nv set interface vlan20-21 ip ipv6 forward on
nv set interface vlan20-21 ip vrf tenant1
nv set interface vlan20-21 type svi
nv set interface vlan21 ip address 45.1.1.1/24
nv set interface vlan21 ip address 2001:c001:b00c::1:0:1/96
nv set interface vlan21 vlan 21
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan mac-learning off
nv set nve vxlan source address 27.0.0.17
nv set platform
nv set router bgp enable on
nv set system global anycast-mac 44:38:39:42:42:21
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp autonomous-system 6300656
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor 27.0.0.11 peer-group rs_client
nv set vrf default router bgp neighbor 27.0.0.11 type numbered
nv set vrf default router bgp neighbor 27.0.0.12 peer-group rs_client
nv set vrf default router bgp neighbor 27.0.0.12 type numbered
nv set vrf default router bgp neighbor p0 capabilities source-address lo
nv set vrf default router bgp neighbor p0 peer-group fabric
nv set vrf default router bgp neighbor p0 type unnumbered
nv set vrf default router bgp neighbor p1 capabilities source-address lo
nv set vrf default router bgp neighbor p1 peer-group fabric
nv set vrf default router bgp neighbor p1 type unnumbered
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn enable off
nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp peer-group fabric timers connection-retry 5
nv set vrf default router bgp peer-group fabric timers hold 30
nv set vrf default router bgp peer-group fabric timers keepalive 10
nv set vrf default router bgp peer-group rs_client address-family ipv4-unicast enable off
nv set vrf default router bgp peer-group rs_client address-family ipv6-unicast enable off
nv set vrf default router bgp peer-group rs_client address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group rs_client address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group rs_client multihop-ttl 5
nv set vrf default router bgp peer-group rs_client remote-as external
nv set vrf default router bgp peer-group rs_client timers connection-retry 5
nv set vrf default router bgp peer-group rs_client timers hold 30
nv set vrf default router bgp peer-group rs_client timers keepalive 10
nv set vrf default router bgp router-id 27.0.0.17
nv set vrf tenant1 evpn enable on
nv set vrf tenant1 evpn vni 20000
nv set vrf tenant1 loopback ip address 15.1.0.1/32
nv set vrf tenant1 loopback ip address 2001:c001:c0de::1/128
nv set vrf tenant1 router bgp address-family ipv4-unicast enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp address-family l2vpn-evpn enable on
nv set vrf tenant1 router bgp autonomous-system 6300656
nv set vrf tenant1 router bgp enable on
nv set vrf tenant1 router bgp route-export to-evpn route-target 6300656:20000
nv set vrf tenant1 router bgp route-import from-evpn route-target ANY:10000
nv set vrf tenant1 router bgp route-import from-evpn route-target auto
nv set vrf tenant1 router bgp router-id 27.0.0.17
```

Tenant-HBN-4 Full Configuration

Tenant-HBN4

```
nv set bridge domain br_default encap 802.1Q
nv set bridge domain br_default type vlan-aware
nv set bridge domain br_default untagged 1
nv set bridge domain br_default vlan 20-21
nv set evpn enable on
nv set interface lo ip address 27.0.0.19/32
nv set interface lo ip address 2001:c001:ff:f00d::13/128
nv set interface lo type loopback
nv set interface p0-1,pf0hpf,pf0vf0-12,pf1hpf,pf1vf0-4 type swp
nv set interface pf0hpf bridge domain br_default access 20
nv set interface pf0vf0 bridge domain br_default access 21
nv set interface vlan20 ip address 45.1.2.1/24
nv set interface vlan20 ip address 2001:c001:b00c::2:0:1/96
nv set interface vlan20 vlan 20
nv set interface vlan20-21 ip ipv4 forward on
nv set interface vlan20-21 ip ipv6 forward on
nv set interface vlan20-21 ip vrf tenant1
nv set interface vlan20-21 type svi
nv set interface vlan21 ip address 45.1.3.1/24
nv set interface vlan21 ip address 2001:c001:b00c::3:0:1/96
nv set interface vlan21 vlan 21
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan mac-learning off
nv set nve vxlan source address 27.0.0.19
nv set platform
nv set router bgp enable on
nv set system global anycast-mac 44:38:39:42:42:23
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp autonomous-system 6300658
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor 27.0.0.11 peer-group rs_client
nv set vrf default router bgp neighbor 27.0.0.11 type numbered
nv set vrf default router bgp neighbor 27.0.0.12 peer-group rs_client
nv set vrf default router bgp neighbor 27.0.0.12 type numbered
nv set vrf default router bgp neighbor p0 capabilities source-address lo
nv set vrf default router bgp neighbor p0 peer-group fabric
nv set vrf default router bgp neighbor p0 type unnumbered
nv set vrf default router bgp neighbor p1 capabilities source-address lo
nv set vrf default router bgp neighbor p1 peer-group fabric
nv set vrf default router bgp neighbor p1 type unnumbered
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn enable off
nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp peer-group fabric timers connection-retry 5
nv set vrf default router bgp peer-group fabric timers hold 30
nv set vrf default router bgp peer-group fabric timers keepalive 10
nv set vrf default router bgp peer-group rs_client address-family ipv4-unicast enable off
nv set vrf default router bgp peer-group rs_client address-family ipv6-unicast enable off
nv set vrf default router bgp peer-group rs_client address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group rs_client address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group rs_client multihop-ttl 5
nv set vrf default router bgp peer-group rs_client remote-as external
nv set vrf default router bgp peer-group rs_client timers connection-retry 5
nv set vrf default router bgp peer-group rs_client timers hold 30
nv set vrf default router bgp peer-group rs_client timers keepalive 10
nv set vrf default router bgp router-id 27.0.0.19
nv set vrf tenant1 evpn enable on
nv set vrf tenant1 evpn vni 20000
nv set vrf tenant1 loopback ip address 15.1.0.2/32
nv set vrf tenant1 loopback ip address 2001:c001:c0de::2/128
nv set vrf tenant1 router bgp address-family ipv4-unicast enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp address-family l2vpn-evpn enable on
nv set vrf tenant1 router bgp autonomous-system 6300658
nv set vrf tenant1 router bgp enable on
nv set vrf tenant1 router bgp route-export to-evpn route-target 6300658:20000
nv set vrf tenant1 router bgp route-import from-evpn route-target ANY:10000
nv set vrf tenant1 router bgp route-import from-evpn route-target auto
nv set vrf tenant1 router bgp router-id 27.0.0.19
```

17.8.4.3.4 Access Control Lists

Access Control Lists (ACLs) are a set of rules that are used to filter network traffic. These rules are used to specify the traffic flows that must be permitted or blocked at networking device interfaces. There are two types of ACLs:

- Stateless ACLs – rules that are applied to individual packets. They inspect each packet individually and permit/block the packets based on the packet header information and the match criteria specified by the rule.
- Stateful ACLs – rules that are applied to traffic sessions/connections. They inspect each packet with respect to the state of the session/connection to which the packet belongs to determine whether to permit/block the packet.

17.8.4.3.4.1 Stateless ACLs

HBN supports configuration of stateless ACLs for IPv4 packets, IPv6 packets, and Ethernet (MAC) frames. The following examples depict how stateless ACLs are configured for each case, with NVUE and with flat files (`cl-acltool`).

NVUE Examples for Stateless ACLs

NVUE IPv4 ACLs Example

The following is an example of an ingress IPv4 ACL that permits DHCP request packets ingressing on the `pf0hpf_if` port towards the DHCP server:

```
root@hbn01-host01:~# nv set acl acl1_ingress type ipv4
root@hbn01-host01:~# nv set acl acl1_ingress rule 100 match ip protocol udp
root@hbn01-host01:~# nv set acl acl1_ingress rule 100 match ip dest-port 67
root@hbn01-host01:~# nv set acl acl1_ingress rule 100 match ip source-port 68
root@hbn01-host01:~# nv set acl acl1_ingress rule 100 action permit
```

Bind the ingress IPv4 ACL to host representor port `pf0hpf_if` of BlueField in the inbound direction:

```
root@hbn01-host01:~# nv set interface pf0hpf_if acl acl1_ingress inbound
root@hbn01-host01:~# nv config apply
```

The following is an example of an egress IPv4 ACL that permits DHCP reply packets egressing out of the `pf0hpf_if` port towards the DHCP client:

```
root@hbn01-host01:~# nv set acl acl2_egress type ipv4
root@hbn01-host01:~# nv set acl acl2_egress rule 200 match ip protocol udp
root@hbn01-host01:~# nv set acl acl2_egress rule 200 match ip dest-port 68
root@hbn01-host01:~# nv set acl acl2_egress rule 200 match ip source-port 67
root@hbn01-host01:~# nv set acl acl2_egress rule 200 action permit
```

Bind the egress IPv4 ACL to host representor port `pf0hpf_if` of BlueField in the outbound direction:

```
root@hbn01-host01:~# nv set interface pf0hpf_if acl acl2_egress outbound
root@hbn01-host01:~# nv config apply
```

NVUE IPv6 ACLs Example

The following is an example of an ingress IPv6 ACL that permits traffic with matching `dest-ip` and `protocol tcp` ingress on port `pf0hpf_if`:

```
root@hbn01-host01:~# nv set acl acl5_ingress type ipv6
root@hbn01-host01:~# nv set acl acl5_ingress rule 100 match ip protocol tcp
root@hbn01-host01:~# nv set acl acl5_ingress rule 100 match ip dest-ip 48:2034::80:9
root@hbn01-host01:~# nv set acl acl5_ingress rule 100 action permit
```

Bind the ingress IPv6 ACL to host representor port `pf0hpf_if` of BlueField in the inbound direction:

```
root@hbn01-host01:~# nv set interface pf0hpf_if acl acl5_ingress inbound
root@hbn01-host01:~# nv config apply
```

The following is an example of an egress IPv6 ACL that permits traffic with matching `source-ip` and `protocol tcp` egressing out of port `pf0hpf_if`:

```
root@hbn01-host01:~# nv set acl acl6_egress type ipv6
root@hbn01-host01:~# nv set acl acl6_egress rule 101 match ip protocol tcp
root@hbn01-host01:~# nv set acl acl6_egress rule 101 match ip source-ip 48:2034::80:9
root@hbn01-host01:~# nv set acl acl6_egress rule 101 action permit
```

Bind the egress IPv6 ACL to host representor port `pf0hpf_if` of BlueField in the outbound direction:

```
root@hbn01-host01:~# nv set interface pf0hpf_if acl acl6_egress outbound
root@hbn01-host01:~# nv config apply
```

NVUE MAC ACLs Example

The following is an example of an ingress MAC ACL that permits traffic with matching `source-mac` and `dest-mac` ingressing to port `pf0hpf_if`:

```
root@hbn01-host01:~# nv set acl acl3_ingress type mac
root@hbn01-host01:~# nv set acl acl3_ingress rule 1 match mac source-mac 00:00:00:00:00:0a
root@hbn01-host01:~# nv set acl acl3_ingress rule 1 match mac dest-mac 00:00:00:00:00:0b
root@hbn01-host01:~# nv set interface pf0hpf_if acl acl3_ingress inbound
```

Bind the ingress MAC ACL to host representor port `pf0hpf_if` of BlueField in the inbound direction:

```
root@hbn01-host01:~# nv set interface pf0hpf_if acl acl3_ingress inbound
root@hbn01-host01:~# nv config apply
```

The following is an example of an egress MAC ACL that permits traffic with matching `source-mac` and `dest-mac` egressing out of port `pf0hpf_if`:

```
root@hbn01-host01:~# nv set acl acl4_egress type mac
root@hbn01-host01:~# nv set acl acl4_egress rule 2 match mac source-mac 00:00:00:00:00:0b
root@hbn01-host01:~# nv set acl acl4_egress rule 2 match mac dest-mac 00:00:00:00:00:0a
root@hbn01-host01:~# nv set acl acl4_egress rule 2 action permit
```

Bind the egress MAC ACL to host representor port `pf0hpf_if` of BlueField in the outbound direction:

```
root@hbn01-host01:~# nv set interface pf0hpf_if acl acl4_egress outbound
root@hbn01-host01:~# nv config apply
```

Flat Files (cl-acltool) Examples for Stateless ACLs

For the same examples cited above, the following are the corresponding ACL rules which must be configured under `/etc/cumulus/acl/policy.d/<rule_name.rules>` followed by invoking `cl-acltool -i`. The rules in `/etc/cumulus/acl/policy.d/<rule_name.rules>` are configured using Linux iptables/ip6tables/ebtables.

Flat Files IPv4 ACLs Example

The following example configures an ingress IPv4 ACL rule matching with DHCP request under `/etc/cumulus/acl/policy.d/<rule_name.rules>` with the ingress interface as the host representor of BlueField followed by invoking `cl-acltool -i`:

```
[iptables]
## ACL acl1_ingress in dir inbound on interface pf1vf1_if ##
-t filter -A FORWARD -m physdev --physdev-in pf1vf1_if -p udp --sport 68 --dport 67 -j ACCEPT
```

The following example configures an egress IPv4 ACL rule matching with DHCP reply under `/etc/cumulus/acl/policy.d/<rule_name.rules>` with the egress interface as the host representor of BlueField followed by invoking `cl-acltool -i`:

```
[iptables]
## ACL acl2_egress in dir outbound on interface pf1vf1_if ##
```

```
-t filter -A FORWARD -m physdev --physdev-out pf1vf1_if -p udp --sport 67 --dport 68 -j ACCEPT
```

Flat File IPv6 ACLs Example

The following example configures an ingress IPv6 ACL rule matching with `dest-ip` and `tcp` protocol under `/etc/cumulus/acl/policy.d/<rule_name.rules>` with the ingress interface as the host representor of BlueField followed by invoking `cl-acltool -i`:

```
[ip6tables]
## ACL acl5_ingress in dir inbound on interface pf0hpf_if ##
-t filter -A FORWARD -m physdev --physdev-in pf0hpf_if -d 48:2034::80:9 -p tcp -j ACCEPT
```

The following example configures an egress IPv6 ACL rule matching with `source-ip` and `tcp` protocol under `/etc/cumulus/acl/policy.d/<rule_name.rules>` with the egress interface as the host representor of BlueField followed by invoking `cl-acltool -i`:

```
[ip6tables]
## ACL acl6_egress in dir outbound on interface pf0hpf_if ##
-t filter -A FORWARD -m physdev --physdev-out pf0hpf_if -s 48:2034::80:9 -p tcp -j ACCEPT
```

Flat Files MAC ACLs Example

The following example configures an ingress MAC ACL rule matching with `source-mac` and `dest-mac` under `/etc/cumulus/acl/policy.d/<rule_name.rules>` with the ingress interface as the host representor of BlueField followed by invoking `cl-acltool -i`:

```
[ebtables]
## ACL acl3_ingress in dir inbound on interface pf0hpf_if ##
-t filter -A FORWARD -m physdev --physdev-in pf0hpf_if -s 00:00:00:00:00:0a/ff:ff:ff:ff:ff:ff -d 00:00:00:00:00:0b/
ff:ff:ff:ff:ff:ff -j ACCEPT
```

The following example configures an egress MAC ACL rule matching with `source-mac` and `dest-mac` under `/etc/cumulus/acl/policy.d/<rule_name.rules>` with egress interface as host representor of BlueField followed by invoking `cl-acltool -i`:

```
[ebtables]
## ACL acl4_egress in dir outbound on interface pf0hpf_if ##
-t filter -A FORWARD -m physdev --physdev-out pf0hpf_if -s 00:00:00:00:00:0b/ff:ff:ff:ff:ff:ff -d
00:00:00:00:00:0a/ff:ff:ff:ff:ff:ff -j ACCEPT
```

17.8.4.3.4.2 Stateful ACLs

Stateful ACLs facilitate monitoring and tracking traffic flows to enforce per-flow traffic filtering (unlike stateless ACLs which filter traffic on a per-packet basis). HBN supports stateful ACLs using reflexive ACL mechanism. Reflexive ACL mechanism is used to allow initiation of connections from "within" the network to "outside" the network and allow only replies to the initiated connections from "outside" the network (or vice versa).

HBN supports stateful ACL configuration for IPv4 traffic. Stateful ACL configuration is supported for TCP, UDP, and ICMP protocols.

Stateful ACLs can be applied for native routed traffic (north-south underlay routed traffic in EVPN deployments), EVPN bridged traffic (east-west overlay bridged/L2 traffic in EVPN deployments) and EVPN routed traffic (east-west overlay routed traffic in EVPN deployments). Stateful ACLs applied for native routed traffic are called "Native-L3 stateful ACLs". Stateful ACLs applied for EVPN bridged traffic and EVPN routed traffic are called "EVPN-L2 stateful ACLs" and "EVPN-L3 stateful ACLs", respectively.

Stateful ACLs in HBN are enabled by default. To enable stateful ACL functionality, use the following NVUE commands:

```
root@hbn03-host00:~# nv set system reflexive-acl enable
root@hbn03-host00:~# nv config apply
```

If using flat-file configuration (and not NVUE), edit the file `/etc/cumulus/nl2docad.d/acl.conf` and set the knob `rflx.reflexive_acl_enable` to `TRUE`. To apply this change, execute:

```
root@hbn03-host00:~# supervisorctl start nl2doca-reload
```

NVUE Example for Stateful ACLs

The following is an example of allowing HTTP (TCP) connection originated by the host, where BlueField is hosted, to an HTTP server (with the IP address 11.11.11.11) on an external network. Two sets of ACLs matching with CONNTRACK state must be configured for a CONNTRACK entry to be established in the kernel which would be offloaded to hardware:

- Configure an ACL rule matching TCP/HTTP connection/flow details with CONNTRACK state of NEW, ESTABLISHED and bind it to the SVI in the inbound direction.
- Configure an ACL rule matching TCP/HTTP connection/flow details with CONNTRACK state of ESTABLISHED and bind it to the SVI in the outbound direction.

Stateful ACLs should be bound to a physical interface. In this example, the physical interface is `pf1vf7_if`.

1. Configure the ingress ACL rule:

   ```
   root@hbn03-host00:~# nv set acl allow_tcp_conn_from_host rule 11 action permit
   root@hbn03-host00:~# nv set acl allow_tcp_conn_from_host rule 11 match conntrack new
   root@hbn03-host00:~# nv set acl allow_tcp_conn_from_host rule 11 match conntrack established
   root@hbn03-host00:~# nv set acl allow_tcp_conn_from_host rule 11 match ip dest-ip 11.11.11.11/32
   root@hbn03-host00:~# nv set acl allow_tcp_conn_from_host rule 11 match ip dest-port 80
   root@hbn03-host00:~# nv set acl allow_tcp_conn_from_host rule 11 match ip protocol tcp
   root@hbn03-host00:~# nv set acl allow_tcp_conn_from_host type ipv4
   ```

2. Bind this ACL to the physical interface in the inbound direction:

   ```
   root@hbn03-host00:~# nv set interface pf1vf7_if acl allow_tcp_conn_from_host inbound
   root@hbn03-host00:~# nv config apply
   ```

3. Configure the egress ACL rule:

   ```
   root@hbn03-host00:~# nv set acl allow_tcp_resp_from_server rule 21 action permit
   root@hbn03-host00:~# nv set acl allow_tcp_resp_from_server rule 21 match conntrack established
   root@hbn03-host00:~# nv set acl allow_tcp_resp_from_server rule 21 match ip protocol tcp
   root@hbn03-host00:~# nv set acl allow_tcp_resp_from_server type ipv4
   root@hbn03-host00:~# nv config apply
   ```

4. Bind this ACL to the physical interface in the outbound direction:

   ```
   root@hbn03-host00:~# nv set interface pf1vf7_if acl allow_tcp_resp_from_server outbound
   root@hbn03-host00:~# nv config apply
   ```

Flat Files (cl-acltool) Example for Stateful ACLs

For the same NVUE example for stateful ACLs cited above (HTTP server at IP address 11.11.11.11 on an external network), the following are the corresponding ACL rules which must be configured under `/etc/cumulus/acl/policy.d/<rule_name.rules>` followed by invoking `cl-acltool -i` to install the rules in BlueField hardware.

1. Configure an ingress ACL rule matching with TCP flow details and CONNTRACK state of NEW, ESTABLISHED under `/etc/cumulus/acl/policy.d/stateful_acl.rules` with the ingress interface as the SVI followed by invoking `cl-acltool -i`:

   ```
   [iptables]
   ## ACL allow_tcp_conn_from_host in dir inbound on interface pf1vf7_if ##
   -t filter -A FORWARD -m physdev --physdev-in pf1vf7_if -p tcp -d 11.11.11.11/32 --dport 80 -m conntrack --
   ctstate EST,NEW -j ACCEPT -m mark --mark 0xdead
   ```

2. Configure an egress ACL rule matching the TCP flow and CONNTRACK state of ESTABLISHED, RELATED under `/etc/cumulus/acl/policy.d/stateful_acl.rules` file with the egress interface as SVI followed by invoking `cl-acltool -i`:

   ```
   [iptables]
   ## ACL allow_tcp_resp_from_server in dir outbound on interface pf1vf7_if ##
   -t filter -A FORWARD -m physdev --physdev-out  pf1vf7_if  -p tcp -s 11.11.11.11/32 --sport 80 -m conntrack
   --ctstate EST -j ACCEPT -m mark --mark 0xdead
   ```

17.8.4.3.5 DHCP Relay on HBN

DHCP is a client server protocol that automatically provides IP hosts with IP addresses and other related configuration information. A DHCP relay (agent) is a host that forwards DHCP packets between clients and servers. DHCP relays forward requests and replies between clients and servers that are not on the same physical subnet.

DHCP relay can be configured using either flat file (supervisord configuration) or through NVUE.

17.8.4.3.5.1 Configuration

HBN is a non-systemd based container. Therefore, the DHCP relay must be configured as explained in the following subsections.

Flat File Configuration (Supervisord)

The HBN initialization script installs default configuration files on BlueField in `/var/lib/hbn/etc/supervisor/conf.d/`. BlueField directory is mounted to `/etc/supervisor/conf.d` which achieves configuration persistence.

By default, DHCP relay is disabled. Default configuration applies to one instance of DHCPv4 relay and DHCPv6 relay in the default VRF.

NVUE Configuration

The user can use NVUE to configure and maintain DHCPv4 and DHCPv6 relays with CLI and REST API. NVUE generates all the required configurations and maintains the relay service.

DHCPv4 Relay Configuration

NVUE Example

The following configuration starts a relay service which listens for the DHCP messages on `p0_if`, `p1_if`, and `vlan482` and relays the requests to DHCP server 10.89.0.1 with `gateway-interface` as `lo`.

```
nv set service dhcp-relay default gateway-interface lo
nv set service dhcp-relay default interface p0_if
nv set service dhcp-relay default interface p1_if
nv set service dhcp-relay default interface vlan482 downstream
nv set service dhcp-relay default server 10.89.0.1
```

Flat Files Example

```
[program: isc-dhcp-relay-default]
command = /usr/sbin/dhcrelay --nl -d -i p0_if -i p1_if -id vlan482 -U lo 10.89.0.1
autostart = true
autorestart = unexpected
startsecs = 3
startretries = 3
exitcodes = 0
stopsignal = TERM
stopwaitsecs = 3
```

Where:

Option	Description
`-i`	Network interface to listen on for requests and replies
`-iu`	Upstream network interface
`-id`	Downstream network interface
`-U [address]%%ifname`	Gateway IP address interface. Use `%%` for `IP%%ifname`. `%` is used as an escape character.
`--loglevel-debug`	Debug logging. Location: `/var/log/syslog`.
`-a`	Append an agent option field to each request before forwarding it to the server with default values for `circuit-id` and `remote-id`
`-r remote-id`	Set a custom remote ID string (max of 255 chars). To use this option, you must also enable the `-a` option.
`--use-pif-circuit-id`	Set the underlying physical interface which receives the packet as the `circuit-id`. To use this option you must also enable the `-a` option.

DHCPv4 Relay Option 82

NVUE Example

The following NVUE command is used to enable option 82 insertion in DHCP packets with default values:

```
nv set service dhcp-relay default agent enable on
```

To provide a custom `remote-id` (e.g., host10) using NVUE:

```
nv set service dhcp-relay default agent remote-id host10
```

To use the underlying physical interface on which the request is received as `circuit-id` using NVUE:

```
nv set service dhcp-relay default agent use-pif-circuit-id enable on
```

Flat Files Example

```
[program: isc-dhcp-relay-default]
command = /usr/sbin/dhcrelay --nl -d -i p0_if -i p1_if -id vlan482 -U lo -a --use-pif-circuit-id -r host10
10.89.0.1
autostart = true
autorestart = unexpected
startsecs = 3
startretries = 3
exitcodes = 0
stopsignal = TERM
stopwaitsecs = 3
```

DHCPv6 Relay Configuration

NVUE Example

The following NVUE command starts the DHCPv6 Relay service which listens for DHCPv6 requests on `vlan482` and sends relayed DHCPv6 requests towards `p0_if` and `p1_if`.

```
nv set service dhcp-relay6 default interface downstream vlan482
nv set service dhcp-relay6 default interface upstream p0_if
nv set service dhcp-relay6 default interface upstream p1_if
```

Flat Files Example

```
[program: isc-dhcp-relay6-default]
command = /usr/sbin/dhcrelay --nl -6 -d -l vlan482 -u p0_if -u p1_if
autostart = true
autorestart = unexpected
startsecs = 3
startretries = 3
exitcodes = 0
stopsignal = TERM
stopwaitsecs = 3
```

Where:

Option	Description
`-l [address]%%ifname[#index]`	Downstream interface. Use `%%` for `IP%%ifname`. `%` is used as escape character.
`-u [address]%%ifname`	Upstream interface. Use `%` `%` for `IP%%ifname`. `%` is used as escape character.
`-6`	IPv6
`--loglevel-debug`	Debug logging located at `/var/log/syslog`

17.8.4.3.5.2 DHCP Relay and VRF Considerations

DHCP relay can be spawned inside a VRF context to handle the DHCP requests in that VRF. There can only be 1 instance each of DHCPv4 relay and DHCPv6 relay per VRF. To achieve that, the user can follow these guidelines:

- DHCPv4 on default VRF:

```
/usr/sbin/dhcrelay --nl -i <interface> -U [address]%%<interface> <server_ip>
```

- DHCPv4 on VRF:

```
/usr/sbin/ip vrf exec <vrf> /usr/sbin/dhcrelay --nl -i <interface> -U [address]%%<interface> <server_ip>
```

- DHCPv6 on default VRF:

```
/usr/sbin/dhcrelay --nl -6 -l <interface> -u <interface>
```

- DHCPv6 on VRF:

```
/usr/sbin/ip vrf exec <vrf> /usr/sbin/dhcrelay --nl -6 -l <interface> -u <interface>
```

17.8.5 HBN Service Troubleshooting

17.8.5.1 HBN Container Stuck in init-sfs

The HBN container starts as `init-sfs` and should transition to `doca-hbn` within 2 minutes as can be seen using `crictl ps`. But sometimes it may remain as `init-sfs`.

This can happen if interface `p0_if` is missing. Run the command `ip -br link show dev p0_if` in BlueField and inside the container to check if `p0_if` is present or not. If its missing, make sure the firmware is upgraded to the latest version. Perform BlueField system-level reset for the new firmware to take effect.

17.8.5.2 Host-side PF/VF Down After BlueField Reboot

In general, the host can use any interface manager to manage host interfaces belonging to BlueField. When the host uses an interface manager other than Netplan or NetworkManager, some ports may remain down after BlueField reboot.

Apply the following workaround if interfaces stay down:

1. Restart openibd:

```
systemctl restart openibd
```

2. Recreate SR-IOV interfaces if they are needed.
3. Replay interface config. For example:
   - If using ifupdown2:

   ```
   ifreload -a
   ```

   - If using Netplan:

   ```
   netplan apply
   ```

17.8.5.3 BGP Session not Establishing

One of the main causes of a BGP session not getting established is a mismatch in MTU configuration. Make sure the MTU on all interfaces is the same. For example, if BGP is failing on `p0` , check and verify that there is a matching MTU value for `p0` , `p0_if_r` , `p0_if` , and the remote peer of `p0` .

17.8.5.4 Generating Support Information

The HBN container image can be collected from `/etc/image-version` using the `hbn-support` command inside container:

```
root@bf2:/tmp# hbn-support
Please send /var/support/hbn_support_doca-hbn-service-bf2-s15-1-ipmi_20240820_211214.txz to Cumulus support.
```

The generated dump would be available under `/var/support` in the HBN container and should contain any process core dump and log files. The generated cores can be found under `/var/support/core` and collected by `hbn-support` . The `/var/support` directory is also mounted on the BlueField Arm side at `/var/lib/hbn/var/support` .

For BlueField, the BFB version can be checked from `/etc/mlnx-release` .

The firmware version can be collect from `mlxfwmanager` .

BlueField support dump can be collect using the `sos` command:

```
root@bf2:/tmp/#sos report -a --all-logs --batch
```

Example output:

```
sos report (version 4.8.0)

This command will collect system configuration and diagnostic
information from this Ubuntu system.
...
...
  Finished running plugins

Creating compressed archive...

Your sos report has been generated and saved in:
        /tmp/sosreport-bf2-s15-1-ipmi-2024-08-20-cpdvegw.tar.xz

 Size   19.37MiB
 Owner  root
 sha256 0890a855623a1a2dd5089c9cd6d57d81e71f3805ac06c2d9fc0dab556ccd5ffc

Please send this file to your support representative.
```

17.8.5.5 SFC Troubleshooting

To troubleshoot flows going through SFC interfaces, the first step is to disable the `nl2doca` service in the HBN container:

```
root@bf2:/tmp# supervisorctl stop nl2doca
nl2doca: stopped
```

Stopping `nl2doca` effectively stops hardware offloading and switches to software forwarding. All packets would appear on `tcpdump` capture on BlueField interfaces.

`tcpdump` can be performed on SF interfaces as well as VLAN, VXLAN, and uplinks to determine where a packet gets dropped or which flow a packet is taking.

17.8.5.6 General nl2doca Troubleshooting

The following steps can be used to make sure the nl2doca daemon is up and running:

1. Make sure there are no errors in the nl2doca log file at `/var/log/hbn/nl2docad.log`.
2. To check the status of the nl2doca daemon under supervisor, run:

   ```
   supervisorctl status nl2doca
   ```

3. Use `ps` to check that the actual nl2doca process is running:

   ```
   ps -eaf | grep nl2doca
   root          18       1  0 06:31 ?        00:00:00 /bin/bash /usr/bin/nl2doca-docker-start
   root        1437      18  0 06:31 ?        00:05:49 /usr/sbin/nl2docad
   ```

4. The core file should be in `/var/support/core/`.
5. Check if the `/cumulus/nl2docad/run/stats/punt` is accessible. Otherwise, nl2doca may be stuck and should be restarted:

   ```
   supervisorctl restart nl2doca
   ```

17.8.5.7 nl2doca Offload Troubleshooting

If a certain traffic flow does not work as expected, disable nl2doca (i.e., disable hardware offloading):

```
supervisorctl stop nl2doca
```

With hardware offloading disabled, you can confirm it is an offloading issue if the traffic starts working. If it is not an offloading issue, use `tcpdump` on various interfaces to see where the packet gets dropped.

Offloaded entries can be checked in following files, which contain the programming status of every IP prefix and MAC address known to system.

- Bridge entries are available in the file `/cumulus/nl2docad/run/software-tables/17`. It includes all the MAC addresses in the system including local and remote MAC addresses. Example format:

  ```
  - flow-entry: 0xaaab0cef4190
        flow-pattern:
          fid: 112
          dst mac: 00:00:5e:00:01:01
        flow-actions:
          SET VRF: 2
          OUTPUT-PD-PORT: 20(TO_RTR_INTF)
          STATS:
            pkts: 1719
            bytes: 191286
  ```

- Router entries are available in the file `/cumulus/nl2docad/run/software-tables/18`. It includes all the IP prefixes known to the system.
  Example format for Entry with ECMP:

  ```
  Entry with ECMP:
  - flow-entry: 0xaaaada723700
    flow-pattern:
       IPV6: LPM
       VRF: 0
       destination-ip: ::/0
    flow-actions :
       ECMP: 2
       STATS:
          pkts: 0
          bytes: 0

  Entry without ECMP: - flow-entry: 0xaaaada7e1400
      flow-pattern:
         IPV4: LPM
         VRF: 0
         destination-ip: 60.1.0.93/32
      flow-actions :
          SET FID: 200
          SMAC: 00:04:4b:a7:88:00
          DMAC: 00:03:00:08:00:12
          OUTPUT-PD-PORT: 19(TO_BR_INTF)
     STATS:
         pkts: 0
         bytes: 0
  ```

- ECMP entries are available in the file `/cumulus/nl2docad/run/software-tables/19`. It includes all the next hops in the system.
  Example format:

  ```
  - ECMP: 2
    ref-count: 2
    num-next-hops: 2
    entries:
    - { index: 0, fid: 4100, src mac: 'b8:ce:f6:99:49:6a', dst mac: '00:02:00:00:00:0a' }
    - { index: 1, fid: 4101, src mac: 'b8:ce:f6:99:49:6b', dst mac: '00:02:00:00:00:0e' }
  ```

To check counters for packets going to the kernel, run:

```
cat /cumulus/nl2docad/run/stats/punt
PUNT miss pkts:3154 bytes:312326
PUNT miss drop pkts:0 bytes:0
PUNT control pkts:31493 bytes:2853186
PUNT control drop pkts:0 bytes:0
ACL PUNT pkts:68 bytes:7364
ACL drop pkts:0 bytes:0
```

For a specific type of packet flow, programming can be referenced in block specific files. The typical flow is as follows:

For example, to check L2 EVPN ENCAP flows for remote MAC `8a:88:d0:b1:92:b1` on port `pf0vf0_if`, the basic offload flow should look as follows: RxPort (`pf0vf0_if`) -> BR (Overlay) -> RTR (Underlay) -> BR (Underlay) -> TxPort (one of the uplink `p0_if` or `p1_if` based on ECMP hash).

Step-by-step procedure:

1. Navigate to the interface file `/cumulus/nl2docad/run/software-tables/20`.
2. Check for the RxPort (`pf0vf0_if`):

   ```
   Interface: pf0vf0_if
       PD PORT: 6
       HW PORT: 16
       NETDEV PORT: 11
       Bridge-id: 61
       Untagged FID: 112
   ```

FID 112 is given to the receive port.

3. Check the bridge table file `/cumulus/nl2docad/run/software-tables/17` with destination MAC `8a:88:d0:b1:92:b1` and FID 112:

```
flow-pattern:
      fid: 112
        dst mac: 8a:88:d0:b1:92:b1
      flow-actions:
        VXLAN ENCAP:
          ENCAP dst ip: 6.0.0.26
          ENCAP vni id: 1000112
        SET VRF: 0
        OUTPUT-PD-PORT: 20(TO_RTR_INTF)
        STATS:
          pkts: 100
          bytes: 10200
```

4. Check the router table file `/cumulus/nl2docad/run/software-tables/18` with destination IP `6.0.0.26` and VRF 0:

```
flow-pattern:
        IPV4: LPM
        VRF: 0
        ip dst: 6.0.0.26/32
      flow-actions :
        ECMP: 1
        OUTPUT PD PORT: 2(TO_BR_INTF)
        STATS:
          pkts: 300
          bytes: 44400
```

5. Check the ECMP table file `/cumulus/nl2docad/run/software-tables/19` with ECMP 1:

```
- ECMP: 1
      ref-count: 7
      num-next-hops: 2
      entries:
        - { index: 0, fid: 4100, src mac: 'b8:ce:f6:99:49:6a', dst mac: '00:02:00:00:00:2f' }
        - { index: 1, fid: 4115, src mac: 'b8:ce:f6:99:49:6b', dst mac: '00:02:00:00:00:33' }
```

6. The ECMP hash calculation picks one of these paths for next-hop rewrite. Check bridge table file for them (`fid=4100, dst mac: 00:02:00:00:00:2f` or `fid=4115, dst mac: 00:02:00:00:00:33`):

```
flow-pattern:
        fid: 4100
        dst mac: 00:02:00:00:00:2f
flow-actions:
    OUTPUT-PD-PORT: 36(p0_if)
    STATS:
       pkts: 1099
       bytes: 162652
```

This will show the packet going out on the uplink.

17.8.5.8 NVUE Troubleshooting

To check the status of the NVUE daemon, run:

```
supervisorctl status nvued
```

To restart the NVUE daemon, run:

```
supervisorctl restart nvued
```

17.9 NVIDIA DOCA Management Service Guide

This guide provides instructions on how to use the DOCA Management Service on top of NVIDIA® BlueField® Networking Platform or ConnectX® Network Adapters.

⚠ DOCA DMS service is currently supported at Alpha level.

17.9.1 Introduction

DOCA Management Service (DMS) is a one-stop shop for the user to configure and operate NVIDIA BlueField and ConnectX devices. DMS governs all scripts/tools of NVIDIA with an easy and industry-standard API created by the OpenConfig community. The user can configure BlueField or ConnectX for any mode whether locally (`ssh`) or remotely (`grpc`). It makes it easy to migrate and bootstrap any customer for any NVIDIA network device.

DMS exposes configurable BlueField/ConnectX parameters over the external interface to support a management station in an automated configuration of the NVIDIA Network Adapters. The exposed interface presents a uniform approach for BF/CX device configuration and keeps hidden details about the internal tools used for the configuration of BlueField or ConnectX features.

The DMS is a Client-Server architecture. Using a daemon, the service handles the discovery of resources, and is ready to receive commands from clients, the user can use DMSc (DMS Client) which delivers as part of the DMS, or use/create any other client.

ⓘ Please refer to the OpenConfig site for an explanation of the OpenConfig protocol.

The Yang models describe a config tree which is easy to navigate and find any "config leaf" using XPath capabilities. Most gNMI/gNOI protocols are common with the OpenConfig community, utilizing gRPC protocol for transferring the command.

⚠ The DOCA Yang model is experimental.

⚠ The gNMI Subscribe mechanism for streaming telemetry is not currently supported yet.

ⓘ DMS can run either on the host machine where BlueField or ConnectX devices are installed or on BlueField Arm itself (when BlueField is operating in DPU mode).

17.9.2 Requirements

DMS requires DOCA to be installed on the target system, where DMS Service will be running:

- DMS for Host - requires DOCA for Host package to be installed on the host system (with doca-networking or doca-all profiles).
- DMS for DPU (BlueField Arm) - requires DOCA Image to be installed on BlueField Arm.

Please follow these instructions to install DOCA: NVIDIA DOCA Installation Guide for Linux.

⚠ DMS supports only Linux-based environments today.

17.9.3 Service Deployment

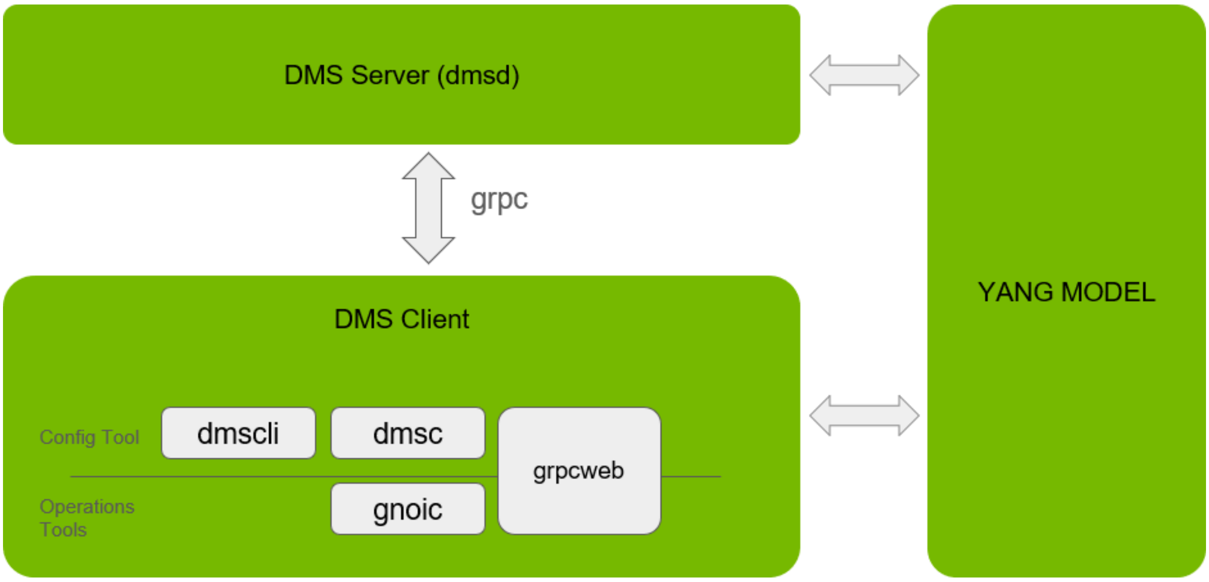
DMS has 3 major components:

- DMSD – Server – DMS server inside the BlueField or on the host with an NVIDIA PCIe device
- DMSC – Client – DOCA provides OpenConfig client. Customers can choose to use this client, any other open-source client, or develop their own (gRPC-based) client.
- Yang files – Yang model files contain the data model used to configure the BlueField device, NVIDIA-specific extension to common OpenConfig YANG Models.

OpenConfig consists of 2 main protocols:

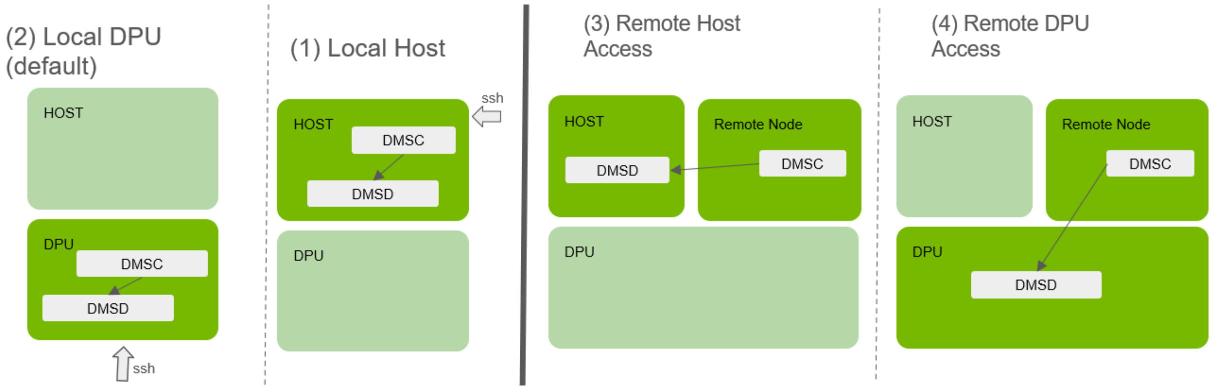
- gNMI – gRPC Network Management Interface, protocol to configure of network device.
- gNOI – gRPC Network Operations Interface, a protocol to perform operational commands on network device (i.e., provision, upgrade, reboot).

The following is an architectural diagram of DMS:



The following diagram presents the DMS mode of operation, as the DMS client can operate from anywhere:

1. Both DMS client and server components are deployed on the Host
2. Both DMS client and server components are deployed on DPU (BlueField Arm)
3. DMS server component is deployed on the Host, while DMS client is deployed remotely (connecting to DMS server over management network)
4. DMS server component is deployed on DPU (BlueField Arm), while DMS client is deployed remotely (connecting to DMS server over management network)



DMSD is a systemd service installed on the DPU by default with the BFB-Bundle and can be enable/disabled using `systemctl`. DMSD can be accessed using the command `dmscli` and provided the `dmsd` user password (default is the root OS password). A systemd template is provided on host packages.

17.9.4 Configuration

To see the full list of flags, user the help flag (i.e., `dmsd -help`, `dmsd -h`).

17.9.4.1 General Flags

- `-bind_address <string>` - Bind to `<address>:<port>` or just `:<port>` (default is `:9339`). Can be localhost for local use case, or an IP address for remote use case.
- `-v <value>` - log level for V logs
- `-target_pci <string>` - The target PCIe address (i.e., `03:00`). Auto-select if only one NVIDIA network device is present; otherwise, the PCIe address must be specified.

17.9.4.2 Security Flags

`-auth string` - this flag has 3 options:

- Shadow
  - Zero-touch, admin not required to create any dedicated additional user for DMS (re-use OS user)
  - Read the hashed password in real time on each client request
  - Use flags `-username -shadow`
  - Example: `-username root -shadow /etc/shadow/`
  - To disable: `-noauth flag`
- Credentials
  - Admin must set a strong password
  - Use flags `-username -password`
  - Example: `-username root -password 123456`
  - To disable: `-noauth flag`
  - Can leave password flag empty to invoke prompt for password at demon boot
- Certificate File
  - The most secure option, based on (m)TLS

- Example: `-ca /tmp/ca.crt -ca_key /tmp/ca.key`
- To disable: `-notls option`

17.9.4.3 Provisioning Flags

- `-target_pci <string>` - The target PCIe address (i.e., `03:00`). Auto-select if only one NVIDIA network device is present; otherwise, the PCIe address must be specified.
- `-image_folder <string>` - Specify image install folder. Can copy images directly to the folder to avoid transfer over the net. Default create folder: `/tmp/dms`.
- `-chunk_size_ack <uint>` - The chunk size of the image to respond with a transfer response in bytes (default: 12000000)
- `-exec_timeout <uint>` - The maximum execution timeout in seconds for a command if not responding (not printing to `stdout`); 0 (default) is unlimited

17.9.5 Description

17.9.5.1 gNMI Command

In DMSC, the gNMI part is powered by the GNMIC project.

> For more information, please refer to GNMIC documentation.

```
dmsc -a localhost:9339 -u root -p <password> --file /opt/mellanox/doca/service/dms/yang <command>
```

Prompt mode with autocomplete options can be invoked using the command `prompt`. It can be accessed using the command `dmscli` and provided the `dmsd` user password (default is the root OS password).

17.9.5.1.1 Get Supported Paths

```
dmsc --file /opt/mellanox/doca/service/dms/yang path --types --descr

/interfaces/interface[name=*]/config/enabled    (type=boolean)
        This leaf contains the configured, desired state of the
        interface.

        Systems that implement the IF-MIB use the value of this
        leaf in the 'running' datastore to set
        IF-MIB.ifAdminStatus to 'up' or 'down' after an ifEntry
        has been initialized, as described in RFC 2863.

        Changes in this leaf in the 'running' datastore are
        reflected in ifAdminStatus, but if ifAdminStatus is
        changed over SNMP, this leaf is not affected.
/interfaces/interface[name=*]/config/mtu        (type=uint16)
        Set the max transmission unit size in octets
        for the physical interface.  If this is not set, the mtu is
        set to the operational default -- e.g., 1514 bytes on an
        Ethernet interface.
/interfaces/interface[name=*]/config/type       (type=identityref)
        The type of the interface.

        When an interface entry is created, a server MAY
        initialize the type leaf with a valid value, e.g., if it
        is possible to derive the type from the name of the
        interface.

        If a client tries to set the type of an interface to a
        value that can never be used by the system, e.g., if the
        type is not supported or if the type does not match the
        name of the interface, the server MUST reject the request.
        A NETCONF server MUST reply with an rpc-error with the
        error-tag 'invalid-value' in this case.
/interfaces/interface[name=*]/ethernet/nvidia/config/inter-packet-gap   (type=uint8)
        Inter packet gap configuration, in 4B unit
/interfaces/interface[name=*]/ethernet/nvidia/config/rate-limit (type=uint16)
        The percentage of bandwidth, in permile units, to be used on the port.
/interfaces/interface[name=*]/name      (type=leafref)
        References the name of the interface
/interfaces/interface[name=*]/nvidia/cc/config/priority[id=*]/id        (type=leafref)

/interfaces/interface[name=*]/nvidia/cc/config/priority[id=*]/np_enabled        (type=boolean)
        Enable CC NP for a given priority on the interface
/interfaces/interface[name=*]/nvidia/cc/config/priority[id=*]/rp_enabled        (type=boolean)
        Enable CC RP for a given priority on the interface
/interfaces/interface[name=*]/nvidia/cc/slot[id=*]/config/enabled       (type=boolean)
        Enable a CC algo slot execution.
/interfaces/interface[name=*]/nvidia/cc/slot[id=*]/id   (type=leafref)
        CC algo slot ID.
/interfaces/interface[name=*]/nvidia/cc/slot[id=*]/param[id=*]/config/value     (type=algo_param_value)
        Parameter value within the CC algo slot.
/interfaces/interface[name=*]/nvidia/cc/slot[id=*]/param[id=*]/id       (type=leafref)
        Parameter ID within the CC algo slot.
/interfaces/interface[name=*]/nvidia/qos/config/pfc     (type=boolean)
        Enables PFC
/interfaces/interface[name=*]/nvidia/qos/config/priority[id=*]/id       (type=prio)
        Priority id.
/interfaces/interface[name=*]/nvidia/qos/config/trust-mode      (type=identityref)
        Trust mode for the interface QoS.
/interfaces/interface[name=*]/nvidia/roce/config/adaptive-retransmission        (type=boolean)
        Enable adaptive retransmission
/interfaces/interface[name=*]/nvidia/roce/config/adaptive-routing-force (type=boolean)
        Force adaptive routing even if feature was not negotiated between a requestor and responder.
/interfaces/interface[name=*]/nvidia/roce/config/rtt-resp-dscp  (type=uint8)
        Defines the DSCP fixed value used if mode is set to FIXED.
/interfaces/interface[name=*]/nvidia/roce/config/rtt-resp-dscp-mode     (type=identityref)
        Defines the method for setting DSCP in RTT response packets.
/interfaces/interface[name=*]/nvidia/roce/config/slow-restart   (type=boolean)
        Enable slow restart when congestion
/interfaces/interface[name=*]/nvidia/roce/config/slow-restart-idle      (type=boolean)
        Enable slow restart when idle
/interfaces/interface[name=*]/nvidia/roce/config/tos    (type=tos)
        ToS value for RoCE traffic.
/interfaces/interface[name=*]/nvidia/roce/config/tx-window      (type=boolean)
        Enable transmission window
/nvidia/cc/config/user-programmable     (type=boolean)
        Enables user-programmable CC functionality.
/nvidia/mode/config/mode        (type=identityref)
        Mode can take one one of several predefined
        values representing operational modes of DPU.
/nvidia/roce/config/adaptive-routing    (type=boolean)
        Enable adaptive routing between a requestor and responder.
/nvidia/roce/config/multipath-dscp      (type=identityref)
        Multipath on transmit, set the DSCP bit to hold the MP eligible info
/nvidia/roce/config/tx-sched-locality-mode      (type=identityref)
        Transmission scheduler adaptation to locality
```

17.9.5.1.2 Get Request

Get requests happen in real-time without cache. Get command require providing the Yang Xpath as described in the following:

```
dmsc <flags> get --path /interfaces/interface[name=p0]/config/mtu
[
  {
    "source": "localhost:9339",
    "timestamp": 1712485149723248511,
    "time": "2024-04-07T10:19:09.723248511Z",
    "updates": [
      {
        "Path": "interfaces/interface[name=p0]/config/mtu",
        "values": {
          "interfaces/interface/config/mtu": "1500"
        }
      }
    ]
  }
]
```

ⓘ To insert params in the path, as an indication of the interface name (p0).

17.9.5.1.3 Set Request

Set requests happen immediately, invoking tools to configure the OS.

Set commands require providing Yang Xpath as described in the following:

```
dmsc <flags> set --update /interfaces/interface[name=p0]/config/mtu:::int:::9216
{
  "source": "localhost:9339",
  "time": "1970-01-01T00:00:00Z",
  "results": [
    {
      "operation": "UPDATE",
      "path": "interfaces/interface[name=p0]/config/mtu"
    }
  ]
}
```

> To insert params in the path, as an indication of the interface name (p0).

> The value provided must be separated by value type and char.

> Currently, only the `--update` flag is supported in set.

> Some leafs' updates take effect only after system reboot. Refer to gNOI system reboot for information.

It is also possible to invoke a command JSON list:

```
dmsc <flags> set --request-file req.json
```

`req.json` example:

```
{
  "updates":
    [
      {
        "path": "/interfaces/interface[name=p0]/config/mtu",
        "value": 9216,
        "encoding": "uint"
      },
      {
        "path": "/interfaces/interface[name=p0]/config/enabled",
        "value": true,
        "encoding": "bool"
      }
    ]
}
```

17.9.5.2 gNOI Commands

In DMSc, the gNOI part is powered by GNOIC project, for full docs refer to GNOIC docs

```
dmsc -a localhost --port 9339 --tls-cert client.crt --tls-key client.key <command>
```

Prompt mode with autocomplete options can be invoked using the command `prompt`.

All commands are blocking unless specified otherwise.

17.9.5.2.1 OS

The following subsections present actions for provisioning a new DOCA Image (BFB) or firmware on BlueField.

17.9.5.2.1.1 Install

This command transmits the file from the client to the server and authenticates the file's validity:

```
dmsc <flags> os install --version <free_text_version> --pkg <bfb|cfg|fw path>
dmsc <flags> os install --version 2_7_0 --pkg DOCA_2.7.0_Ubuntu.bfb
dmsc <flags> os install --version 2_7_0 --pkg config.cfg
dmsc <flags> os install --version 1_3_5_custom.bfb --pkg custom.bfb
```

The file is saved to the folder specified in the `-image_folder` flag (default `/tmp/dms`) if the file authenticates successfully. The file's extension is autodetected and is written automatically if `none` is provided in the `--version` field. Users may copy the file to the folder manually and invoke the command with file extension to authenticate the file. No file transfer is initiated if the file already exists in the folder and the version specified with the extension.

17.9.5.2.1.2 Activate

Activate the command deploy the BFB bundle/firmware to the hardware:

```
dmsc <flags> os activate --version 2_7_0 # Invoke all files under 2_7_0 name
dmsc <flags> os activate --version "2_7_0.bfb;0_0_1.cfg;24_29_0046.fw"
```

The `--version` flag provides a version to search for in the folder specified by the `-image_folder` flag (default `/tmp/dms`). If no extension is provided, the command uses all files under the version name.

To activate separate files, use the `--version` flag separated by semi-colon.

> ⚠ After running the command to activate firmware, firmware reset is automatically invoked.

17.9.5.2.1.3 Verify

Verify command retrieves the firmware and BFB bundle version:

```
dmsc <flags> os verify
```

The return value consists of both versions separated by semi-colon.

> ⚠ Currently, the BFB bundle can only be retrieved if it was installed via DMS.

17.9.5.2.2 System

The following subsections provide actions for rebooting the BFB bundle/firmware on the BlueField.

17.9.5.2.2.1 Reboot Status

To verify BFB is rebooting:

```
dmsc <flags> system reboot-status
```

The value returned is `false` if the system is active. It is `true` if the system is rebooting. If the status cannot be retrieved, the status appears as a failure and the message field indicates what the issue is.

The flag `--reboot_status_check <string>` checks if firmware reboot is needed:

- If set to `fast` (default), a quick test occurs but not accurate (any config can trigger this flag)
- If set to `strict`, a more accurate test occurs but slower
- If set to `none`, then firmware check is skipped

17.9.5.2.2.2 Reboot

To reboot the BlueField Arm and firmware:

```
dmsc <flags> system reboot --delay <uint>s --subcomponent <string> --method <string>
```

This command is non-blocking and returns immediately.

The flag `--delay` specifies the time interval to wait before invoking the reset.

The subcomponent and method are optional. By default, the reboot executes with the lowest reset level and type available.

> ⚠ Currently, DMS supports `--subcomponent ARM --method <WARM|POWERDOWN>` flags.

17.10 NVIDIA OpenvSwitch Acceleration (OVS in DOCA)

> ⓘ Note on naming conventions:
>
> - OVS - Refers to the Open vSwitch distribution within DOCA framework
> - OVS-DOCA - Describes the datapath offloading layer (DPIF) that utilizes the DOCA Flow library for offloading tasks. This layer is a component of OVS, along with additional DPIF implementations that facilitate offloading via DPDK or Kernel, known respectively as OVS-DPDK and OVS-Kernel.

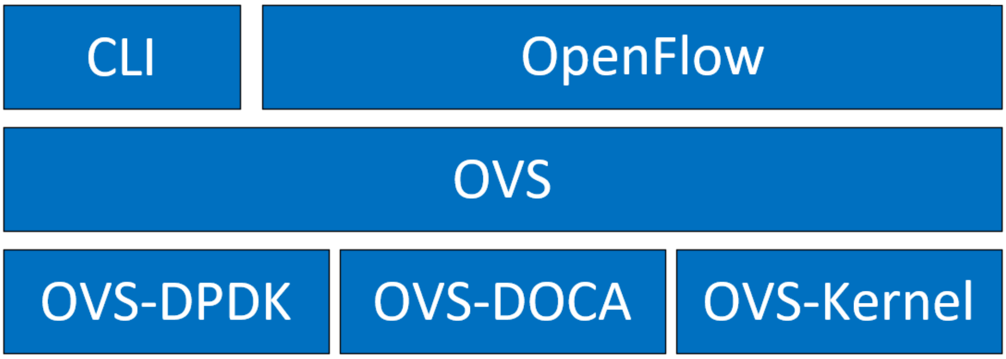
> ✓ NVIDIA advises utilizing the OVS-DOCA DPIF to maximize efficiency, performance, scalability, and feature support.

> ❗ The DPDK and Kernel DPIFs are maintained in their current form primarily for backward compatibility and are not planned to be updated with new features.

17.10.1 Introduction

Open vSwitch (OVS) is a software-based network technology that enhances virtual machine (VM) communication within internal and external networks. Typically deployed in the hypervisor, OVS employs a software-based approach for packet switching, which can strain CPU resources, impacting system performance and network bandwidth utilization. Addressing this, NVIDIA's Accelerated Switching and Packet Processing (ASAP^2) technology offloads OVS data-plane tasks to specialized hardware, like the embedded switch (eSwitch) within the NIC subsystem, while maintaining an unmodified OVS control-plane. This results in notably improved OVS performance without burdening the CPU.

NVIDIA's DOCA-OVS extends the traditional OVS-DPDK and OVS-Kernel data-path offload interfaces (DPIF), introducing OVS-DOCA as an additional DPIF implementation. DOCA-OVS, built upon NVIDIA's networking API, preserves the same interfaces as OVS-DPDK and OVS-Kernel while utilizing the DOCA Flow library with the additional OVS-DOCA DPIF. Unlike the use of the other DPIFs (DPDK, Kernel), OVS-DOCA DPIF exploits unique hardware offload mechanisms and application techniques, maximizing performance and features for NVIDA NICs and DPUs. This mode is especially efficient due to its architecture and DOCA library integration, enhancing e-switch configuration and accelerating hardware offloads beyond what the other modes can achieve.



NVIDIA OVS installation contains all three OVS flavors. The following subsections describe the three flavors (default is OVS-Kernel) and how to configure each of them.

17.10.2 OVS and Virtualized Devices

When OVS is combined with NICs and DPUs (such as NVIDIA® ConnectX®-6 Lx/Dx and NVIDIA® BlueField®-2 and later), it utilizes the hardware data plane of ASAP^2. This data plane can establish connections to VMs using either SR-IOV virtual functions (VFs) or virtual host data path acceleration (vDPA) with virtio.

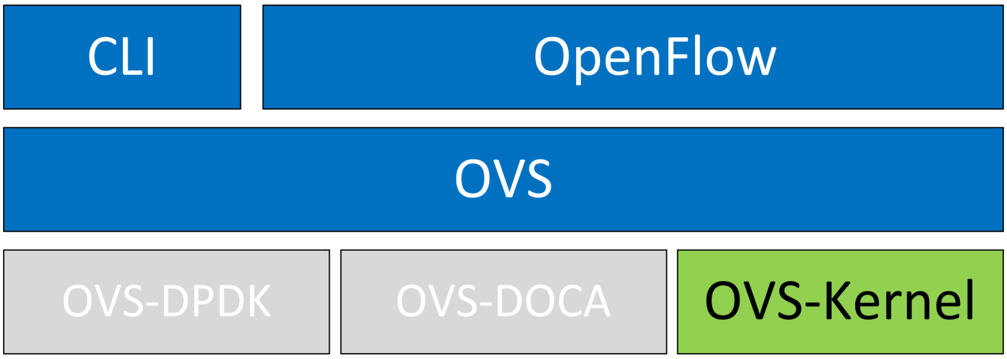
In both scenarios, an accelerator engine within the NIC accelerates forwarding and offloads the OVS rules. This integrated solution accelerates both the infrastructure (via VFs through SR-IOV or virtio) and the data plane. For DPUs (which include a NIC subsystem), an alternate virtualization technology implements full virtio emulation within the DPU, enabling the host server to communicate with the DPU as a software virtio device.

- When using ASAP^2 data plane over SR-IOV virtual functions (VFs), the VF is directly passed through to the VM, with the NVIDIA driver running within the VM.

- When using vDPA, the vDPA driver allows VMs to establish their connections through VirtIO. As a result, the data plane is established between the SR-IOV VF and the standard virtio driver within the VM, while the control plane is managed on the host by the vDPA application.

17.10.3 OVS-Kernel Hardware Acceleration

OVS-Kernel is the default OVS flavor enabled on your NVIDIA device.



17.10.3.1 Switchdev Configuration

1. Unbind the VFs:

```
echo 0000:04:00.2 > /sys/bus/pci/drivers/mlx5_core/unbind
echo 0000:04:00.3 > /sys/bus/pci/drivers/mlx5_core/unbind
```

   VMs with attached VFs must be powered off to be able to unbind the VFs.

2. Change the eSwitch mode from legacy to switchdev on the PF device:

```
# devlink dev eswitch set pci/0000:3b:00.0 mode switchdev
```

   This also creates the VF representor netdevices in the host OS.

   Before changing the mode, make sure that all VFs are unbound.

   To return to SR-IOV legacy mode, run:

```
# devlink dev eswitch set pci/0000:3b:00.0 mode legacy
```

   This also removes the VF representor netdevices.

   On OSes or kernels that do not support devlink, moving to switchdev mode can be done using sysfs:

```
# echo switchdev > /sys/class/net/enp4s0f0/compat/devlink/mode
```

3. At this stage, VF representors have been created. To map a representor to its VF, make sure to obtain the representor's `switchid` and `portname` by running:

```
# ip -d link show eth4
41: enp0s8f0_1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default
 qlen 1000
    link/ether ba:e6:21:37:bc:d4 brd ff:ff:ff:ff:ff:ff promiscuity 0 addrgenmode eui64 numtxqueues 10
 numrxqueues 10 gso_max_size 65536 gso_max_segs 65535 portname pf0vf1 switchid f4ab580003a1420c
```

   Where:
   - `switchid` - used to map representor to device, both device PFs have the same `switchid`
   - `portname` - used to map representor to PF and VF. Value returned is `pf<X>vf<Y>`, where `X` is the PF number and `Y` is the number of VF.
4. Bind the VFs:

```
echo 0000:04:00.2 > /sys/bus/pci/drivers/mlx5_core/bind
echo 0000:04:00.3 > /sys/bus/pci/drivers/mlx5_core/bind
```

17.10.3.2 Switchdev Performance Tuning

Switchdev tuning improves its performance.

17.10.3.2.1 Steering Mode

OVS-kernel supports two steering modes for rule insertion into hardware:

- SMFS (software-managed flow steering) – default mode; rules are inserted directly to the hardware by the software (driver). This mode is optimized for rule insertion.
- DMFS (device-managed flow steering) – rule insertion is done using firmware commands. This mode is optimized for throughput with a small amount of rules in the system.

The steering mode can be configured via sysfs or devlink API in kernels that support it:

- For sysfs:

```
echo <smfs|dmfs> > /sys/class/net/<pf-netdev>/compat/devlink/steering_mode
```

- For devlink:

```
devlink dev param set pci/0000:00:08.0 name flow_steering_mode value "<smfs|dmfs>" cmode runtime
```

Notes:

- The mode should be set prior to moving to switchdev, by echoing to the sysfs or invoking the devlink command.
- Only when moving to switchdev will the driver use the mode configured.
- Mode cannot be changed after moving to switchdev.
- The steering mode is applicable for switchdev mode only (i.e., it does not affect legacy SR-IOV or other configurations).

17.10.3.2.2 Troubleshooting SMFS

mlx5 debugfs supports presenting Software Steering resources. `dr_domain` including its tables, matchers and rules. The interface is read-only.

> New steering rules cannot be inserted/deleted while the dump is being created,

The steering information is dumped in the CSV form in the following format: `<object_type>,<object_ID>, <object_info>,...,<object_info>`.

This data can be read at the following path: `/sys/kernel/debug/mlx5/<BDF>/steering/fdb/<domain_handle>`.

Example:

```
# cat /sys/kernel/debug/mlx5/0000:82:00.0/steering/fdb/dmn_000018644
3100,0x55caa4621c50,0xee802,4,65533
3101,0x55caa4621c50,0xe0100008
```

You can then use the steering dump parser to make the output more human-readable.

The parser can be found in this GitHub repository.

17.10.3.2.3 vPort Match Mode

OVS-kernel support two modes that define how the rules match on vport.

Mode	Description
Metadata	Rules match on metadata instead of vport number (default mode). This mode is needed to support SR-IOV live migration and dual-port RoCE. Matching on Metadata can have a performance impact.
Legacy	Rules match on vport number. In this mode, performance can be higher in comparison to Metadata. It can be used only if SR-IOV live migration or dual port RoCE are enabled/used.

vPort match mode can be controlled via sysfs:

- Set legacy:

  ```
  echo legacy > /sys/class/net/<PF netdev>/compat/devlink/vport_match_mode
  ```

- Set metadata:

  ```
  echo metadata > /sys/class/net/<PF netdev>/compat/devlink/vport_match_mode
  ```

> This mode must be set prior to moving to switchdev.

17.10.3.2.4 Flow Table Large Group Number

Offloaded flows, including connection tracking (CT), are added to the virtual switch forwarding data base (FDB) flow tables. FDB tables have a set of flow groups, where each flow group saves the same traffic pattern flows. For example, for CT offloaded flow, TCP and UDP are different traffic patterns which end up in two different flow groups.

A flow group has a limited size to save flow entries. By default, the driver has 15 big FDB flow groups. Each of these big flow groups can save 4M/(15+1)=256k different 5-tuple flow entries at most. For scenarios with more than 15 traffic patterns, the driver provides a module parameter (`num_of_groups`) to allow customization and performance tuning.

The mode can be controlled via module param or devlink API for kernels that support it:

- Module param:

  ```
  echo <num_of_groups> > /sys/module/mlx5_core/parameters/num_of_groups
  ```

- Devlink:

  ```
  devlink dev param set pci/0000:82:00.0 name fdb_large_groups cmode driverinit value 20
  ```

> The change takes effect immediately if no flows are inside the FDB table (no traffic running and all offloaded flows are aged out). And it can be dynamically changed without reloading the driver. If there are still offloaded flows when changing this parameter, it takes effect after all flows have aged out.

17.10.3.3 Open vSwitch Configuration

OVS configuration is a simple OVS bridge configuration with switchdev.

1. Run the OVS service:

   ```
   systemctl start openvswitch
   ```

2. Create an OVS bridge (named `ovs-sriov` here):

   ```
   ovs-vsctl add-br ovs-sriov
   ```

3. Enable hardware offload (disabled by default):

   ```
   ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
   ```

4. Restart the OVS service:

   ```
   systemctl restart openvswitch
   ```

   This step is required for hardware offload changes to take effect.

5. Add the PF and the VF representor netdevices as OVS ports:

```
ovs-vsctl add-port ovs-sriov enp4s0f0
ovs-vsctl add-port ovs-sriov enp4s0f0_0
ovs-vsctl add-port ovs-sriov enp4s0f0_1
```

Make sure to bring up the PF and representor netdevices:

```
ip link set dev enp4s0f0 up
ip link set dev enp4s0f0_0 up
ip link set dev enp4s0f0_1 up
```

The PF represents the uplink (wire):

```
# ovs-dpctl show
system@ovs-system:
        lookups: hit:0 missed:192 lost:1
        flows: 2
        masks: hit:384 total:2 hit/pkt:2.00
        port 0: ovs-system (internal)
        port 1: ovs-sriov (internal)
        port 2: enp4s0f0
        port 3: enp4s0f0_0
        port 4: enp4s0f0_1
```

6. Run traffic from the VFs and observe the rules added to the OVS data-path:

```
# ovs-dpctl dump-flows

recirc_id(0),in_port(3),eth(src=e4:11:22:33:44:50,dst=e4:1d:2d:a5:f3:9d),
eth_type(0x0800),ipv4(frag=no), packets:33, bytes:3234, used:1.196s, actions:2

recirc_id(0),in_port(2),eth(src=e4:1d:2d:a5:f3:9d,dst=e4:11:22:33:44:50),
eth_type(0x0800),ipv4(frag=no), packets:34, bytes:3332, used:1.196s, actions:3
```

In this example, the ping is initiated from VF0 (OVS port 3) to the outer node (OVS port 2), where the VF MAC is `e4:11:22:33:44:50` and the outer node MAC is `e4:1d:2d:a5:f3:9d`. As previously shown, two OVS rules are added, one in each direction.

Users can also verify offloaded packets by adding `type=offloaded` to the command. For example:

```
ovs-appctl dpctl/dump-flows type=offloaded
```

17.10.3.4 OVS Performance Tuning

17.10.3.4.1 Flow Aging

The aging timeout of OVS is given in milliseconds and can be controlled by running:

```
ovs-vsctl set Open_vSwitch . other_config:max-idle=30000
```

17.10.3.4.2 TC Policy

Specifies the policy used with hardware offloading:

- `none` - adds a TC rule to both the software and the hardware (default)
- `skip_sw` - adds a TC rule only to the hardware
- `skip_hw` - adds a TC rule only to the software

Example:

```
ovs-vsctl set Open_vSwitch . other_config:tc-policy=skip_sw
```

⚠ TC policy should only be used for debugging purposes.

17.10.3.4.3 max-revalidator

Specifies the maximum time (in milliseconds) for the revalidator threads to wait for kernel statistics before executing flow revalidation.

```
ovs-vsctl set Open_vSwitch . other_config:max-revalidator=10000
```

17.10.3.4.4 n-handler-threads

Specifies the number of threads for software datapaths to use to handle new flows.

```
ovs-vsctl set Open_vSwitch . other_config:n-handler-threads=4
```

The default value is the number of online CPU cores minus the number of revalidators.

17.10.3.4.5 n-revalidator-threads

Specifies the number of threads for software datapaths to use to revalidate flows in the datapath.

```
ovs-vsctl set Open_vSwitch . other_config:n-revalidator-threads=4
```

17.10.3.4.5.1 vlan-limit

Limits the number of VLAN headers that can be matched to the specified number.

```
ovs-vsctl set Open_vSwitch . other_config:vlan-limit=2
```

17.10.3.5 Basic TC Rules Configuration

Offloading rules can also be added directly, and not only through OVS, using the `tc` utility.

To create an offloading rule using TC:

1. Create an ingress qdisc (queueing discipline) for each interface that you wish to add rules into:

   ```
   tc qdisc add dev enp4s0f0 ingress
   tc qdisc add dev enp4s0f0_0 ingress
   tc qdisc add dev enp4s0f0_1 ingress
   ```

2. Add TC rules using flower classifier in the following format:

```
tc filter add dev NETDEVICE ingress protocol PROTOCOL prio PRIORITY [chain CHAIN] flower [MATCH_LIST]
[action ACTION_SPEC]
```

A list of supported matches (specifications) and actions can be found in section "Classification Fields (Matches)".

3. Dump the existing `tc` rules using flower classifier in the following format:

```
tc [-s] filter show dev NETDEVICE ingress
```

17.10.3.6 SR-IOV VF LAG

SR-IOV VF LAG allows the NIC's physical functions (PFs) to get the rules that the OVS tries to offload to the bond net-device, and to offload them to the hardware e-switch.

The supported bond modes are as follows:

- Active-backup
- XOR
- LACP

SR-IOV VF LAG enables complete offload of the LAG functionality to the hardware. The bonding creates a single bonded PF port. Packets from the up-link can arrive from any of the physical ports and are forwarded to the bond device.

When hardware offload is used, packets from both ports can be forwarded to any of the VFs. Traffic from the VF can be forwarded to both ports according to the bonding state. This means that when in active-backup mode, only one PF is up, and traffic from any VF goes through this PF. When in XOR or LACP mode, if both PFs are up, traffic from any VF is split between these two PFs.

17.10.3.6.1 SR-IOV VF LAG Configuration on ASAP^2

To enable SR-IOV VF LAG, both physical functions of the NIC must first be configured to SR-IOV switchdev mode, and only afterwards bond the up-link representors.

The following example shows the creation of a bond interface over two PFs:

1. Load the bonding device and subordinate the up-link representor (currently PF) net-device devices:

```
modprobe bonding mode=802.3ad
Ifup bond0 (make sure ifcfg file is present with desired bond configuration)
ip link set enp4s0f0 master bond0
ip link set enp4s0f1 master bond0
```

2. Add the VF representor net-devices as OVS ports. If tunneling is not used, add the bond device as well.

```
ovs-vsctl add-port ovs-sriov bond0
ovs-vsctl add-port ovs-sriov enp4s0f0_0
ovs-vsctl add-port ovs-sriov enp4s0f1_0
```

3. Bring up the PF and the representor netdevices:

```
ip link set dev bond0 up
ip link set dev enp4s0f0_0 up
ip link set dev enp4s0f1_0 up
```

Once the SR-IOV VF LAG is configured, all VFs of the two PFs become part of the bond and behave as described above.

17.10.3.6.2 Using TC with VF LAG

Both rules can be added either with or without shared block:

- With shared block (supported from kernel 4.16 and RHEL/CentOS 7.7 and above):

```
tc qdisc add dev bond0 ingress_block 22 ingress
tc qdisc add dev ens4p0 ingress_block 22 ingress
tc qdisc add dev ens4p1 ingress_block 22 ingress
```

  a. Add drop rule:

```
# tc filter add block 22 protocol arp parent ffff: prio 3 \
    flower \
        dst_mac e4:11:22:11:4a:51 \
        action drop
```

  b. Add redirect rule from bond to representor:

```
# tc filter add block 22 protocol arp parent ffff: prio 3 \
        flower \
        dst_mac e4:11:22:11:4a:50 \
        action mirred egress redirect dev ens4f0_0
```

  c. Add redirect rule from representor to bond:

```
# tc filter add dev ens4f0_0 protocol arp parent ffff: prio 3 \
    flower \
        dst_mac ec:0d:9a:8a:28:42 \
        action mirred egress redirect dev bond0
```

- Without shared block (supported from kernel 4.15 and below):

  a. Add redirect rule from bond to representor:

```
# tc filter add dev bond0 protocol arp parent ffff: prio 1 \
        flower \
        dst_mac e4:11:22:11:4a:50 \
        action mirred egress redirect dev ens4f0_0
```

  b. Add redirect rule from representor to bond:

```
# tc filter add dev ens4f0_0 protocol arp parent ffff: prio 3 \
        flower \
        dst_mac ec:0d:9a:8a:28:42 \
        action mirred egress redirect dev bond0
```

17.10.3.7 Classification Fields (Matches)

OVS-Kernel supports multiple classification fields which packets can fully or partially match.

17.10.3.7.1 Ethernet Layer 2

- Destination MAC
- Source MAC
- Ethertype

Supported on all kernels.

In OVS dump flows:

```
skb_priority(0/0),skb_mark(0/0),in_port(eth6),eth(src=00:02:10:40:10:0d,dst=68:54:ed:00:af:de),eth_type(0x8100),
packets:1981, bytes:206024, used:0.440s, dp:tc, actions:eth7
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol arp pref 1 \
flower \
dst_mac e4:1d:2d:5d:25:35 \
src_mac e4:1d:2d:5d:25:34 \
action mirred egress redirect dev $NIC
```

17.10.3.7.2 IPv4/IPv6

- Source address
- Destination address
- Protocol
    - TCP/UDP/ICMP/ICMPv6
- TOS
- TTL (HLIMIT)

Supported on all kernels.

In OVS dump flows:

```
Ipv4:
ipv4(src=0.0.0.0/0.0.0.0,dst=0.0.0.0/0.0.0.0,proto=17,tos=0/0,ttl=0/0,frag=no)
Ipv6:
ipv6(src=::/::,dst=1:1:1::3:1040:1008,label=0/0,proto=58,tclass=0/0x3,hlimit=64),
```

Using TC rules:

```
IPv4:
tc filter add dev $rep parent ffff: protocol ip pref 1 \
flower \
dst_ip 1.1.1.1 \
src_ip 1.1.1.2 \
ip_proto TCP \
ip_tos 0x3 \
ip_ttl 63 \
action mirred egress redirect dev $NIC


IPv6:
tc filter add dev $rep parent ffff: protocol ipv6 pref 1 \
flower \
dst_ip 1:1:1::3:1040:1009 \
src_ip 1:1:1::3:1040:1008 \
ip_proto TCP \
ip_tos 0x3 \
ip_ttl 63\
action mirred egress redirect dev $NIC
```

17.10.3.7.3 TCP/UDP Source and Destination Ports and TCP Flags

- TCP/UDP source and destinations ports
- TCP flags

Supported on kernel >4.13 and RHEL >7.5.

In OVS dump flows:

```
TCP: tcp(src=0/0,dst=32768/0x8000),
UDP: udp(src=0/0,dst=32768/0x8000),
TCP flags: tcp_flags(0/0)
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol ip pref 1 \
flower \
ip_proto TCP \
dst_port 100 \
src_port 500 \
tcp_flags 0x4/0x7 \
action mirred egress redirect dev $NIC
```

17.10.3.7.4 VLAN

- ID
- Priority
- Inner vlan ID and Priority

Supported kernels: All (QinQ: kernel 4.19 and higher, and RHEL 7.7 and higher).

In OVS dump flows:

```
eth_type(0x8100),vlan(vid=2347,pcp=0),
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol 802.1Q pref 1 \
                    flower \
                    vlan_ethtype 0x800 \
                    vlan_id 100 \
                    vlan_prio 0 \
                    action mirred egress redirect dev $NIC
QinQ:
tc filter add dev $rep parent ffff: protocol 802.1Q pref 1 \
                    flower \
                    vlan_ethtype 0x8100  \
                    vlan_id 100 \
                    vlan_prio 0 \
                    cvlan_id 20 \
                    cvlan_prio 0 \
                    cvlan_ethtype 0x800 \
                    action mirred egress redirect dev $NIC
```

17.10.3.7.5 Tunnel

- ID (Key)
- Source IP address
- Destination IP address
- Destination port
- TOS (supported from kernel 4.19 and above & RHEL 7.7 and above)
- TTL (support from kernel 4.19 and above & RHEL 7.7 and above)

- Tunnel options (Geneve)

Supported kernels:

- VXLAN: All
- GRE: Kernel >5.0, RHEL 7.7 and above
- Geneve: Kernel >5.0, RHEL 7.7 and above

In OVS dump flows:

```
tunnel(tun_id=0x5,src=121.9.1.1,dst=131.10.1.1,ttl=0/0,tp_dst=4789,flags(+key))
```

Using TC rules:

```
# tc filter add dev $rep protocol 802.1Q parent ffff: pref 1
flower \
vlan_ethtype 0x800 \
vlan_id 100 \
vlan_prio 0 \
action mirred egress redirect dev $NIC
QinQ:
# tc filter add dev vxlan100 protocol ip parent ffff: \
                flower \
                         skip_sw \
                         dst_mac e4:11:22:11:4a:51 \
                         src_mac e4+:11:22:11:4a:50 \
                         enc_src_ip 20.1.11.1 \
                         enc_dst_ip 20.1.12.1 \
                         enc_key_id 100 \
                         enc_dst_port 4789 \
                         action tunnel_key unset \
                         action mirred egress redirect dev ens4f0_0
```

17.10.3.8 Supported Actions

17.10.3.8.1 Forward

Forward action allows for packet redirection:

- From VF to wire
- Wire to VF
- VF to VF

Supported on all kernels.

In OVS dump flows:

```
skb_priority(0/0),skb_mark(0/0),in_port(eth6),eth(src=00:02:10:40:10:0d,dst=68:54:ed:00:af:de),eth_type(0x8100),
packets:1981, bytes:206024, used:0.440s, dp:tc, actions:eth7
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol arp pref 1 \
                        flower \
                        dst_mac e4:1d:2d:5d:25:35 \
                        src_mac e4:1d:2d:5d:25:34 \
                        action mirred egress redirect dev $NIC
```

17.10.3.8.2 Drop

Drop action allows to drop incoming packets.

Supported on all kernels.

In OVS dump flows:

```
skb_priority(0/0),skb_mark(0/0),in_port(eth6),eth(src=00:02:10:40:10:0d,dst=68:54:ed:00:af:de),eth_type(0x8100),
packets:1981, bytes:206024, used:0.440s, dp:tc, actions:drop
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol arp pref 1 \
                            flower \
                            dst_mac e4:1d:2d:5d:25:35 \
                            src_mac e4:1d:2d:5d:25:34 \
                            action drop
```

17.10.3.8.3 Statistics

By default, each flow collects the following statistics:

- Packets - number of packets which hit the flow
- Bytes - total number of bytes which hit the flow
- Last used - the amount of time passed since last packet hit the flow

Supported on all kernels.

In OVS dump flows:

```
skb_priority(0/0),skb_mark(0/0),in_port(eth6),eth(src=00:02:10:40:10:0d,dst=68:54:ed:00:af:de),eth_type(0x8100),
packets:1981, bytes:206024, used:0.440s, dp:tc, actions:drop
```

Using TC rules:

```
#tc -s filter show dev $rep ingress

filter protocol ip pref 2 flower chain 0
filter protocol ip pref 2 flower chain 0 handle 0x2
eth_type ipv4
ip_proto tcp
src_ip 192.168.140.100
src_port 80
skip_sw
in_hw
    action order 1: mirred (Egress Redirect to device p0v11_r) stolen
    index 34 ref 1 bind 1 installed 144 sec used 0 sec
    Action statistics:
    Sent 388344 bytes 2942 pkt (dropped 0, overlimits 0 requeues 0)
    backlog 0b 0p requeues 0
```

17.10.3.8.4 Tunnels: Encapsulation/Decapsulation

OVS-kernel supports offload of tunnels using encapsulation and decapsulation actions.

- Encapsulation - pushing of tunnel header is supported on Tx
- Decapsulation - popping of tunnel header is supported on Rx

Supported Tunnels:

- VXLAN (IPv4/IPv6) - supported on all Kernels
- GRE (IPv4/IPv6) - supported on kernel 5.0 and above & RHEL 7.6 and above
- Geneve (IPv4/IPv6) - supported on kernel 5.0 and above & RHEL 7.6 and above

OVS configuration:

In case of offloading tunnel, the PF/bond should not be added as a port in the OVS datapath. It should rather be assigned with the IP address to be used for encapsulation.

The following example shows two hosts (PFs) with IPs 1.1.1.177 and 1.1.1.75, where the PF device on both hosts is enp4s0f0, and the VXLAN tunnel is set with VNID 98:

- On the first host:

```
# ip addr add 1.1.1.177/24 dev enp4s0f1
# ovs-vsctl add-port ovs-sriov vxlan0 -- set interface vxlan0 type=vxlan options:local_ip=1.1.1.177
 options:remote_ip=1.1.1.75 options:key=98
```

- On the second host:

```
# ip addr add 1.1.1.75/24 dev enp4s0f1
# ovs-vsctl add-port ovs-sriov vxlan0 -- set interface vxlan0 type=vxlan options:local_ip=1.1.1.75
 options:remote_ip=1.1.1.177 options:key=98
```

For a GRE IPv4 tunnel, use `type=gre`. For a GRE IPv6 tunnel, use `type=ip6gre`. For a Geneve tunnel, use `type=geneve`.

When encapsulating guest traffic, the VF's device MTU must be reduced to allow the host/hardware to add the encap headers without fragmenting the resulted packet. As such, the VF's MTU must be lowered by 50 bytes from the uplink MTU for IPv4 and 70 bytes for IPv6.

Tunnel offload using TC rules:

```
Encapsulation:
# tc filter add dev ens4f0_0 protocol 0x806 parent ffff: \
                flower \
                        skip_sw \
                        dst_mac e4:11:22:11:4a:51 \
                        src_mac e4:11:22:11:4a:50 \
                action tunnel_key set \
                src_ip 20.1.12.1 \
                dst_ip 20.1.11.1 \
                id 100 \
                action mirred egress redirect dev vxlan100

Decapsulation:
# tc filter add dev vxlan100 protocol 0x806 parent ffff: \
                flower \
                         skip_sw \
                         dst_mac e4:11:22:11:4a:51 \
                         src_mac e4:11:22:11:4a:50 \
                         enc_src_ip 20.1.11.1 \
                         enc_dst_ip 20.1.12.1 \
                         enc_key_id 100 \
                         enc_dst_port 4789 \
                action tunnel_key unset \
                action mirred egress redirect dev ens4f0_0
```

17.10.3.8.5 VLAN Push/Pop

OVS-kernel supports offload of VLAN header push/pop actions:

- Push - pushing of VLAN header is supported on Tx
- Pop - popping of tunnel header is supported on Rx

17.10.3.8.5.1 OVS Configuration

Add a tag=$TAG section for the OVS command line that adds the representor ports. For example, VLAN ID 52 is being used here.

```
# ovs-vsctl add-port ovs-sriov enp4s0f0
# ovs-vsctl add-port ovs-sriov enp4s0f0_0 tag=52
# ovs-vsctl add-port ovs-sriov enp4s0f0_1 tag=52
```

The PF port should not have a VLAN attached. This will cause OVS to add VLAN push/pop actions when managing traffic for these VFs.

Dump Flow Example

```
recirc_id(0),in_port(3),eth(src=e4:11:22:33:44:50,dst=00:02:c9:e9:bb:b2),eth_type(0x0800),ipv4(frag=no), \
packets:0, bytes:0, used:never, actions:push_vlan(vid=52,pcp=0),2

recirc_id(0),in_port(2),eth(src=00:02:c9:e9:bb:b2,dst=e4:11:22:33:44:50),eth_type(0x8100), \
vlan(vid=52,pcp=0),encap(eth_type(0x0800),ipv4(frag=no)), packets:0, bytes:0, used:never, actions:pop_vlan,3
```

VLAN Offload Using TC Rules Example

```
# tc filter add dev ens4f0_0 protocol ip parent ffff: \
                flower \
                        skip_sw \
                        dst_mac e4:11:22:11:4a:51 \
                        src_mac e4:11:22:11:4a:50 \
                action vlan push id 100 \
                action mirred egress redirect dev ens4f0

# tc filter add dev ens4f0 protocol 802.1Q parent ffff: \
                flower \
                        skip_sw \
                        dst_mac e4:11:22:11:4a:51 \
                        src_mac e4:11:22:11:4a:50 \
                        vlan_ethtype 0x800 \
                        vlan_id 100 \
                        vlan_prio 0 \
                action vlan pop \
                action mirred egress redirect dev ens4f0_0
```

17.10.3.8.5.2 TC Configuration

Example of VLAN Offloading with popping header on Tx and pushing on Rx using TC rules:

```
# tc filter add dev ens4f0_0 ingress protocol 802.1Q parent ffff: \
        flower \
                vlan_id 100 \
        action vlan pop \
        action tunnel_key set \
                src_ip 4.4.4.1 \
                dst_ip 4.4.4.2 \
                dst_port 4789 \
                id 42 \
        action mirred egress redirect dev vxlan0

# tc filter add dev vxlan0 ingress protocol all parent ffff: \
        flower \
                enc_dst_ip 4.4.4.1 \
                enc_src_ip 4.4.4.2 \
                enc_dst_port 4789 \
                enc_key_id 42 \
        action tunnel_key unset \
        action vlan push id 100 \
        action mirred egress redirect dev ens4f0_0
```

17.10.3.8.6 Header Rewrite

This action allows for modifying packet fields.

17.10.3.8.7 Ethernet Layer 2

- Destination MAC
- Source MAC

Supported kernels:

- Kernel 4.14 and above
- RHEL 7.5 and above

In OVS dump flows:

```
skb_priority(0/0),skb_mark(0/0),in_port(eth6),eth(src=00:02:10:40:10:0d,dst=68:54:ed:00:af:de),eth_type(0x8100), 
packets:1981, bytes:206024, used:0.440s, dp:tc, actions: set(eth(src=68:54:ed:00:f4:ab,dst=fa:16:3e:dd:69:c4)),eth7
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol arp pref 1 \
                            flower \
                            dst_mac e4:1d:2d:5d:25:35 \
                            src_mac e4:1d:2d:5d:25:34 \
                    action pedit ex \
                    munge eth dst set 20:22:33:44:55:66 \
                    munge eth src set aa:ba:cc:dd:ee:fe \
                    action mirred egress redirect dev $NIC
```

17.10.3.8.8 IPv4/IPv6

- Source address
- Destination address
- Protocol
- TOS
- TTL (HLIMIT)

Supported kernels:

- Kernel 4.14 and above
- RHEL 7.5 and above

In OVS dump flows:

```
Ipv4:
    set(eth(src=de:e8:ef:27:5e:45,dst=00:00:01:01:01:01)),
    set(ipv4(src=10.10.0.111,dst=10.20.0.122,ttl=63))
Ipv6:
    set(ipv6(dst=2001:1:6::92eb:fcbe:f1c8,hlimit=63)),
```

Using TC rules:

```
IPv4:
tc filter add dev $rep parent ffff: protocol ip pref 1 \
                            flower \
                            dst_ip 1.1.1.1 \
                            src_ip 1.1.1.2 \
                            ip_proto TCP \
                            ip_tos 0x3 \
                            ip_ttl 63 \
                    pedit ex \
                    munge ip src set 2.2.2.1 \
                    munge ip dst set 2.2.2.2 \
                    munge ip tos set 0 \
                    munge ip ttl dec \
                    action mirred egress redirect dev $NIC


IPv6:
tc filter add dev $rep parent ffff: protocol ipv6 pref 1 \
                            flower \
                            dst_ip 1:1:1::3:1040:1009 \
                            src_ip 1:1:1::3:1040:1008 \
                            ip_proto tcp \
                            ip_tos 0x3 \
                            ip_ttl 63\
                    pedit ex \
                    munge ipv6 src set 2:2:2::3:1040:1009 \
                    munge ipv6 dst set 2:2:2::3:1040:1008 \
                    munge ipv6 hlimit dec \
                    action mirred egress redirect dev $NIC
```

IPv4 and IPv6 header rewrite is only supported with match on UDP/TCP/ICMP protocols.

17.10.3.8.8.1 TCP/UDP Source and Destination Ports

- TCP/UDP source and destinations ports

Supported kernels:

- Kernel 4.16 and above
- RHEL 7.6 and above

In OVS dump flows:

```
TCP:
        set(tcp(src= 32768/0xffff,dst=32768/0xffff)),
UDP:
        set(udp(src= 32768/0xffff,dst=32768/0xffff)),
```

Using TC rules:

```
TCP:
        tc filter add dev $rep parent ffff: protocol ip pref 1 \
                            flower \
                            dst_ip 1.1.1.1 \
                            src_ip 1.1.1.2 \
                            ip_proto tcp \
                            ip_tos 0x3 \
                            ip_ttl 63 \
                    pedit ex \
                    pedit ex munge ip tcp sport set 200
                    pedit ex munge ip tcp dport set 200
                    action mirred egress redirect dev $NIC
UDP:
        tc filter add dev $rep parent ffff: protocol ip pref 1 \
                            flower \
                            dst_ip 1.1.1.1 \
                            src_ip 1.1.1.2 \
                            ip_proto udp \
                            ip_tos 0x3 \
                            ip_ttl 63 \
                    pedit ex \
                    pedit ex munge ip udp sport set 200
                    pedit ex munge ip udp dport set 200
                    action mirred egress redirect dev $NIC
```

17.10.3.8.8.2 VLAN

- ID

Supported on all kernels.

In OVS dump flows:

```
Set(vlan(vid=2347,pcp=0/0)),
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol 802.1Q pref 1 \
                    flower \
                    vlan_ethtype 0x800 \
                    vlan_id 100 \
                    vlan_prio 0 \
            action vlan modify id 11 pipe
            action mirred egress redirect dev $NIC
```

17.10.3.8.9 Connection Tracking

The TC connection tracking (CT) action performs CT lookup by sending the packet to netfilter conntrack module. Newly added connections may be associated, via the `ct commit` action, with a 32 bit mark, 128 bit label, and source/destination NAT values.

The following example allows ingress TCP traffic from the uplink representor to `vf1_rep`, while assuring that egress traffic from `vf1_rep` is only allowed on established connections. In addition, mark and source IP NAT is applied.

In OVS dump flows:

```
ct(zone=2,nat)
ct_state(+est+trk)
actions:ct(commit,zone=2,mark=0x4/0xffffffff,nat(src=5.5.5.5))
```

Using TC rules:

```
# tc filter add dev $uplink_rep ingress chain 0 prio 1 proto ip \
                    flower \
                    ip_proto tcp   \
                    ct_state -trk \
               action ct zone 2 nat pipe
               action goto chain 2
# tc filter add dev $uplink_rep ingress chain 2 prio 1 proto ip \
                     flower \
                     ct_state +trk+new \
               action ct zone 2 commit mark 0xbb nat src addr 5.5.5.7 pipe \
               action mirred egress redirect dev $vf1_rep
# tc filter add dev $uplink_rep ingress chain 2 prio 1 proto ip \
                    flower \
                    ct_zone 2 \
                    ct_mark 0xbb \
                    ct_state +trk+est \
                action mirred egress redirect dev $vf1_rep

// Setup filters on $vf1_rep, allowing only established connections of zone 2 through, and reverse nat (dst nat in
this case)

# tc filter add dev $vf1_rep ingress chain 0 prio 1 proto ip \
                     flower \
                     ip_proto tcp \
                     ct_state -trk \
                action ct zone 2 nat pipe \
                action goto chain 1
# tc filter add dev $vf1_rep ingress chain 1 prio 1 proto ip \
                     flower \
                     ct_zone 2 \
                     ct_mark 0xbb \
                     ct_state +trk+est \
                action mirred egress redirect dev eth0
```

17.10.3.8.9.1 CT Performance Tuning

- Max offloaded connections - specifies the limit on the number of offloaded connections.
  Example:

  ```
  devlink dev param set pci/${pci_dev} name ct_max_offloaded_conns value $max cmode runtime
  ```

- Allow mixed NAT/non-NAT CT - allows offloading of the following scenario:

  ```
  •   cookie=0x0, duration=21.843s, table=0, n_packets=4838718, n_bytes=241958846, ct_state=-
  trk,ip,in_port=enp8s0f0 actions=ct(table=1,zone=2)
  •   cookie=0x0, duration=21.823s, table=1, n_packets=15363, n_bytes=773526, ct_state=+new+trk,ip,in_port=en
  p8s0f0 actions=ct(commit,zone=2,nat(dst=11.11.11.11)),output:"enp8s0f0_1" •  cookie=0x0, duration=21.806s,
  table=1, n_packets=4767594, n_bytes=238401190, ct_state=+est+trk,ip,in_port=enp8s0f0 actions=ct(zone=2,nat)
  ,output:"enp8s0f0_1"
  ```

  Example:

```
echo enable > /sys/class/net/<device>/compat/devlink/ct_action_on_nat_conns
```

17.10.3.8.10 Forward to Chain (TC Only)

TC interface supports adding flows on different chains. Only chain 0 is accessed by default. Access to the other chains requires using the `goto` action.

In this example, a flow is created on chain 1 without any match and redirect to wire.

The second flow is created on chain 0 and match on source MAC and action `goto` chain 1.

This example simulates simple MAC spoofing:

```
#tc filter add dev $rep parent ffff: protocol all chain 1 pref 1 \
                    flower \
                action mirred egress redirect dev $NIC

#tc filter add dev $rep parent ffff: protocol all chain 1 pref 1 \
                    flower \
                    src_mac aa:bb:cc:aa:bb:cc \
                action goto chain 1
```

17.10.3.9 Port Mirroring: Flow-based VF Traffic Mirroring for ASAP²

Unlike para-virtual configurations, when the VM traffic is offloaded to hardware via SR-IOV VF, the host-side admin cannot snoop the traffic (e.g., for monitoring).

ASAP² uses the existing mirroring support in OVS and TC along with the enhancement to the offloading logic in the driver to allow mirroring the VF traffic to another VF.

The mirrored VF can be used to run traffic analyzer (e.g., tcpdump, wireshark, etc.) and observe the traffic of the VF being mirrored.

The following example shows the creation of port mirror on the following configuration:

```
# ovs-vsctl show
  09d8a574-9c39-465c-9f16-47d81c12f88a
      Bridge br-vxlan
                  Port "enp4s0f0_1"
                     Interface "enp4s0f0_1"
                  Port "vxlan0"
                      Interface "vxlan0"
                                  type: vxlan
                                  options: {key="100", remote_ip="192.168.1.14"}
                  Port "enp4s0f0_0"
                      Interface "enp4s0f0_0"
                  Port "enp4s0f0_2"
                      Interface "enp4s0f0_2"
                  Port br-vxlan
                      Interface br-vxlan
                                  type: internal
      ovs_version: "2.14.1"
```

- To set `enp4s0f0_0` as the mirror port and mirror all the traffic:

```
# ovs-vsctl -- --id=@p get port enp4s0f0_0 \
            -- --id=@m create mirror name=m0 select-all=true output-port=@p \
            -- set bridge br-vxlan mirrors=@m
```

- To set `enp4s0f0_0` as the mirror port, only mirror the traffic, and set `enp4s0f0_1` as the destination port:

```
# ovs-vsctl -- --id=@p1 get port enp4s0f0_0 \
            -- --id=@p2 get port enp4s0f0_1 \
            -- --id=@m create mirror name=m0 select-dst-port=@p2 output-port=@p1 \
            -- set bridge br-vxlan mirrors=@m
```

- To set `enp4s0f0_0` as the mirror port, only mirror the traffic, and set `enp4s0f0_1` as the source port:

```
# ovs-vsctl -- --id=@p1 get port enp4s0f0_0 \
            -- --id=@p2 get port enp4s0f0_1 \
            -- --id=@m create mirror name=m0 select-src-port=@p2 output-port=@p1 \
            -- set bridge br-vxlan mirrors=@m
```

- To set `enp4s0f0_0` as the mirror port and mirror all the traffic on `enp4s0f0_1`:

```
# ovs-vsctl -- --id=@p1 get port enp4s0f0_0 \
            -- --id=@p2 get port enp4s0f0_1 \
            -- --id=@m create mirror name=m0 select-dst-port=@p2 select-src-port=@p2 output-port=@p1 \
            -- set bridge br-vxlan mirrors=@m
```

To clear the mirror port:

```
ovs-vsctl clear bridge br-vxlan mirrors
```

Mirroring using TC:

- Mirror to VF:

```
tc filter add dev $rep parent ffff: protocol arp pref 1 \
                        flower \
                        dst_mac e4:1d:2d:5d:25:35 \
                        src_mac e4:1d:2d:5d:25:34 \
                        action mirred egress mirror dev $mirror_rep pipe \
                        action mirred egress redirect dev $NIC
```

- Mirror to tunnel:

```
tc filter add dev $rep parent ffff: protocol arp pref 1 \
                        flower \
                        dst_mac e4:1d:2d:5d:25:35 \
                        src_mac e4:1d:2d:5d:25:34 \
                action tunnel_key set \
                src_ip 1.1.1.1 \
                dst_ip 1.1.1.2 \
                dst_port 4789 \
                id 768 \
                pipe \
                action mirred egress mirror dev vxlan100 pipe \
                action mirred egress redirect dev $NIC
```

17.10.3.10 Forward to Multiple Destinations

Forwarding to up 32 destinations (representors and tunnels) is supported using TC:

- Example 1 – forwarding to 32 VFs:

```
tc filter add dev $NIC parent ffff: protocol arp pref 1 \
                        flower \
                        dst_mac e4:1d:2d:5d:25:35 \
                        src_mac e4:1d:2d:5d:25:34 \
                        action mirred egress mirror dev $rep0 pipe \
                        action mirred egress mirror dev $rep1 pipe \
...
                        action mirred egress mirror dev $rep30 pipe \
                        action mirred egress redirect dev $rep31
```

- Example 2 – forwarding to 16 tunnels:

```
tc filter add dev $rep parent ffff: protocol arp pref 1 \
                        flower \
                        dst_mac e4:1d:2d:5d:25:35 \
                        src_mac e4:1d:2d:5d:25:34 \
                        action tunnel_key set src_ip $ip_src dst_ip $ip_dst \
                        dst_port 4789 id 0 nocsum \
                        pipe action mirred egress mirror dev vxlan0 pipe \
                        action tunnel_key set src_ip $ip_src dst_ip $ip_dst \
                        dst_port 4789 id 1 nocsum \
                        pipe action mirred egress mirror dev vxlan0 pipe \
                        ...
                        action tunnel_key set src_ip $ip_src dst_ip $ip_dst \
                        dst_port 4789 id 15 nocsum \
                        pipe action mirred egress redirect dev vxlan0
```

TC supports up to 32 actions.

If header rewrite is used, then all destinations should have the same header rewrite.

If VLAN push/pop is used, then all destinations should have the same VLAN ID and actions.

17.10.3.11 sFlow

sFlow allows for monitoring traffic sent between two VMs on the same host using an sFlow collector.

The following example assumes the environment is configured as described later.

```
# ovs-vsctl show
  09d8a574-9c39-465c-9f16-47d81c12f88a
      Bridge br-vxlan
                  Port "enp4s0f0_1"
                    Interface "enp4s0f0_1"
                  Port "vxlan0"
                    Interface "vxlan0"
                                type: vxlan
                                options: {key="100", remote_ip="192.168.1.14"}
                  Port "enp4s0f0_0"
                    Interface "enp4s0f0_0"
                  Port "enp4s0f0_2"
                    Interface "enp4s0f0_2"
                  Port br-vxlan
                    Interface br-vxlan
                                type: internal
      ovs_version: "2.14.1"
```

To sample all traffic over the OVS bridge:

```
# ovs-vsctl -- --id=@sflow create sflow agent=\"$SFLOW_AGENT\" \
                                        target=\"$SFLOW_TARGET:$SFLOW_PORT\" \
                                        header=$SFLOW_HEADER \
                                        sampling=$SFLOW_SAMPLING polling=10 \
               -- set bridge br-vxlan sflow=@sflow
```

Parameter	Description
SFLOW_AGENT	Indicates that the sFlow agent should send traffic from SFLOW_AGENT 's IP address
SFLOW_TARGET	Remote IP address of the sFlow collector
SFLOW_HEADER	Size of packet header to sample (in bytes)
SFLOW_SAMPLING	Sample rate

To clear the sFlow configuration:

```
# ovs-vsctl clear bridge br-vxlan sflow
```

To list the sFlow configuration:

```
# ovs-vsctl list sflow
```

sFlow using TC:

```
Sample to VF
tc filter add dev $rep parent ffff: protocol arp pref 1 \
                        flower \
                        dst_mac e4:1d:2d:5d:25:35 \
                        src_mac e4:1d:2d:5d:25:34 \
                        action sample rate 10 group 5 trunc 96 \
                        action mirred egress redirect dev $NIC
```

> ⚠ A userspace application is needed to process the sampled packet from the kernel. An example is available on Github.

17.10.3.12 Rate Limit

OVS-kernel supports offload of VF rate limit using OVS configuration and TC.

The following example sets the rate limit to the VF related to representor `eth0` to 10Mb/s:

- OVS:

```
ovs-vsctl set interface eth0 ingress_policing_rate=10000
```

- TC:

```
tc_filter add dev eth0 root prio 1 protocol ip matchall skip_sw action police rate 10mbit burst 20k
```

17.10.3.13 Kernel Requirements

This kernel config should be enabled to support switchdev offload.

- `CONFIG_NET_ACT_CSUM` - needed for action csum
- `CONFIG_NET_ACT_PEDIT` - needed for header rewrite
- `CONFIG_NET_ACT_MIRRED` - needed for basic forward
- `CONFIG_NET_ACT_CT` - needed for CT (supported from kernel 5.6)
- `CONFIG_NET_ACT_VLAN` - needed for action vlan push/pop
- `CONFIG_NET_ACT_GACT`
- `CONFIG_NET_CLS_FLOWER`
- `CONFIG_NET_CLS_ACT`
- `CONFIG_NET_SWITCHDEV`
- `CONFIG_NET_TC_SKB_EXT` - needed for CT (supported from kernel 5.6)
- `CONFIG_NET_ACT_CT` - needed for CT (supported from kernel 5.6)
- `CONFIG_NFT_FLOW_OFFLOAD`
- `CONFIG_NET_ACT_TUNNEL_KEY`

- `CONFIG_NF_FLOW_TABLE` - needed for CT (supported from kernel 5.6)
- `CONFIG_SKB_EXTENSIONS` - needed for CT (supported from kernel 5.6)
- `CONFIG_NET_CLS_MATCHALL`
- `CONFIG_NET_ACT_POLICE`
- `CONFIG_MLX5_ESWITCH`

17.10.3.14 VF Metering

OVS-kernel supports offloading of VF metering (TX and RX) using sysfs. Metering of number of packets per second (PPS) and bytes per second (BPS) is supported.

The following example sets Rx meter on VF 0 with value 10Mb/s BPS:

```
echo 10000000 > /sys/class/net/enp4s0f0/device/sriov/0/meters/rx/bps/rate
echo 65536 > /sys/class/net/enp4s0f0/device/sriov/0/meters/rx/bps/burst
```

The following example sets Tx meter on VF 0 with value 1000 PPS:

```
echo 1000 > /sys/class/net/enp4s0f0/device/sriov/0/meters/tx/pps/rate
echo 100 > /sys/class/net/enp4s0f0/device/sriov/0/meters/tx/pps/burst
```

⚠ Both `rate` and `burst` must not be zero and `burst` may need to be adjusted according to the requirements.

The following counters can be used to query the number dropped packet/bytes:

```
cat /sys/class/net/enp8s0f0/device/sriov/0/meters/rx/pps/packets_dropped
cat /sys/class/net/enp8s0f0/device/sriov/0/meters/rx/pps/bytes_dropped
cat /sys/class/net/enp8s0f0/device/sriov/0/meters/rx/bps/packets_dropped
cat /sys/class/net/enp8s0f0/device/sriov/0/meters/rx/bps/bytes_dropped
cat /sys/class/net/enp8s0f0/device/sriov/0/meters/tx/pps/packets_dropped
cat /sys/class/net/enp8s0f0/device/sriov/0/meters/tx/pps/bytes_dropped
cat /sys/class/net/enp8s0f0/device/sriov/0/meters/tx/bps/packets_dropped
cat /sys/class/net/enp8s0f0/device/sriov/0/meters/tx/bps/bytes_dropped
```

17.10.3.15 Representor Metering

ⓘ Metering for uplink and VF representors traffic is supported.

Traffic going to a representor device can be a result of a miss in the embedded switch (eSwitch) FDB tables. This means that a packet which arrives from that representor into the eSwitch has not matched against the existing rules in the hardware FDB tables and must be forwarded to software to be handled there and is, therefore, forwarded to the originating representor device driver.

The meter allows to configure the max rate [packets per second] and max burst [packets] for traffic going to the representor driver. Any traffic exceeding values provided by the user are dropped in hardware. There are statistics that show the number of dropped packets.

The configuration of representor metering is done via `miss_rl_cfg`.

- Full path of the `miss_rl_cfg` parameter: `/sys/class/net//rep_config/miss_rl_cfg`
- Usage: `echo "<rate> <burst>" > /sys/class/net//rep_config/miss_rl_cfg`.

- `rate` is the max rate of packets allowed for this representor (in packets/sec units)
- `burst` is the max burst size allowed for this representor (in packets units)
- Both values must be specified. Both of their default values is 0, signifying unlimited rate and burst.

To view the amount of packets and bytes dropped due to traffic exceeding the user-provided rate and burst, two read-only sysfs for statistics are available:

- `/sys/class/net//rep_config/miss_rl_dropped_bytes` - counts how many FDB-miss bytes are dropped due to reaching the miss limits
- `/sys/class/net//rep_config/miss_rl_dropped_packets` - counts how many FDB-miss packets are dropped due to reaching the miss limits

17.10.3.16 OVS Metering

There are two types of meters, kpps (kilobits per second) and pktps (packets per second). OVS-Kernel supports offloading both of them.

The following example is to offload a kpps meter.

1. Create OVS meter with a target rate:

   ```
   ovs-ofctl -O OpenFlow13 add-meter ovs-sriov meter=1,kbps,band=type=drop,rate=204800
   ```

2. Delete the default rule:

   ```
   ovs-ofctl del-flows ovs-sriov
   ```

3. Configure OpenFlow rules:

   ```
   ovs-ofctl -O OpenFlow13 add-flow ovs-sriov 'ip,dl_dst=e4:11:22:33:44:50,actions= meter:1,output:enp4s0f0_0'
   ovs-ofctl -O OpenFlow13 add-flow ovs-sriov 'ip,dl_src=e4:11:22:33:44:50,actions= output:enp4s0f0'
   ovs-ofctl -O OpenFlow13 add-flow ovs-sriov 'arp,actions=normal'
   ```

   Here, the VF bandwidth on the receiving side is limited by the rate configured in step 1.
4. Run iperf server and be ready to receive UDP traffic. On the outer node, run iperf client to send UDP traffic to this VF. After traffic starts, check the offloaded meter rule:

   ```
   ovs-appctl dpctl/dump-flows --names type=offloaded

   recirc_id(0),in_port(enp4s0f0),eth(dst=e4:11:22:33:44:50),eth_type(0x0800),ipv4(frag=no), packets:11626587,
   bytes:17625889188, used:0.470s, actions:meter(0),enp4s0f0_0
   ```

To verify metering, iperf client should set the target bandwidth with a number which is larger than the meter rate configured. Then it should apparent that packets are received with the limited rate on the server side and the extra packets are dropped by hardware.

17.10.3.17 Multiport eSwitch Mode

The multiport eswitch mode allows adding rules on a VF representor with an action forwarding the packet to the physical port of the physical function. This can be used to implement failover or forward packets based on external information such as the cost of the route.

1. To configure multiport eswitch mode, the nvconfig parameter `LAG_RESOURCE_ALLOCATION` must be set.
2. After the driver loads, configure multiport eSwitch for each PF where `enp8s0f0` and `enp8s0f1` represent the netdevices for the PFs:

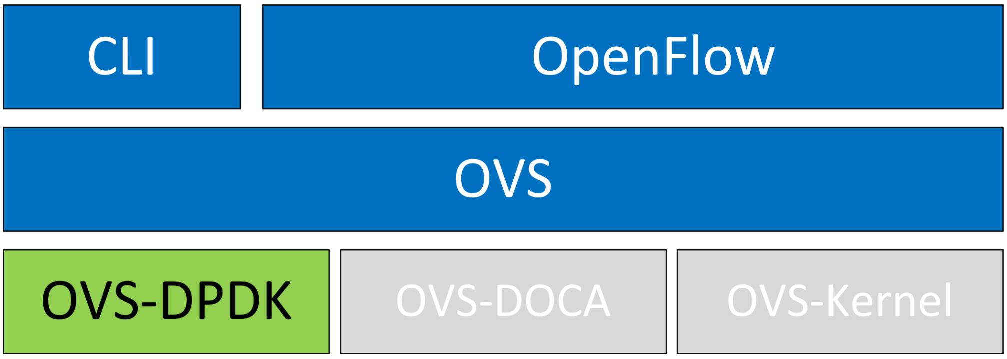
   ```
   echo multiport_esw > /sys/class/net/enp8s0f0/compat/devlink/lag_port_select_mode
   echo multiport_esw > /sys/class/net/enp8s0f1/compat/devlink/lag_port_select_mode
   ```

   The mode becomes operational after entering switchdev mode on both PFs.

Rule example:

```
tc filter add dev enp8s0f0_0 prot ip root flower dst_ip 7.7.7.7 action mirred egress redirect dev enp8s0f1
```

17.10.4 OVS-DPDK Hardware Acceleration



17.10.4.1 OVS-DPDK Hardware Offloads Configuration

To configure OVS-DPDK HW offloads:

1. Unbind the VFs:

   ```
   echo 0000:04:00.2 > /sys/bus/pci/drivers/mlx5_core/unbind
   echo 0000:04:00.3 > /sys/bus/pci/drivers/mlx5_core/unbind
   ```

   ⚠ VMs with attached VFs must be powered off to be able to unbind the VFs.

2. Change the e-switch mode from legacy to switchdev on the PF device (make sure all VFs are unbound). This also creates the VF representor netdevices in the host OS.

   ```
   echo switchdev > /sys/class/net/enp4s0f0/compat/devlink/mode
   ```

   To revert to SR-IOV legacy mode:

   ```
   echo legacy > /sys/class/net/enp4s0f0/compat/devlink/mode
   ```

   ⚠ This command removes the VF representor netdevices.

3. Bind the VFs:

```
echo 0000:04:00.2 > /sys/bus/pci/drivers/mlx5_core/bind
echo 0000:04:00.3 > /sys/bus/pci/drivers/mlx5_core/bind
```

4. Run the OVS service:

```
systemctl start openvswitch
```

5. Enable hardware offload (disabled by default):

```
ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-init=true
ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
```

6. Configure the DPDK whitelist:

```
ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-extra="-a
0000:01:00.0,representor=[0],dv_flow_en=1,dv_esw_en=1,dv_xmeta_en=1"
```

   Where `representor=[0-N]`.

7. Restart the OVS service:

```
systemctl restart openvswitch
```

   This step is required for the hardware offload changes to take effect.

8. Create OVS-DPDK bridge:

```
ovs-vsctl --no-wait add-br br0-ovs -- set bridge br0-ovs datapath_type=netdev
```

9. Add PF to OVS:

```
ovs-vsctl add-port br0-ovs pf -- set Interface pf type=dpdk options:dpdk-devargs=0000:88:00.0
```

10. Add representor to OVS:

```
ovs-vsctl add-port br0-ovs representor -- set Interface representor type=dpdk options:dpdk-devargs=0000:88:0
0.0,representor=[0]
```

   Where `representor=[0-N]`.

17.10.4.2 Offloading VXLAN Encapsulation/Decapsulation Actions

vSwitch in userspace requires an additional bridge. The purpose of this bridge is to allow use of the kernel network stack for routing and ARP resolution.

The datapath must look up the routing table and ARP table to prepare the tunnel header and transmit data to the output port.

17.10.4.2.1 Configuring VXLAN Encap/Decap Offloads

The configuration is done with:

- PF on 0000:03:00.0 PCIe and MAC 98:03:9b:cc:21:e8
- Local IP 56.56.67.1 – br-phy interface is configured to this IP
- Remote IP 56.56.68.1

To configure OVS-DPDK VXLAN:

1. Create a br-phy bridge:

```
ovs-vsctl add-br br-phy -- set Bridge br-phy datapath_type=netdev -- br-set-external-id br-phy bridge-id br-phy -- set bridge br-phy fail-mode=standalone other_config:hwaddr=98:03:9b:cc:21:e8
```

2. Attach PF interface to br-phy bridge:

```
ovs-vsctl add-port br-phy p0 -- set Interface p0 type=dpdk options:dpdk-devargs=0000:03:00.0
```

3. Configure IP to the bridge:

```
ip addr add 56.56.67.1/24 dev br-phy
```

4. Create a br-ovs bridge:

```
ovs-vsctl add-br br-ovs -- set Bridge br-ovs datapath_type=netdev -- br-set-external-id br-ovs bridge-id br-ovs -- set bridge br-ovs fail-mode=standalone
```

5. Attach representor to br-ovs:

```
ovs-vsctl add-port br-ovs pf0vf0 -- set Interface pf0vf0 type=dpdk options:dpdk-devargs=0000:03:00.0,representor=[0]
```

6. Add a port for the VXLAN tunnel:

```
ovs-vsctl add-port ovs-sriov vxlan0 -- set interface vxlan0 type=vxlan options:local_ip=56.56.67.1 options:remote_ip=56.56.68.1 options:key=45 options:dst_port=4789
```

17.10.4.3 CT Offload

CT enables stateful packet processing by keeping a record of currently open connections. OVS flows using CT can be accelerated using advanced NICs by offloading established connections.

To view offloaded connections, run:

```
ovs-appctl dpctl/offload-stats-show
```

17.10.4.4 SR-IOV VF LAG

To configure OVS-DPDK SR-IOV VF LAG:

1. Enable SR-IOV in the NIC firmware:

```
// It is recommended to query the parameters first to determine if change is needed, to save unnecessary reboot
mst start
mlxconfig -d <mst device> -y set PF_NUM_OF_VF_VALID=0 SRIOV_EN=1 NUM_OF_VFS=8
```

If configuration changes were made, unless the NIC is BlueField DPU Mode, perform a warm reboot of the Server OS. Otherwise, please perform BlueField System-Level Reset.

2. Allocate the desired number of VFs per port:

```
echo $n > /sys/class/net/<net name>/device/sriov_numvfs
```

3. Unbind all VFs:

```
echo <VF PCI> >/sys/bus/pci/drivers/mlx5_core/unbind
```

4. Change both devices' mode to switchdev:

```
devlink dev eswitch set pci/<PCI> mode switchdev
```

5. Create Linux bonding using kernel modules:

```
modprobe bonding mode=<desired mode>
```

> Other bonding parameters can be added here. The supported bond modes are: Active-backup, XOR and LACP.

6. Bring all PFs and VFs down:

```
ip link set <PF/VF> down
```

7. Attach both PFs to the bond:

```
ip link set <PF> master bond0
```

8. To use VF-LAG with OVS-DPDK, add the bond master (PF) to the bridge:

```
ovs-vsctl add-port br-phy p0 -- set Interface p0 type=dpdk options:dpdk-devargs=0000:03:00.0  options:dpdk-lsc-interrupt=true
```

9. Add representor `$N` of PF0 or PF1 to a bridge:

```
ovs-vsctl add-port br-phy rep$N -- set Interface rep$N type=dpdk options:dpdk-devargs=<PF0 PCI>,representor=pf0vf$N
```

Or:

```
ovs-vsctl add-port br-phy rep$N -- set Interface rep$N type=dpdk options:dpdk-devargs=<PF0 PCI>,representor=pf1vf$N
```

17.10.4.5 VirtIO Acceleration Through VF Relay: Software and Hardware vDPA

> Hardware vDPA is enabled by default. If your hardware does not support vDPA, the driver will fall back to Software vDPA.

To check which vDPA mode is activated on your driver, run: `ovs-ofctl -O OpenFlow14 dump-ports br0-ovs` and look for `hw-mode` flag.

⚠ This feature has not been accepted to the OVS-DPDK upstream yet, making its API subject to change.

In user space, there are two main approaches for communicating with a guest (VM), either through SR-IOV or virtio.

PHY ports (SR-IOV) allow working with port representor, which is attached to the OVS and a matching VF is given with pass-through to the guest. HW rules can process packets from up-link and direct them to the VF without going through SW (OVS). Therefore, using SR-IOV achieves the best performance.

However, SR-IOV architecture requires the guest to use a driver specific to the underlying HW. Specific HW driver has two main drawbacks:

- Breaks virtualization in some sense (guest is aware of the HW). It can also limit the type of images supported.
- Gives less natural support for live migration.

Using a virtio port solves both problems, however, it reduces performance and causes loss of some functionalities, such as, for some HW offloads, working directly with virtio. The netdev type dpdkvdpa solves this conflict as it is similar to the regular DPDK netdev yet introduces several additional functionalities.

dpdkvdpa translates between the PHY port to the virtio port. It takes packets from the Rx queue and sends them to the suitable Tx queue, and allows transfer of packets from the virtio guest (VM) to a VF and vice-versa, benefitting from both SR-IOV and virtio.

To add a vDPA port:

```
ovs-vsctl add-port br0 vdpa0 -- set Interface vdpa0 type=dpdkvdpa \
options:vdpa-socket-path=<sock path> \
options:vdpa-accelerator-devargs=<vf pci id> \
options:dpdk-devargs=<pf pci id>,representor=[id] \
options: vdpa-max-queues =<num queues> \
options: vdpa-sw=<true/false>
```

⚠ `vdpa-max-queues` is an optional field. When the user wants to configure 32 vDPA ports, the maximum queues number is limited to 8.

17.10.4.5.1 vDPA Configuration in OVS-DPDK Mode

Prior to configuring vDPA in OVS-DPDK mode, perform the following:

1. Generate the VF:

   ```
   echo 0 > /sys/class/net/enp175s0f0/device/sriov_numvfs
   echo 4 > /sys/class/net/enp175s0f0/device/sriov_numvfs
   ```

2. Unbind each VF:

```
echo <pci> > /sys/bus/pci/drivers/mlx5_core/unbind
```

3. Switch to switchdev mode:

```
echo switchdev >> /sys/class/net/enp175s0f0/compat/devlink/mode
```

4. Bind each VF:

```
echo <pci> > /sys/bus/pci/drivers/mlx5_core/bind
```

5. Initialize OVS:

```
ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-init=true
ovs-vsctl --no-wait set Open_vSwitch . other_config:hw-offload=true
```

To configure vDPA in OVS-DPDK mode:

1. OVS configuration:

```
ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-extra="-a
0000:01:00.0,representor=[0],dv_flow_en=1,dv_esw_en=1,dv_xmeta_en=1" /usr/share/openvswitch/scripts/ovs-ctl
restart
```

2. Create OVS-DPDK bridge:

```
ovs-vsctl add-br br0-ovs -- set bridge br0-ovs datapath_type=netdev
ovs-vsctl add-port br0-ovs pf -- set Interface pf type=dpdk options:dpdk-devargs=0000:01:00.0
```

3. Create vDPA port as part of the OVS-DPDK bridge:

```
ovs-vsctl add-port br0-ovs vdpa0 -- set Interface vdpa0 type=dpdkvdpa options:vdpa-socket-path=/var/run/
virtio-forwarder/sock0 options:vdpa-accelerator-devargs=0000:01:00.2 options:dpdk-devargs=0000:01:00.0,repr
esentor=[0] options: vdpa-max-queues=8
```

To configure vDPA in OVS-DPDK mode on BlueField DPUs, set the bridge with the software or hardware vDPA port:

- To create the OVS-DPDK bridge on the Arm side:

```
ovs-vsctl add-br br0-ovs -- set bridge br0-ovs datapath_type=netdev
ovs-vsctl add-port br0-ovs pf -- set Interface pf type=dpdk options:dpdk-devargs=0000:af:00.0
ovs-vsctl add-port br0-ovs rep-- set Interface rep type=dpdk options:dpdk-devargs=0000:af:00.0,representor=
[0]
```

- To create the OVS-DPDK bridge on the host side:

```
ovs-vsctl add-br br1-ovs -- set bridge br1-ovs datapath_type=netdev protocols=OpenFlow14
ovs-vsctl add-port br0-ovs vdpa0 -- set Interface vdpa0 type=dpdkvdpa options:vdpa-socket-path=/var/run/
virtio-forwarder/sock0 options:vdpa-accelerator-devargs=0000:af:00.2
```

> To configure SW vDPA, add `options:vdpa-sw=true` to the command.

17.10.4.5.2 Software vDPA Configuration in OVS-Kernel Mode

Software vDPA can also be used in configurations where hardware offload is done through TC and not DPDK.

1. OVS configuration:

```
ovs-vsctl set Open_vSwitch . other_config:dpdk-extra="-a
0000:01:00.0,representor=[0],dv_flow_en=1,dv_esw_en=0,idv_xmeta_en=0,isolated_mode=1"
/usr/share/openvswitch/scripts/ovs-ctl restart
```

2. Create OVS-DPDK bridge:

```
ovs-vsctl add-br br0-ovs -- set bridge br0-ovs datapath_type=netdev
```

3. Create vDPA port as part of the OVS-DPDK bridge:

```
ovs-vsctl add-port br0-ovs vdpa0 -- set Interface vdpa0 type=dpdkvdpa options:vdpa-socket-path=/var/run/
virtio-forwarder/sock0 options:vdpa-accelerator-devargs=0000:01:00.2 options:dpdk-devargs=0000:01:00.0,repr
esentor=[0] options: vdpa-max-queues=8
```

4. Create Kernel bridge:

```
ovs-vsctl add-br br-kernel
```

5. Add representors to Kernel bridge:

```
ovs-vsctl add-port br-kernel enp1s0f0_0
ovs-vsctl add-port br-kernel enp1s0f0
```

17.10.4.6 Large MTU/Jumbo Frame Configuration

To configure MTU/jumbo frames:

1. Verify that the Kernel version on the VM is 4.14 or above:

```
cat /etc/redhat-release
```

2. Set the MTU on both physical interfaces in the host:

```
ifconfig ens4f0 mtu 9216
```

3. Send a large size packet and verify that it is sent and received correctly:

```
tcpdump -i ens4f0 -nev icmp &
ping 11.100.126.1 -s 9188 -M do -c 1
```

4. Enable `host_mtu` in XML and add the following values:

```
host_mtu=9216,csum=on,guest_csum=on,host_tso4=on,host_tso6=on
```

Example:

```
<qemu:commandline>
<qemu:arg value='-chardev'/>
<qemu:arg value='socket,id=charnet1,path=/tmp/sock0,server'/>
<qemu:arg value='-netdev'/>
<qemu:arg value='vhost-user,chardev=charnet1,queues=16,id=hostnet1'/>
<qemu:arg value='-device'/>
<qemu:arg value='virtio-net-
pci,mq=on,vectors=34,netdev=hostnet1,id=net1,mac=00:21:21:24:02:01,bus=pci.0,addr=0xC,page-per-
vq=on,rx_queue_size=1024,tx_queue_size=1024,host_mtu=9216,csum=on,guest_csum=on,host_tso4=on,host_tso6=on'/
>
</qemu:commandline>
```

5. Add the `mtu_request=9216` option to the OVS ports inside the container and restart the OVS:

   ```
   ovs-vsctl add-port br0-ovs pf -- set Interface pf type=dpdk options:dpdk-devargs=0000:c4:00.0 mtu_request=9
   216
   ```

   Or:

   ```
   ovs-vsctl add-port br0-ovs vdpa0 -- set Interface vdpa0 type=dpdkvdpa options:vdpa-socket-path=/tmp/sock0
   options:vdpa-accelerator-devargs=0000:c4:00.2 options:dpdk-devargs=0000:c4:00.0,representor=[0]
   mtu_request=9216
   /usr/share/openvswitch/scripts/ovs-ctl restart
   ```

6. Start the VM and configure the MTU on the VM:

   ```
   ifconfig eth0 11.100.124.2/16 up
   ifconfig eth0 mtu 9216
   ping 11.100.126.1 -s 9188 -M do -c1
   ```

17.10.4.7 E2E Cache

> This feature is supported at beta level.

OVS offload rules are based on a multi-table architecture. E2E cache enables merging the multi-table flow matches and actions into one joint flow.

This improves CT performance by using a single-table when an exact match is detected.

To set the E2E cache size (default is 4k):

```
ovs-vsctl set open_vswitch . other_config:e2e-size=<size>
systemctl restart openvswitch
```

To enable E2E cache (disabled by default):

```
ovs-vsctl set open_vswitch . other_config:e2e-enable=true
systemctl restart openvswitch
```

To run E2E cache statistics:

```
ovs-appctl dpctl/dump-e2e-stats
```

To run E2E cache flows:

```
ovs-appctl dpctl/dump-e2e-flows
```

17.10.4.8 Geneve Encapsulation/Decapsulation

Geneve tunneling offload support includes matching on extension header.

To configure OVS-DPDK Geneve encap/decap:

1. Create a br-phy bridge:

   ```
   ovs-vsctl --may-exist add-br br-phy -- set Bridge br-phy datapath_type=netdev -- br-set-external-id br-phy
   bridge-id br-phy -- set bridge br-phy fail-mode=standalone
   ```

2. Attach PF interface to br-phy bridge:

   ```
   ovs-vsctl add-port br-phy pf -- set Interface pf type=dpdk options:dpdk-devargs=<PF PCI>
   ```

3. Configure IP to the bridge:

   ```
   ifconfig br-phy <$local_ip_1> up
   ```

4. Create a br-int bridge:

   ```
   ovs-vsctl --may-exist add-br br-int -- set Bridge br-int datapath_type=netdev -- br-set-external-id br-int
    bridge-id br-int -- set bridge br-int fail-mode=standalone
   ```

5. Attach representor to br-int:

   ```
   ovs-vsctl add-port br-int rep$x -- set Interface rep$x type=dpdk options:dpdk-devargs=<PF
   PCI>,representor=[$x]
   ```

6. Add a port for the Geneve tunnel:

   ```
   ovs-vsctl add-port br-int geneve0 -- set interface geneve0 type=geneve options:key=<VNI>
   options:remote_ip=<$remote_ip_1> options:local_ip=<$local_ip_1>
   ```

17.10.4.9 Parallel Offloads

OVS-DPDK supports parallel insertion and deletion of offloads (flow and CT). While multiple threads are supported (only one is used by default).

To configure multiple threads:

```
ovs-vsctl set Open_vSwitch . other_config:n-offload-threads=3
systemctl restart openvswitch
```

> Refer to the OVS user manual for more information.

17.10.4.9.1 sFlow

sFlow allows monitoring traffic sent between two VMs on the same host using an sFlow collector.

To sample all traffic over the OVS bridge, run the following:

```
# ovs-vsctl -- --id=@sflow create sflow agent=\"$SFLOW_AGENT\" \
                                        target=\"$SFLOW_TARGET:$SFLOW_HEADER\" \
                                        header=$SFLOW_HEADER \
                                        sampling=$SFLOW_SAMPLING polling=10 \
                -- set bridge sflow=@sflow
```

Parameter	Description
SFLOW_AGENT	Indicates that the sFlow agent should send traffic from SFLOW_AGENT 's IP address
SFLOW_TARGET	Remote IP address of the sFlow collector
SFLOW_PORT	Remote IP destination port of the sFlow collector
SFLOW_HEADER	Size of packet header to sample (in bytes)
SFLOW_SAMPLING	Sample rate

To clear the sFlow configuration, run:

```
# ovs-vsctl clear bridge br-vxlan mirrors
```

⚠ Currently sFlow for OVS-DPDK is supported without CT.

17.10.4.10 CT CT NAT

To enable ct-ct-nat offloads in OVS-DPDK (disabled by default), run:

```
ovs-vsctl set open_vswitch . other_config:ct-action-on-nat-conns=true
```

If disabled, ct-ct-nat configurations are not fully offloaded, improving connection offloading rate for other cases (ct and ct-nat).

If enabled, ct-ct-nat configurations are fully offloaded but ct and ct-nat offloading would be slower to create.

17.10.4.11 OpenFlow Meters (OpenFlow13+)

OpenFlow meters in OVS are implemented according to RFC 2697 (Single Rate Three Color Marker—srTCM).

- The srTCM meters an IP packet stream and marks its packets either green, yellow, or red. The color is decided on a Committed Information Rate (CIR) and two associated burst sizes, Committed Burst Size (CBS), and Excess Burst Size (EBS).
- A packet is marked green if it does not exceed the CBS, yellow if it exceeds the CBS but not the EBS, and red otherwise.
- The volume of green packets should never be smaller than the CIR.

To configure a meter in OVS:

1. Create a meter over a certain bridge, run:

   ```
   ovs-ofctl -O openflow13 add-meter $bridge
   meter=$id,$pktps/$kbps,band=type=drop,rate=$rate,[burst,burst_size=$burst_size]
   ```

   Parameters:

Parameter	Description
`bridge`	Name of the bridge on which the meter should be applied.
`id`	Unique meter ID (32 bits) to be used as an identifier for the meter.
`pktps` / `kbps`	Indication if the meter should work according to packets or kilobits per second.
`rate`	Rate of `pktps` / `kbps` of allowed data transmission.
`burst`	If set, enables burst support for meter bands through the `burst_size` parameter.
`burst_size`	If burst is specified for the meter entry, configures the maximum burst allowed for the band in kilobits/packets, depending on whether `kbps` or `pktps` has been specified. If unspecified, the switch is free to select some reasonable value depending on its configuration. Currently, if burst is not specified, the `burst_size` parameter is set the same as `rate` .

2. Add the meter to a certain OpenFlow rule. For example:

```
ovs-ofctl -O openflow13 add-flow $bridge "table=0,actions=meter:$id,normal"
```

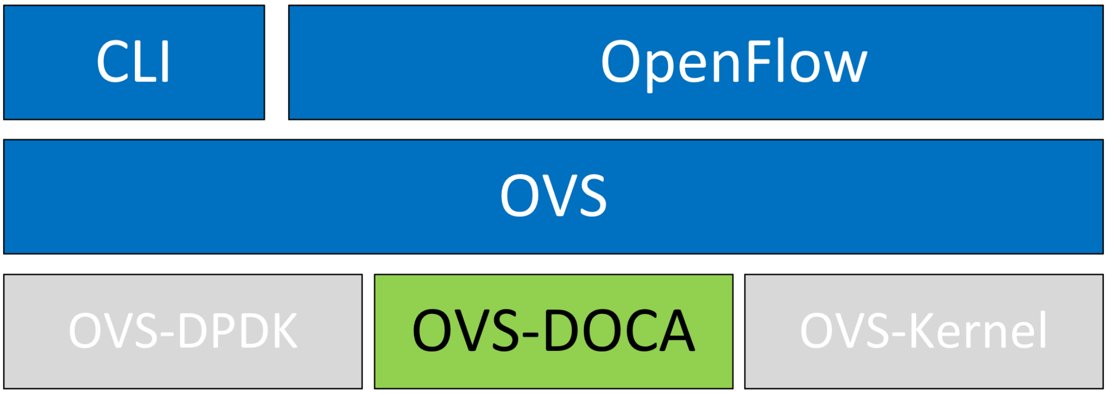
3. View the meter statistics:

```
ovs-ofctl -O openflow13 meter-stats $bridge meter=$id
```

4. For more information, refer to official OVS documentation.

17.10.5 OVS-DOCA Hardware Acceleration

OVS-DOCA is designed on top of NVIDIA's networking API to preserve the same OpenFlow, CLI, and data interfaces (e.g., vdpa, VF passthrough), as well as datapath offloading APIs, also known as OVS-DPDK and OVS-Kernel. While all OVS flavors make use of flow offloads for hardware acceleration, due to its architecture and use of DOCA libraries, the OVS-DOCA mode provides the most efficient performance and feature set among them, making the most out of NVIDA NICs and DPUs.



The following subsections provide the necessary steps to launch/deploy OVS DOCA.

17.10.5.1 Configuring OVS-DOCA

To configure OVS DOCA HW offloads:

1. Unbind the VFs:

   ```
   echo 0000:04:00.2 > /sys/bus/pci/drivers/mlx5_core/unbind
   echo 0000:04:00.3 > /sys/bus/pci/drivers/mlx5_core/unbind
   ```

   > VMs with attached VFs must be powered off to be able to unbind the VFs.

2. Change the e-switch mode from `legacy` to `switchdev` on the PF device (make sure all VFs are unbound):

   ```
   echo switchdev > /sys/class/net/enp4s0f0/compat/devlink/mode
   ```

   > This command also creates the VF representor netdevices in the host OS.

   To revert to SR-IOV `legacy` mode:

   ```
   echo legacy > /sys/class/net/enp4s0f0/compat/devlink/mode
   ```

3. Bind the VFs:

   ```
   echo 0000:04:00.2 > /sys/bus/pci/drivers/mlx5_core/bind
   echo 0000:04:00.3 > /sys/bus/pci/drivers/mlx5_core/bind
   ```

4. Configure huge pages:

   ```
   mkdir -p /hugepages
   mount -t hugetlbfs hugetlbfs /hugepages
   echo 4096 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
   ```

5. Run the Open vSwitch service:

   ```
   systemctl start openvswitch
   ```

6. Enable DOCA mode and hardware offload (disabled by default):

   ```
   ovs-vsctl --no-wait set Open_vSwitch . other_config:doca-init=true
   ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
   ```

7. Restart the Open vSwitch service.

   ```
   systemctl restart openvswitch
   ```

   > This step is required for HW offload changes to take effect.

8. Create OVS-DOCA bridge:

```
ovs-vsctl --no-wait add-br br0-ovs -- set bridge br0-ovs datapath_type=netdev
```

9. Add PF to OVS:

```
ovs-vsctl add-port br0-ovs enp4s0f0 -- set Interface enp4s0f0 type=dpdk
```

10. Add representor to OVS:

```
ovs-vsctl add-port br0-ovs enp4s0f0_0 -- set Interface enp4s0f0_0 type=dpdk
```

> The legacy option to add DPDK ports without using a related netdev by providing dpdk-devargs still exists:
>
> a. Add a PF port:
>
> ```
> ovs-vsctl add-port br0-ovs pf -- set Interface pf type=dpdk options:dpdk-devargs=0000:88:00.0
> ```
>
> b. Add a VF representor port:
>
> ```
> ovs-vsctl add-port br0-ovs representor -- set Interface representor type=dpdk options:dpdk-devargs=0000:88:00.0,representor=[0]
> ```
>
> c. Add a SF representor port:
>
> ```
> ovs-vsctl add-port br0-ovs representor -- set Interface representor type=dpdk options:dpdk-devargs=0000:88:00.0,representor=sf[0]
> ```
>
> d. Add a BlueField host PF representor port:
>
> ```
> ovs-vsctl add-port br0-ovs hpf -- set Interface hpf type=dpdk options:dpdk-devargs=0000:88:00.0,representor=[65535]
> ```

11. Optional configuration:

    a. To set port MTU, run:

    ```
    ovs-vsctl set interface enp4s0f0 mtu_request=9000
    ```

    > ⚠ OVS restart is required for changes to take effect.

    b. To set VF/SF MAC, run:

    ```
    ovs-vsctl add-port br0-ovs enp4s0f0 -- set Interface enp4s0f0 type=dpdk options:dpdk-vf-mac=00:11:22:33:44:55
    ```

    > ⚠ Unbinding and rebinding the VFs/SFs is required for the change to take effect.

17.10.5.2 Notable Differences Between OVS-DPDK and OVS-DOCA

OVS-DOCA shares most of its structure with OVS-DPDK. To benefit from the DOCA offload design, some of the behavior of userland datapath and ports are however modified.

17.10.5.2.1 Eswitch Dependency

Configured in `switchdev` mode, the physical port and all supported functions share a single general domain to execute the offloaded flows, the `eswitch`.

All ports on the same eswitch are dependent on its physical function. If this main physical function is deactivated (e.g., removed from OVS or its link set down), dependent ports are disabled as well.

17.10.5.2.2 Pre-allocated Offload Tables

To offer the highest insertion speed, DOCA offloads pre-allocate offload structures (entries and containers).

When starting the vSwitch daemon, offloads are thus configured with sensible defaults. If different numbers of offloads are required, configuration entries specific to OVS-DOCA are available and are described in the next section.

17.10.5.2.3 Unsupported CT-CT-NAT

The special ct-ct-nat mode that can be configured in OVS-kernel and OVS-DPDK is not supported by OVS-DOCA.

17.10.5.3 OVS-DOCA Specific vSwitch Configuration

The following configuration is particularly useful or specific to OVS-DOCA mode.

> The full list of OVS vSwitch configuration is documented in `man ovs-vswitchd.conf.db`.

17.10.5.3.1 other_config

The following table provides `other_config` configurations which are global to the vSwitch (non-exhaustive list, check manpage for more):

Configuration	Description
`other_config:doca-init`	• Optional string, either true or false • Set this value to true to enable DOCA Flow HW offload • The default value is false. Changing this value requires restarting the daemon. • This is only relevant for userspace datapath

Configuration	Description
`other_config:hw-offload-ct-size`	• Optional string, containing an integer, at least 0 • Only for the DOCA offload provider on netdev datapath • Configure the usable amount of connection tracking (CT) offload entries • The default value is 250000. Changing this value requires restarting the daemon. • Setting a value of 0 disables CT offload • Changing this configuration affects the OVS memory usage as CT tables are allocated on OVS start • Maximum number of supported connections is 2M Setting this parameter to more than 2M might result in failures.
`other_config:hw-offload-ct-ipv6-enabled`	• Optional string, either true or false • Only for the DOCA offload provider on netdev datapath • Set this value to true to enable IPv6 CT offload • The default value is false. Changing this value requires restarting the daemon. • Changing this configuration affects the OVS memory usage as CT tables are allocated on OVS start
`other_config:doca-congestion-threshold`	• Optional string, containing an integer, in range 30 to 90 • The occupancy rate of DOCA offload structures that triggers a resize, as a percentage • Default to 80, but only relevant if `other_config:doca-init` is true. Changing this value requires restarting the daemon.
`other_config:ctl-pipe-size`	• Optional string, containing an integer • The initial size of DOCA control pipes • Default to 0, which is DOCA's internal default value
`other_config:ctl-pipe-infra-size`	• Optional string, containing an integer • The initial size of infrastructure DOCA control pipes: root, post-hash, post-ct, post-meter, split, miss. • Default to 0, which fallbacks to `other_config:ctl-pipe-size`
`other_config:pmd-quiet-idle`	• Optional string, either true or false • Allow the PMD threads to go into quiescent mode when idling. If no packets are received or waiting to be processed and sent, enter a continuous quiescent period. End this period as soon as a packet is received. • This option is disabled by default
`other_config:pmd-maxsleep`	• Optional string, containing an integer, in range 0 to 10,000 • Specifies the maximum sleep time in microseconds per iteration for a PMD thread which has received zero or a small amount of packets from the Rx queues it is polling. • The actual sleep time requested is based on the load of the Rx queues that the PMD polls and may be less than the maximum value • The default value is 0 microseconds, which means that the PMD does not sleep regardless of the load from the Rx queues that it polls • To avoid requesting very small sleeps (e.g., less than 10 μs) the value is rounded up to the nearest 10 μs • The maximum value is 10000 microseconds.

Configuration	Description
`other_config:dpdk-max-memzones`	• Optional string, containing an integer • Specifies the maximum number of memzones that can be created in DPDK • The default is empty, keeping DPDK's default. Changing this value requires restarting the daemon.
`other_config:pmd-cpu-mask`	With PMD multi-threading support, OVS creates one PMD thread for each NUMA node by default if there is at least one DPDK interface added to OVS from that NUMA node. However, in cases where there are multiple ports/rxqs producing traffic, performance can be improved by creating multiple PMD threads running on separate cores. These PMD threads can share the workload by each being responsible for different ports/rxqs. Assignment of ports/rxqs to PMD threads is done automatically. A set bit in the mask means a PMD thread is created and pinned to the corresponding CPU core. For example, to run PMD threads on cores 1 and 2, run: `$ ovs-vsctl set Open_vSwitch . other_config:pmd-cpu-mask=0x6`

17.10.5.3.2 netdev-dpdk

The following table provides `netdev-dpdk` configurations which only userland (DOCA or DPDK) netdevs support (non-exhaustive list, check manpage for more):

Configuration	Description
`options:iface-name`	• Specifies the interface name of the port • Providing this option accelerates processing the port reconfiguration by querying the sysfs to check if the interface exists before DPDK attempts to probe the port

17.10.5.4 Offloading VXLAN Encapsulation/Decapsulation Actions

vSwitch in userspace rather than kernel-based Open vSwitch requires an additional bridge. The purpose of this bridge is to allow use of the kernel network stack for routing and ARP resolution.

The datapath must look up the routing table and ARP table to prepare the tunnel header and transmit data to the output port.

VXLAN encapsulation/decapsulation offload configuration is done with:

- PF on `0000:03:00.0` PCIe and MAC `98:03:9b:cc:21:e8`
- Local IP `56.56.67.1` - the `br-phy` interface is configured to this IP
- Remote IP `56.56.68.1`

To configure OVS DOCA VXLAN:

1. Create a `br-phy` bridge:

```
ovs-vsctl add-br br-phy -- set Bridge br-phy datapath_type=netdev -- br-set-external-id br-phy bridge-id
br-phy -- set bridge br-phy fail-mode=standalone other_config:hwaddr=98:03:9b:cc:21:e8
```

2. Attach PF interface to `br-phy` bridge:

   ```
   ovs-vsctl add-port br-phy enp4s0f0 -- set Interface enp4s0f0 type=dpdk
   ```

3. Configure IP to the bridge:

   ```
   ip addr add 56.56.67.1/24 dev br-phy
   ```

4. Create a `br-ovs` bridge:

   ```
   ovs-vsctl add-br br-ovs -- set Bridge br-ovs datapath_type=netdev -- br-set-external-id br-ovs bridge-id
   br-ovs -- set bridge br-ovs fail-mode=standalone
   ```

5. Attach representor to `br-ovs`:

   ```
   ovs-vsctl add-port br-ovs enp4s0f0_0 -- set Interface enp4s0f0_0 type=dpdk
   ```

6. Add a port for the VXLAN tunnel:

   ```
   ovs-vsctl add-port ovs-sriov vxlan0 -- set interface vxlan0 type=vxlan options:local_ip=56.56.67.1
   options:remote_ip=56.56.68.1 options:key=45 options:dst_port=4789
   ```

17.10.5.4.1 VXLAN GBP Extension

The VXLAN group-based policy (GBP) model outlines an application-focused policy framework that specifies connectivity requirements for applications, independent of the network's physical layout.

Setting GBP extension for a VXLAN port allows for matching on and setting a GBP ID per flow. To enable GBP extension when the port `vxlan0` is first added:

```
ovs-vsctl add-port br-int vxlan0 -- set interface vxlan0 type=vxlan options:key=30 options:remote_ip=10.0.30.1
options:exts=gbp
```

It is also possible to enable GBP extension for an existing VXLAN port:

```
ovs-vsctl set interface vxlan1 options:exts=gbp
```

This approach has a limitation that it does not take effect until after the OVS `vswitchd` service is restarted. In cases where there are multiple VXLAN ports, they must all share the same GBP extension configuration in their port options. A mixed configuration with some VXLAN ports having the GBP extension enabled and others disabled is not supported.

When GBP extension is enabled, the following OpenFlow rules which match on a GBP ID 32 or set a GBP ID 64 in the actions, can be offloaded:

```
ovs-ofctl add-flow br-int table=0,priority=100,in_port=vxlan0,tun_gbp_id=32 actions=output:pf0vf0
ovs-ofctl add-flow br-int table=0,priority=100,in_port=pf0vf0 actions=load:64->NXM_NX_TUN_GBP_ID[],output:vxlan0
```

17.10.5.5 Offloading Connection Tracking

Connection tracking enables stateful packet processing by keeping a record of currently open connections.

OVS flows utilizing connection tracking can be accelerated using advanced NICs by offloading established connections.

To view offload statistics, run:

```
ovs-appctl dpctl/offload-stats-show
```

17.10.5.6 SR-IOV VF LAG

To configure OVS-DOCA SR-IOV VF LAG:

1. Enable SR-IOV on the NICs:

   ```
   // It is recommended to query the parameters first to determine if a change is needed, to save potentially
   unnecessary reboot.
   mst start
   mlxconfig -d <mst device> -y set PF_NUM_OF_VF_VALID=0   SRIOV_EN=1 NUM_OF_VFS=8
   ```

   > If configuration did change, perform a BlueField system reboot for the `mlxconfig` settings to take effect.

2. Allocate the desired number of VFs per port:

   ```
   echo $n > /sys/class/net/<net name>/device/sriov_numvfs
   ```

3. Unbind all VFs:

   ```
   echo <VF PCI> >/sys/bus/pci/drivers/mlx5_core/unbind
   ```

4. Change both NICs' mode to SwitchDev:

   ```
   devlink dev eswitch set pci/<PCI> mode switchdev
   ```

5. Create Linux bonding using kernel modules:

   ```
   modprobe bonding mode=<desired mode>
   ```

   > Other bonding parameters can be added here. The supported bond modes are Active-Backup, XOR, and LACP.

6. Bring all PFs and VFs down:

   ```
   ip link set <PF/VF> down
   ```

7. Attach both PFs to the bond:

```
ip link set <PF> master bond0
```

8. Bring PFs and bond link up:

```
ip link set <PF0> up
ip link set <PF1> up
ip link set bond0 up
```

9. Add the bond interface to the bridge as `type=dpdk` :

```
ovs-vsctl add-port br-phy bond0 -- set Interface bond0 type=dpdk options:dpdk-lsc-interrupt=true
```

> The legacy option to work with VF-LAG in OVS-DPDK is to add the bond master (PF) interface to the bridge:
>
> ```
> ovs-vsctl add-port br-phy p0 -- set Interface p0 type=dpdk options:dpdk-devargs=<PF0-
> PCI>,dv_flow_en=2,dv_xmeta_en=4 options:dpdk-lsc-interrupt=true
> ```

10. Add representor of PF0 or PF1 to a bridge:

```
ovs-vsctl add-port br-phy enp4s0f0_0 -- set Interface enp4s0f0_0 type=dpdk
```

Or:

```
ovs-vsctl add-port br-phy enp4s0f1_0 -- set Interface enp4s0f1_0 type=dpdk
```

> The legacy option to add DPDK ports:
>
> ```
> ovs-vsctl add-port br-phy rep$N -- set Interface rep$N type=dpdk options:dpdk-devargs=<PF0-
> PCI>,representor=pf0vf$N,dv_flow_en=2,dv_xmeta_en=4
> ```
>
> Or:
>
> ```
> ovs-vsctl add-port br-phy rep$N -- set Interface rep$N type=dpdk options:dpdk-devargs=<PF0-
> PCI>,representor=pf1vf$N,dv_flow_en=2,dv_xmeta_en=4
> ```

17.10.5.7 Multiport eSwitch Mode

Multiport eswitch mode allows adding rules on a VF representor with an action, forwarding the packet to the physical port of the physical function. This can be used to implement failover or to forward packets based on external information such as the cost of the route.

1. To configure multiport eswitch mode, the nvconig parameter `LAG_RESOURCE_ALLOCATION=1` must be set in the BlueField Arm OS, according to the following instructions:

```
mst start
mlxconfig -d /dev/mst/mt*conf0 -y s  LAG_RESOURCE_ALLOCATION=1
```

2. Perform a BlueField system reboot for the `mlxconfig` settings to take effect.

3. After the driver loads, and before moving to switchdev mode, configure multiport eswitch for each PF where p0 and p1 represent the netdevices for the PFs:

```
devlink dev param set pci/0000:03:00.0 name esw_multiport value 1 cmode runtime
devlink dev param set pci/0000:03:00.1 name esw_multiport value 1 cmode runtime
```

   > The mode becomes operational after entering switchdev mode on both PFs.

4. This mode can be activated by default in BlueField by adding the following line into `/etc/mellanox/mlnx-bf.conf`:

```
ENABLE_ESWITCH_MULTIPORT="yes"
```

While in this mode, the second port is not an eswitch manager, and should be add to OVS using this command:

```
ovs-vsctl add-port br-phy enp4s0f1 -- set interface enp4s0f1 type=dpdk
```

VFs for the second port can be added using this command:

```
ovs-vsctl add-port br-phy enp4s0f1_0 -- set interface enp4s0f1_0 type=dpdk
```

> The legacy option to add DPDK ports:
>
> ```
> ovs-vsctl add-port br-phy p1 -- set interface p1 type=dpdk options:dpdk-devargs="0000:08:00.0,dv_xmeta_en
> =4,dv_flow_en=2,representor=pf1
> ```
>
> VFs for the second port can be added using this command:
>
> ```
> ovs-vsctl add-port br-phy p1vf0 -- set interface p1 type=dpdk options:dpdk-devargs="0000:08:00.0,dv_xmeta
> _en=4,dv_flow_en=2,representor=pf1vf0
> ```

17.10.5.8 Offloading Geneve Encapsulation/Decapsulation

Geneve tunneling offload support includes matching on extension header.

> OVS-DOCA Geneve option limitations:
>
> - Only 1 Geneve option is supported
> - Max option len is 7
> - To change the Geneve option currently being matched and encapsulated, users must remove all ports or restart OVS and configure the new option
> - Matching on Geneve options can work with `FLEX_PARSER` profile 0 (the default profile). Working with `FLEX_PARSER` profile 8 is also supported as well. To configure it, run:

```
mst start
mlxconfig -d <mst device> s FLEX_PARSER_PROFILE_ENABLE=8
```

Perform a BlueField system reboot for the `mlxconfig` settings to take effect.

To configure OVS-DOCA Geneve encapsulation/decapsulation:

1. Create a `br-phy` bridge:

```
ovs-vsctl --may-exist add-br br-phy -- set Bridge br-phy datapath_type=netdev -- br-set-external-id br-phy
bridge-id br-phy -- set bridge br-phy fail-mode=standalone
```

2. Attach a PF interface to `br-phy` bridge:

```
ovs-vsctl add-port br-phy enp4s0f0 -- set Interface enp4s0f0 type=dpdk
```

3. Configure an IP to the bridge:

```
ifconfig br-phy <$local_ip_1> up
```

4. Create a `br-int` bridge:

```
ovs-vsctl add-port br-int enp4s0f0_0 -- set Interface enp4s0f0_0 type=dpdk
```

5. Attach a representor to `br-int`:

```
ovs-vsctl add-port br-int rep$x -- set Interface rep$x type=dpdk options:dpdk-devargs=<PF
PCI>,representor=[$x],dv_flow_en=2,dv_xmeta_en=4
```

6. Add a port for the Geneve tunnel:

```
ovs-vsctl add-port br-int geneve0 -- set interface geneve0 type=geneve options:key=<VNI>
options:remote_ip=<$remote_ip_1> options:local_ip=<$local_ip_1>
```

17.10.5.9 GRE Tunnel Offloads

To configure OVS-DOCA GRE encapsulation/decapsulation:

1. Create a `br-phy` bridge:

```
ovs-vsctl --may-exist add-br br-phy -- set Bridge br-phy datapath_type=netdev -- br-set-external-id br-phy
bridge-id br-phy -- set bridge br-phy fail-mode=standalone
```

2. Attach a PF interface to `br-phy` bridge:

```
ovs-vsctl add-port br-phy enp4s0f0 -- set Interface enp4s0f0 type=dpdk
```

3. Configure an IP to the bridge:

```
ifconfig br-phy <$local_ip_1> up
```

4. Create a `br-int` bridge:

   ```
   ovs-vsctl --may-exist add-br br-int -- set Bridge br-int datapath_type=netdev -- br-set-external-id br-int
   bridge-id br-int -- set bridge br-int fail-mode=standalone
   ```

5. Attach a representor to `br-int`:

   ```
   ovs-vsctl add-port br-int enp4s0f0_0 -- set Interface enp4s0f0_0 type=dpdk
   ```

Add a port for the Geneve tunnel:

```
ovs-vsctl add-port br-int gre0 -- set interface gre0 type=gre options:key=<VNI> options:remote_ip=<$remote_ip_1>
options:local_ip=<$local_ip_1>
```

17.10.5.10 Slow Path Rate Limiting/SW-Meter

Slow path rate limiting allows controlling the rate of traffic that bypasses hardware offload rules and is subsequently processed by software.

To configure slow path rate limiting:

1. Create a `br-phy` bridge:

   ```
   ovs-vsctl --may-exist add-br br-phy -- set Bridge br-phy datapath_type=netdev -- br-set-external-id br-phy
   bridge-id br-phy -- set bridge br-phy fail-mode=standalone
   ```

2. Attach a PF interface to `br-phy` bridge:

   ```
   ovs-vsctl add-port br-phy pf0 -- set Interface pf0 type=dpdk
   ```

3. Rate limit `pf0vf0` to 10Kpps with 6K burst size:

   ```
   ovs-vsctl set interface pf0 options:sw-meter=pps:10k:6k
   ```

4. Restart OVS:

   ```
   systemctl restart openvswitch-switch.service
   ```

A dry-run option is also supported to allow testing different software meter configurations in a production environment. This allows gathering statistics without impacting the actual traffic flow. These statistics can then be analyzed to determine appropriate rate limiting thresholds. When the dry-run option is enabled, traffic is not dropped or rate-limited, allowing normal operations to continue without disruption. However, the system simulates the rate limiting process and increment counters as though packets are being dropped.

To enable slow path rate limiting dry-run:

1. Create a `br-phy` bridge:

   ```
   ovs-vsctl --may-exist add-br br-phy -- set Bridge br-phy datapath_type=netdev -- br-set-external-id br-phy
   bridge-id br-phy -- set bridge br-phy fail-mode=standalone
   ```

2. Attach a PF interface to `br-phy` bridge:

```
ovs-vsctl add-port br-phy pf0 -- set Interface pf0 type=dpdk
```

3. Rate limit `pf0vf0` to 10Kpps with 6K burst size:

```
ovs-vsctl set interface pf0 options:sw-meter=pps:10k:6k
```

4. Set the `sw-meter-dry-run` option:

```
ovs-vsctl set interface pf0vf0 options:sw-meter-dry-run=true
```

5. Restart OVS:

```
systemctl restart openvswitch-switch.service
```

17.10.5.11 Hairpin

Hairpin allows forwarding packets from wire to wire.

To configure hairpin :

1. Create a `br-phy` bridge:

```
ovs-vsctl --may-exist add-br br-phy -- set Bridge br-phy datapath_type=netdev -- br-set-external-id br-phy
bridge-id br-phy -- set bridge br-phy fail-mode=standalone
```

2. Attach a PF interface to `br-phy` bridge:

```
ovs-vsctl add-port br-phy pf0 -- set Interface pf0 type=dpdk
```

3. Add hairpin OpenFlow rule:

```
ovs-ofctl add-flow br-phy"in_port=pf0,ip,actions=in_port"
```

17.10.5.12 OpenFlow Meters

OVS-DOCA supports OpenFlow meter action as covered in this document in section "OpenFlow Meters". In addition, OVS-DOCA supports chaining multiple meter actions together in a single datapth rule.

The following is an example configuration of such OpenFlow rules:

```
ovs-ofctl add-flow br-phy -O OpenFlow13 "table=0,priority=1,in_port=pf0vf0_r,ip actions=meter=1,resubmit(,1)"
ovs-ofctl add-flow br-phy -O OpenFlow13 "table=1,priority=1,in_port=pf0vf0_r,ip actions=meter=2,normal"
```

Meter actions are applied sequentially, first using meter ID 1 and then using meter ID 2.

Use case examples for such a configuration:

- Rate limiting the same logical flow with different meter types—bytes per second and packets per second

- Metering a group of flows. As meter IDs can be used by multiple flows, it is possible to re-use meter ID 2 from this example with other logical flows; thus, making sure that their cumulative bandwidth is limited by the meter.

17.10.5.13 DP-HASH Offloads

OVS supports group configuration. The "select" type executes one bucket in the group, balancing across the buckets according to their weights. To select a bucket, for each live bucket, OVS hashes flow data with the bucket ID and multiplies that by the bucket weight to obtain a "score". The bucket with the highest score is selected.

For more details, refer to the ovs-ofctl man.

For example:

- `ovs-ofctl add-group br-int 'group_id=1,type=select,bucket=<port1>'`
- `ovs-ofctl add-flow br-int in_port=<port0>,actions=group=1`

Limitations:

- Offloads are supported on IP traffic only (IPv4 or IPv6)

17.10.5.14 sFlow

The sFlow standard outlines a method for capturing traffic data in switched or routed networks. It employs sampling technology to gather statistics from the device, making it suitable for high-speed networks.

With a predetermined sampling rate, one out of every N packets is captured. While this sampling method does not yield completely accurate results, it does offer acceptable accuracy.

To activate sampling for 0.2% of all traffic traversing an OVS bridge named `br-int`, run:

```
ovs-vsctl -- --id=@sflow create sflow agent=lo target=127.0.0.1:6343 header=96 sampling=512 -- set bridge br-int
sflow=@sflow
```

With this sFlow configuration on the bridge, captured packets are mirrored to an sFlow collector application that listens on the default sFlow port, 6343, on localhost.

sFlow collector applications fall outside the scope of this guide.

It is possible to set the sampling rate to 1 while configuring sFlow on a bridge, which effectively mirrors all traffic to the sFlow collector.

17.10.5.15 OVS-DOCA Known Limitations

- Only one insertion thread is supported (`n-offload-threads=1`)
- Only 250K connection are offloadable by default (can be configured)

> ⚠ The maximum number of supported connections is 2M.

- Only 8 CT zones are supported by CT offload
- When using two PFs with 127 VFs each and adding their representors to OVS bridge, the user must configure `dpdk-memzones` :

```
ovs-vsctl set o . other_config:dpdk-max-memzones=6500
restart ovs
```

- In an OVS topology that includes both physical and internal bridges, sFlow offloads are only supported on the internal bridge when employing a VXLAN tunnel. Utilizing sFlow on the physical bridge leads to only partial offload of flows in this scenario.

17.10.5.16 OVS-DOCA Debugging

Additional debugging information can be enabled in the vSwitch log file using the `dbg` log level:

```
    (
        topics='netdev|ofproto|ofp|odp|doca'
        IFS=$'\n'; for topic in $(ovs-appctl vlog/list | grep -E "$topics" | cut -d' ' -f1)
        do
            printf "$topic:file:dbg "
        done
    ) | xargs ovs-appctl vlog/set
```

The listed topics are relevant to DOCA offload operations.

Coverage counters specific to the DOCA offload provider have been added. The following command should be used to check them:

```
ovs-appctl coverage/show # Print the current non-zero coverage counters
```

The following table provides the meaning behind these DOCA-specific counters:

Counter	Description
`doca_async_queue_full`	The asynchronous offload insertion queue was full while the daemon attempted to insert a new offload. The queue will have been flushed and insertion attempted again. This is not a fatal error but is the sign of a slowed down hardware.
`doca_async_queue_blocked`	The asynchronous offload insertion queue has remained full even after several attempts to flush its currently enqueued requests. While not a fatal error, it should never happen during normal offload operations and should be considered a bug.
`doca_async_add_failed`	An asynchronous insertion failed specifically due to its asynchronous nature. This is not expected to happen and should be considered a bug.
`doca_pipe_resize`	The number of time a DOCA pipe has been resized. This is normal and expected as DOCA pipes receives more entries.

Counter	Description
`doca_pipe_resize_over_10_ms`	A DOCA pipe resize took longer than 10ms to complete. It can happen infrequently. If a sudden drop in insertion rate is measured, this counter could help identify the root cause.

17.10.5.17 OVS-DOCA Build

To build OVS-DOCA from provided sources and pre-installed DOCA with the same version packages, run:

```
$ ./boot.sh
$ ./configure --prefix=/usr --localstatedir=/var --sysconfdir=/etc --with-dpdk=static --with-doca=static
$ make -j 10
$ make install
```

A helper build script is bundled with OVS-DOCA sources that can be used as follows:

```
$ ./build.sh --install-ovs
```

17.10.5.18 Scaling Megaflows

Megaflows aggregate multiple microflows into a single flow entry, reduce the load on the flow table, and improve packet processing efficiency. Scaling megaflows in OVS is crucial for optimizing network performance and ensuring efficient handling of high traffic volumes. By default, OVS-DOCA can handle up to 200k megaflows.

To effectively manage and scale megaflows, several key configurations in the `other_config` section of OVS can be adjusted:

- The `flow-limit` parameter sets the maximum number of flows that can be stored in the flow table, helping to control memory usage and prevent overflow.
- The `max-revalidator` parameter defines the longest duration (in milliseconds) that re-validator threads will wait before initiating flow revalidation. It is crucial to understand that this represents the upper limit, and the actual timeout employed by OVS is the lesser of the `max-idle` and `max-revalidator` values. Modifying this parameter is generally not recommended without a thorough understanding of its effects. For systems with less powerful CPUs, setting a higher `max-revalidator` value is suggested to compensate for reduced computational capacity and ensure revalidation completes.

Fine-tuning these settings can improve the scalability and performance of an OVS deployment, allowing it to manage a greater number of megaflows efficiently.

- To set `flow-limit` (default is 200k):

  ```
  $ ovs-vsctl set o . other_config:flow-limit=<desired_value>
  ```

- To set `max-revalidator` (default is 250ms).

```
$ ovs-vsctl set o . other_config:max-revalidator=<desired_value>
```

17.10.6 OVS Metrics

OVS exposes Prometheus metrics through its control socket (experimental feature). These metrics can be accessed using the command:

```
ovs-appctl metrics/show
```

A terminal dashboard is also installed with OVS, `ovs-metrics`. This script is dependent on the OVS Python API (package `python3-openvswitch`). Its default mode currently watches over a set of offload-related metrics.

17.10.7 OVS Inside BlueField

17.10.7.1 Verifying Host Connection on Linux

When the DPU is connected to another DPU on another machine, manually assign IP addresses with the same subnet to both ends of the connection.

1. Assuming the link is connected to `p3p1` on the other host, run:

```
$ ifconfig p3p1 192.168.200.1/24 up
```

2. On the host which the DPU is connected to, run:

```
$ ifconfig p4p2 192.168.200.2/24 up
```

3. Have one ping the other. This is an example of the DPU pinging the host:

```
$ ping 192.168.200.1
```

17.10.7.2 Verifying Connection from Host to BlueField

There are two SFs configured on the BlueField device, `enp3s0f0s0` and `enp3s0f1s0`, and their representors are part of the built-in bridge. These interfaces will get IP addresses from the DHCP server if it is present. Otherwise it is possible to configure IP address from the host. It is possible to access BlueField via the SF netdev interfaces.

For example:

1. Verify the default OVS configuration. Run:

```
# ovs-vsctl show
5668f9a6-6b93-49cf-a72a-14fd64b4c82b
    Bridge ovsbr1
        Port pf0hpf
            Interface pf0hpf
        Port ovsbr1
            Interface ovsbr1
                type: internal
        Port p0
            Interface p0
        Port en3f0pf0sf0
            Interface en3f0pf0sf0
    Bridge ovsbr2
        Port en3f1pf1sf0
            Interface en3f1pf1sf0
        Port ovsbr2
            Interface ovsbr2
                type: internal
        Port pf1hpf
            Interface pf1hpf
        Port p1
            Interface p1
    ovs_version: "2.14.1"
```

2. Verify whether the SF netdev received an IP address from the DHCP server. If not, assign a static IP. Run:

```
# ifconfig enp3s0f0s0
enp3s0f0s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.200.125  netmask 255.255.255.0  broadcast 192.168.200.255
        inet6 fe80::8e:bcff:fe36:19bc  prefixlen 64  scopeid 0x20<link>
        ether 02:8e:bc:36:19:bc  txqueuelen 1000  (Ethernet)
        RX packets 3730  bytes 1217558 (1.1 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 22  bytes 2220 (2.1 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

3. Verify the connection of the configured IP address. Run:

```
# ping 192.168.200.25 -c 5
PING 192.168.200.25 (192.168.200.25) 56(84) bytes of data.
64 bytes from 192.168.200.25: icmp_seq=1 ttl=64 time=0.228 ms
64 bytes from 192.168.200.25: icmp_seq=2 ttl=64 time=0.175 ms
64 bytes from 192.168.200.25: icmp_seq=3 ttl=64 time=0.232 ms
64 bytes from 192.168.200.25: icmp_seq=4 ttl=64 time=0.174 ms
64 bytes from 192.168.200.25: icmp_seq=5 ttl=64 time=0.168 ms

--- 192.168.200.25 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 91ms
rtt min/avg/max/mdev = 0.168/0.195/0.232/0.031 ms
```

17.10.7.3 Verifying Host Connection on Windows

Set IP address on the Windows side for the RShim or Physical network adapter, please run the following command in Command Prompt:

```
PS C:\Users\Administrator> New-NetIPAddress -InterfaceAlias "Ethernet 16" -IPAddress "192.168.100.1" -PrefixLength
22
```

To get the interface name, please run the following command in Command Prompt:

```
PS C:\Users\Administrator> Get-NetAdapter
```

Output should give us the interface name that matches the description (e.g. NVIDIA BlueField Management Network Adapter).

```
Ethernet 2          NVIDIA ConnectX-4 Lx Ethernet Adapter        6 Not Present  24-8A-07-0D-E8-1D
Ethernet 6          NVIDIA ConnectX-4 Lx Ethernet Ad...#2       23 Not Present  24-8A-07-0D-E8-1C
Ethernet 16         NVIDIA BlueField Management Netw...#2       15 Up           CA-FE-01-CA-FE-02
```

Once IP address is set, Have one ping the other.

```
C:\Windows\system32>ping 192.168.100.2

Pinging 192.168.100.2 with 32 bytes of data:
Reply from 192.168.100.2: bytes=32 time=148ms TTL=64
Reply from 192.168.100.2: bytes=32 time=152ms TTL=64
Reply from 192.168.100.2: bytes=32 time=158ms TTL=64
Reply from 192.168.100.2: bytes=32 time=158ms TTL=64
```

17.11 NVIDIA DOCA Telemetry Service Guide

This guide provides instructions on how to use the DOCA Telemetry Service (DTS) container on top of NVIDIA® BlueField® DPU.

17.11.1 Introduction

DOCA Telemetry Service (DTS) collects data from built-in providers and from external telemetry applications. The following providers are available:

- Data providers:
  - sysfs
  - ethtool
  - tc (traffic control)
- Aggregation providers:
  - fluent_aggr
  - prometheus_aggr

⚠ Sysfs provider is enabled by default.

DTS stores collected data into binary files under the `/opt/mellanox/doca/services/telemetry/data` directory. Data write is disabled by default due to BlueField storage restrictions.

DTS can export the data via Prometheus Endpoint (pull) or Fluent Bit (push).

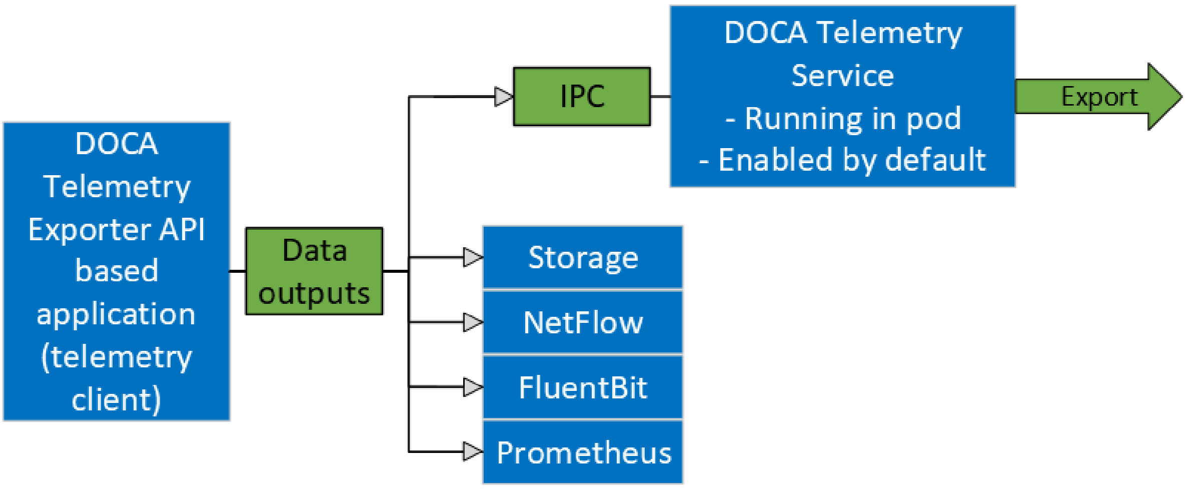
DTS allows exporting NetFlow packets when data is collected from the DOCA Telemetry Exporter NetFlow API client application. NetFlow exporter is enabled from `dts_config.ini` by setting NetFlow collector IP/address and port.

17.11.2 Service Deployment

17.11.2.1 Available Images

17.11.2.1.1 Built-in DOCA Service Image

DOCA Telemetry Service is enabled by default on the DPU and is shipped as part of the BlueField image. That is, every BlueField image contains a fixed service version so as to provide out-of-the-box support for programs based on the DOCA Telemetry Exporter library.



17.11.2.1.2 DOCA Service on NGC

In addition to the built-in image shipped with the BlueField boot image, DTS is also available on NGC, NVIDIA's container catalog. This is useful in case a new version of the service has been released and the user wants to upgrade from the built-in image. For service-specific configuration steps and deployment instructions, refer to the service's container page.

> ⓘ For more information about the deployment of DOCA containers on top of the BlueField DPU, refer to NVIDIA DOCA Container Deployment Guide.

17.11.2.2 DPU Deployment

As mentioned above, DTS starts automatically on BlueField boot. This is done according to the `.yaml` file located at `/etc/kubelet.d/doca_telemetry_standalone.yaml`. Removing the `.yaml` file from this path stops the automatic DTS boot.

DTS files can be found under the directory `/opt/mellanox/doca/services/telemetry/`.

- Container folder mounts:
  - `config`
  - `data`
  - `ipc_sockets`
- Backup files:
  - `doca_telemetry_service_${version}_arm64.tar.gz` - DTS image
  - `doca_telemetry_standalone.yaml` - copy of the default boot `.yaml` file

17.11.2.3 Host Deployment

DTS supports x86_64 hosts. The providers and exporters all run from a single docker container.

1. Initialize and configure host DTS with the desired DTS version:

```
export DTS_IMAGE=nvcr.io/nvidia/doca/doca_telemetry:<desired-DTS-version>
docker run -v "/opt/mellanox/doca/services/telemetry/config:/config" --rm --name doca-telemetry-init -it
$DTS_IMAGE /bin/bash -c "DTS_CONFIG_DIR=host /usr/bin/telemetry-init.sh"
```

Per NGC policy, the "latest" tag does not exist. This means that when deploying DTS, the user must pick the desired tag from NGC and ensure that the `DTS_IMAGE` variable points to the full image. Example from version `1.16.5-doca2.6.0-host`:

```
export DTS_IMAGE=nvcr.io/nvidia/doca/doca_telemetry:1.16.5-doca2.6.0-host
```

2. Run with:

```
docker run -d --net=host --uts=host --ipc=host                                       \
              --privileged                                                           \
              --ulimit stack=67108864 --ulimit memlock=-1                            \
              --device=/dev/mst/                                                     \
              --device=/dev/infiniband/                                              \
              --gpus all                                                             \
              -v "/opt/mellanox/doca/services/telemetry/config:/config"              \
              -v "/opt/mellanox/doca/services/telemetry/ipc_sockets:/tmp/ipc_sockets" \
              -v "/opt/mellanox/doca/services/telemetry/data:/data"                  \
              -v "/usr/lib/mft:/usr/lib/mft"                                         \
              -v "/sys/kernel/debug:/sys/kernel/debug"                               \
              --rm --name doca-telemetry -it $DTS_IMAGE /usr/bin/telemetry-run.sh
```

The following mounts are required by specific services only:

- `hcaperf` provider:
  - `--device=/dev/mst/`
  - `-v "/usr/lib/mft:/usr/lib/mft"`
  - `-v "/sys/kernel/debug:/sys/kernel/debug"`
- UCX/RDMA export modes:
  - `--device=/dev/infiniband/`
- GPU providers (`nvidia-smi` and `dcgm`):
  - `--gpu all`

17.11.2.4 Deployment with Grafana Monitoring

Refer to section "Deploying with Grafana Monitoring".

17.11.3 Configuration

The configuration of DTS is placed under `/opt/mellanox/doca/services/telemetry/config` by DTS during initialization. The user can interact with the `dts_config.ini` file and `fluent_bit_configs` folder. `dts_config.ini` contains the main configuration for the service and must be used to enable/disable providers, exporters, data writing. More details are provided in the corresponding sections. For every update in this file, DST must be restarted. Interaction with `fluent_bit_configs` folder is described in section Fluent Bit.

17.11.3.1 Init Scripts

The `InitContainers` section of the `.yaml` file has 2 scripts for config initialization:

- `/usr/bin/telemetry-init.sh` - generates the default configuration files if, and only if, the `/opt/mellanox/doca/services/telemetry/config` folder is empty.
- `/usr/bin/enable-fluent-forward.sh` - configures the destination host and port for Fluent Bit forwarding. The script requires that both the host and port are present, and only in this case it would start. The script overwrites the `/opt/mellanox/doca/services/telemetry/config/fluent_bit_configs` folder and configures the `.exp` file.

17.11.3.2 Enabling Fluent Bit Forwarding

To enable Fluent Bit forward, add the destination host and port to the command line found in the `initContainers` section of the `.yaml` file:

```
command: ["/bin/bash", "-c", "/usr/bin/telemetry-init.sh && /usr/bin/enable-fluent-forward.sh -i=127.0.0.1
-p=24224"]
```

> ⚠ The host and port shown above are just an example. See section Fluent Bit to learn about manual configuration.

17.11.3.3 Generating Configuration

The configuration folder `/opt/mellanox/doca/services/telemetry/config` starts empty by default. Once the service starts, the initial scripts run as a part of the initial container and create configuration as described in section Enabling Fluent Bit Forwarding.

17.11.3.4 Resetting Configuration

Resetting the configuration can be done by deleting the content found in the configuration folder and restarting the service to generate the default configuration.

17.11.3.5 Enabling Providers

Providers are enabled from the `dts_config.ini` configuration file. Uncomment the `enable-provider=$provider-name` line to allow data collection for this provider. For example, uncommenting the following line enables the `ethtool` provider:

```
#enable-provider=ethtool
```

> ⚠ More information about telemetry providers can be found under the Providers section.

17.11.3.5.1 Remote Collection

Certain providers or components are unable to execute properly within the container due to various container limitations. Therefore, they would have to perform remote collection or execution.

The following steps enable remote collection:

1. Activate DOCA privileged executer (DPE), as DPE is how remote collection is achieved:

   ```
   systemctl start dpe
   ```

2. Add `grpc` before `provider-name` (i.e., `enable-provider=grpc.$provider-name`). For example, the following line configures remote collection of the `hcaperf` provider:

   ```
   enable-provider=grpc.hcaperf
   ```

3. If there are any configuration lines that are provider-specific, then add the `grpc` prefix as well. Building upon the previous example:

   ```
   grpc.hcaperf.mlx5_0=sample
   grpc.hcaperf.mlx5_1=sample
   ```

17.11.3.6 Enabling Data Write

Uncomment the following line in `dts_config.ini`:

```
#output=/data
```

> Changes in `dts_config.ini` force the main DTS process to restart in 60 seconds to apply the new settings.

17.11.3.7 Enabling IPC with Non-container Program

For information on enabling IPC between DTS and an application that runs outside of a container, refer to section "Using IPC with Non-container Application" in the DOCA Telemetry Exporter.

17.11.4 Description

17.11.4.1 Providers

DTS supports on-board data collection from `sysf`, `ethtool`, and `tc` providers. Fluent and Prometheus aggregator providers can collect the data from other applications.

Other providers are available based on different conditions (e.g., specific container mounts or host only such as `amber`, `ppcc_eth`, etc). Such providers are described with their dependencies in their corresponding sections.

17.11.4.1.1 Sysfs Counters List

The sysfs provider has several components: `ib_port` , `hw_port` , `mr_cache` , `eth` , `hwmon` and `bf_ptm` . By default, all the components (except `bf_ptm`) are enabled when the provider is enabled:

```
#disable-provider=sysfs
```

The components can be disabled separately. For instance, to disable `eth` :

```
enable-provider=sysfs
disable-provider=sysfs.eth
```

> `ib_port` and `ib_hvw` are state counters which are collected per port. These counters are only collected for ports whose state is active.

- `ib_port` counters:

```
{hca_name}:{port_num}:ib_port_state
{hca_name}:{port_num}:VL15_dropped
{hca_name}:{port_num}:excessive_buffer_overrun_errors
{hca_name}:{port_num}:link_downed
{hca_name}:{port_num}:link_error_recovery
{hca_name}:{port_num}:local_link_integrity_errors
{hca_name}:{port_num}:multicast_rcv_packets
{hca_name}:{port_num}:multicast_xmit_packets
{hca_name}:{port_num}:port_rcv_constraint_errors
{hca_name}:{port_num}:port_rcv_data
{hca_name}:{port_num}:port_rcv_errors
{hca_name}:{port_num}:port_rcv_packets
{hca_name}:{port_num}:port_rcv_remote_physical_errors
{hca_name}:{port_num}:port_rcv_switch_relay_errors
{hca_name}:{port_num}:port_xmit_constraint_errors
{hca_name}:{port_num}:port_xmit_data
{hca_name}:{port_num}:port_xmit_discards
{hca_name}:{port_num}:port_xmit_packets
{hca_name}:{port_num}:port_xmit_wait
{hca_name}:{port_num}:symbol_error
{hca_name}:{port_num}:unicast_rcv_packets
{hca_name}:{port_num}:unicast_xmit_packets
```

- `ib_hw` counters:

```
{hca_name}:{port_num}:hw_state
{hca_name}:{port_num}:hw_duplicate_request
{hca_name}:{port_num}:hw_implied_nak_seq_err
{hca_name}:{port_num}:hw_lifespan
{hca_name}:{port_num}:hw_local_ack_timeout_err
{hca_name}:{port_num}:hw_out_of_buffer
{hca_name}:{port_num}:hw_out_of_sequence
{hca_name}:{port_num}:hw_packet_seq_err
{hca_name}:{port_num}:hw_req_cqe_error
{hca_name}:{port_num}:hw_req_cqe_flush_error
{hca_name}:{port_num}:hw_req_remote_access_errors
{hca_name}:{port_num}:hw_req_remote_invalid_request
{hca_name}:{port_num}:hw_resp_cqe_error
{hca_name}:{port_num}:hw_resp_cqe_flush_error
{hca_name}:{port_num}:hw_resp_local_length_error
{hca_name}:{port_num}:hw_resp_remote_access_errors
{hca_name}:{port_num}:hw_rnr_nak_retry_err
{hca_name}:{port_num}:hw_rx_atomic_requests
{hca_name}:{port_num}:hw_rx_dct_connect
{hca_name}:{port_num}:hw_rx_icrc_encapsulated
{hca_name}:{port_num}:hw_rx_read_requests
{hca_name}:{port_num}:hw_rx_write_requests
```

- `ib_mr_cache` counters:

```
{hca_name}:mr_cache:size_{n}:cur
{hca_name}:mr_cache:size_{n}:limit
{hca_name}:mr_cache:size_{n}:miss
{hca_name}:mr_cache:size_{n}:size
```

Where `n` ranges from 0 to 24.

- `eth` counters:

```
{hca_name}:{device_name}:eth_collisions
{hca_name}:{device_name}:eth_multicast
{hca_name}:{device_name}:eth_rx_bytes
{hca_name}:{device_name}:eth_rx_compressed
{hca_name}:{device_name}:eth_rx_crc_errors
{hca_name}:{device_name}:eth_rx_dropped
{hca_name}:{device_name}:eth_rx_errors
{hca_name}:{device_name}:eth_rx_fifo_errors
{hca_name}:{device_name}:eth_rx_frame_errors
{hca_name}:{device_name}:eth_rx_length_errors
{hca_name}:{device_name}:eth_rx_missed_errors
{hca_name}:{device_name}:eth_rx_nohandler
{hca_name}:{device_name}:eth_rx_over_errors
{hca_name}:{device_name}:eth_rx_packets
{hca_name}:{device_name}:eth_tx_aborted_errors
{hca_name}:{device_name}:eth_tx_bytes
{hca_name}:{device_name}:eth_tx_carrier_errors
{hca_name}:{device_name}:eth_tx_compressed
{hca_name}:{device_name}:eth_tx_dropped
{hca_name}:{device_name}:eth_tx_errors
{hca_name}:{device_name}:eth_tx_fifo_errors
{hca_name}:{device_name}:eth_tx_heartbeat_errors
{hca_name}:{device_name}:eth_tx_packets
{hca_name}:{device_name}:eth_tx_window_errors
```

- BlueField-2 `hwmon` counters:

```
{hwmon_name}:{l3cache}:CYCLES
{hwmon_name}:{l3cache}:HITS_BANK0
{hwmon_name}:{l3cache}:HITS_BANK1
{hwmon_name}:{l3cache}:MISSES_BANK0
{hwmon_name}:{l3cache}:MISSES_BANK1
{hwmon_name}:{pcie}:IN_C_BYTE_CNT
{hwmon_name}:{pcie}:IN_C_PKT_CNT
{hwmon_name}:{pcie}:IN_NP_BYTE_CNT
{hwmon_name}:{pcie}:IN_NP_PKT_CNT
{hwmon_name}:{pcie}:IN_P_BYTE_CNT
{hwmon_name}:{pcie}:IN_P_PKT_CNT
{hwmon_name}:{pcie}:OUT_C_BYTE_CNT
{hwmon_name}:{pcie}:OUT_C_PKT_CNT
{hwmon_name}:{pcie}:OUT_NP_BYTE_CNT
{hwmon_name}:{pcie}:OUT_NP_PKT_CNT
{hwmon_name}:{pcie}:OUT_P_PKT_CNT
{hwmon_name}:{tile}:MEMORY_READS
{hwmon_name}:{tile}:MEMORY_WRITES
{hwmon_name}:{tile}:MSS_NO_CREDIT
{hwmon_name}:{tile}:VICTIM_WRITE
{hwmon_name}:{tilenet}:CDN_DIAG_C_OUT_OF_CRED
{hwmon_name}:{tilenet}:CDN_REQ
{hwmon_name}:{tilenet}:DDN_REQ
{hwmon_name}:{tilenet}:NDN_REQ
{hwmon_name}:{trio}:TDMA_DATA_BEAT
{hwmon_name}:{trio}:TDMA_PBUF_MAC_AF
{hwmon_name}:{trio}:TDMA_RT_AF
{hwmon_name}:{trio}:TPIO_DATA_BEAT
{hwmon_name}:{triogen}:TX_DAT_AF
{hwmon_name}:{triogen}:TX_DAT_AF
```

- BlueField-3 `hwmon` counters:

```
{hwmon_name}:{llt}:GDC_BANK0_RD_REQ
{hwmon_name}:{llt}:GDC_BANK1_RD_REQ
{hwmon_name}:{llt}:GDC_BANK0_WR_REQ
{hwmon_name}:{llt}:GDC_BANK1_WR_REQ
{hwmon_name}:{llt_miss}:GDC_MISS_MACHINE_RD_REQ
{hwmon_name}:{llt_miss}:GDC_MISS_MACHINE_WR_REQ
{hwmon_name}:{mss}:SKYLIB_DDN_TX_FLITS
{hwmon_name}:{mss}:SKYLIB_DDN_RX_FLITS
```

- BlueField-3 `bf_ptm` counters:

```
bf:ptm:active_power_profile
bf:ptm:atx_power_available
bf:ptm:core_temp
bf:ptm:ddr_temp
bf:ptm:error_state
bf:ptm:power_envelope
bf:ptm:power_throttling_event_count
bf:ptm:power_throttling_state
```

```
bf:ptm:thermal_throttling_event_count
bf:ptm:thermal_throttling_state
bf:ptm:throttling_state
bf:ptm:total_power
bf:ptm:vr0_power
bf:ptm:vr1_power
```

17.11.4.1.1.1 Port Counters

The following parameters are located in `/sys/class/infiniband/mlx5_0/ports/1/counters`.

Counter	Description	InfiniBand Spec Name	Group
`port_rcv_data`	The total number of data octets, divided by 4, (counting in double words, 32 bits), received on all VLs from the port.	`PortRcvData`	Informative
`port_rcv_packets`	Total number of packets (this may include packets containing Errors. This is 64 bit counter.	`PortRcvPkts`	Informative
`port_multicast_rcv_packets`	Total number of multicast packets, including multicast packets containing errors.	`PortMultiCastRcvPkts`	Informative
`port_unicast_rcv_packets`	Total number of unicast packets, including unicast packets containing errors.	`PortUnicastRcvPkts`	Informative
`port_xmit_data`	The total number of data octets, divided by 4, (counting in double words, 32 bits), transmitted on all VLs from the port.	`PortXmitData`	Informative
`port_xmit_packets` `port_xmit_packets_64`	Total number of packets transmitted on all VLs from this port. This may include packets with errors. This is 64 bit counter.	`PortXmitPkts`	Informative
`port_rcv_switch_relay_errors`	Total number of packets received on the port that were discarded because they could not be forwarded by the switch relay.	`PortRcvSwitchRelayErrors`	Error
`port_rcv_errors`	Total number of packets containing an error that were received on the port.	`PortRcvErrors`	Informative
`port_rcv_constraint_errors`	Total number of packets received on the switch physical port that are discarded.	`PortRcvConstraintErrors`	Error
`local_link_integrity_errors`	The number of times that the count of local physical errors exceeded the threshold specified by `LocalPhyErrors`.	`LocalLinkIntegrityErrors`	Error
`port_xmit_wait`	The number of ticks during which the port had data to transmit but no data was sent during the entire tick (either because of insufficient credits or because of lack of arbitration).	`PortXmitWait`	Informative

Counter	Description	InfiniBand Spec Name	Group
`port_multicast_xmit_packets`	Total number of multicast packets transmitted on all VLs from the port. This may include multicast packets with errors.	`PortMultiCastXmitPkts`	Informative
`port_unicast_xmit_packets`	Total number of unicast packets transmitted on all VLs from the port. This may include unicast packets with errors.	`PortUnicastXmitPkts`	Informative
`port_xmit_discards`	Total number of outbound packets discarded by the port because the port is down or congested.	`PortXmitDiscards`	Error
`port_xmit_constraint_errors`	Total number of packets not transmitted from the switch physical port.	`PortXmitConstraintErrors`	Error
`port_rcv_remote_physical_errors`	Total number of packets marked with the EBP delimiter received on the port.	`PortRcvRemotePhysicalErrors`	Error
`symbol_error`	Total number of minor link errors detected on one or more physical lanes.	`SymbolErrorCounter`	Error
`VL15_dropped`	Number of incoming VL15 packets dropped due to resource limitations (e.g., lack of buffers) of the port.	`VL15Dropped`	Error
`link_error_recovery`	Total number of times the Port Training state machine has successfully completed the link error recovery process.	`LinkErrorRecoveryCounter`	Error
`link_downed`	Total number of times the Port Training state machine has failed the link error recovery process and downed the link.	`LinkDownedCounter`	Error

17.11.4.1.1.2 Hardware Counters

The hardware counters, found under `/sys/class/infiniband/mlx5_0/ports/1/hw_counters/`, are counted per function and exposed on the function. Some counters are not counted per function. These counters are commented with a relevant comment.

Counter	Description	Group
`duplicate_request`	Number of received packets. A duplicate request is a request that had been previously executed.	Error
`implied_nak_seq_err`	Number of time the requested decided an ACK. with a PSN larger than the expected PSN for an RDMA read or response.	Error
`lifespan`	The maximum period in ms which defines the aging of the counter reads. Two consecutive reads within this period might return the same values	Informative

Counter	Description	Group
`local_ack_timeout_err`	The number of times QP's ack timer expired for RC, XRC, DCT QPs at the sender side. The QP retry limit was not exceed, therefore it is still recoverable error.	Error
`np_cnp_sent`	The number of CNP packets sent by the Notification Point when it noticed congestion experienced in the RoCEv2 IP header (ECN bits).	Informative
`np_ecn_marked_roce_packets`	The number of RoCEv2 packets received by the notification point which were marked for experiencing the congestion (ECN bits where '11' on the ingress RoCE traffic) .	Informative
`out_of_buffer`	The number of drops occurred due to lack of WQE for the associated QPs.	Error
`out_of_sequence`	The number of out of sequence packets received.	Error
`packet_seq_err`	The number of received NAK sequence error packets. The QP retry limit was not exceeded.	Error
`req_cqe_error`	The number of times requester detected CQEs completed with errors.	Error
`req_cqe_flush_error`	The number of times requester detected CQEs completed with flushed errors.	Error
`req_remote_access_errors`	The number of times requester detected remote access errors.	Error
`req_remote_invalid_request`	The number of times requester detected remote invalid request errors.	Error
`resp_cqe_error`	The number of times responder detected CQEs completed with errors.	Error
`resp_cqe_flush_error`	The number of times responder detected CQEs completed with flushed errors.	Error
`resp_local_length_error`	The number of times responder detected local length errors.	Error
`resp_remote_access_errors`	The number of times responder detected remote access errors.	Error
`rnr_nak_retry_err`	The number of received RNR NAK packets. The QP retry limit was not exceeded.	Error
`rp_cnp_handled`	The number of CNP packets handled by the Reaction Point HCA to throttle the transmission rate.	Informative
`rp_cnp_ignored`	The number of CNP packets received and ignored by the Reaction Point HCA. This counter should not raise if RoCE Congestion Control was enabled in the network. If this counter raise, verify that ECN was enabled on the adapter. See HowTo Configure DCQCN (RoCE CC) values for ConnectX-4 (Linux).	Error
`rx_atomic_requests`	The number of received ATOMIC request for the associated QPs.	Informative
`rx_dct_connect`	The number of received connection request for the associated DCTs.	Informative

Counter	Description	Group
`rx_read_requests`	The number of received READ requests for the associated QPs.	Informative
`rx_write_requests`	The number of received WRITE requests for the associated QPs.	Informative
`rx_icrc_encapsulated`	The number of RoCE packets with ICRC errors.	Error
`roce_adp_retrans`	Counts the number of adaptive retransmissions for RoCE traffic	Informative
`roce_adp_retrans_to`	Counts the number of times RoCE traffic reached timeout due to adaptive retransmission	Informative
`roce_slow_restart`	Counts the number of times RoCE slow restart was used	Informative
`roce_slow_restart_cnps`	Counts the number of times RoCE slow restart generated CNP packets	Informative
`roce_slow_restart_trans`	Counts the number of times RoCE slow restart changed state to slow restart	Informative
`roce_adp_retrans_to`	Counts the number of adaptive retransmissions for RoCE traffic	Informative
`roce_slow_restart`	Counts the number of times RoCE traffic reached timeout due to adaptive retransmission	Informative

17.11.4.1.1.3 Debug Status Counters

The following parameters are located in `/sys/class/net/<interface>/debug`.

Parameter	Description	Default

`lro_timeout`	Sets the LRO timer period value in usecs which will be used as LRO session expiration time. For example:
	`# cat /sys/class/net/eth2/debug/lro_timeout`
	`Actual timeout: 32`
	`Supported timeout: 8 16 32 1024`
		32

`link_down_reason`	Link down reason will allow the user to query the reason which is preventing the link from going up. For example:
	`$ cat /sys/class/net/ethXX/debug/link_down_reason`
	`monitor_opcode: 0x0`
	`status_message: The port is Active.`
	Refer to the adapter PRM for all possible options (PDDR register).
		N/A

17.11.4.1.2 Power Thermal Counters

The `bf_ptm` component collects BlueField-3 power thermal counters using remote collection. It is disabled by default and can be enabled as follows:

1. Load kernel module `mlxbf-ptm`:

```
modprobe -v mlxbf-ptm
```

2. Enable component using remote collection:

```
enable-provider=grpc.sysfs.bf_ptm
```

> DPE server should be active before changing the `dts_config.ini` file. See section "Remote Collection" for details.

17.11.4.1.3 Ethtool Counters

Ethtool counters is the generated list of counters which corresponds to Ethtool utility. Counters are generated on a per-device basis.

There are several counter groups, depending on where the counter is counted:

- Ring – software ring counters
- Software port – an aggregation of software ring counters
- vPort counters – traffic counters and drops due to steering or no buffers. May indicate BlueField issues. These counters include Ethernet traffic counters (including raw Ethernet) and RDMA/RoCE traffic counters.
- Physical port counters – the physical port connecting BlueField to the network. May indicate device issues or link or network issues. This measuring point holds information on standardized counters like IEEE 802.3, RFC2863, RFC 2819, RFC 3635 and additional counters like flow control, FEC, and more. Physical port counters are not exposed to virtual machines.
- Priority port counters – a set of the physical port counters, per priory per port

Each group of counters may have different counter types:

- Traffic informative counters - counters which counts traffic. These counters can be used for load estimation of for general debug.
- Traffic acceleration counters - counters which counts traffic accelerated by NVIDIA drivers or by hardware. The counters are an additional layer to the informative counter set and the same traffic is counted in both informative and acceleration counters. Acceleration counters are marked with [A].
- Error counters - increment of these counters might indicate a problem

The following acceleration mechanisms have dedicated counters:

- TCP segmentation offload (TSO) – increasing outbound throughput and reducing CPU utilization by allowing the kernel to buffer multiple packets in a single large buffer. The BlueField splits the buffer into packet and transmits it.
- Large receive offload (LRO) – increasing inbound throughput and reducing CPU utilization by aggregation of multiple incoming packets of a single stream to a single buffer
- CHECKSUM – calculation of TCP checksum (by the BlueField). The following checksum offloads are available (refer to skbuff.h for detailed explanation)
    - `CHECKSUM_UNNECESSARY`

    - `CHECKSUM_NONE` - no checksum acceleration was used
    - `CHECKSUM_COMPLETE` - device provided checksum on the entire packet
    - `CHECKSUM_PARTIAL` - device provided checksum
- CQE compress - compression of completion queue events (CQE) used for sparing bandwidth on PCIe and hence achieve better performance.

17.11.4.1.3.1 Ring/Software Port Counters

The following counters are available per ring or software port.

These counters provide information on the amount of traffic accelerated by the BlueField. The counters tally the accelerated traffic in addition to the standard counters which tally that (i.e. accelerated traffic is counted twice).

The counter names in the table below refers to both ring and port counters. the notation for ring counters includes the `[i]` index without the braces. the notation for port counters does not include the `[i]`. a counter name `rx[i]_packets` will be printed as `rx0_packets` for ring 0 and `rx_packets` for the software port

Counter	Description	Type
`rx[i]_packets`	The number of packets received on ring i.	Informative
`rx[i]_bytes`	The number of bytes received on ring i.	Informative
`tx[i]_packets`	The number of packets transmitted on ring i.	Informative
`tx[i]_bytes`	The number of bytes transmitted on ring i.	Informative
`tx[i]_tso_packets`	The number of TSO packets transmitted on ring i [A].	Acceleration
`tx[i]_tso_bytes`	The number of TSO bytes transmitted on ring i [A].	Acceleration
`tx[i]_tso_inner_packets`	The number of TSO packets which are indicated to be carry internal encapsulation transmitted on ring i [A]	Acceleration
`tx[i]_tso_inner_bytes`	The number of TSO bytes which are indicated to be carry internal encapsulation transmitted on ring i [A].	Acceleration
`rx[i]_lro_packets`	The number of LRO packets received on ring i [A].	Acceleration
`rx[i]_lro_bytes`	The number of LRO bytes received on ring i [A].	Acceleration
`rx[i]_csum_unnecessary`	Packets received with a `CHECKSUM_UNNECESSARY` on ring i [A].	Acceleration
`rx[i]_csum_none`	Packets received with `CHECKSUM_NONE` on ring i [A].	Acceleration
`rx[i]_csum_complete`	Packets received with a `CHECKSUM_COMPLETE` on ring i [A].	Acceleration
`rx[i]_csum_unnecessary_inner`	Packets received with inner encapsulation with a CHECK_SUM UNNECESSARY on ring i [A].	Acceleration
`tx[i]_csum_partial`	Packets transmitted with a `CHECKSUM_PARTIAL` on ring i [A].	Acceleration
`tx[i]_csum_partial_inner`	Packets transmitted with inner encapsulation with a CHECKSUM_PARTIAL on ring i [A].	Acceleration

Counter	Description	Type
`tx[i]_csum_none`	Packets transmitted with no hardware checksum acceleration on ring i.	Informative
`tx[i]_stopped` `tx_queue_stopped` [1]	Events where SQ was full on ring i. If this counter is increased, check the amount of buffers allocated for transmission.	Error
`tx[i]_wake` `tx_queue_wake` [1]	Events where SQ was full and has become not full on ring i.	Error
`tx[i]_dropped` `tx_queue_dropped` [1]	Packets transmitted that were dropped due to DMA mapping failure on ring i. If this counter is increased, check the amount of buffers allocated for transmission.	Error
`rx[i]_wqe_err`	The number of wrong opcodes received on ring i.	Error
`tx[i]_nop`	The number of no WQEs (empty WQEs) inserted to the SQ (related to ring i) due to the reach of the end of the cyclic buffer. When reaching near to the end of cyclic buffer the driver may add those empty WQEs to avoid handling a state the a WQE start in the end of the queue and ends in the beginning of the queue. This is a normal condition.	Informative
`rx[i]_mpwqe_frag`	The number of WQEs that failed to allocate compound page and hence fragmented MPWQE's (multipacket WQEs) were used on ring i. If this counter raise, it may suggest that there is no enough memory for large pages, the driver allocated fragmented pages. This is not abnormal condition.	Informative
`rx[i]_mpwqe_filler_cqes`	The number of filler CQEs events that where issued on ring i. The counter name before kernel 4.19 was `rx[i]_mpwqe_filler`.	Informative
`rx[i]_cqe_compress_blks`	The number of receive blocks with CQE compression on ring i [A].	Acceleration
`rx[i]_cqe_compress_pkts`	The number of receive packets with CQE compression on ring i [A].	Acceleration
`rx[i]_cache_reuse`	The number of events of successful reuse of a page from a driver's internal page cache	Acceleration
`rx[i]_cache_full`	The number of events of full internal page cache where driver can't put a page back to the cache for recycling (page will be freed)	Acceleration
`rx[i]_cache_empty`	The number of events where cache was empty - no page to give. driver shall allocate new page	Acceleration
`rx[i]_cache_busy`	The number of events where cache head was busy and cannot be recycled. driver allocated new page	Acceleration
`rx[i]_xmit_more`	The number of packets sent with xmit_more indication set on the skbuff (no doorbell)	Acceleration
`tx[i]_cqes`	The number of completions received on the CQ of TX ring.	Informative

Counter	Description	Type
`ch[i]_poll`	The number of invocations of NAPI poll of channel.	Informative
`ch[i]_arm`	The number of times the NAPI poll function completed and armed the completion queues on channel ⓘ Supported from kernel 4.19.	Informative
`ch[i]_aff_change`	The number of times the NAPI poll function explicitly stopped execution on a CPU due to a change in affinity, on channel.	Informative
`rx[i]_congst_umr`	The number of times an outstanding UMR request is delayed due to congestion, on ring. ⓘ Supported from kernel 4.19.	Error
`ch[i]_events`	The number of hard interrupt events on the completion queues of channel.	Informative
`rx[i]_mpwqe_filler_strides`	The number of strides consumed by filler CQEs on ring.	Informative
`rx[i]_xdp_tx_xmit`	The number of packets forwarded back to the port due to XDP program XDP_TX action (bouncing). these packets are not counted by other software counters. These packets are counted by physical port and vPort counters.	Informative
`rx[i]_xdp_tx_full`	The number of packets that should have been forwarded back to the port due to `XDP_TX` action but were dropped due to full tx queue. these packets are not counted by other software counters. These packets are counted by physical port and vPort counters You may open more rx queues and spread traffic rx over all queues and/or increase rx ring size.	Error
`rx[i]_xdp_tx_err`	The number of times an XDP_TX error such as frame too long and frame too short occurred on `XDP_TX` ring of RX ring.	Error
`rx[i]_xdp_tx_cqes` `rx_xdp_tx_cqe` [1]	The number of completions received on the CQ of the XDP-TX ring.	Informative
`rx[i]_xdp_drop`	The number of packets dropped due to XDP program `XDP_DROP` action. these packets are not counted by other software counters. These packets are counted by physical port and vPort counters.	Informative
`rx[i]_xdp_redirect`	The number of times an XDP redirect action has been triggered on ring.	Acceleration
`tx[i]_xdp_xmit`	The number of packets redirected to the interface (due to XDP redirect). These packets are not counted by other software counters. These packets are counted by physical port and vPort counters.	Informative

Counter	Description	Type
`tx[i]_xdp_full`	The number of packets redirected to the interface (due to XDP redirect) but were dropped due to the Tx queue being full. These packets are not counted by other software counters. Users may enlarge Tx queues.	Informative
`tx[i]_xdp_err`	The number of packets redirected to the interface (due to XDP redirect) but were dropped due to an error (e.g., frame too long and frame too short).	Error
`tx[i]_xdp_cqes`	The number of completions received for packets redirected to the interface (due to XDP redirect) on the CQ.	Informative
`rx[i]_cache_waive`	The number of cache evacuation. This can occur due to page move to another NUMA node or page was pfmemalloc-ed and should be freed as soon as possible.	Acceleration

[1] The corresponding ring and global counters do not share the same name (i.e., do not follow the common naming scheme). ↩ ↩ ↩ ↩

17.11.4.1.3.2 vPort Counters

Counters on the eswitch port that is connected to the vNIC.

Counter	Description	Type
`rx_vport_unicast_packets`	Unicast packets received, steered to a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
`rx_vport_unicast_bytes`	Unicast bytes received, steered to a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
`tx_vport_unicast_packets`	Unicast packets transmitted, steered from a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
`tx_vport_unicast_bytes`	Unicast bytes transmitted, steered from a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
`rx_vport_multicast_packets`	Multicast packets received, steered to a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
`rx_vport_multicast_bytes`	Multicast bytes received, steered to a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
`tx_vport_multicast_packets`	Multicast packets transmitted, steered from a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
`tx_vport_multicast_bytes`	Multicast bytes transmitted, steered from a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative

Counter	Description	Type
`rx_vport_broadcast_packets`	Broadcast packets received, steered to a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
`rx_vport_broadcast_bytes`	Broadcast bytes received, steered to a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
`tx_vport_broadcast_packets`	Broadcast packets transmitted, steered from a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
`tx_vport_broadcast_bytes`	Broadcast packets transmitted, steered from a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
`rx_vport_rdma_unicast_packets`	RDMA unicast packets received, steered to a port (counters counts RoCE/UD/RC traffic) [A]	Acceleration
`rx_vport_rdma_unicast_bytes`	RDMA unicast bytes received, steered to a port (counters counts RoCE/UD/RC traffic) [A]	Acceleration
`tx_vport_rdma_unicast_packets`	RDMA unicast packets transmitted, steered from a port (counters counts RoCE/UD/RC traffic) [A]	Acceleration
`tx_vport_rdma_unicast_bytes`	RDMA unicast bytes transmitted, steered from a port (counters counts RoCE/UD/RC traffic) [A]	Acceleration
`rx_vport_ rdma _multicast_packets`	RDMA multicast packets received, steered to a port (counters counts RoCE/UD/RC traffic) [A]	Acceleration
`rx_vport_ rdma _multicast_bytes`	RDMA multicast bytes received, steered to a port (counters counts RoCE/UD/RC traffic) [A]	Acceleration
`tx_vport_ rdma _multicast_packets`	RDMA multicast packets transmitted, steered from a port (counters counts RoCE/UD/RC traffic) [A]	Acceleration
`tx_vport_ rdma _multicast_bytes`	RDMA multicast bytes transmitted, steered from a port (counters counts RoCE/UD/RC traffic) [A]	Acceleration
`rx_steer_missed_packets`	Number of packets received by the NIC but discarded due to not matching any flow in the NIC flow table. ⓘ Supported from kernel 4.16.	Error
`rx_packets`	Representor only: packets received, that were handled by the hypervisor. ⓘ Supported from kernel 4.18.	Informative

Counter	Description	Type
`rx_bytes`	Representor only: bytes received, that were handled by the hypervisor. ⓘ Supported from kernel 4.18.	Informative
`tx_packets`	Representor only: packets transmitted which have been handled by the hypervisor. ⓘ Supported from kernel 4.18.	Informative
`tx_bytes`	Representor only: bytes transmitted which have been handled by the hypervisor. ⓘ Supported from kernel 4.18.	Informative

17.11.4.1.3.3 Physical Port Counters

The physical port counters are the counters on the external port connecting adapter to the network. This measuring point holds information on standardized counters like IEEE 802.3, RFC2863, RFC 2819, RFC 3635 and additional counters like flow control, FEC and more.

Counter	Description	Type
`rx_packets_phy`	The number of packets received on the physical port. This counter doesn't include packets that were discarded due to FCS, frame size and similar errors.	Informative
`tx_packets_phy`	The number of packets transmitted on the physical port.	Informative
`rx_bytes_phy`	The number of bytes received on the physical port, including Ethernet header and FCS.	Informative
`tx_bytes_phy`	The number of bytes transmitted on the physical port.	Informative
`rx_multicast_phy`	The number of multicast packets received on the physical port.	Informative
`tx_multicast_phy`	The number of multicast packets transmitted on the physical port.	Informative
`rx_broadcast_phy`	The number of broadcast packets received on the physical port.	Informative
`tx_broadcast_phy`	The number of broadcast packets transmitted on the physical port.	Informative
`rx_crc_errors_phy`	The number of dropped received packets due to frame check sequence (FCS) error on the physical port. If this counter is increased in high rate, check the link quality using `rx_symbol_error_phy` and `rx_corrected_bits_phy` counters below.	Error
`rx_in_range_len_errors_phy`	The number of received packets dropped due to length/type errors on a physical port.	Error

Counter	Description	Type
`rx_out_of_range_len_phy`	The number of received packets dropped due to length greater than allowed on a physical port. If this counter is increasing, it implies that the peer connected to the adapter has a larger MTU configured. Using same MTU configuration shall resolve this issue.	Error
`rx_oversize_pkts_phy`	The number of dropped received packets due to length which exceed MTU size on a physical port. If this counter is increasing, it implies that the peer connected to the adapter has a larger MTU configured. Using same MTU configuration shall resolve this issue.	Error
`rx_symbol_err_phy`	The number of received packets dropped due to physical coding errors (symbol errors) on a physical port.	Error
`rx_mac_control_phy`	The number of MAC control packets received on the physical port.	Informative
`tx_mac_control_phy`	The number of MAC control packets transmitted on the physical port.	Informative
`rx_pause_ctrl_phy`	The number of link layer pause packets received on a physical port. If this counter is increasing, it implies that the network is congested and cannot absorb the traffic coming from to the adapter.	Informative
`tx_pause_ctrl_phy`	The number of link layer pause packets transmitted on a physical port. If this counter is increasing, it implies that the NIC is congested and cannot absorb the traffic coming from the network.	Informative
`rx_unsupported_op_phy`	The number of MAC control packets received with unsupported opcode on a physical port.	Error
`rx_discards_phy`	The number of received packets dropped due to lack of buffers on a physical port. If this counter is increasing, it implies that the adapter is congested and cannot absorb the traffic coming from the network.	Error
`tx_discards_phy`	The number of packets which were discarded on transmission, even no errors were detected. the drop might occur due to link in down state, head of line drop, pause from the network, etc.	Error
`tx_errors_phy`	The number of transmitted packets dropped due to a length which exceed MTU size on a physical port.	Error
`rx_undersize_pkts_phy`	The number of received packets dropped due to length which is shorter than 64 bytes on a physical port. If this counter is increasing, it implies that the peer connected to the adapter has a non-standard MTU configured or malformed packet had arrived.	Error
`rx_fragments_phy`	The number of received packets dropped due to a length which is shorter than 64 bytes and has FCS error on a physical port. If this counter is increasing, it implies that the peer connected to the adapter has a non-standard MTU configured.	Error

Counter	Description	Type
`rx_jabbers_phy`	The number of received packets d due to a length which is longer than 64 bytes and had FCS error on a physical port.	Error
`rx_64_bytes_phy`	The number of packets received on the physical port with size of 64 bytes.	Informative
`rx_65_to_127_bytes_phy`	The number of packets received on the physical port with size of 65 to 127 bytes.	Informative
`rx_128_to_255_bytes_phy`	The number of packets received on the physical port with size of 128 to 255 bytes.	Informative
`rx_256_to_511_bytes_phy`	The number of packets received on the physical port with size of 256 to 512 bytes.	Informative
`rx_512_to_1023_bytes_phy`	The number of packets received on the physical port with size of 512 to 1023 bytes.	Informative
`rx_1024_to_1518_bytes_phy`	The number of packets received on the physical port with size of 1024 to 1518 bytes.	Informative
`rx_1519_to_2047_bytes_phy`	The number of packets received on the physical port with size of 1519 to 2047 bytes.	Informative
`rx_2048_to_4095_bytes_phy`	The number of packets received on the physical port with size of 2048 to 4095 bytes.	Informative
`rx_4096_to_8191_bytes_phy`	The number of packets received on the physical port with size of 4096 to 8191 bytes.	Informative
`rx_8192_to_10239_bytes_phy`	The number of packets received on the physical port with size of 8192 to 10239 bytes.	Informative
`link_down_events_phy`	The number of times where the link operative state changed to down. In case this counter is increasing it may imply on port flapping. You may need to replace the cable/transceiver.	Error
`rx_out_of_buffer`	Number of times receive queue had no software buffers allocated for the adapter's incoming traffic.	Error
`module_bus_stuck`	The number of times that module's I^2C bus (data or clock) short-wire was detected. You may need to replace the cable/transceiver. ⓘ Supported from kernel 4.10.	Error
`module_high_temp`	The number of times that the module temperature was too high. If this issue persists, you may need to check the ambient temperature or replace the cable/transceiver module. ⓘ Supported from kernel 4.10.	Error
`module_bad_shorted`	The number of times that the module cables were shorted. You may need to replace the cable/transceiver module. ⓘ Supported from kernel 4.10.	Error

Counter	Description	Type
`module_unplug`	The number of times that module was ejected. ⓘ Supported from kernel 4.10.	Informative
`rx_buffer_passed_thres_phy`	The number of events where the port receive buffer was over 85% full. ⓘ Supported from kernel 4.14.	Informative
`tx_pause_storm_warning_events`	The number of times the device was sending pauses for a long period of time. ⓘ Supported from kernel 4.15.	Informative
`tx_pause_storm_error_events`	The number of times the device was sending pauses for a long period of time, reaching time out and disabling transmission of pause frames. on the period where pause frames were disabled, drop could have been occurred. ⓘ Supported from kernel 4.15.	Error
`rx[i]_buff_alloc_err / rx_buff_alloc_err`	Failed to allocate a buffer to received packet (or SKB) on port (or per ring)	Error
`rx_bits_phy`	This counter provides information on the total amount of traffic that could have been received and can be used as a guideline to measure the ratio of errored traffic in `rx_pcs_symbol_err_phy` and `rx_corrected_bits_phy` .	Informative
`rx_pcs_symbol_err_phy`	This counter counts the number of symbol errors that wasn't corrected by FEC correction algorithm or that FEC algorithm was not active on this interface. If this counter is increasing, it implies that the link between the NIC and the network is suffering from high BER, and that traffic is lost. You may need to replace the cable/transceiver. The error rate is the number of `rx_pcs_symbol_err_phy` divided by the number of `rx_phy_bits` on a specific time frame.	Error
`rx_corrected_bits_phy`	The number of corrected bits on this port according to active FEC (RS/FC). If this counter is increasing, it implies that the link between the NIC and the network is suffering from high BER. The corrected bit rate is the number of `rx_corrected_bits_phy` divided by the number of `rx_phy_bits` on a specific time frame	Error

Counter	Description	Type
phy_raw_errors_lane[l]	This counter counts the number of physical raw errors per lane [l] index. The counter counts errors before FEC corrections. If this counter is increasing, it implies that the link between the NIC and the network is suffering from high BER, and that traffic might be lost. You may need to replace the cable/ transceiver. Please check in accordance with rx_corrected_bits_phy . ⓘ Supported from kernel 4.20.	Error

17.11.4.1.3.4 Priority Port Counters

The following counters are physical port counters that being counted per L2 priority (0-7).

> ⓘ p in the counter name represents the priority.

Counter	Description	Type
rx_prio[p]_bytes	The number of bytes received with priority p on the physical port.	Informative
rx_prio[p]_packets	The number of packets received with priority p on the physical port.	Informative
tx_prio[p]_bytes	The number of bytes transmitted on priority p on the physical port.	Informative
tx_prio[p]_packets	The number of packets transmitted on priority p on the physical port.	Informative
rx_prio[p]_pause	The number of pause packets received with priority p on a physical port. If this counter is increasing, it implies that the network is congested and cannot absorb the traffic coming from the adapter. **Note:** This counter is available only if PFC was enabled on priority p. Refer to HowTo Configure PFC on ConnectX-4.	Informative
rx_prio[p]_pause_duration	The duration of pause received (in microSec) on priority p on the physical port. The counter represents the time the port did not send any traffic on this priority. If this counter is increasing, it implies that the network is congested and cannot absorb the traffic coming from the adapter. **Note:** This counter is available only if PFC was enabled on priority p. Refer to HowTo Configure PFC on ConnectX-4.	Informative
rx_prio[p]_pause_transition	The number of times a transition from Xoff to Xon on priority p on the physical port has occurred. **Note:** This counter is available only if PFC was enabled on priority p. Refer to HowTo Configure PFC on ConnectX-4.	Informative

Counter	Description	Type
`tx_prio[p]_pause`	The number of pause packets transmitted on priority p on a physical port. If this counter is increasing, it implies that the adapter is congested and cannot absorb the traffic coming from the network. **Note:** This counter is available only if PFC was enabled on priority p. Refer to HowTo Configure PFC on ConnectX-4.	Informative
`tx_prio[p]_pause_duration`	The duration of pause transmitter (in microSec) on priority p on the physical port. **Note:** This counter is available only if PFC was enabled on priority p. Refer to HowTo Configure PFC on ConnectX-4.	Informative
`rx_prio[p]_buf_discard`	The number of packets discarded by device due to lack of per host receive buffers. ⓘ Supported from kernel 5.3.	Informative
`rx_prio[p]_cong_discard`	The number of packets discarded by device due to per host congestion. ⓘ Supported from kernel 5.3.	Informative
`rx_prio[p]_marked`	The number of packets ecn marked by device due to per host congestion. ⓘ Supported from kernel 5.3.	Informative
`rx_prio[p]_discard`	The number of packets discarded by device due to lack of receive buffers. ⓘ Supported from kernel 5.6.	Infornative

17.11.4.1.3.5 Device Counters

Counter	Description	Type
`rx_pci_signal_integrity`	Counts physical layer PCIe signal integrity errors, the number of transitions to recovery due to Framing errors and CRC (dlp and tlp). If this counter is raising, try moving the adapter card to a different slot to rule out a bad PCIe slot. Validate that you are running with the latest firmware available and latest server BIOS version.	Error
`tx_pci_signal_integrity`	Counts physical layer PCIe signal integrity errors, the number of transition to recovery initiated by the other side (moving to recovery due to getting TS/EIEOS). If this counter is raising, try moving the adapter card to a different slot to rule out a bad PCI slot. Validate that you are running with the latest firmware available and latest server BIOS version.	Error

Counter	Description	Type
`outbound_pci_buffer_overflow`	The number of packets dropped due to pci buffer overflow. If this counter is raising in high rate, it might indicate that the receive traffic rate for a host is larger than the PCIe bus and therefore a congestion occurs. ⓘ Supported from kernel 4.14.	Informative
`outbound_pci_stalled_rd`	The percentage (in the range 0...100) of time within the last second that the NIC had outbound non-posted reads requests but could not perform the operation due to insufficient posted credits. ⓘ Supported from kernel 4.14.	Informative
`outbound_pci_stalled_wr`	The percentage (in the range 0...100) of time within the last second that the NIC had outbound posted writes requests but could not perform the operation due to insufficient posted credits. ⓘ Supported from kernel 4.14.	Informative
`outbound_pci_stalled_rd_events`	The number of seconds where `outbound_pci_stalled_rd` was above 30%. ⓘ Supported from kernel 4.14.	Informative
`outbound_pci_stalled_wr_events`	The number of seconds where `outbound_pci_stalled_wr` was above 30%. ⓘ Supported from kernel 4.14.	Informative
`dev_out_of_buffer`	The number of times the device owned queue had not enough buffers allocated.	Error

17.11.4.1.3.6 Full List of Counters

```
# ethtool -S eth5

NIC statistics:
rx_packets: 10
rx_bytes: 3420
tx_packets: 18
tx_bytes: 1296
tx_tso_packets: 0
tx_tso_bytes: 0
tx_tso_inner_packets: 0
tx_tso_inner_bytes: 0
tx_added_vlan_packets: 0
tx_nop: 0
rx_lro_packets: 0
rx_lro_bytes: 0
rx_ecn_mark: 0
rx_removed_vlan_packets: 0
rx_csum_unnecessary: 0
rx_csum_none: 0
rx_csum_complete: 10
rx_csum_unnecessary_inner: 0
rx_xdp_drop: 0
```

```
rx_xdp_redirect: 0
rx_xdp_tx_xmit: 0
rx_xdp_tx_full: 0
rx_xdp_tx_err: 0
rx_xdp_tx_cqe: 0
tx_csum_none: 18
tx_csum_partial: 0
tx_csum_partial_inner: 0
tx_queue_stopped: 0
tx_queue_dropped: 0
tx_xmit_more: 0
tx_recover: 0
tx_cqes: 18
tx_queue_wake: 0
tx_udp_seg_rem: 0
tx_cqe_err: 0
tx_xdp_xmit: 0
tx_xdp_full: 0
tx_xdp_err: 0
tx_xdp_cqes: 0
rx_wqe_err: 0
rx_mpwqe_filler_cqes: 0
rx_mpwqe_filler_strides: 0
rx_buff_alloc_err: 0
rx_cqe_compress_blks: 0
rx_cqe_compress_pkts: 0
rx_page_reuse: 0
rx_cache_reuse: 0
rx_cache_full: 0
rx_cache_empty: 2688
rx_cache_busy: 0
rx_cache_waive: 0
rx_congst_umr: 0
rx_arfs_err: 0
ch_events: 75
ch_poll: 75
ch_arm: 75
ch_aff_change: 0
ch_eq_rearm: 0
rx_out_of_buffer: 0
rx_if_down_packets: 15
rx_steer_missed_packets: 0
rx_vport_unicast_packets: 0
rx_vport_unicast_bytes: 0
tx_vport_unicast_packets: 0
tx_vport_unicast_bytes: 0
rx_vport_multicast_packets: 2
rx_vport_multicast_bytes: 172
tx_vport_multicast_packets: 12
tx_vport_multicast_bytes: 936
rx_vport_broadcast_packets: 37
rx_vport_broadcast_bytes: 9270
tx_vport_broadcast_packets: 6
tx_vport_broadcast_bytes: 360
rx_vport_rdma_unicast_packets: 0
rx_vport_rdma_unicast_bytes: 0
tx_vport_rdma_unicast_packets: 0
tx_vport_rdma_unicast_bytes: 0
rx_vport_rdma_multicast_packets: 0
rx_vport_rdma_multicast_bytes: 0
tx_vport_rdma_multicast_packets: 0
tx_vport_rdma_multicast_bytes: 0
tx_packets_phy: 0
rx_packets_phy: 0
rx_crc_errors_phy: 0
tx_bytes_phy: 0
rx_bytes_phy: 0
tx_multicast_phy: 0
tx_broadcast_phy: 0
rx_multicast_phy: 0
rx_broadcast_phy: 0
rx_in_range_len_errors_phy: 0
rx_out_of_range_len_phy: 0
rx_oversize_pkts_phy: 0
rx_symbol_err_phy: 0
tx_mac_control_phy: 0
rx_mac_control_phy: 0
rx_unsupported_op_phy: 0
rx_pause_ctrl_phy: 0
tx_pause_ctrl_phy: 0
rx_discards_phy: 0
tx_discards_phy: 0
tx_errors_phy: 0
rx_undersize_pkts_phy: 0
rx_fragments_phy: 0
rx_jabbers_phy: 0
rx_64_bytes_phy: 0
rx_65_to_127_bytes_phy: 0
rx_128_to_255_bytes_phy: 0
rx_256_to_511_bytes_phy: 0
rx_512_to_1023_bytes_phy: 0
rx_1024_to_1518_bytes_phy: 0
rx_1519_to_2047_bytes_phy: 0
rx_2048_to_4095_bytes_phy: 0
rx_4096_to_8191_bytes_phy: 0
rx_8192_to_10239_bytes_phy: 0
link_down_events_phy: 0
rx_prio0_bytes: 0
rx_prio0_packets: 0
tx_prio0_bytes: 0
tx_prio0_packets: 0
rx_prio1_bytes: 0
rx_prio1_packets: 0
tx_prio1_bytes: 0
```

```
tx_prio1_packets: 0
rx_prio2_bytes: 0
rx_prio2_packets: 0
tx_prio2_bytes: 0
tx_prio2_packets: 0
rx_prio3_bytes: 0
rx_prio3_packets: 0
tx_prio3_bytes: 0
tx_prio3_packets: 0
rx_prio4_bytes: 0
rx_prio4_packets: 0
tx_prio4_bytes: 0
tx_prio4_packets: 0
rx_prio5_bytes: 0
rx_prio5_packets: 0
tx_prio5_bytes: 0
tx_prio5_packets: 0
rx_prio6_bytes: 0
rx_prio6_packets: 0
tx_prio6_bytes: 0
tx_prio6_packets: 0
rx_prio7_bytes: 0
rx_prio7_packets: 0
tx_prio7_bytes: 0
tx_prio7_packets: 0
module_unplug: 0
module_bus_stuck: 0
module_high_temp: 0
module_bad_shorted: 0
ch0_events: 9
ch0_poll: 9
ch0_arm: 9
ch0_aff_change: 0
ch0_eq_rearm: 0
ch1_events: 23
ch1_poll: 23
ch1_arm: 23
ch1_aff_change: 0
ch1_eq_rearm: 0
ch2_events: 8
ch2_poll: 8
ch2_arm: 8
ch2_aff_change: 0
ch2_eq_rearm: 0
ch3_events: 19
ch3_poll: 19
ch3_arm: 19
ch3_aff_change: 0
ch3_eq_rearm: 0
ch4_events: 8
ch4_poll: 8
ch4_arm: 8
ch4_aff_change: 0
ch4_eq_rearm: 0
ch5_events: 8
ch5_poll: 8
ch5_arm: 8
ch5_aff_change: 0
ch5_eq_rearm: 0
rx0_packets: 0
rx0_bytes: 0
rx0_csum_complete: 0
rx0_csum_unnecessary: 0
rx0_csum_unnecessary_inner: 0
rx0_csum_none: 0
rx0_xdp_drop: 0
rx0_xdp_redirect: 0
rx0_lro_packets: 0
rx0_lro_bytes: 0
rx0_ecn_mark: 0
rx0_removed_vlan_packets: 0
rx0_wqe_err: 0
rx0_mpwqe_filler_cqes: 0
rx0_mpwqe_filler_strides: 0
rx0_buff_alloc_err: 0
rx0_cqe_compress_blks: 0
rx0_cqe_compress_pkts: 0
rx0_page_reuse: 0
rx0_cache_reuse: 0
rx0_cache_full: 0
rx0_cache_empty: 448
rx0_cache_busy: 0
rx0_cache_waive: 0
rx0_congst_umr: 0
rx0_arfs_err: 0
rx0_xdp_tx_xmit: 0
rx0_xdp_tx_full: 0
rx0_xdp_tx_err: 0
rx0_xdp_tx_cqes: 0
rx1_packets: 10
rx1_bytes: 3420
rx1_csum_complete: 10
rx1_csum_unnecessary: 0
rx1_csum_unnecessary_inner: 0
rx1_csum_none: 0
rx1_xdp_drop: 0
rx1_xdp_redirect: 0
rx1_lro_packets: 0
rx1_lro_bytes: 0
rx1_ecn_mark: 0
rx1_removed_vlan_packets: 0
rx1_wqe_err: 0
rx1_mpwqe_filler_cqes: 0
rx1_mpwqe_filler_strides: 0
```

```
rx1_buff_alloc_err: 0
rx1_cqe_compress_blks: 0
rx1_cqe_compress_pkts: 0
rx1_page_reuse: 0
rx1_cache_reuse: 0
rx1_cache_full: 0
rx1_cache_empty: 448
rx1_cache_busy: 0
rx1_cache_waive: 0
rx1_congst_umr: 0
rx1_arfs_err: 0
rx1_xdp_tx_xmit: 0
rx1_xdp_tx_full: 0
rx1_xdp_tx_err: 0
rx1_xdp_tx_cqes: 0
rx2_packets: 0
rx2_bytes: 0
rx2_csum_complete: 0
rx2_csum_unnecessary: 0
rx2_csum_unnecessary_inner: 0
rx2_csum_none: 0
rx2_xdp_drop: 0
rx2_xdp_redirect: 0
rx2_lro_packets: 0
rx2_lro_bytes: 0
rx2_ecn_mark: 0
rx2_removed_vlan_packets: 0
rx2_wqe_err: 0
rx2_mpwqe_filler_cqes: 0
rx2_mpwqe_filler_strides: 0
rx2_buff_alloc_err: 0
rx2_cqe_compress_blks: 0
rx2_cqe_compress_pkts: 0
rx2_page_reuse: 0
rx2_cache_reuse: 0
rx2_cache_full: 0
rx2_cache_empty: 448
rx2_cache_busy: 0
rx2_cache_waive: 0
rx2_congst_umr: 0
rx2_arfs_err: 0
rx2_xdp_tx_xmit: 0
rx2_xdp_tx_full: 0
rx2_xdp_tx_err: 0
rx2_xdp_tx_cqes: 0
...
tx0_packets: 1
tx0_bytes: 60
tx0_tso_packets: 0
tx0_tso_bytes: 0
tx0_tso_inner_packets: 0
tx0_tso_inner_bytes: 0
tx0_csum_partial: 0
tx0_csum_partial_inner: 0
tx0_added_vlan_packets: 0
tx0_nop: 0
tx0_csum_none: 1
tx0_stopped: 0
tx0_dropped: 0
tx0_xmit_more: 0
tx0_recover: 0
tx0_cqes: 1
tx0_wake: 0
tx0_cqe_err: 0
tx1_packets: 5
tx1_bytes: 300
tx1_tso_packets: 0
tx1_tso_bytes: 0
tx1_tso_inner_packets: 0
tx1_tso_inner_bytes: 0
tx1_csum_partial: 0
tx1_csum_partial_inner: 0
tx1_added_vlan_packets: 0
tx1_nop: 0
tx1_csum_none: 5
tx1_stopped: 0
tx1_dropped: 0
tx1_xmit_more: 0
tx1_recover: 0
tx1_cqes: 5
tx1_wake: 0
tx1_cqe_err: 0
tx2_packets: 0
tx2_bytes: 0
tx2_tso_packets: 0
tx2_tso_bytes: 0
tx2_tso_inner_packets: 0
tx2_tso_inner_bytes: 0
tx2_csum_partial: 0
tx2_csum_partial_inner: 0
tx2_added_vlan_packets: 0
tx2_nop: 0
tx2_csum_none: 0
tx2_stopped: 0
tx2_dropped: 0
tx2_xmit_more: 0
tx2_recover: 0
tx2_cqes: 0
tx2_wake: 0
tx2_cqe_err: 0
...
```

17.11.4.1.4 Traffic Control Info

The following TC objects are supported and reported regarding the ingress filters:

- Filters
    - flower
- Actions
    - mirred
    - tunnel_key

The info is provided as one of the following events:

- Basic filter event
- Flower/IPv4 filter event
- Flower/IPv6 filter event
- Basic action event
- Mirred action event
- Tunnel_key/IPv4 action event
- Tunnel_key/IPv6 action event

General notes:

- Actions always belong to a filter, so action events share the filter event's ID via the `event_id` data member
- Basic filter event only contains textual *kind* (so users can see which real life objects' support they are lacking)
- Basic action event only contains textual *kind* and some basic common statistics if available

17.11.4.1.5 Amber Provider

Amber data for both InfiniBand and Ethernet MST devices in amBER format.

MST device names can be found under `/dev/mst/` .

`/dev/mst` should be accessible within DTS container.

The following config files are available:

```
amber_devices=DEV1,DEV2,DEV3   # Default:all, or set comma separated list of devices under /dev/mst
amber_update_interval_sec=30   # Sample rate for collection amber counters
```

17.11.4.1.6 PPCC_ETH Provider

Programmable congestion control counters are based on an algorithm defined by an end-user, although default algorithms are also available.

Counters are collected per MST device and algorithm parameters.

ⓘ MST device names can be found under `/dev/mst/` .

⚠ `/dev/mst` should be accessible within the DTS container.

The counter list depends on the installed MFT version.

⚠ `/usr/lib64/mft` or `/usr/lib/mft` should be mounted to the DTS container to get the counter list according to the installed MFT version. If not mounted, the internal DTS version of the counters is used.

A comma-separated list of device names is required to enable this provider:

```
ppcc_eth_devices=mt41692_pciconf0,mt41692_pciconf0.1
```

The following algorithm parameters are available:

```
ppcc_algo_slot=1
ppcc_algo_param_index=0
local_port=1
pnat=0
lp_msb=0
```

ⓘ For more details, consult the official PPCC documentation.

⚠ Some of the `algo_slots` are not implemented:

- If there are no counters to collect, the device is ignored
- If there are no devices to collect, the provider is disabled

17.11.4.1.7 Fluent Aggregator

`fluent_aggr` listens on a port for Fluent Bit Forward protocol input connections. Received data can be streamed via a Fluent Bit exporter.

The default port is 42442. This can be changed by updating the following option:

```
fluent-aggr-port=42442
```

17.11.4.1.8 Prometheus Aggregator

`prometheus_aggr` polls data from a list of Prometheus endpoints.

Each endpoint is listed in the following format:

```
prometheus_aggr_endpoint.{N}={host_name},{host_port_url},{poll_inteval_msec}
```

Where N starts from 0.

Aggregated data can be exported via a Prometheus Aggr Exporter endpoint.

17.11.4.1.9 Network Interfaces

`ifconfig` collects network interface data. To enable, set:

```
enable-provider=ifconfig
```

If the Prometheus endpoint is enabled, add the following configuration to cache every collected network interface and arrange the index according to their names:

```
prometheus-fset-indexes=name
```

Metrices are collected for each network interface as follows:

```
name
rx_packets
tx_packets
rx_bytes
tx_bytes
rx_errors
tx_errors
rx_dropped
tx_dropped
multicast
collisions
rx_length_errors
rx_over_errors
rx_crc_errors
rx_frame_errors
rx_fifo_errors
rx_missed_errors
tx_aborted_errors
tx_carrier_errors
tx_fifo_errors
tx_heartbeat_errors
tx_window_errors
rx_compressed
tx_compressed
rx_nohandler
```

17.11.4.1.10 HCA Performance

`hcaperf` collects HCA performance data. Since it requires access to an RDMA device, it must use remote collection on the DPU. On the host, the user runs the container in privileged mode and RDMA device mount.

The counter list is device dependent.

17.11.4.1.10.1 hcaperf DPU Configuration

To enable `hcaperf` in remote collection mode, set:

```
enable-provider=grpc.hcaperf

# specify HCAs to sample
grpc.hcaperf.mlx5_0=sample
grpc.hcaperf.mlx5_1=sample
```

⚠ DPE server should be active before changing the `dts_config.ini` file. See section "Remote Collection" for details.

17.11.4.1.10.2 hcaperf Host Configuration

To enable `hcaperf` in regular mode, set:

```
enable-provider=hcaperf

# specify HCAs to sample
hcaperf.mlx5_0=sample
hcaperf.mlx5_1=sample
```

17.11.4.1.11 NVIDIA System Management Interface

The `nvidia-smi` provider collects GPU and GPU process information provided by the NVIDIA system management interface.

This provider is supported only on x86_64 hosts with installed GPUs. All GPU cards supported by `nvidia-smi` are supported by this provider.

The counter list is GPU dependent. Additionally, per-process information is collected for the first 20 (by default) `nvidia_smi_max_processes` processes.

Counters can be either collected as string data "as is" in `nvidia-smi` or converted to numbers when `nvsmi_with_numeric_fields` is set.

To enable `nvidia-smi` provider and change parameters, set:

```
enable-provider=nvidia-smi

# Optional parameters:
#nvidia_smi_max_processes=20
#nvsmi_with_numeric_fields=1
```

17.11.4.1.12 NVIDIA Data Center GPU Manager

The `dcgm` provider collects GPU information provided by the NVIDIA data center GPU manager (DCGM) API.

This provider is supported only on x86_64 hosts with installed GPUs, and requires running the `nv-hostengine` service (refer to DCGM documentation for details).

DCGM counters are split into several groups by context:

- GPU - basic GPU information (always)
- COMMON – common fields that can be collected from all devices
- PROF – profiling fields
- ECC - ECC errors
- NVLINK / NVSWITCH / VGPU – fields depending on the device type

To enable DCGM provider and counter groups, set:

```
enable-provider=dcgm

dcgm_events_enable_common_fields=1
#dcgm_events_enable_prof_fields=0
#dcgm_events_enable_ecc_fields=0
#dcgm_events_enable_nvlink_fields=0
#dcgm_events_enable_nvswitch_fields=0
```

```
#dcgm_events_enable_vgpu_fields=0
```

17.11.4.1.13 BlueField Performance

The `bfperf` provider collects calculated performance counters of BlueField Arm cores. It requires the executable `bfperf_pmc`, which is integrated in the DOCA BFB bundle of BlueField-3, as well as an active DPE.

To enable BlueField performance provider, set:

```
enable-provider=bfperf
```

> ⚠ When running, the `bfperf` provider is expected to recurrently reset the counters of the `sysfs.hwmon` component. Consider disabling it if `bfperf` is enabled.

17.11.4.1.14 Ngauge

Ngauge is comprised of two providers which gather diagnostic data counters from network interface cards (NICs). These providers support the same counters (as defined in a YAML file), but they differ in usage and collection frequency:

- Low frequency provider is defined in `dts_config.ini` and is controlled by DTS collection loop
- High frequency provider is defined in `dts_high_freq_config.ini` and operates in a distinct flow for a limited duration

The `fwctl` and `mlx5_fwctl` drivers (supported on NVIDIA networking devices from BlueField-3 and ConnectX-7 and onward) are required for firmware interaction, and are part of MLNX_OFED driver. To load them, run:

```
modprobe -a fwctl mlx5_fwctl
```

Both providers get the counter set from a YAML file.

17.11.4.1.14.1 Ngauge Low Frequency

To enable the Ngauge low frequency provider, set:

```
enable-provider=ngauge_low_freq
```

To verify that the YAML file name matches the connected NIC's type:

```
ngauge-yml-file=/config/ngauge_configs/all-single-port.yml
```

To configure the Ngauge timestamp collection type, set the following:

```
ngauge-timestamp-collection-type=<method>
```

Where `<method>` can be one of the following:

- `no_counters` - Do not collect timestamp counters. Default.
- `start_and_end` - Collect sample start and end timestamps
- `per_counter` - Collect every counter collection timestamp

To configure the clock firmware should use when collecting time stamps, set the following:

```
ngauge-timestamp-source=<clock>
```

Where `<clock>` can be one of the following:

- `RTC` - Real-time clock. Default.
- `RFC` - Free-running clock

17.11.4.1.14.2 Ngauge High Frequency

This provider is designed to support higher sampling frequencies with sub-millisecond resolution. Due to the large scale of the collected data, this provider is aimed to run ad-hoc, for a limited time period, unlike the usual DTS providers which are configured with the DTS configuration file `/opt/mellanox/doca/services/telemetry/config/dts_config.ini`.

If the DTS standard flow constitutes an endless collect-export loop, then High Frequency Telemetry (HFT) is an additional external flow designed for the Ngauge high-frequency provider, based on the HFT configuration, located in `/opt/mellanox/doca/services/telemetry/config/dts_high_freq_config.ini`. This file defines the HFT session timing parameters, provider settings, and export settings. This means that an HFT session can export to different endpoints and/or protocols than those DTS used in the standard collection loop.The standard DTS configuration file references the HFT configuration file, enabling DTS to monitor the file's status. The HFT configuration file is also the trigger for the HFT session. That is, when the HFT configuration file is modified, the current HFT session is removed, and a new HFT session is configured (if defined). Removing the HFT configuration file stops pending sessions.

Required HFT Parameters

This table provides the details of the required HFT parameters. Refer to section "HFT Configuration File Example" for more helpful tips.

Option	Description
`start-time`	HFT session start time. If not used, the session starts immediately. UTC epoch timestamp (in microseconds). Syntax: HH:MM:SS / HH:MM
`end-time`	HFT session end time. Ignored if `start-time` is missing. If not used, `end-time` is calculated using `num-iterations`. UTC epoch timestamp (in microseconds). Syntax: HH:MM:SS / HH:MM
`num-iterations`	Number of iterations. If not used, `start-time` and `end-time` are required, and the number of iterations is calculated.

Option	Description
`sample-time-us`	Time interval between iterations (in microseconds)
`provider`	Provider to use. Should be `ngauge_high_freq`.
`file-write`	Whether to write collected telemetry to files. If enabled, could potentially write several MB of data every second.
`data-root`	Root folder for file writing. Ignored if `file-write=false`.
`provider.ngauge-num-samples`	Number of samples to collect in one iteration. Affects the buffer used by the firmware for diagnostic data.
`provider.ngauge-sample-period`	Sample period between samples (in nanoseconds). This option specifies the sample interval per iteration, as the provider collects N samples during each iteration.
`provider.ngauge-yml-file`	The Ngauge counters YAML file to use

17.11.4.1.14.3 Provider Compatibility

Both low and high frequency providers can run concurrently. The low frequency provider samples at the DTS standard frequency (defined in `dts_config.ini`), and the high frequency provider samples counters based on the HFT configuration file (`dts_high_freq_config.ini`).

To allow both providers to run concurrently, verify that the counters, the timestamp collection type, and the timestamp collection source are identical. Otherwise, when the high frequency provider starts sampling, the low frequency provider hangs until the end of the HFT session.

HFT Configuration File Example

```
## DTS configuration file for ad-hoc high frequency collection
## When modified, the file is parsed and applied.
## Note that the folders path is the container path, not the host path.

## Each section defines a collection. A file may have several sections, each one defines a high frequency
collection.
## Section names must be unique and will be used as collection name by clx.
[hft-collection-session]

### Time between samples in microseconds
sample-time-us=100000

### Start time of high frequency collection. Can be in the format HH:MM:SS or HH:MM or as epoch timestamp in
microseconds
### Note - in container, the time is in UTC
start-time=18:00:00

### End time of high frequency collection. Can be in the format HH:MM:SS or HH:MM or as epoch timestamp in
microseconds
### Note - in container, the time is in UTC
end-time=18:01:00
### Alternatively, you can set the number of iterations. This and start_time field will determine the end time
#num-iterations=300
  ### Data provider to use
provider=ngauge_high_freq

### Write data to file system. Could potentially fill up the disk
file-write=false

### Root directory to store the data
# Ignored if file-write is set to false
data-root=/data

### Enable busy wait between iterations, for a more accurate sample time (default is false)
#busy-wait-sampling=true

### Set prometheus endpoint to enable http endpoint
#prometheus-endpoint=http://0.0.0.0:9112

### Set fluentbit config dir to enable fluentbit export
```

```
#fluentbit-config-dir=/config/fluent_bit_configs

### Set open telemetry receiver to enable open telemetry export
#open-telemetry-receiver=http://0.0.0.0:9502/v1/metrics

### Set remote write receiver to enable remote write export
#remote-write-receiver=http://0.0.0.0:9090/api/v1/write

### Provider specific parameters. Format is 'provider.$KEY=$VALUE'.
### The options below are specific to the ngauge high frequency provider

# Number of samples to collect on each iteration
provider.ngauge-num-samples=1000

# The time period (in nanoseconds) between samples
provider.ngauge-sample-period-nsec=1000

# The YAML file with the configuration for the ngauge provider
provider.ngauge-yml-file=/config/ngauge_configs/all-dual-port.yml

# Ngauge timestamp collection type. Options are ['no_counters', 'start_and_end', 'per_counter']. default:
'no_counters'
#provider.ngauge-timestamp-collection-type=start_and_end

# Ngauge timestamp source. Options are ['RTC', 'FRC']. default: 'RTC'
#provider.ngauge-timestamp-source=FRC
```

17.11.4.1.14.4 Ngauge YAML File

For Ngauge compatibility, the counter set is defined in a YAML file.

There are 4 existing YAML files within a DTS container (one per permutation of BlueField-3 and ConnectX-7 with dual or single ports). The path to the YAMLs folder is `/opt/mellanox/doca/services/telemetry/config/ngauge_configs` which is mounted to `/config/ngauge_configs` .

By default, YAML files include a counter set that is not device-specific. This implies that the same counter set is utilized across all devices by default.

It is possible to assign a specific device within a YAML file; however, this requires maintaining a separate copy of the YAML file for each device. To manage multiple devices, use the `ngauge-yml-dir` option to specify a directory for YAML files, where each `.yml` / `.yaml` file is utilized. This folder should be available to the container under `/opt/mellanox/doca/services/telemetry/config` .

The following list describes the expected entries in the YAML file:

- `counters` - sequence of counters to collect
  - `id` - counter data ID
  - `desc` - counter description (optional)
  - `unit` - name of unit to collect from (optional)
  - `name` - name of counter to use (optional). If not specified, the generated name is based on the counter description. Otherwise, it is based on the data ID.
- `device` - name of the mlx device to collect (optional). If not used, the provider requires a single file containing a list of counters, which it then applies to all available devices on the host.

YAML File Example

The following is the default `all-dual-port.yml` provided in DTS:

```
counters:
  - id: 0x1020000100000000
    desc: RX bytes port 0
    unit: RX port
  - id: 0x1020000100000001
    desc: RX bytes port 1
    unit: RX port
  - id: 0x1020000300000000
```

```
    desc: RX packets port 0
    unit: RX port
  - id: 0x1020000300000001
    desc: RX packets port 1
    unit: RX port
  - id: 0x1140000100000000
    desc: TX bytes port 0
    unit: TX port
  - id: 0x1140000100000001
    desc: TX bytes port 1
    unit: TX port
  - id: 0x1140000300000000
    desc: TX packets port 0
    unit: TX port
  - id: 0x1140000300000001
    desc: TX packets port 1
    unit: TX port
  - id: 0x1100000100000000
    desc: CNP sent packets port 0
    unit: TX Transport
  - id: 0x1100000100000001
    desc: CNP sent packets port 1
    unit: TX Transport
  - id: 0x1080000400000000
    desc: CNP handled packets port 0
    unit: RX Transport
  - id: 0x1080000400000001
    desc: CNP handled packets port 1
    unit: RX Transport
  - id: 0x1080000500000000
    desc: ECN RoCE packets port 0
    unit: RX Transport
  - id: 0x1080000500000001
    desc: ECN RoCE packets port 1
    unit: RX Transport
  - id: 0x1160000b00000000
    desc: PCIe link latency total read ns
    unit: PCIe
    cutoff_min: 1
    cutoff_max: 2e6
  - id: 0x1160000c00000000
    desc: PCIe link latency total read packets
    unit: PCIe
    cutoff_min: 1
    cutoff_max: 3000
  - id: 0x1160000d00000000
    desc: PCIe link latency max read ns
    unit: PCIe
    cutoff_min: 1
    cutoff_max: 3000
  - id: 0x1160000e00000000
    desc: PCIe link latency min read ns
    unit: PCIe
    cutoff_min: 1
    cutoff_max: 3000
```

The NVIDIA Adapters Programmer's Reference Manual (PRM) "Diagnostic Data" section defines the rules for data IDs.

17.11.4.1.14.5 Counters

The following counters are available from the DTS default YAML files (and correspond the YAML file example):

```
cnp_handled_packets_port_0
cnp_handled_packets_port_1
cnp_sent_packets_port_0
cnp_sent_packets_port_1
ecn_roce_packets_port_0
ecn_roce_packets_port_1
pcie_link_latency_max_read_ns
pcie_link_latency_min_read_ns
pcie_link_latency_total_read_ns
pcie_link_latency_total_read_packets
rx_bytes_port_0
rx_bytes_port_1
rx_packets_port_0
rx_packets_port_1
tx_bytes_port_0
tx_bytes_port_1
tx_packets_port_0
tx_packets_port_1
```

17.11.4.2 Data Outputs

DTS can send the collected data to the following outputs:

- Data writer (saves binary data to disk)
- Fluent Bit (push-model streaming)
- Prometheus endpoint (keeps the most recent data to be pulled)

17.11.4.2.1 Data Writer

The data writer is disabled by default to save space on BlueField. Steps for activating data write during debug can be found under section Enabling Data Write.

The schema folder contains JSON-formatted metadata files which allow reading the binary files containing the actual data. The binary files are written according to the naming convention shown in the following example (`apt install tree`):

```
tree /opt/mellanox/doca/services/telemetry/data/
/opt/mellanox/doca/services/telemetry/data/
    {year}
        {mmdd}
             {hash}
                  {source_id}
                      {source_tag}{timestamp}.bin
                  {another_source_id}
                       {another_source_tag}{timestamp}.bin
    schema
        schema_{MD5_digest}.json
```

New binary files appears when the service starts or when binary file age/size restriction is reached. If no schema or no data folders are present, refer to the Troubleshooting section.

> `source_id` is usually set to the machine hostname. `source_tag` is a line describing the collected counters, and it is often set as the provider's name or name of user-counters.

Reading the binary data can be done from within the DTS container using the following command:

```
crictl exec -it <Container ID> /opt/mellanox/collectx/bin/clx_read -s /data/schema /data/path/to/datafile.bin
```

> The path to the data file must be an absolute path.

Example output:

```
{
    "timestamp": 1634815738799728,
    "event_number": 0,
    "iter_num": 0,
    "string_number": 0,
    "example_string": "example_str_1"
}
{
    "timestamp": 1634815738799768,
    "event_number": 1,
    "iter_num": 0,
    "string_number": 1,
    "example_string": "example_str_2"
}
…
```

17.11.4.2.2 Prometheus

The Prometheus endpoint keeps the most recent data to be pulled by the Prometheus server and is enabled by default.

To check that data is available, run the following command on BlueField:

```
curl -s http://0.0.0.0:9100/metrics
```

The command dumps every counter in the following format:

```
counter_name {list of meta fields} counter_value timestamp
```

Additionally, endpoint supports JSON and CSV formats:

```
curl -s http://0.0.0.0:9100/json/metrics
curl -s http://0.0.0.0:9100/csv/metrics
```

⚠ The default port for Prometheus can be changed in `dts_config.ini`.

17.11.4.2.3 Configuration Details

Prometheus is configured as a part of `dts_config.ini`.

By default, the Prometheus HTTP endpoint is set to port 9100. Comment this line out to disable Prometheus export.

```
prometheus=http://0.0.0.0:9100
```

Prometheus can use the data field as an index to keep several data records with different index values. Index fields are added to Prometheus labels.

```
# Comma-separated counter set description for Prometheus indexing:
#prometheus-indexes=idx1,idx2

# Comma-separated fieldset description for prometheus indexing
#prometheus-fset-indexes=idx1,idx2
```

The default `fset` index is `device_name`. It allows Prometheus to keep ethtool data up for both the `p0` and `p1` devices.

```
prometheus-fset-indexes=device_name
```

If `fset` index is not set, the data from `p1` overwrites `p0`'s data.

For quick name filtering, the Prometheus exporter supports being provided with a comma-separated list of counter names to be ignored:

```
#prometheus-ignore-names=counter_name1,counter_name_2
```

For quick filtering of data by tag, the Prometheus exporter supports being provided with a comma-separated list of data source tags to be ignored.

Users should add tags for all streaming data since the Prometheus exporter cannot be used for streaming. By default, `FI_metrics` are disabled.

```
prometheus-ignore-tags=FI_metrics
```

17.11.4.2.4 Prometheus Aggregator Exporter

Prometheus aggregator exporter is an endpoint that keeps the latest aggregated data using `prometheus_aggr`.

This exporter labels data according to its source.

To enable this provider, users must set 2 parameters in `dts_config.ini`:

```
prometheus-aggr-exporter-host=0.0.0.0
prometheus-aggr-exporter-port=33333
```

17.11.4.2.5 Fluent Bit

Fluent Bit allows streaming to multiple destinations. Destinations are configured in `.exp` files that are documented in-place and can be found under:

```
/opt/mellanox/doca/services/telemetry/config/fluent_bit_configs
```

Fluent Bit allows exporting data via "Forward" protocol which connects to the Fluent Bit/FluentD instance on customer side.

Export can be enabled manually:

1. Uncomment the line with `fluent_bit_configs=…` in `dts_config.ini`.
2. Set `enable=1` in required `.exp` files for the desired plugins.
3. Additional configurations can be set according to instructions in the `.exp` file if needed.
4. Restart the DTS.
5. Set up receiving instance of Fluent Bit/FluentD if needed.
6. See the data on the receiving side.

Export file destinations are set by configuring `.exp` files or creating new ones. It is recommended to start by going over documented example files. Documented examples exist for the following supported plugins:

- forward
- file
- stdout
- kafka
- es (elastic search)
- influx

> ⚠ All `.exp` files are disabled by default if not configured by `initContainer` entry point through `.yaml` file.

> ⚠ To forward the data to several destinations, create several `forward_{num}.exp` files. Each of these files must have their own destination host and port.

17.11.4.2.5.1 Export File Configuration Details

Each export destination has the following fields:

- `name` - configuration name
- `plugin_name` - Fluent Bit plugin name
- `enable` - 1 or 0 values to enable/disable this destination
- `host` - the host for Fluent Bit plugin
- `port` - port for Fluent Bit plugin
- `msgpack_data_layout` - the msgpacked data format. Default is `flb_std`. The other option is custom. See section Msgpack Data Layout for details.
- `plugin_key=val` - key-value pairs of Fluent Bit plugin parameter (optional)
- `counterset` / `fieldset` - file paths (optional). See details in section Cset/Fset Filtering.
- `source_tag=source_tag1,source_tag2` - comma-separated list of data page source tags for filtering. The rest tags are filtered out during export. Event tags are event provider names. All counters can be enabled/disabled only simultaneously with a `counters` keyword.

> Use `#` to comment a configuration line.

17.11.4.2.5.2 Msgpack Data Layout

Data layout can be configured using `.exp` files by setting `msgpack_data_layout=layout`. There are two available layouts: Standard and Custom.

The standard `flb_std` data layout is an array of 2 fields:

- timestamp double value
- a plain dictionary (key-value pairs)

The standard layout is appropriate for all Fluent Bit plugins. For example:

```
[timestamp_val, {"timestamp"->ts_val, type=>"counters/events", "source"=>"source_val", "key_1"=>val_1,
"key_2"=>val_2,...}]
```

The custom data layout is a dictionary of meta-fields and counter fields. Values are placed into a separate plain dictionary. Custom data format can be dumped with `stdout_raw` output plugin of Fluent-Bit installed or can be forwarded with `forward` output plugin.

Counters example:

```
{"timestamp"=>timestamp_val, "type"=>"counters", "source"=>"source_val", "values"=> {"key_1"=>val_1,
"key_2"=>val_2,...}}
```

Events example:

```
{"timestamp"=>timestamp_val, "type"=>"events", "type_name"=>"type_name_val", "source"=>" source_val",
"values"=>{"key_1"=>val_1, "key_2"=>val_2,...}}
```

17.11.4.2.5.3 Cset/Fset Filtering

Each export file can optionally use one `cset` and one `fset` file to filter UFM telemetry counters and events data.

- `cset` contains tokens per line to filter data with `"type"="counters"`.
- `fset` contains several blocks started with the header line `[event_type_name]` and tokens under that header. An Fset file is used to filter data with `"type"="events"`.

> Event type names could be prefixed to apply the same tokens to all fitting types. For example, to filter all ethtool events, use `[ethtool_event_*]`.

If several tokens must be matched simultaneously, use `<tok1>+<tok2>+<tok3>`. Exclusive tokens are available as well. For example, the line `<tok1>+<tok2>-<tok3>-<tok4>` filters names that match both tok1 and tok2 and do not match tok3 or tok4.

The following are the details of writing `cset` files:

```
# Put tokens on separate lines
# Tokens are the actual name 'fragments' to be matched
# port$ # match names ending with token "port"
# ^port # match names starting with token "port"
# ^port$ # include name that is exact token "port
# port+xmit # match names that contain both tokens "port" and "xmit"
# port-support # match names that contain the token "port" and do not match the "-" token "support"
#
# Tip: To disable counter export put a single token line that fits nothing
```

The following are the details of writing `fset` files:

```
# Put your events here
# Usage:
#
# [type_name_1]
# tokens
# [type_name_2]
# tokens
# [type_name_3]
# tokens
# ...
# Tokens are the actual name 'fragments' to be matched
# port$ # match names ending with token "port"
# ^port # match names starting with token "port"
# ^port$ # include name that is exact token "port
# port+xmit # match names that contain both tokens "port" and "xmit"
# port-support # match names that contain the token "port" and do not match the "-" token "support"

# The next example will export all the "tc" events and all events with type prefix "ethtool_" "ethtool" are
filtered with token "port":
# [tc]
#
# [ethtool_*]
# packet

# To know which event type names are available check export and find field "type_name"=>"ethtool_event_p0"
# ...
# Corner cases:
# 1. Empty fset file will export all events.
# 2. Tokens written above/without [event_type] will be ignored.
# 3. If cannot open fset file, warning will be printed, all event types will be exported.
```

17.11.4.2.6 NetFlow Exporter

NetFlow exporter must be used when data is collected as NetFlow packets from the telemetry client applications. In this case, DOCA Telemetry Exporter NetFlow API sends NetFlow data packages to DTS via IPC. DTS uses NetFlow exporter to send data to the NetFlow collector (3rd party service).

To enable NetFlow exporter, set `netflow-collector-ip` and `netflow-collector-port` in `dts_config.ini`. `netflow-collector-ip` could be set either to IP or an address.

For additional information, refer to the `dts_config.ini` file.

17.11.5 DOCA Privileged Executer

DOCA Privileged Executer (DPE) is a daemon that allows specific DOCA services (DTS included) to access BlueField information that is otherwise inaccessible from a container due to technology limitations or permission granularity issues.

When enabled, DPE enriches the information collected by DTS. However, DTS can still be used if DPE is disabled (default).

17.11.5.1 DPE Usage

DPE is controlled by systemd, and can be used as follows:

- To check DPE status:

  ```
  sudo systemctl status dpe
  ```

- To start DPE:

  ```
  sudo systemctl start dpe
  ```

- To stop DPE:

  ```
  sudo systemctl stop dpe
  ```

DPE logs can be found in `/var/log/doca/telemetry/dpe.log`.

17.11.5.2 DPE Configuration File

DPE can be configured by the user. This section covers the syntax and implications of its configuration file.

> The DPU telemetry collected by DTS does not require for this configuration file to be used.

The DPE configuration file allows users to define the set of commands that DPE should support. This may be done by passing the `-f` option in the following line of `/etc/systemd/system/dpe.service`:

```
ExecStart=/opt/mellanox/doca/services/telemetry/dpe/bin/dpeserver -vvv
```

To use the configuration file:

```
ExecStart=/opt/mellanox/doca/services/telemetry/dpe/bin/dpeserver -vvv -f /path/to/dpe_config.ini
```

The configuration file supports the following sections:

- `[server]` - list of key=value lines for general server configuration. Allowed keys: `socket`.
- `[commands]` - list of bash command lines that are not using custom RegEx
- `[commands_regex]` - list of bash command lines that are using custom RegEx
- `[regex_macros]` - custom RegEx definitions used in the `commands_regex` section

Consider the following example configuration file:

```
[server]
socket=/tmp/dpe.sock

[commands]
hostname
cat /etc/os-release

[commands_regex]
crictl inspect $HEXA        # resolved as "crictl inspect [a-f0-9]+"
lspci $BDF                  # resolved as "lspci ([0-9a-f]{4}\:|)[0-9a-f]{2}\:[0-9a-f]{2}\.[0-9a-f]"

[regex_macros]
HEXA=[a-f0-9]+
BDF=([0-9a-f]{4}\:|)[0-9a-f]{2}\:[0-9a-f]{2}\.[0-9a-f]
```

> ⚠ DPE is shipped with a preconfigured file that matches the commands used by the standalone DTS version included in the same DOCA installation. The file is located in `/opt/mellanox/doca/services/telemetry/dpe/etc/dpe_config.ini`.

> ⚠ Using a DPE configuration file allows for a fine-grained control over the interface exposed by it to the rest of the DOCA services. However, even when using the pre-supplied configuration file mentioned above, one should remember that it has been configured to match a fixed DTS version. That is, replacing the standalone DTS version with a new one downloaded from NGC means that the used configuration file might not cover additional features added in the new DTS version.

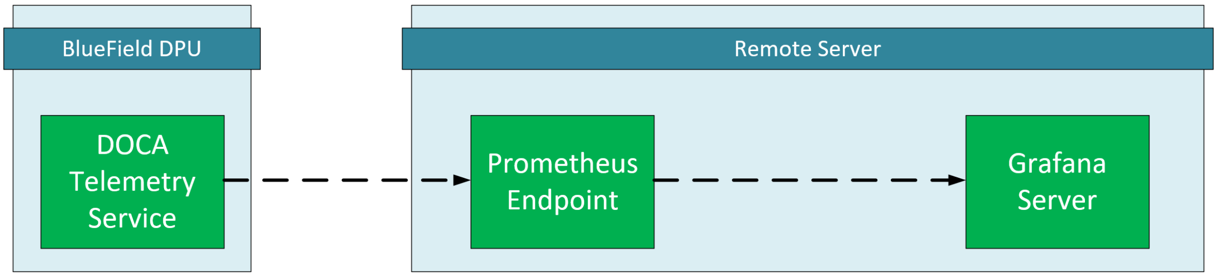
17.11.6 Deploying with Grafana Monitoring

This chapter provides an overview and deployment configuration of DOCA Telemetry Service with Grafana.

17.11.6.1 Grafana Deployment Prerequisites

- BlueField DPU running DOCA Telemetry Service.
- Optional remote server to host Grafana and Prometheus.
- Prometheus installed on the host machine. Please refer to the Prometheus website for more information.
- Grafana installed on the host machine. Please refer to Grafana Labs website for more information.

17.11.6.2 Grafana Deployment Configuration



17.11.6.2.1 DTS Configuration (DPU Side)

Configuring DTS to export the sysfs counter using the Prometheus plugin:

> Sysfs is used as an example, other counters are available.

1. Make sure the sysfs counter is enabled.

```
vim /opt/mellanox/doca/services/telemetry/config/dts_config.ini
enable-provider=sysfs
```

2. Enable Prometheus exporter by setting the `prometheus` address and port.

```
vim /opt/mellanox/doca/services/telemetry/config/dts_config.ini
prometheus=http://0.0.0.0:9100
```

> In this example, the Prometheus plugin exports data on localhost port 9100, this is an arbitrary value and can changed.

> DTS must be restarted to apply changes.

17.11.6.2.2 Prometheus Configuration (Remote Server)

Please download Prometheus for your platform.

Prometheus is configured via command-line flags and a configuration file, `prometheus.yml`.

1. Open the `prometheus.yml` file and configure the DPU as the endpoint target.

```
vim prometheus.yml
# metrics_path defaults to '/metrics'
# scheme defaults to 'http'.

static_configs:
- targets: ["<dpu-ip>:<prometheus-port>"]
```

Where:

- `<dpu-ip>` is the DPU IP address. Prometheus reaches to this IP to pull data.

    - `<prometheus-port>` the exporter port that set in DTS configuration.

2. Run Prometheus server:

```
./prometheus --config.file="prometheus.yml"
```

> ✓ Prometheus services are available as Docker images. Please refer to Using Docker in Prometheus' Installation guide.

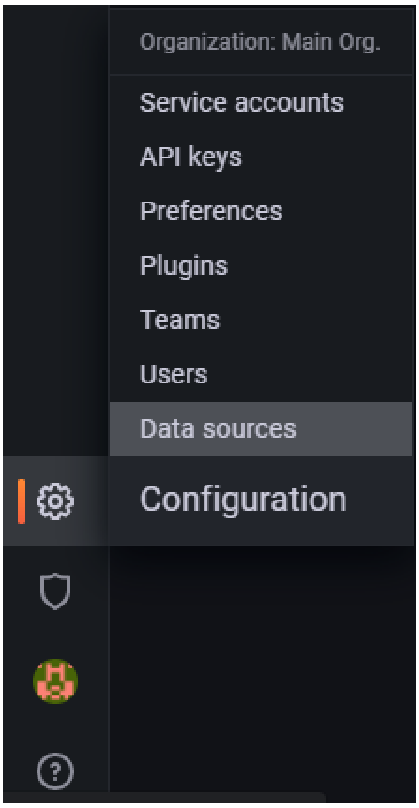
17.11.6.2.3 Grafana Configuration (Remote Server)

Please download and install Grafana for your platform.

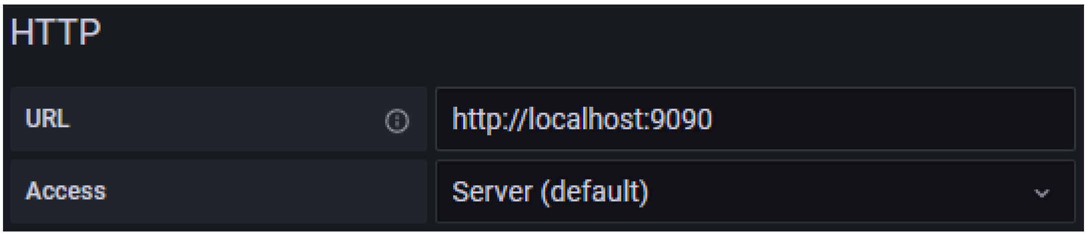
1. Setup Grafana. Please refer to Install Grafana guide in Grafana documentation.
2. Log into the Grafana dashboard at http://localhost:3000.

> ⚠ Port 3000 is the default port number set by Grafana. This can be changed if needed. The default credentials are admin/admin.

3. Add Prometheus as data source by navigating to Settings → Data sources → Add data source → Prometheus.



4. Configure the Prometheus data source. Under the HTTP section, set the Prometheus server address.



HTTP

URL: http://localhost:9090

Access: Server (default)

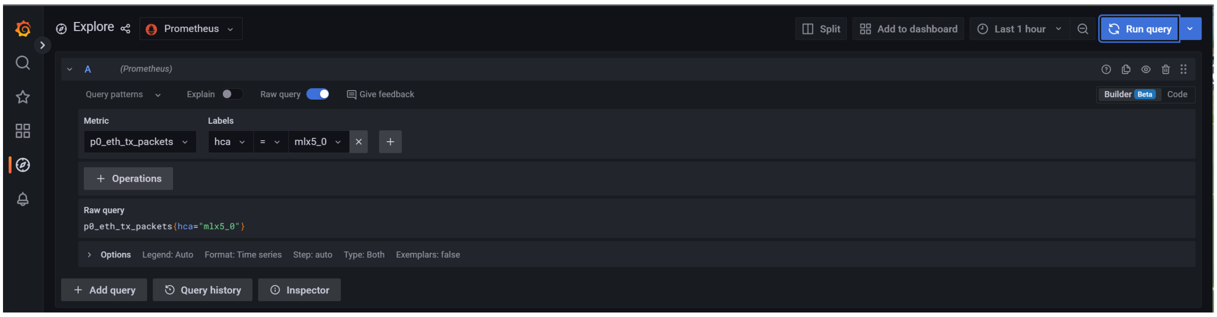
The Prometheus server's default listen port is 9090. Prometheus and Grafana are both running on the same server, thus the address is localhost.

5. Save and test.

17.11.6.3 Exploring Telemetry Data

Go to the Explore page on the left-hand side, and choose a Prometheus provider.

Choose a metric to display and specify a label. The label can be used to filter out data based on the source and HCA devices.



Metric: p0_eth_tx_packets

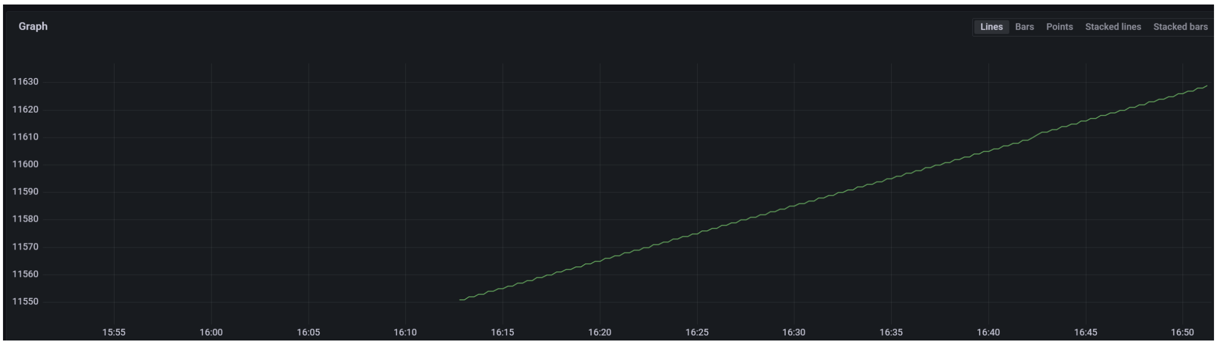
Labels: hca = mlx5_0

Raw query

```
p0_eth_tx_packets{hca="mlx5_0"}
```

Options Legend: Auto Format: Time series Step: auto Type: Both Exemplars: false

Graph display after selecting a metric and specifying a label to filter by:



17.11.7 Troubleshooting

On top of the Troubleshooting section in the NVIDIA DOCA Container Deployment Guide, here are additional troubleshooting tips for DTS:

- For general troubleshooting, refer to the NVIDIA DOCA Troubleshooting Guide.
- If the pod's state fails to be marked as "Ready", refer to `/var/log/syslog`.
- Check if the service is configured to write data to the disk as this may cause the system to run out of disk space.

- If a PIC bus error occurs, configure the following files inside the container:

```
crictl exec -it <container-id> /bin/bash
# Add to /config/clx.env the following line:
"
export UCX_TLS=tcp
"
```

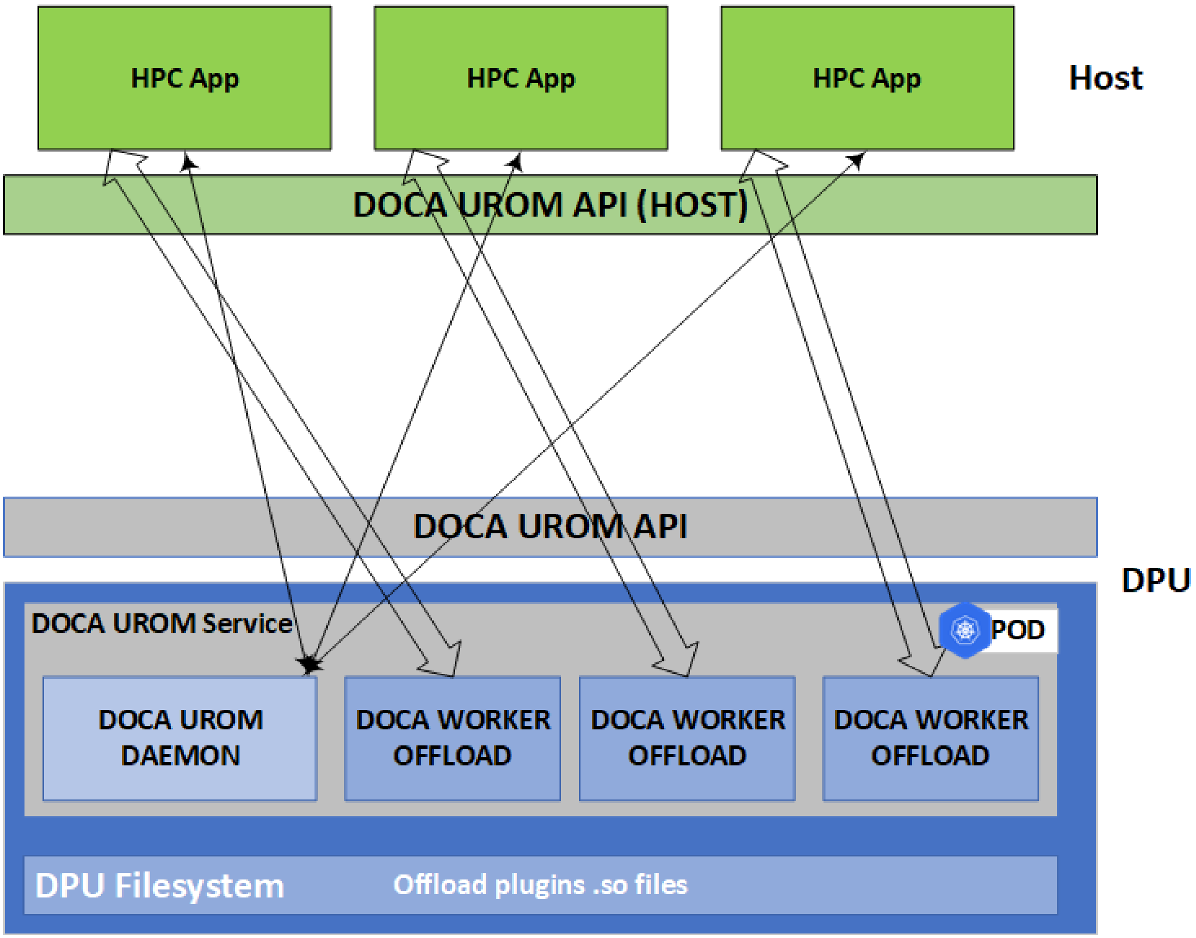
17.12 NVIDIA DOCA UROM Service Guide

This guide provides instructions on how to use the DOCA UROM Service on top of the NVIDIA® BlueField® networking platform.

17.12.1 Introduction

The DOCA UROM service provides a framework for offloading significant portions of HPC software stack directly from the host and to the BlueField device.

Using a daemon, the service handles the discovery of resources, the coordination between the host and BlueField, and the spawning, management, and teardown of the BlueField workers themselves.



The first step in initiating an offload request involves the UROM host application establishing a connection with the UROM service. Upon receiving the plugin discovery command, the UROM service responds by providing the application with a list of plugins available on the BlueField. The application then attaches the plugin IDs that correspond to the desired workers to their network identifiers. Finally, the service triggers UROM worker plugin instances on the BlueField to execute the parallel computing tasks. Within the service's Kubernetes pod, workers are spawned by the daemon in response to these offload requests. Each computation can utilize either a single library or multiple computational libraries.

17.12.2 Requirements

Before deploying the UROM service container, ensure that the following prerequisites are satisfied:

- Allocate huge pages as needed by DOCA (this requires root privileges):

  ```
  $ sudo echo 2048 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
  ```

- Or alternatively:

  ```
  $ sudo echo '2048' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
  $ sudo mkdir /mnt/huge
  $ sudo mount -t hugetlbfs nodev /mnt/huge
  ```

17.12.3 Service Deployment

For information about the deployment of DOCA containers on top of the BlueField, refer to the NVIDIA BlueField Container Deployment Guide.

Service-specific configuration steps and deployment instructions can be found under the service's container page.

17.12.4 Description

17.12.4.1 Plugin Discovery and Reporting

When the application initiates a connection request to the DOCA UROM Service, the daemon reads the `UROM_PLUGIN_PATH` environment variable. This variable stores directory paths to `.so` files for the plugins with multiple paths separated by semicolons. The daemon scans these paths sequentially and tries loading each `.so` file. Once the daemon finishes the scan, it reports the available BlueField plugins to the host application.

The host application gets the list of available plugins as a list of `doca_urom_service_plugin_info` structures:

```
struct doca_urom_service_plugin_info {
    uint64_t id;        // Unique ID to send commands to the plugin
    uint64_t version;   // Plugin version
    char plugin_name[DOCA_UROM_PLUGIN_NAME_MAX_LEN]; // .so filename
};
```

The UROM daemon is responsible for generating unique identifiers for the plugins, which are necessary to enable the worker to distinguish between different plugin tasks.

17.12.4.2 Loading Plugin in Worker

During the spawning of UROM workers by the UROM daemon, the daemon attaches a list of desired plugins in the worker command line. Each plugin is passed in a format of `so_path:id`.

As part of worker bootstrapping, the flow iterates all `.so` files and tries to load them by using `dlopen` system call and look for `urom_plugin_get_iface()` symbol to get the plugin operations interface.

17.12.4.3 Yaml File

The `.yaml` file downloaded from NGC can be easily edited according to users' needs:

```
 env:
      # Service-Specific command line arguments
      - name: SERVICE_ARGS
        value: "-l 60 -m 4096"
      - name: UROM_PLUGIN_PATH
        value: "/opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox/;/opt/mellanox/doca/samples/doca_urom/
plugins/worker_graph/"
```

- The `SERVICE_ARGS` are the runtime arguments received by the service:
    - `-l`, `--log-level <value>` - sets the (numeric) log level for the program `<10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>`
    - `--sdk-log-level` - sets the SDK (numeric) log level for the program `<10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>`
    - `-m`, `--max-msg-size` - specify UROM communication channel maximum message size
- The `UROM_PLUGIN_PATH` is an env variable that stores directory paths to `.so` files for the plugins

For each plugin on the BlueField, it is necessary to add a volume mount inside the service container. For example:

```
volumes:
  - name: urom-sandbox-plugin
    hostPath:
      path: /opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox
      type: DirectoryOrCreate
...
volumeMounts:
    - mountPath: /opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox
      name: urom-sandbox-plugin
```

17.12.5 Troubleshooting

When troubleshooting a container deployment issues, it is highly recommended to follow the deployment steps and tips found in the "Review Container Deployment" section of the NVIDIA BlueField Container Deployment Guide.

One could also check the `/var/log/doca/urom` log files for more details about the running cycles of service components (daemon and workers).

The log file name for workers is `urom_worker_<pid>_dev.log` and for the daemon it is `urom_daemon_dev.log`.

17.12.5.1 Pod is Marked as "Ready" and No Container is Listed

17.12.5.1.1 Error

When deploying the container, the pod's STATE is marked as `Ready` and an image is listed, however, no container can be seen running:

```
$ sudo crictl pods
POD ID              CREATED             STATE               NAME
NAMESPACE           ATTEMPT             RUNTIME
3162b71e67677       4 seconds ago       Ready               doca-urom-my-dpu                                default
0                  (default)

$ sudo crictl images
IMAGE                                     TAG                 IMAGE ID            SIZE
k8s.gcr.io/pause                          3.2                 2a060e2e7101d       487kB
nvcr.io/nvidia/doca/doca_urom             1.0.0-doca2.7.0     2af1e539eb7ab       86.8MB

$ sudo crictl ps
CONTAINER           IMAGE               CREATED             STATE               NAME                     ATTEMPT
POD ID              POD
```

17.12.5.1.2 Solution

In most cases, the container did start but immediately exited. This could be checked using the following command:

```
$ sudo crictl ps -a
CONTAINER           IMAGE               CREATED             STATE               NAME                     ATTEMPT
POD ID              POD
556bb78281e1d       2af1e539eb7ab       6 seconds ago       Exited              doca-urom                1
          3162b71e67677       doca-urom-my-dpu
```

Should the container fail (i.e., reporting a state of `Exited`), it is recommended to examine the UROM's main log at `/var/log/doca/urom/urom_daemon_dev.log`.

In addition, for a short period of time after termination, the container logs could also be viewed using the container's ID:

```
$ sudo crictl logs 556bb78281e1d
...
```

17.12.5.2 Pod is Not Listed

17.12.5.2.1 Error

When placing the container's YAML file in the Kubelet's input folder, the service pod is not listed in the list of pods:

```
$ sudo crictl pods
POD ID              CREATED             STATE               NAME
NAMESPACE           ATTEMPT             RUNTIME
```

17.12.5.2.2 Solution

In most cases, the pod has not started because of the absence of the requested hugepages. This can be verified using the following command:

```
$ sudo journalctl -u kubelet -e. . .
Oct 04 12:12:19 <my-dpu> kubelet[2442376]: I1004 12:12:19.905064 2442376 predicate.go:103] "Failed to admit pod,
unexpected error while attempting to recover from admission failure" pod="default/doca-urom-service-<my-dpu>" err="
preemption: error finding a set of pods to preempt: no set of running pods found to reclaim resources: [(res:
hugepages-2Mi, q: 104563999874), ]"
```

17.13 NVIDIA DOCA SNAP Virtio-fs Service Guide

This guide provides instructions on using the DOCA SNAP Virtio-fs service on top of the NVIDIA® BlueField®-3 DPU.

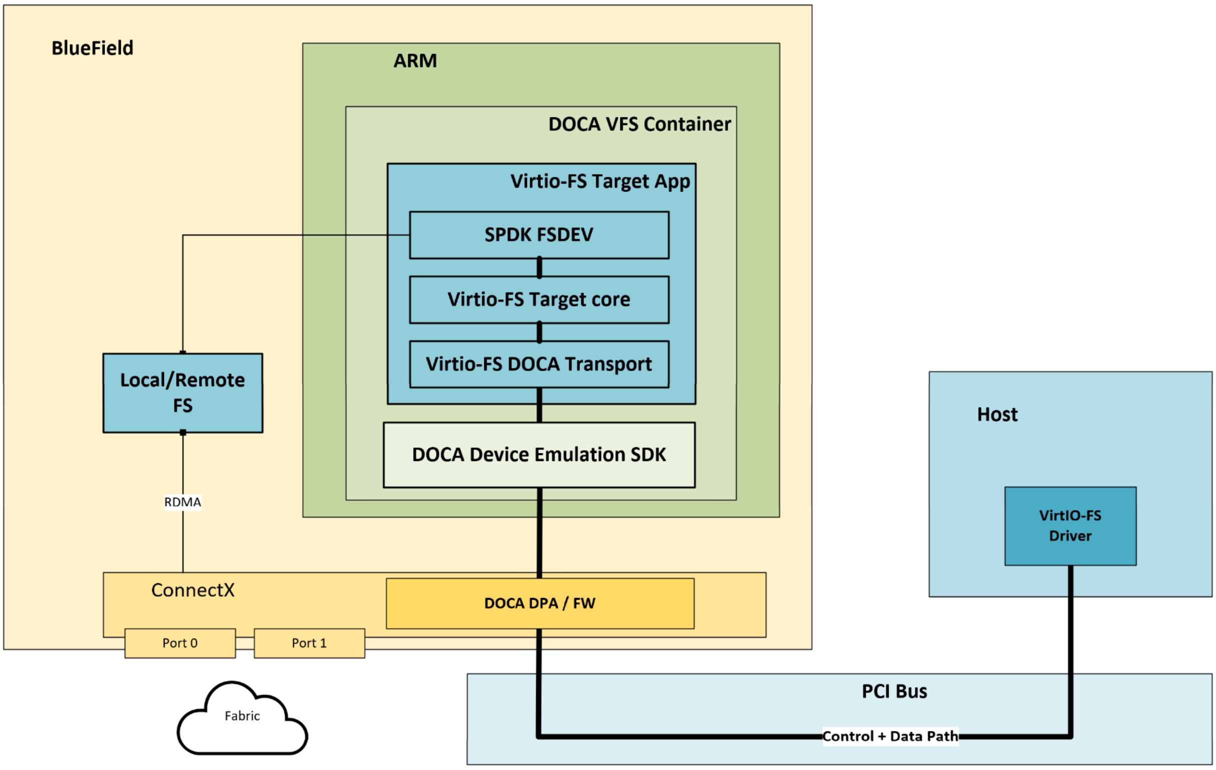
17.13.1 Introduction

⚠ The DOCA SNAP Virtio-fs Service is currently supported at alpha level.

NVIDIA® BlueField® enables hardware-accelerated software-defined virtio-fs PCIe device emulation. This leverages the power of BlueField networking platforms (DPUs or SuperNICs) to provide high-performance file system access in bare-metal and virtualized environments. Using BlueField, users can offload and accelerate networked file system operations from the host/guest, freeing up resources for other tasks and improving overall system efficiency. In this solution, the host/guest uses its own standard virtio-fs driver which is fully isolated from the networked filesystem mounted within the BlueField.

Built upon the DOCA and SPDK frameworks, virtio-fs device emulation on BlueField devices offers a comprehensive set of libraries for BlueField-based solutions and for storage solutions. This architecture consists of several key components:

- DOCA DevEmu subsystem and DOCA Virtio-fs library – These core libraries are responsible for the low-level hardware management and the translation of virtio descriptors carrying FUSE (filesystem in userspace) requests into abstract virtio-fs requests, which are then processed by the SPDK virtio-fs DOCA transport component.
- SPDK virtio-fs transport – This component is responsible for the interaction with the low-level DOCA components and translating the incoming abstract DOCA SNAP Virtio-fs requests into generic virtio-fs request which are then processed by the virtio-fs target core.
- SPDK virtio-fs target – This component implements and manages the virtio-fs device, transports, and the interface with a backend file system. Upon arrival on a new generic virtio-fs request from the transport, it processes and translates the requests according to virtio-fs and FUSE specifications, translating FUSE-based commands into the generic filesystem protocol.
- SPDK FSdev – This component provides generic filesystem abstraction and interfaces with the low-level filesystem modules implementing a specific backend filesystem protocol.



17.13.1.1 DOCA SNAP Virtio-fs as Container

The DOCA SNAP Virtio-fs container image may be downloaded from NVIDIA NGC and easily deployed on the BlueField using a YAML file. The YAML file points to the docker image that includes DOCA SNAP Virtio-fs binaries aligned with the latest `spdk.nvda` version.

DOCA SNAP Virtio-fs is not pre-installed on the BFB but can be downloaded manually on demand. For instructions on how to install the DOCA SNAP Virtio-fs container, refer to section "DOCA SNAP Virtio-fs Container Deployment".

17.13.2 Release Notes

The release notes provide information for the DOCA SNAP Virtio-fs Service such as changes and new features, software known issues, and bug fixes.

17.13.2.1 Changes and New Features

17.13.2.1.1 Key Features in Version 1.0.0-doca2.8.0

- NVIDIA® BlueField®-3 support
- Virtio-fs emulation
- Virtio-fs hotplug emulation support
- Container support

17.13.2.2 Limitations

The following features are currently not supported and are still under development in this version of the application:

- Crash Recovery – the ability to handle device recovery.
- Live Update
- Live Migration
- Dynamic MSIX
- 254 Queues per Emulation Function: (Currently, only 62 queues per emulation function are supported)
- Build Custom Container - SDK: The ability to build custom container via the SDK is not yet available.

17.13.2.3 Known Issues

17.13.2.3.1 DOCA SNAP Virtio-fs Issues

The following are known limitations of DOCA SNAP Virtio-fs software version.

Ref #	Issue
-	Description: The following FUSE commands are currently unsupported: GETLK, SETLK if FUSE_LK_FLOCK not set, SETLKW, ACCESS, BMAP, IOCTL, POLL, SETUPMAPPING, REMOVEMAPPING, DAX, and SYNCFS.
	Workaround: N/A
	Keywords: FUSE
	Discovered in version: 1.0.0-doca2.8.0
-	Description: App or controller restart is not allowed if the controller has processed FUSE commands.
	Workaround: Unload the virtio-fs driver on the host, then restart the app or controller.
	Keywords: FUSE
	Discovered in version: 1.0.0-doca2.8.0
-	Description: Currently, only a single hotplug function is supported.
	Workaround: N/A
	Keywords: Hotplug
	Discovered in version: 1.0.0-doca2.8.0
-	Description: Currently, the only supported protocol is NFS-over-TCP. NFS-over-RDMA is not supported.
	Workaround: N/A
	Keywords: NFS-over-TCP; NFS-over-RDMA
	Discovered in version: 1.0.0-doca2.8.0
-	Description: Due to the lack of recovery support, it is not possible to perform any negative/resilience operations.

Ref #	Issue
	Workaround: N/A
	Keywords: Recovery; negative/resilience operations
	Discovered in version: 1.0.0-doca2.8.0

17.13.2.3.2 OS or Vendor Issues

The following are not DOCA SNAP Virtio-fs limitations.

Ref #	Issue
-	Description: After FLR, the virtio-fs driver does not create virt queues again, resulting in IO failures.
	Workaround: FLR should only be performed without any mount over virtio-fs on the host. To run IO after FLR, reload the virtio-fs host.
	Keywords: Driver; FLR
	Discovered in version: 1.0.0-doca2.8.0
-	Description: On the host, when the virtio-fs mount is idle (i.e., no I/O operations), dmesg logs are filled with repeated AppArmor DENIED messages. These messages indicate that the ntpd service is being denied access to specific files by AppArmor. The ntpd service is trying to access `/snap/bin/` and `/etc/ssl/openssl.cnf`, but the AppArmor profile for ntpd does not permit these accesses, resulting in denied requests.

	Workaround: Modify the AppArmor profile for ntpd to grant the required read permissions.
	1. Locate the AppArmor profile for ntpd. It is typically located in `/etc/apparmor.d/` and named `usr.sbin.ntpd`.
	2. Edit the profile and add the required permissions by using a text editor. For example:
	a. Run:
	`sudo nano /etc/apparmor.d/usr.sbin.ntpd`
	b. Add the following lines to allow ntpd to read the necessary files:
	`/snap/bin/ r,`
	`` /etc/ssl/openssl.cnf r` ``
	3. Apply the changes by reloading the AppArmor profile:
	`sudo apparmor_parser -r /etc/apparmor.d/usr.sbin.ntpd`

	Keywords: AppArmor, ntpd
	Discovered in version: 1.0.0-doca2.8.0
-	Description: With a kernel version older than 6.10, if loading and unloading of the `virtio_pci` and `virtiofs` drivers is done in a loop, unloading the driver may hang.

Ref #	Issue
	Workaround: Add a delay of 1 second between loading and unloading of the drivers.
	Keywords: virtio_pci; virtiofs
	Discovered in version: 1.0.0-doca2.8.0

17.13.3 DOCA SNAP Virtio-fs Deployment

This section describes how to deploy DOCA SNAP Virtio-fs as a container.

> DOCA SNAP Virtio-fs does not come pre-installed with the BFB bundle.

17.13.3.1 Installing Full DOCA Image on BlueField

To install the BFB on BlueField:

```
[host] sudo bfb-install --rshim <rshimN> --bfb <image_path.bfb>
```

For more information, please refer to section "Installing Full DOCA Image on DPU" in the NVIDIA DOCA Installation Guide for Linux.

17.13.3.2 Firmware Installation

```
[dpu] sudo /opt/mellanox/mlnx-fw-updater/mlnx_fw_updater.pl --force-fw-update
```

For more information, please refer to section "Upgrading Firmware" in the NVIDIA DOCA Installation Guide for Linux.

17.13.3.3 Firmware Configuration

> Firmware configuration may expose new emulated PCIe functions, which can be later used by the host's OS. As such, the user must make sure all exposed PCIe functions (static/hotplug) are backed by a supporting virtio-fs software configuration. Otherwise, these functions would malfunction and host behavior would be anomalous.

1. Clear the firmware config before implementing the required configuration:

   ```
   [dpu] mst start
   [dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 reset
   ```

2. Verify the firmware configuration:

   ```
   [dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 query
   ```

   Output example:

```
mlxconfig -d /dev/mst/mt41692_pciconf0 -e query | grep VIRTIO_FS
Configurations:                                      Default             Current             Next Boot
*        VIRTIO_FS_EMULATION_ENABLE                  False(0)            True(1)             True(1)
         VIRTIO_FS_EMULATION_NUM_VF                  0                   0                   0
*        VIRTIO_FS_EMULATION_NUM_PF                  0                   2                   2
         VIRTIO_FS_EMU_SUBSYSTEM_VENDOR_ID           6900                6900                6900
         VIRTIO_FS_EMULATION_SUBSYSTEM_ID            4186                4186                4186
*        VIRTIO_FS_EMULATION_NUM_MSIX                2                   3                   3
```

The output provides 5 columns (listed from left to right):

- Non-default configuration marker (`*`)
- Firmware configuration name
- Default firmware value
- Current firmware value
- Firmware value after reboot – shows configuration update pending system reboot

3. To enable storage emulation options, BlueField must be set to work in internal CPU model:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s INTERNAL_CPU_MODEL=1
```

4. To enable the firmware config with virtio-fs emulation PF:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s VIRTIO_FS_EMULATION_ENABLE=1 VIRTIO_FS_EMULATION_NUM_PF=1
VIRTIO_FS_EMULATION_NUM_MSIX=3
```

For a complete list of the DOCA SNAP Virtio-fs firmware configuration options, refer to "Appendix – BlueField Firmware Configuration".

Power cycle is required to apply firmware configuration changes.

17.13.3.3.1 RDMA/RoCE Firmware Configuration

RoCE communication is blocked for the default interfaces of BlueField OS's (named ECPFs), `mlx5_0` and `mlx5_1` typically. If RoCE traffic is required, scalable functions (or SFs) must be added which are network functions which support RoCE transport.

To enable RDMA/RoCE:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PER_PF_NUM_SF=1
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PF_SF_BAR_SIZE=8 PF_TOTAL_SF=2
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0.1 s PF_SF_BAR_SIZE=8 PF_TOTAL_SF=2
```

This is not required when working over TCP or RDMA over InfiniBand.

17.13.3.3.2 Hot-plug Firmware Configuration

When PCIe switch emulation is enabled, BlueField can support 1 hotplug virtio-fs function. These PCIe functions are shared among all BlueField users and applications and may hold hot-plugged devices of type NVMe, virtio-blk, virtio-fs, and more (e.g., virtio-net).

To enable PCIe switch emulation and configure 1 hot-plugged ports to be used, run:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PCI_SWITCH_EMULATION_ENABLE=1 PCI_SWITCH_EMULATION_NUM_PORT=2
```

`PCI_SWITCH_EMULATION_NUM_PORT` equals 1 plus the number of hot-plugged PCIe functions.

> ⚠ On AMD machines, hotplug is not guaranteed to work and enabling `PCI_SWITCH_EMULATION_ENABLE` may impact SR-IOV capabilities.

17.13.3.4 DOCA SNAP Virtio-fs Container Deployment

DOCA SNAP Virtio-fs container is available on the DOCA SNAP Virtio-fs NVIDIA™ NGC page.

To deploy DOCA SNAP Virtio-fs container on top of BlueField, the following procedure is required:

1. Setup preparation and DOCA SNAP Virtio-fs resource download for container deployment. See section "Preparation Steps" for details.
2. Adjust the `doca_vfs.yaml` for advanced configuration if needed according to section "Adjusting YAML Configuration".
3. Deploy the container. The image is automatically pulled from NGC. See section "Spawning DOCA SNAP Virtio-fs Container" for details.

17.13.3.4.1 Preparation Steps

17.13.3.4.1.1 Step 0: Connect to NGC Container Registry

The Early Adopters (EA) NGC is used to distribute the DOCA SNAP Virtio-fs container before it becomes available in the public NGC:

1. Use the welcome email that invites you to continue the activation and sign-in process.
2. Follow the instruction under section "Joining an Org or Team with a New NVIDIA Account" in the NGC Private Registry User Guide.
3. Under "[EA] DOCA", select "doca_vfs" and click "Continue".
4. In the top-right corner, click your user account icon and select "Setup".
5. Click "Get API key" to open the "Setup" → "API Key" page.
6. Click "Generate API Key" to generate your API key.
7. Click "Confirm" to generate the key.
8. Save the key for later usage.
9. Convert your API key to an auth token using the following command:

   ```
   echo -n '$oauthtoken:<your-api-key>' | base64 -w 128
   ```

10. Update `containerd`'s configuration file at `/etc/containerd/config.toml` by removing the comments from the following configuration lines and inserting your authentication token where indicated:

    ```
    # [plugins."io.containerd.grpc.v1.cri".registry.configs]
    # [plugins."io.containerd.grpc.v1.cri".registry.configs."nvcr.io".auth]
    #     auth = "<your-auth-token>"
    ```

11. Restart the `containerd` service:

```
[dpu] systemctl restart containerd.service
```

17.13.3.4.1.2 Step 1: Allocate Hugepages

Allocate `4GiB` hugepages for the DOCA SNAP Virtio-fs container according to the DPU OS's `Hugepagesize` value:

1. Query the `Hugepagesize` value:

   ```
   [dpu] grep Hugepagesize /proc/meminfo
   ```

   In Ubuntu, the value should be 2048KB.
2. Append the following line to the end of the `/etc/sysctl.conf` file:

   ```
   vm.nr_hugepages = 2048
   ```
3. Run the following:

   ```
   [dpu] sysctl --system
   ```

> ⚠ If live upgrade is utilized in this deployment, it is necessary to allocate twice the amount of resources listed above for the upgraded container.

> ❗ If other applications are running concurrently within the setup and are consuming hugepages, make sure to allocate a number of hugepages appropriate to accommodate all applications.

17.13.3.4.1.3 Step 2: Create /etc/virtiofs Folder

The folder `/etc/virtiofs` is used by the container for automatic configuration after deployment.

> ⚠ The default YAML configuration only mounts the `/etc/virtiofs` folder for exposure and sharing between the container and the BlueField. This folder is used to expose configuration files or local file backends (e.g., AIO FSdev) from the DPU to the container.

17.13.3.4.2 Downloading YAML from Early Access NGC

The `.yaml` configuration file for the DOCA SNAP Virtio-fs container, `doca_vfs.yaml`, is uploaded to EA DOCA NGC.

To download the file, in the top-left corner in NGC page, click "PRIVATE REGISTERY" → "Resources" → "DOCA SNAP Virtio-fs Container Resources" → "File Browser".

Download latest version file and move it to BlueField.

⚠ Internet connectivity is necessary to download DOCA SNAP Virtio-fs resources.

17.13.3.4.3 Adjusting YAML Configuration

The `.yaml` file can easily be edited for advanced configuration.

- The DOCA SNAP Virtio-fs `.yaml` file is configured by default to support Ubuntu setups (i.e., `Hugepagesize` = 2048 kB) by using hugepages-2Mi.
  To support other setups, edit the hugepages section according to the relevant `Hugepagesize` value for the BlueField OS. For example, to support CentOS 8.x configure `Hugepagesize` to 512MB:

  ```
   limits:
      hugepages-512Mi: "<number-of-hugepages>Gi"
  ```

- The following example edits the `.yaml` file to request 8 CPU cores for the DOCA SNAP Virtio-fs container:

  ```
  resources:
      cpu: "8"
    limits:
      cpu: "8"
  env:
    - name: APP_ARGS
      value: "-m 0xff"
  ```

  ⚠ If all BlueField-3 cores are requested, the user must verify no other containers are in conflict over CPU resources.

- To automatically configure the DOCA SNAP Virtio-fs container upon deployment:
  a. Add the `spdk_rpc_init.conf` file under `/etc/virtiofs/`. File example:

     ```
     fsdev_aio_create aio0 /etc/virtiofs/test
     virtio_fs_transport_create -t DOCA
     virtio_fs_transport_start -t DOCA
     virtio_fs_device_create --transport-name DOCA --dev-name vfsdev0 --tag docatag --fsdev aio0 --num-request-queues 1 --queue-size 32 --driver-platform x86_64
     virtio_fs_doca_device_modify --dev-name vfsdev0 --manager mlx5_0 --vuid "MT2251XZ02WZVFSS0D0F3"
     virtio_fs_device_start --dev-name vfsdev0
     ```

  b. Edit the `.yaml` file accordingly (uncomment):

     ```
     env:
       - name: SPDK_RPC_INIT_CONF
         value: "/etc/virtiofs/spdk_rpc_init.conf"
     ```

     ⚠ It is user responsibility to make sure DOCA SNAP Virtio-fs configuration matches firmware configuration. That is, an emulated controller must be opened on all existing (static/hotplug) emulated PCIe functions (either through automatic or manual configuration). A PCIe function without a supporting controller is considered malfunctioned, and host behavior with it is anomalous.

17.13.3.4.4 Spawning DOCA SNAP Virtio-fs Container

Run the Kubernetes tool:

```
[dpu] systemctl restart containerd
[dpu] systemctl restart kubelet
[dpu] systemctl enable kubelet
[dpu] systemctl enable containerd
```

Copy the updated `doca_vfs.yaml` file to the `/etc/kubelet.d` directory.

Kubelet automatically pulls the container image from NGC described in the YAML file and spawns a pod executing the container.

```
cp doca_vfs.yaml /etc/kubelet.d/
```

The DOCA SNAP Virtio-fs Service starts initialization immediately, which may take a few seconds.

To verify whether DOCA SNAP Virtio-fs is running, send `spdk_rpc.py spdk_get_version` to confirm whether DOCA SNAP Virtio-fs is operational or still initializing.

17.13.3.4.5 Debug and Log

Unable to render include or excerpt-include. Could not retrieve page.

17.13.3.4.6 Stop, Start, Restart DOCA SNAP Virtio-fs Container

- To stop the container, remove the `.yaml` file form `/etc/kubelet.d/`.
- To start the container, copy the `.yaml` file to the same path:

  ```
  cp doca_vfs.yaml /etc/kubelet.d
  ```

- To restart the container (with sig-term), use the `-t` (timeout) option:

  ```
  crictl stop -t 10 <container-id>
  ```

  General Kublet Comment

  After containers in a pod exit, the kubelet restarts them with an exponential back-off delay (10s, 20s, 40s, etc.) which is capped at five minutes. Once a container has run for 10 minutes without an issue, the kubelet resets the restart back-off timer for that container.

17.13.3.5 DOCA SNAP Virtio-fs with SNAP Support

The DOCA SNAP Virtio-fs container, along with associated packages, natively supports DOCA SNAP 4.x.x, which is implemented as an SPDK subsystem module, allowing the concurrent operation of both virtio-fs and virtio-blk as a unified service. Additionally, DOCA SNAP is deployed as part of the DOCA SNAP Virtio-fs deployment.

> Refer to NVIDIA BlueField-3 SNAP for NVMe and Virtio-blk documentation here.

17.13.4 RPC Commands

Like other standard SPDK applications, the remote procedure call (RPC) protocol is used to control the DOCA SNAP Virtio-fs Service and supports JSON-based RPC protocol commands to control any resources and create, delete, query, or modify commands easily from the CLI.

DOCA SNAP Virtio-fs supports all standard SPDK RPC commands in addition to an extended DOCA SNAP Virtio-fs-specific command set. Standard SPDK commands are executed by the `spdk_rpc.py` tool.

To invoke the extended DOCA SNAP Virtio-fs-specific command set, users must add the `--plugin rpc_virtio_fs_tgt` flag to the SPDK's `rpc.py` command. The SPDK RPC plugin `rpc_virtio_fs_tgt.py` is implemented as an RPC plugin. This flag is not needed when working with containers.

The following is an example of an RPC when using DOCA SNAP Virtio-fs from the source:

```
/opt/nvidia/spdk-subsystem/src/spdk/install-$(hostname)/bin/spdk_rpc --plugin spdk.rpc.rpc_virtio_fs_tgt --help
```

> Users may need to define the path to the virtio-fs-target folder using the `PYTHONPATH` environment variable. More details on the RPC plugins can be found in SPDK's official documentation.

> Full `spdk_rpc.py` command set documentation can be found in the SPDK official documentation site.

DOCA SNAP Virtio-fs extended commands are detailed in the following subsections.

17.13.4.1 Using JSON-based RPC Protocol

The JSON-based RPC protocol can be used with the `rpc.py` script inside the DOCA SNAP Virtio-fs container and `crictl` tool.

> The DOCA SNAP Virtio-fs container is CRI-compatible.

- To query the active container ID:

  ```
  crictl ps -s running -q --name virtiofs
  ```

- To post RPCs to the container using `crictl`:

```
crictl exec <container-id> spdk_rpc.py -v <RPC-method>
```

The flag `-v` controls verbosity. For example:

```
crictl exec 0379ac2c4f34c spdk_rpc.py -v virtio_fs_doca_get_functions
```

Alternatively, an alias can be used:

```
crictl exec -it $(crictl ps -s running -q --name virtiofs) spdk_rpc.py -v virtio_fs_doca_get_functions
```

- To open a bash shell to the container that can be used to post RPCs:

```
crictl exec -it <container-id> bash
```

17.13.4.2 PCIe Function Management

Emulated PCIe functions are managed through DOCA devices called emulation managers. Emulation managers have special privileges to control, manipulate, and expose the emulated PCIe devices towards the host PCIe subsystem.

To operate a virtio-fs device/function by the DOCA transport, it is necessary to locate the appropriate emulation manager for it. The emulation manager maintains a list of the emulated PCIe functions it controls. Each of those functions is assigned a globally unique serial called a vendor unique identifier or VUID (e.g., MT2251XZ02WZVFSS0D0F2), which serves as unambiguous reference for identification and tracking purposes.

Command	Description
virtio_fs_doca_get_managers	List emulation managers for virtio-fs
virtio_fs_doca_get_functions	List functions for virtio-fs

17.13.4.2.1 virtio_fs_doca_get_managers

List emulation managers for virtio-fs. This method has no input parameters.

Example response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": [
    {
      "name": "mlx5_0"
    }
  ]
}
```

17.13.4.2.2 virtio_fs_doca_get_functions

List functions for virtio-fs with their characteristics. The user may specify no parameters to list all emulated virtio-fs functions managed by any emulation manager device, or specify an emulation manager device name to list virtio-fs functions managed by that emulation manager device.

Example response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": [
    {
      "manager": "mlx5_0",
      "Function List": [
        {
          "hot pluggable": "false",
          "pci_address": "0000:29:00.2",
          "vuid": "MT2251XZ02WZVFSS0D0F2",
          "function_type": "PF"
        }
      ]
    }
  ]
}
```

Parameter Name	Optional/Mandatory	Type	Description
`manager`	Optional	String	Emulation manager device name to list emulated virtio-fs functions specific to it

17.13.4.3 Hot-pluggable PCIe Functions Management

Hotplug PCIe functions are configured dynamically at runtime using RPCs.

The commands outlined in the following subsections hot plug a new PCIe function to the system.

17.13.4.3.1 virtio_fs_doca_get_functions

List DOCA transport functions for virtio-fs with their characteristics.

Users may specify no parameters to list all emulated virtio-fs functions managed by any emulation manager device, or an emulation manager device name to list virtio-fs functions managed by a specific emulation manager device.

Parameter Name	Optional/Mandatory	Type	Description
`manager`	Mandatory	String	Emulation manager device name for creating a new Virtio FS function

17.13.4.3.2 virtio_fs_doca_function_create

Create a DOCA virtio FS function. The return value of this method is a VUID.

Parameter Name	Optional/Mandatory	Type	Description
`manager`	Mandatory	String	Emulation manager device name for creating a new virtio-fs function

17.13.4.3.3 virtio_fs_doca_function_destroy

Destroy a DOCA SNAP Virtio-fs function.

> This function should not be associated to any virtio-fs device.

Parameter Name	Optional/Mandatory	Type	Description
`manager`	Mandatory	String	Emulation manager device name for destroying a virtio-fs function
`vuid`	Mandatory	String	VUID of the function to destroy

17.13.4.3.4 virtio_fs_doca_device_hotplug

Hot plug a DOCA SNAP Virtio-fs device. The virtio-fs device must be started.

Parameter Name	Optional/Mandatory	Type	Description
`dev_name`	Mandatory	String	Virtio-fs device name to hot plug
`wait-for-done`	Optional	Flag	If used, the method waits until the device is visible by the host PCIe subsystem. Otherwise, only issue hot-plug operation and exit.

17.13.4.3.5 virtio_fs_doca_device_hotunplug

Hot unplug a DOCA virtio FS device. The virtio FS device must be started.

Parameter Name	Optional/Mandatory	Type	Description
`dev_name`	Mandatory	String	Virtio-fs device name to hot unplug
`wait-for-done`	Optional	Flag	If exists, the method waits until the device is non-visible by the host PCIe subsystem. Otherwise, only issue hot-unplug operation and exit.

17.13.4.4 SPDK FSdev Module Configuration

17.13.4.4.1 fsdev_set_opts

Set SPDK FSdev module options.

Parameter Name	Optional/Mandatory	Type	Description
`fsdev_io_pool_size`	Mandatory	int	SPDK FSdev IO objects pool size
`fsdev_io_cache_size`	Mandatory	int	SPDK FSdev IO per-thread objects cache size

17.13.4.4.2 fsdev_get_opts

Get SPDK FSdev module options.

17.13.4.5 SPDK FSDEV Management

DOCA SNAP Virtio-fs uses the SPDK file system (FSdev) device framework as a backend for its virtio-fs controllers. Therefore, an SPDK FSdev must created and configured in advance.

Although the SPDK FSdev framework is generic and allows different types of the backend file system devices to be implemented. Currently, the only available backend device is AIO. This is the file system device that provides passthrough access to a local folder using either the Linux-native async I/O or POSIX async I/O.

17.13.4.5.1 fsdev_get_fsdevs

Get information about the SPDK filesystem devices (fsdevs). The user may specify no parameters to list all filesystem devices, or a filesystem device may be specified by name.

Parameter Name	Optional/Mandatory	Type	Description
`name`	Optional	string	Name of the fsdev of interest

17.13.4.5.2 fsdev_aio_create

Create an SPDK AIO FSdev,

Parameter Name	Optional/Mandatory	Type	Description
`name`	Mandatory	string	Name of the AIO FSdev to create
`root_path`	Mandatory	string	Path on the system directory to be exposed as an SPDK filesystem
`enable_xattr`	Optional	bool	Enable extended attributes if set to `true` ; `false` by default
`enable_writeback_cache`	Optional	bool	Enable the writeback cache if set to `true` ; `false` by default
`max_write`	Optional	int	Maximum write size in bytes; `0x00020000` by default
`enable_skip_rw`	Optional	bool	Enable skipping read/write IOs if set to `true` ; `false` by default ⚠ For debug purposes only.

17.13.4.5.3 fsdev_aio_delete

Delete an AIO FSdev.

Parameter Name	Optional/Mandatory	Type	Description
`name`	Mandatory	string	Name of the AIO FSdev to delete

17.13.4.6 Virtio-fs Emulation Management

Virtio-fs emulation is a protocol belonging to the virtio family of devices. These mount points are found in virtual environments yet by design look like physical mount points to the user within the virtual machine. Each virtio-fs mount point (e.g., virtio-fs PCIe entry) exposed to the host, whether it is PF or VF, must be backed by a virtio-fs controller.

> ⚠ Probing a virtio-fs driver on the host without an already functioning virtio-fs controller may cause the host to hang until such controller is opened successfully (no timeout mechanism exists).

Command	Description
virtio_fs_transport_create	Create a virtio-fs transport
virtio_fs_transport_destroy	Destroy a virtio-fs transport
virtio_fs_transport_start	Start a virtio-fs transport
virtio_fs_transport_stop	Stop a virtio-fs transport
virtio_fs_get_transports	Display virtio-fs transports or requested transport
virtio_fs_device_create	Create a virtio-fs device
virtio_fs_device_start	Start a virtio-fs device
virtio_fs_device_stop	Stop a virtio-fs device
virtio_fs_device_destroy	Destroy a virtio-fs device
virtio_fs_get_devices	Display virtio-fs devices with their characteristics
virtio_fs_doca_device_modify	Modify a virtio-fs device created from DOCA transport

17.13.4.6.1 virtio_fs_transport_create

Create a virtio-fs transport. This RPC includes all the common parameters/options for all transports. The transport becomes operational once it is started.

Parameter Name	Optional/Mandatory	Type	Description
`transport_name`	Mandatory	String	Transport type name. For DOCA SNAP Virtio-fs, `transport_name` should be DOCA.

17.13.4.6.2 virtio_fs_transport_destroy

Destroy a virtio-fs transport.

> ⚠ The transport must be stopped for destruction.

Parameter Name	Optional/Mandatory	Type	Description
`transport_name`	Mandatory	String	Transport type name. For DOCA SNAP Virtio-fs, `transport_name` should be DOCA.

17.13.4.6.3 virtio_fs_transport_start

Start a virtio-fs transport. This RPC finalizes the transport configuration. From this point, the transport is fully operational and can be used to create new devices.

Parameter Name	Optional/Mandatory	Type	Description
`transport_name`	Mandatory	String	Transport type name. For DOCA SNAP Virtio-fs, `transport_name` should be DOCA.

17.13.4.6.4 virtio_fs_transport_stop

Stop a virtio-fs transport. This RPC makes the transport configurable again.

⚠ A transport cannot be stopped if any devices are associated to it.

Parameter Name	Optional/Mandatory	Type	Description
`transport_name`	Mandatory	String	Transport type name. For DOCA SNAP Virtio-fs, `transport_name` should be DOCA.

17.13.4.6.5 virtio_fs_get_transports

Display virtio-fs transports or requested transport.

Parameter Name	Optional/Mandatory	Type	Description
`transport_name`	Optional	String	Transport type name. For DOCA SNAP Virtio-fs, `transport_name` should be DOCA.

17.13.4.6.6 virtio_fs_device_create

Create a virtio-fs device. This RPC creates a device with common parameters which are acceptable to all the transport types. To configure transport-specific parameters, users should use the `virtio_fs_doca_device_modify` command. The device becomes operational once it is started.

Parameter Name	Optional/Mandatory	Type	Description
`transport_name`	Mandatory	String	Transport type name. For DOCA SNAP Virtio-fs, `transport_name` should be DOCA.
`dev_name`	Mandatory	String	Virtio-fs device name to use
`tag`	Optional	String	Virtio-fs tag according to the virtio specification. ⚠ Must be provided during the `virtio_fs_device_create` RPC before the `virtio_fs_device_start` RPC.
`num_request_queues`	Optional	Number	Virtio-fs `num_request_queues` according to the virtio specification (default 31, range 1-62)
`queue_size`	Optional	Number	The maximal queue size for all virtio queues (default 64, range 1-256)
`fsdev`	Optional	String	The name of the SPDK filesystem backend device ⚠ Must be provided during the `virtio_fs_device_create` RPC before the `virtio_fs_device_start` RPC. ⚠ RPC does not verify if FSdev is valid. If a wrong FSdev is attached to the device, the user would experience failure during mount of the FS on the host.
`packed_vq`	Optional	Bool	Expose packed virtqueues feature to the driver for negotiation.
`driver_platform`	Optional	String	Set the driver's platform architecture. Possible values: `native` ; `x86` ; `x86_64` ; `aarch32` ; `aarch64` . Using the `native` platform option sets the driver platform to be identical to the device platform.

17.13.4.6.7 virtio_fs_device_start

Start a virtio-fs device. This RPC finalizes the device configuration. From this point, the transport is fully operational.

Parameter Name	Optional/Mandatory	Type	Description
`dev_name`	Mandatory	String	Virtio-fs device name

17.13.4.6.8 virtio_fs_device_stop

Stop a virtio-fs device.

Parameter Name	Optional/Mandatory	Type	Description
`dev_name`	Mandatory	String	Virtio-fs device name

17.13.4.6.9 virtio_fs_device_destroy

Destroy a virtio-fs device.

⚠ The device must be stopped before destruction.

Parameter Name	Optional/Mandatory	Type	Description
`dev_name`	Mandatory	String	Virtio-fs device name

17.13.4.6.10 virtio_fs_device_modify

Modify a virtio-fs device. This RPC is used to modify/set common properties of the device which are acceptable to all the transports.

Parameter Name	Optional/Mandatory	Type	Description
`dev_name`	Mandatory	String	Virtio-fs device name to use
`tag`	Optional	String	Virtio-fs tag according to the virtio specification ⚠ Must be provided during `virtio_fs_device_create` or `virtio_fs_device_modify` RPCs, before `virtio_fs_device_start` RPC.
`num_request_queues`	Optional	Number	Virtio-fs `num_request_queues` according to the virtio specification (default 31; range 1-62)
`queue_size`	Optional	Number	The maximal queue size for all virtio queues (default 64; range 1-256)

Parameter Name	Optional/Mandatory	Type	Description
`fsdev`	Optional	String	The name of the SPDK filesystem backend device ⚠ Must be provided during `virtio_fs_device_create` or `virtio_fs_device_modify` RPCs, before `virtio_fs_device_start` RPC. ⚠ RPC does not verify if FSdev is valid. If a wrong FSdev is attached to the device, the user would experience failure during mount of the FS on the host.
`packed_vq`	Optional	Bool	Expose packed virtqueues feature to the driver for negotiation
`driver_platform`	Optional	String	Set the driver's platform architecture. Possible values: `native` ; `x86` ; `x86_64` ; `aarch32` ; `aarch64` . Using the `native` platform option sets the driver platform to be identical to the device platform.

17.13.4.6.11 virtio_fs_get_devices

Display virtio-fs devices with their characteristics.

- The user may specify no parameters to list the virtio-fs devices associated with all transports
- The user may specify the name of a transport to list the virtio-fs devices associated with it
- The user may specify the name of a virtio-fs device to display its characteristics

Transport name and device name parameters should be mutually exclusive.

Example response:

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": [
        {
           "name": "vfsdev0",
           "transport_name": "DOCA",
           "state": "idle",
          "fsdev": "aio0",
          "tag": "docatag",
          "queue_size": 256,
          "num_request_queues": 1,
          "packed_ring": true
         }
    ]
}
```

Parameter Name	Optional/Mandatory	Type	Description
`transport_name`	Optional	String	Name of transport whose associated virtio-fs devices to list

Parameter Name	Optional/Mandatory	Type	Description
`dev_name`	Optional	String	Virtio-fs device name

17.13.4.6.12 virtio_fs_doca_device_modify

Modify a virtio-fs device created from DOCA transport.

> ⓘ This RPC is for configuring DOCA target specific parameters.

Parameter Name	Optional/Mandatory	Type	Description
`dev_name`	Mandatory	String	Virtio-fs device name
`manager`	Optional (must be provided before start)	String	Emulation manager
`vuid`	Optional (must be provided before start)	String	Vendor unique identifier ⚠ VUID validation is not done. If an invalid VUID is set, `virtio_fs_device_start` RPC fails.

17.13.4.7 Configuration Example

17.13.4.7.1 Static Function - Bring up

The following is an example of creating virtio-fs DOCA transport and associating it to a virtio-fs device using a static physical function.

- In BlueField:
  a. Create an AIO FSdev backend:

```
rpc.py fsdev_aio_create aio0 /etc/virtiofs
```

  b. Create and start the DOCA transport:

```
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_transport_create -t DOCA
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_transport_start -t DOCA
```

  c. Get transport information:

```
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_get_transports
```

  d. Get managers information:

```
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_get_managers
```

e. Get function information, including their VUIDs:

```
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_get_functions
```

f. Create the virtio-fs device associated with DOCA transport:

```
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_create --transport-name DOCA --dev-name
vfsdev0 --tag doca_test --fsdev aio0 --num-request-queues 8 --queue-size 256 --driver-platform
x86_64
```

g. Set and modify virtio-fs parameters (VUID must be provided before calling `virtio_fs_device_start` RPC):

```
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_device_modify --dev-name vfsdev0 --manager
mlx5_0 --vuid MT2333XZ0VJQVFSS0D0F2
```

h. Start the virtio-fs device:

```
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_start --dev-name vfsdev0
```

i. Get device information:

```
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_get_devices
```

- In VM/host:
  - To mount a device with the tag `docatag` and load `virtio_pci` driver if not loaded:

```
mkdir "/tmp/test"
modprobe -v virtioFS
mount -t virtiofs docatag /tmp/test
```

17.13.4.7.2 Static Function – Teardown

- In BlueField:

a. Get device information:

```
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_get_devices
```

b. Stop and destroy the virtio-fs device:

```
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_stop --dev-name vfsdev0
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_destroy --dev-name vfsdev0
```

c. Stop and destroy the DOCA transport:

```
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_transport_stop -t DOCA
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_transport_destroy -t DOCA
```

- In VM/host:
  - To unmount the device:

```
umount /tmp/test
modprobe -rv virtiofs
```

17.13.4.7.3 Hotplug Function

The following is an example of creating virtio-fs DOCA transport, creating a virtio-fs function, associating it to a virtio-fs device, and hot-plugging it:

- In BlueField:
  a. Create AIO FSdev backend:

  ```
  rpc.py fsdev_aio_create aio0 /etc/virtiofs
  ```

  b. Create and start the DOCA transport:

  ```
  rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_transport_create -t DOCA
  rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_transport_start -t DOCA
  ```

  c. Get transport information:

  ```
  rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_get_transports
  ```

  d. Get managers information:

  ```
  rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_get_managers
  ```

  Some managers would show hotplug capability.

  e. Get functions information:

  ```
  rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_get_functions
  ```

  f. Create virtio-fs function:

  ```
  rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_function_create --manager mlx5_0
  ```

  Returns VUID `MT2333XZ0VJQVFSS0D0F2`.

  g. Get functions information:

  ```
  rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_get_functions
  ```

  Returns the function that has been created with the appropriate VUID.

  h. Create the virtio-fs device associated with DOCA transport:

  ```
  rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_create --transport-name DOCA --dev-name
  vfsdev0 --tag doca_test --fsdev aio0 --num-request-queues 8 --queue-size 256 --driver-platform
  x86_64
  ```

  i. Set and modify virtio-fs parameters:

  ```
  rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_create --transport-name DOCA --dev-name
  vfsdev0 --tag doca_test --fsdev aio0 --num-request-queues 8 --queue-size 256 --driver-platform
  x86_64
  ```

  The VUID must be provided before calling the `virtio_fs_device_start` RPC.

  j. Start the virtio FS device:

  ```
  rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_start --dev-name vfsdev0
  ```

k. Get device information:

```
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_get_devices
```

The output for `vfsdev0` would show it is not yet plugged.

l. Hot plug the DOCA device to the host and wait until it becomes visible by the host:

```
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_device_hotplug --dev-name vfsdev0 --wait-for-
done
```

- In VM/host:
  - To mount a device with the tag `docatag` and load `virtio_pci` driver if not loaded:

```
mkdir "/tmp/test"
modprobe -v virtiofs
mount -t virtiofs docatag /tmp/test
```

17.13.4.7.4 Hot-unplug Function

The following is an example of how to cleanup and destroy the flow described under section "Hotplug Function":

- In VM/host:
  - To unmount the device:

```
umount /tmp/test
modprobe -rv virtiofs
```

- In BlueField:

a. Get device information:

```
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_get_devices
```

b. Hot unplug the DOCA device from the host and wait until it becomes non-visible by the host:

```
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_device_hotunplug --dev-name vfsdev0 --wait-for-
done
```

c. Get device information:

```
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_get_devices
```

d. Stop and destroy the virtio-fs DOCA device:

```
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_stop --dev-name vfsdev0
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_destroy --dev-name vfsdev0
```

e. Destroy the virtio-fs function:

```
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_function_destroy --manager mlx5_0 --vuid
MT2333XZ0VJQVFSS0D0F2
```

f. Stop and destroy the DOCA transport:

```
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_transport_stop -t DOCA
rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_transport_destroy -t DOCA
```

17.13.5 Appendix – BlueField Firmware Configuration

Before configuring DOCA SNAP Virtio-fs, the user must ensure that all firmware configuration requirements are met. By default, virtio-fs is disabled and must be enabled by running both common DOCA SNAP Virtio-fs configurations and additional protocol-specific configurations depending on the expected usage of the application (e.g., hot-plug, SR-IOV, UEFI boot, etc).

After configuration is finished, the host must be power cycled for the changes to take effect.

> To verify that all configuration requirements are satisfied, users may query the current/next configuration by running the following:
>
> ```
> mlxconfig -d /dev/mst/mt41692_pciconf0 -e query
> ```

17.13.5.1 System Configuration Parameters

Parameter	Description	Possible Values
`INTERNAL_CPU_MODEL`	Enable BlueField to work in internal CPU model Must be set to `1` for storage emulations.	0/1
`PCI_SWITCH_EMULATION_ENABLE`	Enable PCIe switch for emulated PFs	0/1
`PCI_SWITCH_EMULATION_NUM_PORT`	The maximum number of hotplug emulated PFs which equals `PCI_SWITCH_EMULATION_NUM_PORT` minus 2. For example, if `PCI_SWITCH_EMULATION_NUM_PORT=16` , then the maximum number of hotplug emulated PFs would be 14. One switch port is reserved for all static PFs.	[0,3-16]

17.13.5.2 RDMA/RoCE Configuration

BlueField's RDMA/RoCE communication is blocked for BlueField's default OS interfaces (nameds ECPFs, typically mlx5_0 and mlx5_1). If RoCE traffic is required, additional network functions (scalable functions) must be added which support RDMA/RoCE traffic.

> The following is not required when working over TCP or even RDMA/IB.

To enable RoCE interfaces, run the following from within the BlueField device:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PER_PF_NUM_SF=1
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PF_SF_BAR_SIZE=8 PF_TOTAL_SF=2
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0.1 s PF_SF_BAR_SIZE=8 PF_TOTAL_SF=2
```

17.13.5.3 Virtio-fs Configuration

Due to virtio-fs protocol limitations, using bad configuration while working with static virtio-fs PFs may cause the host server OS to fail on boot.

Before continuing, make sure you have configured:

- A working channel to access Arm even when the host is shut down. Setting such channel is out of the scope of this document. Please refer to NVIDIA BlueField DPU BSP documentation for more details.
- Use the initial configure file to create a controller on the static virtio-fs PF.

  For more information, please refer to section "Virtio-fs Emulation Management"

Parameter	Description	Possible Values
`VIRTIO_VFS_EMULATION_ENABLE`	Enable virtio-fs device emulation	0/1
`VIRTIO_VFS_EMULATION_NUM_PF`	Number of static emulated virtio-fs PFs See WARNING above.	[0-4]
`VIRTIO_VFS_EMULATION_NUM_MSIX`	Number of MSIX assigned to emulated virtio-fs PF/VF	[0-63]

17.13.6 Appendix - Host OS Configuration

With Linux environment on host OS, additional kernel boot parameters may be required to support DOCA SNAP Virtio-fs related features:

- To use PCIe hotplug, `pci=realloc` must be added
- `modprobe.blacklist=virtio_pci,virtiofs` for the virtio-fs driver which is not built-in
- `modprobe.blacklist=virtio_pci` for the `virtio_pci` driver which is not built-in

To view boot parameter values, run:

```
cat /proc/cmdline
```

It is recommended to use the following command with virtio-fs:

```
[dpu] cat /proc/cmdline BOOT_IMAGE … pci=realloc modprobe.blacklist=virtio_pci,virtiofs
```

17.13.6.1 Intel Server Performance Optimizations

```
cat /proc/cmdline
BOOT_IMAGE=(hd0,msdos1)/vmlinuz-5.15.0_mlnx root=UUID=91528e6a-b7d3-4e78-9d2e-9d5ad60e8273 ro crashkernel=auto
resume=UUID=06ff0f35-0282-4812-894e-111ae8d76768 rhgb quiet pci=realloc modprobe.blacklist=virtio_pci,virtiofs
```

17.13.6.2 AMD Server Performance Optimizations

```
cat /proc/cmdline
cat /proc/cmdline BOOT_IMAGE=(hd0,msdos1)/vmlinuz-5.15.0_mlnx root=UUID=91528e6a-b7d3-4e78-9d2e-9d5ad60e8273 ro
crashkernel=auto resume=UUID=06ff0f35-0282-4812-894e-111ae8d76768 rhgb quiet pci=realloc
modprobe.blacklist=virtio_pci,virtiofs
```

17.13.7 References

Title	Description
NVIDIA DOCA	NVIDIA DOCA™ SDK enables developers to rapidly create applications and services on top of NVIDIA® BlueField® networking platform, leveraging industry-standard APIs
NVIDIA BlueField BSP	BlueField Board Support Package includes the bootloaders and other essentials for loading and setting software components
BlueField DPU Hardware User Manual	This document provides details as to the interfaces of the BlueField DPU, specifications, required software and firmware for operating the device, and a step-by-step plan for bringing the DPU up
NVIDIA BlueField BSP Documentation	This document provides product release notes as well as information on the BlueField software distribution and how to develop and/or customize applications, system software, and file system images for the BlueField platform
DOCA Device Emulation	DOCA Device Emulation library documentation. The DOCA Device Emulation subsystem provides a low-level software API for users to develop PCIe devices and their controllers.
DOCA DevEmu Virtio-fs	DOCA Device Emulation Virtio-fs library documentation. The DOCA DevEmu Virtio-fs library is part of the DOCA DevEmu Virtio subsystem. It provides low-level software APIs that provide building blocks for developing and manipulating virtio filesystem devices using the device emulation capability of BlueField platforms.
DOCA DevEmu PCI	DOCA Device Emulation PCI library documentation. DOCA DevEmu PCI is part of the DOCA Device Emulation subsystem. It provides low-level software APIs that allow management of an emulated PCIe device using the emulation capability of NVIDIA® BlueField® networking platforms.

18 API References

This section contains the following pages:

- NVIDIA DOCA Driver APIs
- NVIDIA DOCA Library APIs

18.1 NVIDIA DOCA Driver APIs

The driver APIs for this DOCA version are available here.

18.2 NVIDIA DOCA Library APIs

The library APIs for this DOCA version are available here.

19 Miscellaneous (Runtime)

This section contains the following pages:

- NVIDIA DOCA Glossary
- NVIDIA DOCA Crypto Acceleration
- NVIDIA DOCA Services Fluent Logger
- NVIDIA DOCA DPU CLI
- NVIDIA DOCA Emulated Devices
- NVIDIA BlueField Modes of Operation
- DOCA Switching
- NVIDIA DOCA with OpenSSL
- NVIDIA BlueField DPU Scalable Function User Guide
- NVIDIA TLS Offload Guide
- NVIDIA DOCA Troubleshooting Guide
- NVIDIA DOCA Virtual Functions User Guide

19.1 NVIDIA DOCA Glossary

Term	Description
ACS	Access control services
ASAP^2	Accelerated Switching and Packet Processing
ASN	Autonomous system number
ATF	Arm-trusted firmware
BAR	Base address register
BDF address	Bus, device, function address. This is the device's PCIe bus address to uniquely identify the specific device.
BFB	BlueField bootstream
BGP	Border gateway protocol
BMC	Board management controller
BUF	Buffer
BSP	BlueField support package
CBS	Committed burst size
CIR	Committed information rate
CMDQ	Command queue
CPDS	Control pipe dynamic size
CQE	Completion queue events
CTX	Context
DEK	Data encryption key
DMA	Direct memory access
DOCA	DPU SDK

Term	Description
DPA	Data path accelerator; an auxiliary processor designed to accelerate data-path operations
DPCP	Direct packet control plane
DPDK	Data plane development kit
DPI	Deep packet inspection
DPIF	Datapath offload interface
DPU	Data processing unit, the third pillar of the data center with CPU and GPU. BlueField is available as a DPU and as a SuperNIC.
DW	Dword
EBS	Excess burst size
ECE	Enhanced connection establishment
ECPF	Embedded CPU physical function
EIR	Excess information rate
EM	Exact match
eMMC	Embedded multi-media card
ESP	EFI system partition
ESP	Encapsulating security payload
EU	Execution unit. HW thread; a logical DPA processing unit.
FLR	Function level reset
FIFO	First-in-first-out
FIPS	Federal Information Processing Standards
FPGA	Field-programmable gate arrays
FW	Firmware
GDAKIN	GPUDirect async kernel-initiated network
GDB	GNU debugger
HCA	Host-channel adapter
Host	When referring to "the host" this documentation is referring to the **server host**. When referring to the Arm based host, the documentation will specifically call out "Arm host". • Server host OS refers to the Host Server OS (Linux or Windows) • Arm host refers to the AARCH64 Linux OS which is running on the BlueField Arm Cores
HW	Hardware
hwmon	Hardware monitoring
IB	InfiniBand
ICM	Interface configuration memory
ICV	Integrity check value
IDE	Integrated development environment
IKE	Internet key exchange

Term	Description
IR	Intermediate representation
IRQ	Interrupt request
KPI	Key performance indicator
LRO	Large receive offload
LSO	Large send offload
LTO	Link-time optimization
MFT	Mellanox firmware tools
MLNX_OFED	Mellanox OpenFabrics Enterprise Distribution
MPU	Message passing interface
MSB	Most significant bit
MSI-X	Message signaled interrupts extended
MSS	Maximum segment size
MSS	Memory subsystem
MST	Mellanox software tools
MTU	Maximum transmission unit
NAT	Network address translation
NIC	Network interface card
NIST	National Institute of Standards and Technology
NP	Notification point
NS	Namespace
NUMA	Non-uniform memory access
OOB	Out-of-band
OS	Operating system
OVS	Open vSwitch
PBA	Pending bit array
PBS	Peak burst size
PCIe	PCI Express; Peripheral Component Interconnect Express
PF	Physical function
PE	Progress engine
PHC	Physical hardware clock
PIR	Peak information rate
PK	Platform key
PKA	Public key accelerator
POC	Proof of concept
PUD	Process under debug
RD	Route distinguisher

Term	Description
RDMA	Remote direct memory access
RDMA CM	RDMA connection manager
RegEx	Regular expression
REQ	Request
RES	Response
RN	Request node RN-F - Fully coherent request node RN-D - IO coherent request node with DVM support RN-I - IO coherent request node
RNG	Random number generator/generation
RoCE	Ethernet and RDMA over converged Ethernet
RP	Reaction point
RQ	Receive queue
RShim	Random shim
RSP	Remote serial protocol
RT	Route target
RTOS	Real-time operating system
RTT	Round-trip time
RX	Receive
RXP	Regular expression processor
SA	Security association
SBSA	Server base system architecture
SDK	Software development kit
SF	Sub-function or scalable function
SFC	Services function chaining
SG	Scatter-gather
SHA	Secure hash algorithm
SN	Sequence number
SNAP	Storage-defined network-accelerated processing
SPDK	Storage performance development kit
SPI	Security parameters index
SQ	Send queue
SR-IOV	Single-root IO virtualization
SuperNIC	a configuration of a DPU that is specific for E-W networking. BlueField has a SuperNIC configuration
SVI	Switch virtual interface
Sync event	Synchronization event
TAI	International Atomic Time

Term	Description
TIR	Transport interface receive
TIS	Transport interface send
TLS	Transport layer security
TSO	TCP segmentation offload
TX	Transmit
UDS	Unix domain socket
UEFI	Unified extensible firmware interface
UTC	Coordinated Universal Time
vDPA	Virtual data path acceleration
VF	Virtual function
VFE	Virtio full emulation
VM	Virtual machine
VMA	NVIDIA® Messaging Accelerator
VNI	• Virtual network identifier • VXLAN network identifier
VPI	Virtual protocol interconnect
VRF	Virtual routing and forwarding
VTEP	VXLAN tunnel endpoint
WorkQ or workq	Work queue
WQE	Work queue elements
WR	Write
XLIO	NVIDIA® Accelerated IO

19.2 NVIDIA DOCA Crypto Acceleration

NVIDIA® BlueField® DPU incorporates several Public Key Acceleration (PKA) engines to offload the processor of the Arm host, providing high-performance computation of PK algorithms. BlueField's PKA is useful for a wide range of security applications. It can assist with SSL acceleration, or a secure high-performance PK signature generator/checker and certificate related operations.

BlueField's PKA software libraries implement a simple, complete framework for crypto public key infrastructure (PKI) acceleration. It provides direct access to hardware resources from the user space, and makes available a number of arithmetic operations—some basic (e.g., addition and multiplication), and some complex (e.g., modular exponentiation and modular inversion)—and high-level operations such as RSA, Diffie-Hallman, Elliptic Curve Cryptography, and the Federal Digital Signature Algorithm (DSA as documented in FIPS-186) public-private key systems.

Some of the use cases for the BlueField PKA involve integrating OpenSSL software applications with BlueField's PKA hardware. The BlueField PKA dynamic engine for OpenSSL allows applications integrated with OpenSSL (e.g., StrongSwan) to accomplish a variety of security-related goals and to accelerate the cryptographic processing with the BlueField PKA hardware. OpenSSL versions ≥1.0.0, ≤1.1.1, and 3.0.2 are supported.

> With CentOS 7.6, only OpenSSL 1.1 (not 1.0) works with PKA engine and keygen. Use `openssl11` with PKA engine and keygen.

The engine supports the following operations:

- RSA
- DH
- DSA
- ECDSA
- ECDH
- Random number generation that is cryptographically secure.

Up to 4096-bit keys for RSA, DH, and DSA operations are supported. Elliptic Curve Cryptography support of (nist) prime curves for 160, 192, 224, 256, 384 and 521 bits.

For example:

To sign a file using BlueField's PKA engine:

```
$ openssl dgst -engine pka -sha256 -sign <privatekey> -out <signature> <filename>
```

To verify the signature, execute:

```
$ openssl dgst -engine pka -sha256 -verify <publickey> -signature <signature> <filename>
```

For further details on BlueField PKA, please refer to "PKA Driver Design and Implementation Architecture Document" and/or "PKA Programming Guide". Directions and instructions on how to integrate the BlueField PKA software libraries are provided in the README files on our PKA GitHub.

19.3 NVIDIA DOCA Services Fluent Logger

This guide provides instructions on how to use the logging infrastructure for DOCA services on top of NVIDIA® BlueField® DPU.

19.3.1 Introduction

Fluent Bit is a fast log collector that collects information from multiple sources and then forwards the data onward using Fluent.

On NVIDIA DPUs, the Fluent Bit logger can be easily configured to collect system data and the logs from the different DOCA services.

19.3.2 Deployment

The deployment is based on a recommended configuration template for the existing Fluent Bit container.

For information about the deployment of DOCA containers on top of the BlueField DPU, refer to NVIDIA DOCA Container Deployment Guide.

The following is an example YAML file for deploying the Fluent Bit pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: fluent-bit
spec:
  hostNetwork: true
  containers:
  - name: fluent-bit
    image: fluent/fluent-bit:latest
    imagePullPolicy: Always
    # Example resource definitions
    resources:
      requests:
        memory: "100Mi"
        cpu: "200m"
      limits:
        memory: "200Mi"
        cpu: "300m"
    volumeMounts:
    - name: varlog
      mountPath: /var/log
    - name: config-file
      mountPath: /fluent-bit/etc/fluent-bit.conf
  volumes:
  - name: varlog
    hostPath:
      path: /var/log
  - name: config-file
    hostPath:
      path: /opt/mellanox/doca/services/fluent-bit.conf
      type: File
```

As explained in the "Configuration" section, Fluent Bit uses a configuration file. As such, to ensure that the example YAML file is shared from the DPU to the deployed Fluent Bit container, use the following:

```
path: /opt/mellanox/doca/services/fluent-bit.conf
```

> ⚠ The path below is just an example for where the user can place the `fluent-bit.conf` file. The file could be placed in a different directory on the DPU as long as the YAML file points to the updated location.

19.3.3 Configuration

The Fluent Bit configuration file should have the following sections:

- `[SERVICE]` - to define the service specifications
- `[INPU]` - to define folders to collect logs from (there could be multiple inputs)
- `[OUTPUT]` - IP and port to stream the data to

Example configuration file:

```
[SERVICE]
    Flush         2
    Log_Level     info
    Daemon        off
    Parsers_File  parsers.conf
    HTTP_Server   On
    HTTP_Listen   0.0.0.0
    HTTP_Port     2020

[INPUT]
    Name              tail
    Tag               kube.*
    Path              /var/log/containers/*.log
    Parser            docker
    Mem_Buf_Limit     5MB
    Skip_Long_Lines   On
    Refresh_Interval  10

[INPUT]
    Name              tail
    Tag               sys.*
    Path              /var/log/doca/*/*.log
    Mem_Buf_Limit     5MB
    Skip_Long_Lines   On
    Refresh_Interval  10

[OUTPUT]
    Name  es
    Match *
    Host 10.20.30.40
    Port 9201
    Index fluent_bit
    Type cpu_metrics
```

> The most important field to pay attention to is `Path` for the `INPUT` section. DOCA services report their logs to a unique directory under `/var/log/doca/<service_name>/*.log` per the respective DOCA service. As such, the configuration above defines the `/var/log/doca/*/*.log` input definition.

More information about the full specifications can be found in the official Fluent Bit manual.

19.3.4 Troubleshooting

For container-related troubleshooting, refer to the "Troubleshooting" section in the NVIDIA DOCA Container Deployment Guide.

For general troubleshooting, refer to the NVIDIA DOCA Troubleshooting Guide.

When copying the above YAML file, it is possible that the container infrastructure logs give an error related to RFC 1123". These errors are usually a result of a spacing error in the file, which sometimes occur when copying the file as is from this page. To fix this issue, make sure that only the space character ('` `') is used as a spacer in the file and not other whitespace characters that might have been added during the copy operation.

19.4 NVIDIA DOCA DPU CLI

This guide provides quick access to a useful set of CLI commands and utilities on the NVIDIA® BlueField® DPU environment.

19.4.1 Introduction

This guide provides a concise guide on useful commands for DOCA deployment and configuration.

The tables in this guide provide two categories of commands:

- General commands for Linux/networking environment
- DOCA/DPU-specific commands

> For more information about these commands, such as usage instructions, flag options, arguments and so on, use the `-h` option after the command or use the manual (e.g., `man lspci`).

19.4.2 General Commands

Command	Description
`ifconfig`	Used to configure kernel-resident network interfaces. It is used at boot time to set up interfaces as necessary. After that, it is usually only needed when debugging or when system tuning is needed. If no arguments are given, `ifconfig` displays the status of the currently active interfaces. If a single interface argument is given, it displays the status of the given interface only. If a single `-a` argument is given, it displays the status of all interfaces, even those that are down. Otherwise, it configures an interface.
`ethtool <devname>`	Used to query and control network device driver and hardware settings, particularly for wired Ethernet devices. `<devname>` is the name of the network device on which `ethtool` should operate. ⚠ This command shows the speed of the network card of the DPU.
`lspci`	Displays information about PCIe buses in the system and devices connected to them. By default, it shows a brief list of devices.
`tcpdump`	Dump traffic on a network. Usage: `tcpdump -i <interface>` where `<interface>` is any port interface (physical/SF rep/VF port rep).
`ovs-vsctl`	Utility for querying and configuring `ovs-vswitchd`. The `ovs-vsctl` program supports the model of a bridge implemented by Open vSwitch in which a single bridge supports ports on multiple VLANs.
`mount 10.0.0.10:/vol/myshare/ myshare/`	Used for mounting a work directory on the DPU. ⚠ Must be used after creating a new directory named `myshare` under root (i.e., `mkdir /myshare`)
`scp`	Secure copy (remote file copy program). Useful for copying files from BlueField to the host and vice versa.
`iperf`	Used for server-client connection. Useful to check if the network connection achieves the speed of the network card on the DPU (line rate).

19.4.3 DPU/DOCA Commands

Command	Description
`ibdev2netdev`	Displays available `mlnx` interfaces

Command	Description
`mst`	Used to start MST service, to stop it, and for other operations with NVIDIA devices like reset and enabling remote access
`cat /etc/mlnx-release`	Displays the full BlueField image (bfb) version
`cat /etc/os-release`	Displays the details of the underlying OS installed on BlueField
`ibv_devinfo`	Displays the current InfiniBand connected devices and relevant information. Useful for checking current firmware version.
`ipmitool power cycle`	Power cycle ⚠ Prior to performing a power cycle, make sure to do a graceful shutdown.
`echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages`	DPDK setup. Allocates hugepages for DPDK environment abstraction layer (EAL).
`mlxdevm tool`	The mlxdevm tool is found under `/opt/mellanox/iproute2/sbin/`. With this tool it is possible to create an SF and set its state to active, configure a HW address and set it to trusted, deploy the created SF and print info about it.
`/opt/mellanox/iproute2/sbin/mlxdevm port add pci/<pci_address> flavour pcisf pfnum <correspondig_physical_function_number> sfnum <unique_sf_number>`	Creates an SF in the flavor of the given PF with the given unique SF number. Example: ``/opt/mellanox/iproute2/sbin/mlxdevm port add pci/0000:03`:00.0 flavour pcisf pfnum 0 sfnum 4``
`/opt/mellanox/iproute2/sbin/mlxdevm port show`	Displays information about the available SFs
`/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/<sf_index> hw_addr <HW_address> trust on state active`	Configures SF capabilities such as setting the HW address, making it "trusted", and setting its state to active. `<sf_index>` the SF. To obtain this index, you may run `mlxdevm port show`. Example: `/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/229377 hw_addr 02:25:f2:8d:a2:4c trust on state active`

`$ echo mlx5_core.sf.<next_serial> > /sys/bus/auxiliary/drivers/mlx5_core.sf_cfg/unbind`
`$ echo mlx5_core.sf. <next_serial> > /sys/bus/auxiliary/drivers/mlx5_core.sf/bind`
	These two commands deploy the created SF. The first command unbinds the SF from the default driver, while the second command binds the SF to the actual driver. The deployment phase should be done after the capabilities of the SF are configured. The SF is identified by `<next_serial>` which can be obtained by running the command below.

Command	Description
`ls /sys/bus/auxiliary/devices/mlx5_core.sf.*`	Displays additional information about the created SFs and their "next serial numbers". For example, if `mlx5_core.sf.2` exists in the output of the command, then running `cat /sys/bus/auxiliary/devices/mlx5_core.sf.2/sfnum` would output the sfnum related to `mlx5_core.sf.2`.

`/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/<pci_address>/<sf_index> state inactive`
`/opt/mellanox/iproute2/sbin/mlxdevm port del pci/<pci_address>/<sf_index>`
	These two commands must be executed to delete a given SF. First, users must set the state of the SF to inactive, and only then should it be deleted.

`/opt/mellanox/iproute2/sbin/mlxdevm port help`	Displays additional information about operations that can be used on created SF ports
`crictl pods`	Displays currently active K8S pods, and their IDs (it might take up to 20-30 seconds for the pod to start)
`crictl ps`	Displays currently active containers and their IDs
`crictl ps -a`	Displays all containers, including containers that recently finished their execution
`crictl logs <container-id>`	Examines the logs of a given container
`crictl exec -it <container-id> /bin/bash`	Attaches a shell to a running container
`journalctl -u kubelet`	Examines the Kubelet logs. Useful when a pod/container fails to spawn.
`crictl stopp <pod-id>`	Stops a running K8S pod
`crictl stop <container-id>`	Stops a running container
`crictl rmi <image-id>`	Removes a container image from the local K8S registry

19.5 NVIDIA DOCA Emulated Devices

Unable to render include or excerpt-include. Could not retrieve page.

19.6 NVIDIA BlueField Modes of Operation

This document describes the modes of operation available for NVIDIA® BlueField® networking platforms (DPUs or SuperNICs).

19.6.1 Introduction

Unable to render include or excerpt-include. Could not retrieve page.

19.7 DOCA Switching

NVIDIA® BlueField® and NVIDIA® ConnectX® platforms provide robust support for diverse applications through hardware-based offloads, offering unparalleled scalability, performance, and efficiency.

This section lists the extensive switching capabilities enabled by DOCA libraries and services on these platforms. It includes detailed configurations of Open Virtual Switch (OVS) such as the setup of representors, virtualization options, and optional bridge configurations. These subsections guide users through the steps to effectively implement these software components.

19.7.1 DOCA Representors Model

⚠ This model is only applicable when the BlueField is operating DPU mode.

BlueField® DPU uses netdev representors to map each one of the host side physical and virtual functions:

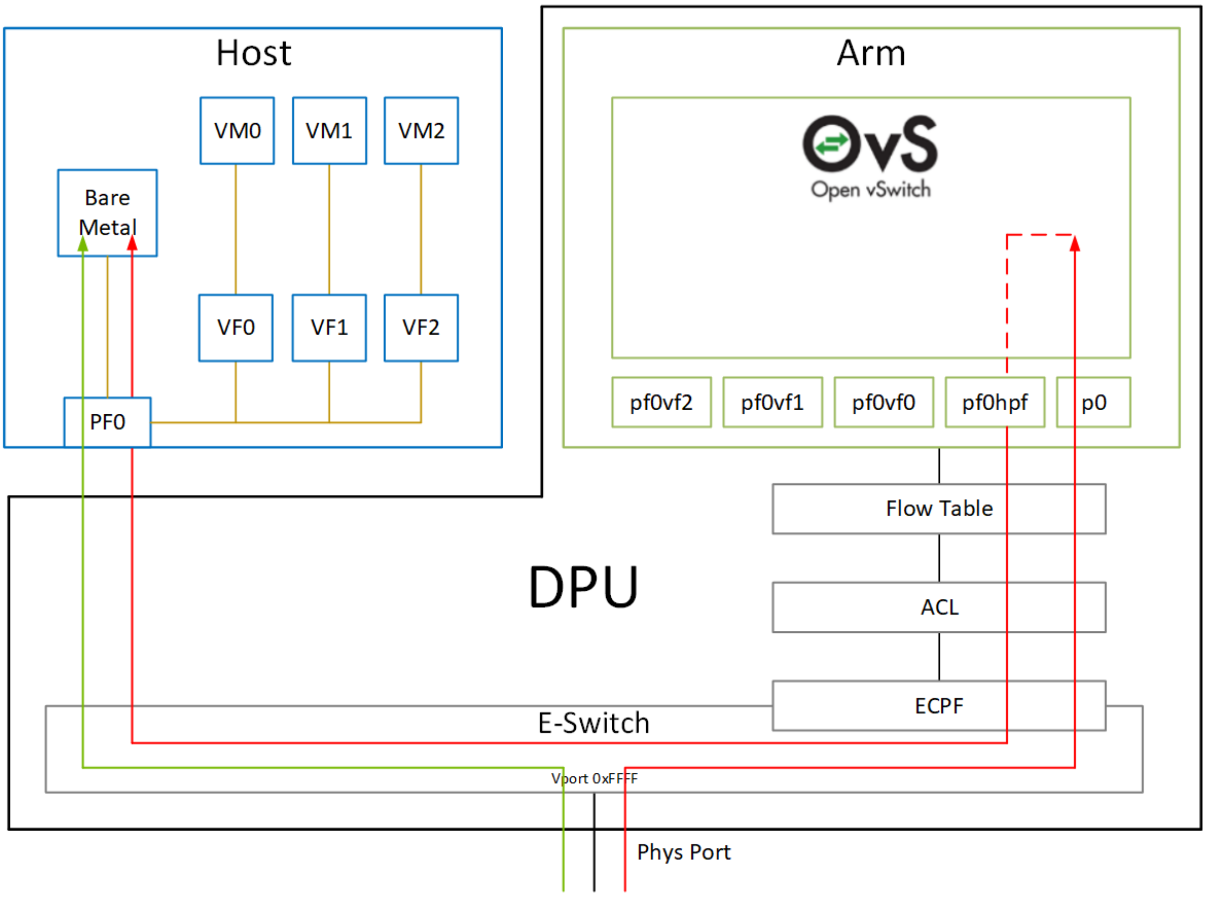
1. Serve as the tunnel to pass traffic for the virtual switch or application running on the Arm cores to the relevant PF or VF on the Arm side.
2. Serve as the channel to configure the embedded switch with rules to the corresponding represented function.

Those representors are used as the virtual ports being connected to OVS or any other virtual switch running on the Arm cores.

When in ECPF ownership mode, we see 2 representors for each one of the DPU's network ports: one for the uplink, and another one for the host side PF (the PF representor created even if the PF is not probed on the host side). For each one of the VFs created on the host side a corresponding representor would be created on the Arm side. The naming convention for the representors is as follows:

- Uplink representors: `p<port_number>`
- PF representors: `pf<port_number>hpf`
- VF representors: `pf<port_number>vf<function_number>`

The diagram below shows the mapping of between the PCIe functions exposed on the host side and the representors. For the sake of simplicity, we show a single port model (duplicated for the second port).



The red arrow demonstrates a packet flow through the representors, while the green arrow demonstrates the packet flow when steering rules are offloaded to the embedded switch. More details on that are available in the switch offload section.

⚠ The MTU of host functions (PF/VF) must be smaller than the MTUs of both the uplink and corresponding PF/VF representor. For example, if the host PF MTU is set to 9000, both uplink and PF representor must be set to above 9000.

This section contains the following pages:

- Virtio Acceleration through Hardware vDPA
- Bridge Offload
- Link Aggregation
- Controlling Host PF and VF Parameters

⚠ DOCA also provides OpenvSwitch Acceleration (OVS in DOCA) which implements a virtual switch service, designed to work with NVIDIA NICs and DPUs to utilize ASAP^2 (Accelerated Switching and Packet Processing) technology for data-path acceleration, providing the most efficient performance and feature set due to its architecture and use of DOCA libraries.

19.7.2 Virtio Acceleration through Hardware vDPA

19.7.2.1 Hardware vDPA Installation

Hardware vDPA requires QEMU v2.12 (or with upstream 6.1.0) and DPDK v20.11 as minimal versions.

To install QEMU:

1. Clone the sources:

```
git clone https://git.qemu.org/git/qemu.git
cd qemu
git checkout v2.12
```

2. Build QEMU:

```
mkdir bin
cd bin
../configure --target-list=x86_64-softmmu --enable-kvm
make -j24
```

To install DPDK:

1. Clone the sources:

```
git clone git://dpdk.org/dpdk
cd dpdk
git checkout v20.11
```

2. Install dependencies (if needed):

```
yum install cmake gcc libnl3-devel libudev-devel make pkgconfig valgrind-devel pandoc libibverbs libmlx5
libmnl-devel -y
```

3. Configure DPDK:

```
export RTE_SDK=$PWD
make config T=x86_64-native-linuxapp-gcc
cd build
sed -i 's/\(CONFIG_RTE_LIBRTE_MLX5_PMD=\)n/\1y/g' .config
sed -i 's/\(CONFIG_RTE_LIBRTE_MLX5_VDPA_PMD=\)n/\1y/g' .config
```

4. Build DPDK:

```
make -j
```

5. Build the vDPA application:

```
cd $RTE_SDK/examples/vdpa/
make -j
```

19.7.2.2 Hardware vDPA Configuration

To configure huge pages:

```
mkdir -p /hugepages
mount -t hugetlbfs hugetlbfs /hugepages
echo <more> > /sys/devices/system/node/node0/hugepages/hugepages-1048576kB/nr_hugepages
echo <more> > /sys/devices/system/node/node1/hugepages/hugepages-1048576kB/nr_hugepages
```

To configure a vDPA VirtIO interface in an existing VM's xml file (using `libvirt`):

1. Open the VM's configuration XML for editing:

   ```
   virsh edit <domain name>
   ```

2. Perform the following:
   a. Change the top line to:

      ```
      <domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
      ```

   b. Assign a memory amount and use 1GB page size for huge pages (size must be the same as that used for the vDPA application), so that the memory configuration looks as follows.

      ```
      <memory unit='KiB'>4194304</memory>
      <currentMemory unit='KiB'>4194304</currentMemory>
      <memoryBacking>
        <hugepages>
          <page size='1048576' unit='KiB'/>
        </hugepages>
      </memoryBacking>
      ```

   c. Assign an amount of CPUs for the VM CPU configuration, so that the `vcpu` and `cputune` configuration looks as follows:

      ```
      <vcpu placement='static'>5</vcpu>
      <cputune>
        <vcpupin vcpu='0' cpuset='14'/>
        <vcpupin vcpu='1' cpuset='16'/>
        <vcpupin vcpu='2' cpuset='18'/>
        <vcpupin vcpu='3' cpuset='20'/>
        <vcpupin vcpu='4' cpuset='22'/>
      </cputune>
      ```

   d. Set the memory access for the CPUs to be shared, so that the `cpu` configuration looks as follows:

      ```
      <cpu mode='custom' match='exact' check='partial'>
        <model fallback='allow'>Skylake-Server-IBRS</model>
        <numa>
          <cell id='0' cpus='0-4' memory='8388608' unit='KiB' memAccess='shared'/>
        </numa>
      </cpu>
      ```

   e. Set the emulator in use to be the one built in step 2, so that the emulator configuration looks as follows:

      ```
      <emulator><path to qemu executable></emulator>
      ```

   f. Add a virtio interface using QEMU command line argument entries, so that the new interface snippet looks as follows:

      ```
      <qemu:commandline>
        <qemu:arg value='-chardev'/>
        <qemu:arg value='socket,id=charnet1,path=/tmp/sock-virtio0'/>
        <qemu:arg value='-netdev'/>
        <qemu:arg value='vhost-user,chardev=charnet1,queues=16,id=hostnet1'/>
        <qemu:arg value='-device'/>
      ```

```
  <qemu:arg value='virtio-net-pci,mq=on,vectors=6,netdev=hostnet1,id=net1,mac=e4:11:c6:d3:45:f2,bus
=pci.0,addr=0x6,
   page-per-vq=on,rx_queue_size=1024,tx_queue_size=1024'/>
</qemu:commandline>
```

In this snippet, the vhostuser socket file path, the amount of queues, the MAC and the PCIe slot of the virtio device can be configured.

19.7.2.3 Running Hardware vDPA

Hardware vDPA supports switchdev mode only.

1. Create the ASAP^2 environment:
    a. Create the VFs.
    b. Enter switchdev mode.
    c. Set up OVS.
2. Run the vDPA application:

```
cd $RTE_SDK/examples/vdpa/build
./vdpa -w <VF PCI BDF>,class=vdpa --log-level=pmd,info -- -i
```

3. Create a vDPA port via the vDPA application CLI:

```
create /tmp/sock-virtio0 <PCI DEVICE BDF>
```

The vhostuser socket file path must be the one used when configuring the VM.

4. Start the VM:

```
virsh start <domain name>
```

For further information on the vDPA application, visit the Vdpa Sample Application DPDK documentation.

19.7.3 Bridge Offload

Bridge offload is supported switchdev mode only.

Bridge offload is supported from kernel version 5.15 onward.

A Linux bridge is an in-kernel software network switch (based on and implementing a subset of IEEE 802.1D standard) used to connect Ethernet segments together in a protocol-independent manner. Packets are forwarded based on L2 Ethernet header addresses.

mlx5 provides the ability to offload bridge dataplane unicast packet forwarding and VLAN management to hardware.

19.7.3.1 Basic Configuration

1. Initialize the ASAP^2 environment:
   a. Create the VFs.
   b. Enter switchdev mode.
2. Create a bridge and add mlx5 representors to bridge:

```
ip link add name bridge0 type bridge
ip link set enp8s0f0_0 master bridge0
```

19.7.3.2 Configuring VLAN

1. Enable VLAN filtering on the bridge:

```
ip link set bridge0 type bridge vlan_filtering 1
```

2. Configure port VLAN matching (trunk mode). In this configuration, only packets with specified VID are allowed.

```
bridge vlan add dev enp8s0f0_0 vid 2
```

3. Configure port VLAN tagging (access mode). In this configuration, VLAN header is pushed/popped upon reception/transmission on port.

```
bridge vlan add dev enp8s0f0_0 vid 2 pvid untagged
```

19.7.3.3 VF LAG Support

Bridge supports offloading on bond net device that is fully initialized with mlx5 uplink representors and is in single (shared) FDB LAG mode. Details about initialization of LAG are provided in section "SR-IOV VF LAG".

To add a bonding net device to bridge:

```
ip link set bond0 master bridge0
```

For further information on interacting with Linux bridge via iproute2 bridge tool, refer to man 8 bridge.

19.7.4 Link Aggregation

Unable to render include or excerpt-include. Could not retrieve page.

19.7.5 Controlling Host PF and VF Parameters

Unable to render include or excerpt-include. Could not retrieve page.

19.8 NVIDIA DOCA with OpenSSL

This guide provides instructions on using DOCA SHA for OpenSSL implementations.

19.8.1 Introduction

The `doca_sha_offload_engine` is an OpenSSL dynamic engine with the ability of offloading SHA calculation. It can offload the OpenSSL one-shot SHA-1, SHA-256, and SHA-512. It supports synchronous mode and asynchronous mode by leveraging the OpenSSL `async_jobs` library. For more information on the `async_jobs` library, please refer to official OpenSSL documentation.

This engine is based on the `doca_sha` library and the OpenSSL dynamic engine interface API. For more information on the OpenSSL dynamic engine, please refer to official OpenSSL documentation.

This engine can be called by an OpenSSL application through the OpenSSL high-level algorithm call interface, `EVP_Digest`. For more information on the `EVP_Digest`, please refer to official OpenSSL documentation.

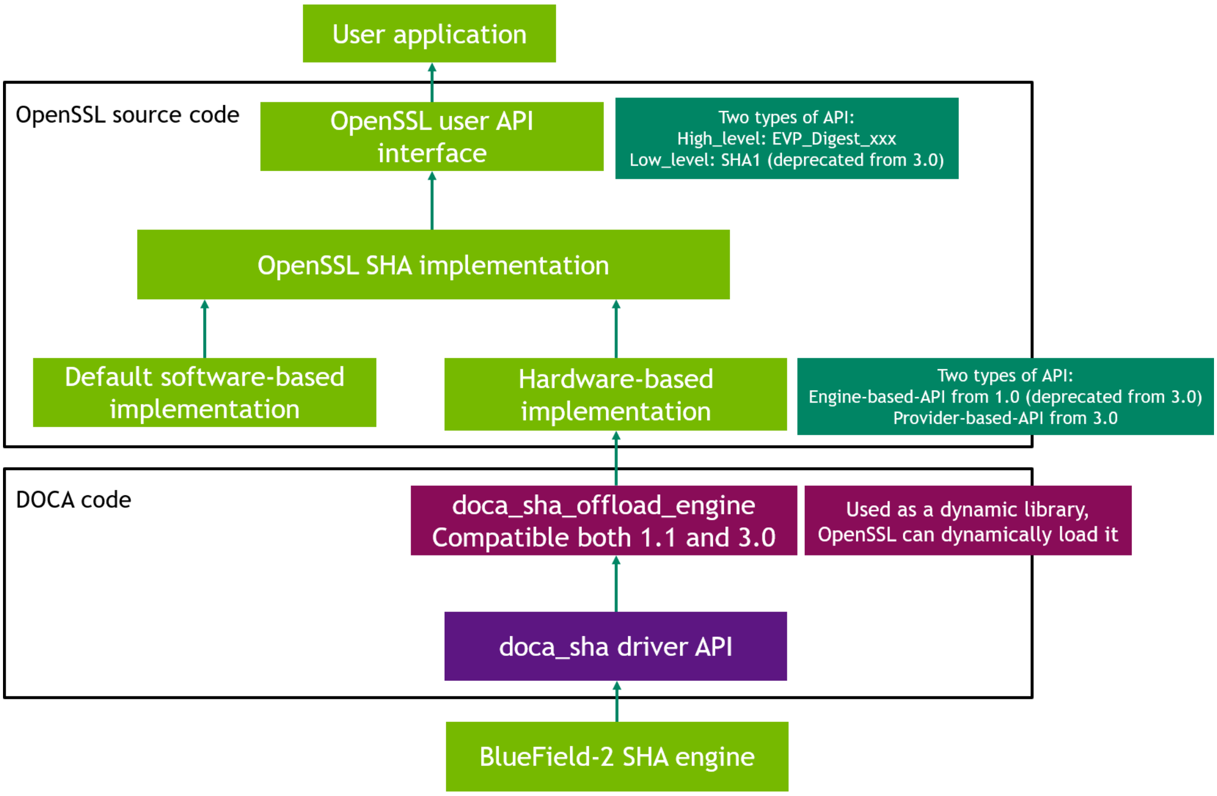
19.8.2 Prerequisites

- Hardware-based `doca_sha` engine which can be verified by calling `doca_sha_get_hardware_supported()`
- Installed OpenSSL version ≥ 1.1.1

19.8.3 Architecture

The following diagram shows the software hierarchy of `doca_sha_offload_engine` and its location in the whole DOCA repository.

From the perspective of OpenSSL, this engine is an instantiation of the OpenSSL dynamic engine interface API by leveraging the `doca_sha` library.


User application
OpenSSL source code
OpenSSL user API interface
Two types of API:
High_level: EVP_Digest_xxx
Low_level: SHA1 (deprecated from 3.0)
OpenSSL SHA implementation
Default software-based implementation
Hardware-based implementation
Two types of API:
Engine-based-API from 1.0 (deprecated from 3.0)
Provider-based-API from 3.0
DOCA code
doca_sha_offload_engine
Compatible both 1.1 and 3.0
Used as a dynamic library,
OpenSSL can dynamically load it
doca_sha driver API
BlueField-2 SHA engine

19.8.4 Capabilities and Limitations

- Only one-shot OpenSSL SHA is supported
- The maximum message length ≤ 2GB, the same as `doca_sha` library

19.8.5 OpenSSL Command Line Verification

Verify that the engine can be loaded:

```
$ openssl engine dynamic -pre NO_VCHECK:1 -pre SO_PATH:${DOCA_DIR}/infrastructure/doca_sha_offload_engine/
libdoca_sha_offload_engine.so -pre LOAD -vvv -t -c
(dynamic) Dynamic engine loading support
[Success]: SO_PATH:${DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_sha_offload_engine.so
[Success]: LOAD
Loaded: (doca_sha_offload_engine) Openssl SHA offloading engine based on doca_sha
 [SHA1, SHA256, SHA512]
     [ available ]
     set_pci_addr: set the pci address of the doca_sha_engine
          (input flags): STRING
```

- For SHA-1:

```
$ echo "hello world" | openssl dgst -sha1 -engine {DOCA_DIR}/infrastructure/doca_sha_offload_engine/
libdoca_sha_offload_engine.so -engine_impl
```

- For SHA-256:

```
$ echo "hello world" | openssl dgst -sha256 -engine {DOCA_DIR}/infrastructure/doca_sha_offload_engine/
libdoca_sha_offload_engine.so -engine_impl
```

- For SHA-512:

```
$ echo "hello world" | openssl dgst -sha512 -engine {DOCA_DIR}/infrastructure/doca_sha_offload_engine/
libdoca_sha_offload_engine.so -engine_impl
```

19.8.6 OpenSSL Throughput Test

`openssl-speed` is the OpenSSL throughput benchmark tool. For more information, consult official OpenSSL documentation. `doca_sha_offload_engine` throughput can also be measured using `openssl-speed`.

- SHA-1, each job 10000 bytes, using engine:

```
$ openssl speed -evp sha1 -bytes 10000 -elapsed --engine {DOCA_DIR}/infrastructure/doca_sha_offload_engine/
libdoca_sha_offload_engine.so
```

- SHA-256, each job 10000 bytes, using engine, `async_jobs=256`:

```
$ openssl speed -evp sha256 -bytes 10000 -elapsed --engine {DOCA_DIR}/infrastructure/
doca_sha_offload_engine/libdoca_sha_offload_engine.so -async_jobs 256
```

- SHA-512, each job 10000 bytes, using engine, `async_jobs=256`, `threads=8`:

```
$ openssl speed -evp sha512 -bytes 10000 -elapsed --engine {DOCA_DIR}/infrastructure/
doca_sha_offload_engine/libdoca_sha_offload_engine.so -async_jobs 256 -multi 8
```

19.8.7 Using DOCA SHA Offload Engine in OpenSSL Application

More information on the dynamic engine usage can be found in the official OpenSSL documentation.

1. To load the `doca_sha_offload_engine` (optionally, set engine PCIe address):

```
ENGINE *e;
const char *doca_engine_path = "${DOCA_DIR}/infrastructure/doca_sha_offload_engine/
libdoca_sha_offload_engine.so";
const char *default_doca_pci_addr = "03:00.0";
ENGINE_load_dynamic();
e = ENGINE_by_id(doca_engine_path);
ENGINE_ctrl_cmd_string(e, "set_pci_addr", doca_engine_pci_addr, 0);
ENGINE_init(e);
ENGINE_set_default_digests(e);
```

2. To perform SHA calculation by calling the OpenSSL high-level function EVP_XXX:

```
const EVP_MD *evp_md = EVP_sha1();
EVP_MD_CTX *mdctx = EVP_MD_CTX_create();
EVP_DigestInit_ex(mdctx, evp_md, e);
EVP_DigestUpdate(mdctx, msg, msg_len);
EVP_DigestFinal_ex(mdctx, digest, digest_len);
EVP_MD_CTX_destroy(mdctx);
```

3. To unload the engine:

```
ENGINE_unregister_digests(e);
ENGINE_finish(e);
ENGINE_free(e);
```

19.9 NVIDIA BlueField DPU Scalable Function User Guide

This document provides an overview and configuration of scalable functions (sub-functions, or SFs) for NVIDIA® BlueField® DPU.

19.9.1 Introduction

Scalable functions (SFs), or sub-functions, are very similar to virtual functions (VFs) which are part of a Single Root I/O Virtualization (SR-IOV) solution. I/O virtualization is one of the key features used in data centers today. It improves the performance of enterprise servers by giving virtual machines direct access to hardware I/O devices. The SR-IOV specification allows one PCI Express (PCIe) device to present itself to the host as multiple distinct "virtual" devices. This is done with a new PCIe capability structure added to a traditional PCIe function (i.e., a physical function or PF).

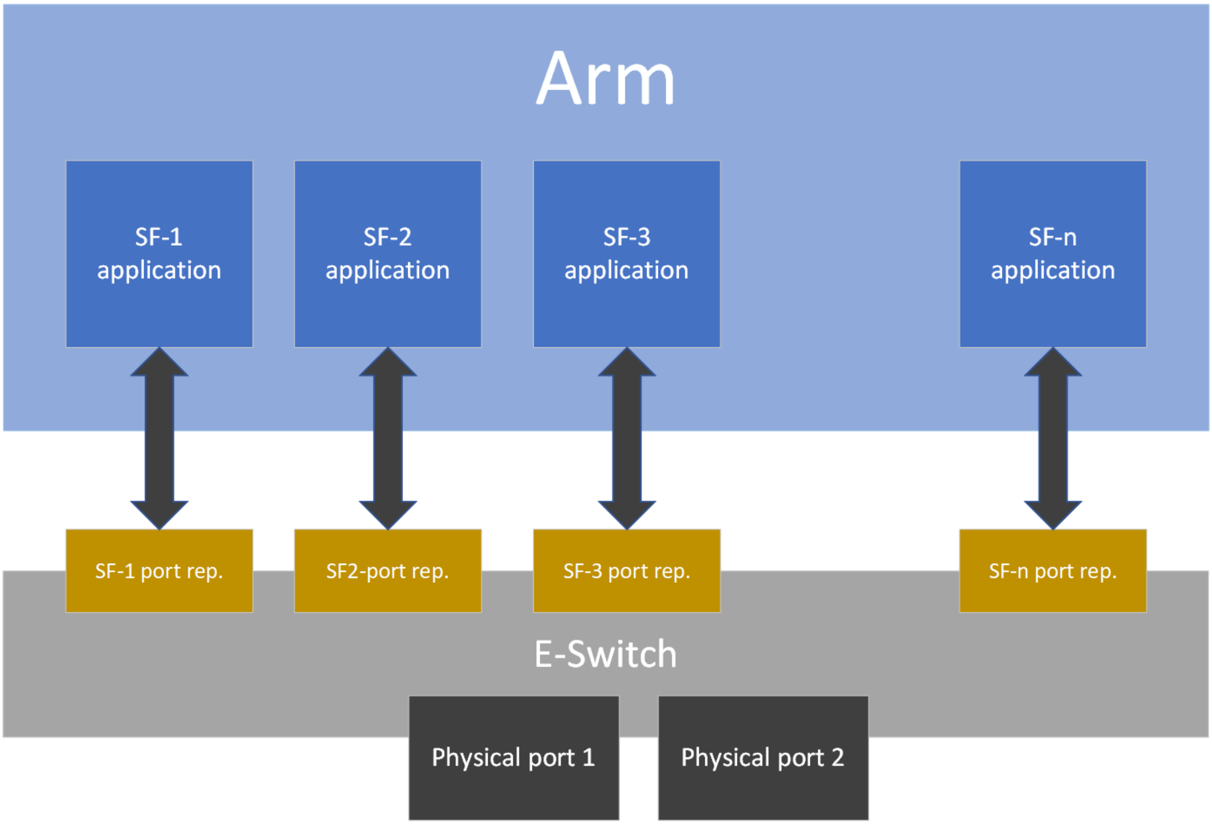
The PF provides control over the creation and allocation of new VFs. VFs share the device's underlying hardware and PCIe. A key feature of the SR-IOV specification is that VFs are very lightweight so that many of them can be implemented in a single device.

To utilize the capabilities of VF in the BlueField, SFs are used. SFs allow support for a larger number of functions than VFs, and more importantly, they allow running multiple services concurrently on the DPU.

An SF is a lightweight function which has a parent PCIe function on which it is deployed. The SF, therefore, has access to the capabilities and resources of its parent PCIe function and has its own function capabilities and its own resources. This means that an SF would also have its own dedicated queues (i.e., txq, rxq).

SFs co-exist with PCIe SR-IOV virtual functions (on the host) but also do not require enabling PCIe SR-IOV.

SFs support E-Switch representation offload like existing PF and VF representors. An SF shares PCIe-level resources with other SFs and/or with its parent PCIe function.



19.9.2 Prerequisites

Refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField related software.

- Make sure your firmware version is 20.30.1004 or higher
- To enable SF support on the device, change the PCIe address for each port:

```
$ mlxconfig -d 0000:03:00.0 s PF_BAR2_ENABLE=0 PER_PF_NUM_SF=1 PF_TOTAL_SF=236
 PF_SF_BAR_SIZE=10

PF_BAR2_ENABLE: if this config is set, then all PFs and ECPFs have the same number of SFs. This should be
off (deprecated).
If set. PF_TOTAL_SF and PF_SF_BAR_SIZE won't work.
PER_PF_NUM_SF: If this config is set, each PF and ECPF configure/control its own number of SFs.
THE ABOVE TWO CONFIGS AFFECS BOTH BF AND HOST, TREAT WITH CARE!
Also, only one of them can be set. It is INVALID to set them both

PF_TOTAL_SF: maximum number of SFs we wish to configure for the given PF/ECPF.
PF_SF_BAR_SIZE: size of each SF at the BAR2. The size is in powers of 2 in KB.
For example: PF_SF_BAR_SIZE=10 means each SF is taking 1MB of the BAR.
             PF_TOTAL_SF=14 means this PCI function can create up to 14 SFs.
             In total: FW will allocate 14MB of BAR2.
```

> ⚠ Perform a BlueField system-level reset for the `mlxconfig` settings to take effect.

19.9.3 SF Configuration

To use an SF, a 3-step setup sequence must be followed first:

1. Create.

2. Configure.
3. Deploy.

Create
Configure
Deploy
Use

These steps can be performed using `mlxdevm` tool.

> ⓘ When working on top of an upstream-based kernel, on which the `mlxdevm` tool is unavailable, please refer to the Upstream Guide on Scalable Functions for instructions on using the `devlink` tool which should be used instead.

19.9.3.1 Configuration Using mlxdevm Tool

1. Create the SF.
   SFs are managed using the `mlxdevm` tool supplied with iproute2 package. The tool is found at `/opt/mellanox/iproute2/sbin/mlxdevm`.
   An SF is created using the `mlxdevm` tool. The SF is created by adding a port of `pcisf` flavor.
   To create an SF port representor, run:

   ```
   /opt/mellanox/iproute2/sbin/mlxdevm port add pci/<pci_address> flavour pcisf pfnum <corresponding pfnum>
   sfnum <sfnum>
   ```

   > ⚠ Each SF must have a unique number (`<sfnum>`).

   For example:

   ```
   /opt/mellanox/iproute2/sbin/mlxdevm port add pci/0000:03:00.0 flavour pcisf pfnum 0 sfnum 4
   ```

   Output example:

   ```
   pci/0000:30:00.0/229409: type eth netdev eth0 flavour pcisf controller 0 pfnum 0 sfnum 4
      function:
       hw_addr 00:00:00:00:00:00 state inactive opstate detached roce true max_uc_macs 128 trust off
   ```

   The number 229409 is required to complete the following two steps (i.e., configuration and deployment).
   `pci/0000:03:00.0/229409` is called the SF index.
   `pci/<pci_address>/<sf_index>` can be replaced with `<representor_name>`. For example:

   ```
   pci/0000:03:00.0/229409 = en3f0pf0sf4
   ```

   To see information about the created SF such as its MAC address, trust mode, or state (active/inactive), run the following command:

```
/opt/mellanox/iproute2/sbin/mlxdevm port show
```

Output example:

```
pci/0000:30:00.0/229409: type eth netdev en3f0pf0sf4 eth0 flavor pcisf controller 0 pfnum 0 sfnum 4
    function:
      hw_addr 00:00:00:00:00:00 state inactive opstate detached roce true max_uc_macs 128 trust off
```

2. Configure the SF.
   A subfunction representor (SF port representor) is created but it is not deployed yet. Users should configure the hardware address (e.g., MAC address), set trust mode to on, and activate the SF before deploying it.
   The following steps can be executed as separate commands (at any order) or combined as one:
   - To configure the hardware address, run:

     ```
     /opt/mellanox/iproute2/sbin/mlxdevm port function set pci/<pci_address>/<sf_index>  hw_addr <MAC
     address>
     ```

   - To set the trust mode to on, run:

     ```
     /opt/mellanox/iproute2/sbin/mlxdevm port function set pci/<pci_address>/<sf_index> trust on
     ```

   - To activate the created SF, run:

     ```
     /opt/mellanox/iproute2/sbin/mlxdevm port function set pci/<pci_address>/<sf_index> state active
     ```

   Alternatively, to configure the MAC address, set trust mode on, and set the state as active, run:

   ```
   /opt/mellanox/iproute2/sbin/mlxdevm port function set pci/<pci_address>/<sf_index>  hw_addr <mac_address>
   trust on state active
   ```

   For example:

   ```
   /opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/229409 hw_addr 00:00:00:00:04:0
    trust on state active
   ```

   > The SF capabilities above must be set before deploying the SF.

3. Deploy the SF.
   To unbind the SF from the default config driver and bind the actual SF driver, run:

   ```
   echo mlx5_core.sf.<next_serial> >  /sys/bus/auxiliary/drivers/mlx5_core.sf_cfg/unbind
   echo mlx5_core.sf.<next_serial>  > /sys/bus/auxiliary/drivers/mlx5_core.sf/bind
   ```

   For example:

   ```
   echo mlx5_core.sf.4  > /sys/bus/auxiliary/drivers/mlx5_core.sf_cfg/unbind
   echo mlx5_core.sf.4  > /sys/bus/auxiliary/drivers/mlx5_core.sf/bind
   ```

`<next_serial>` is a number produced by the firmware when creating the SF (this is the gvmi number of the SF). `mlxdevm` tool when creating the SF. To obtain it, refer to the useful commands provided below.

Useful commands:

- To see the available sub-functions, run:

  ```
  $ devlink dev show
  ```

  For example, if you run the command before creating, configuring, and deploying the SF (using the steps detailed earlier), the output would appear as follows:

  ```
  pci/0000:03:00.0
  pci/0000:03:00.1
  auxiliary/mlx5_core.sf.2
  auxiliary/mlx5_core.sf.3
  ```

  After creating, configuring, and deploying the SF, the output would be:

  ```
  pci/0000:03:00.0
  pci/0000:03:00.1
  auxiliary/mlx5_core.sf.2
  auxiliary/mlx5_core.sf.3
  auxiliary/mlx5_core.sf.4
  ```

  Note that the `<next_serial>` number is 4 for the created SF.

- To see the `sfnum` of each sub-function, run:

  ```
  cat /sys/bus/auxiliary/devices/mlx5_core.sf.<next_serial>/sfnum
  ```

  For example:

  ```
  cat /sys/bus/auxiliary/devices/mlx5_core.sf.4/sfnum
  ```

  Example output:

  ```
  cat /sys/bus/auxiliary/devices/mlx5_core.sf.4/sfnum
  4
  ```

- To remove an SF, you must first make its state inactive and only then remove the SF representor.
  To make the SF's state inactive, run:

  ```
  /opt/mellanox/iproute2/sbin/mlxdevm port function set pci/<pci_address>/<sf_index> state inactive
  ```

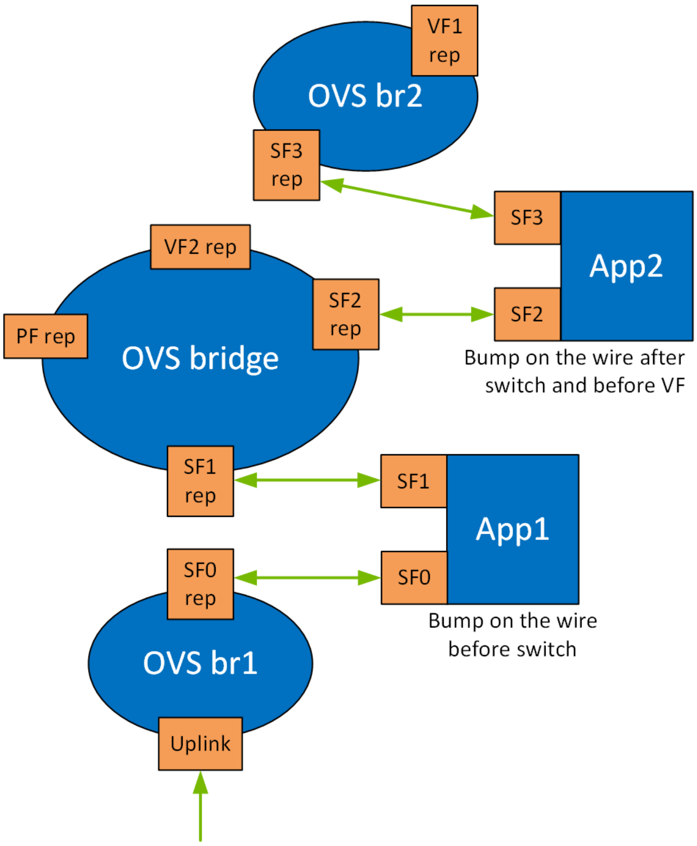
  To delete the SF port representor, run:

  ```
  /opt/mellanox/iproute2/sbin/mlxdevm port del pci/<pci_address>/<sf_index>
  ```

  For example:

  ```
  /opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/229409 state inactive
  /opt/mellanox/iproute2/sbin/mlxdevm port del pci/0000:03:00.0/229409
  ```

4. Use the SF.



Running the application on the DPU requires OVS configuration. By creating SFs, an SF representor for the OVS is also created and named `en3f0pf*sf*`. Therefore, each representor needs to be connected to the correct OVS bridge.

> ⚠ Two SFs related to the same PCIe are necessary for the configuration in the illustration.

The following example configures 2 SFs and adds their representors to the OVS.

a. Create, configure, and deploy the SFs. Run:

```
/opt/mellanox/iproute2/sbin/mlxdevm port add pci/0000:03:00.0 flavour pcisf pfnum 0 sfnum 4
/opt/mellanox/iproute2/sbin/mlxdevm port add pci/0000:03:00.0 flavour pcisf pfnum 0 sfnum 5
```

   Using the command `mlxdevm port show`, you can see the SF indices of the created SFs.

```
/opt/mellanox/iproute2/sbin/mlxdevm port show
```

   Output example:

```
pci/0000:30:00.0/229409: type eth netdev en3f0pf0sf4 flavour pcisf controller 0 pfnum 0 sfnum 4
   function:
    hw_addr 00:00:00:00:00:00 state inactive opstate detached roce true max_uc_macs 128 trust off
pci/0000:30:00.0/229410: type eth netdev en3f0pf0sf5 flavour pcisf controller 0 pfnum 0 sfnum 5
   function:
    hw_addr 00:00:00:00:00:00 state inactive opstate detached roce true max_uc_macs 128 trust off
```

b. Configure the MAC address, set trust mode on, and activate the created SFs:

```
/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/229409 hw_addr
02:25:f2:8d:a2:4c trust on state active
/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/229410 hw_addr
02:25:f2:8d:a2:5c trust on state active
```

Using `ifconfig`, you may see that there are 2 added network interfaces: `en3f0pf0sf4` and `en3f0pf0sf5` for the two respective SF port representors.

c. Delete existing OVS bridges (optional).
   For example, run the following command to delete an OVS bridge called `ovsbr1`:

```
ovs-vsctl del-br ovsbr1
```

d. Create two bridges `sf_bridge1` and `sf_bridge2` and configure them as follows:

```
ovs-vsctl add-br sf_bridge1
ovs-vsctl add-br sf_bridge2
ovs-vsctl add-port sf_bridge1 p0
ovs-vsctl add-port sf_bridge2 pf0hpf
```

e. Add the port representors to the OVS bridges:

```
ovs-vsctl add-port sf_bridge1 en3f0pf0sf4
ovs-vsctl add-port sf_bridge2 en3f0pf0sf5
```

The OVS bridges after adding the SF representors:

```
Bridge sf_bridge1
     Port p0
         Interface p0
     Port sf_bridge1
         Interface sf_bridge1
             type: internal
     Port en3f0pf0sf4
         Interface en3f0pf0sf4
Bridge sf_bridge2
     Port sf_bridge2
         Interface sf_bridge2
             type: internal
     Port en3f0pf0sf5
         Interface en3f0pf0sf5
     Port pf0hpf
         Interface pf0hpf
ovs_version: "2.14.1"
```

⚠ The interface might be down by default. Remember to `ifconfing` the interface to "up" status.

⚠ When deleting the SF port representor, you must also de-attach it from the bridge it is connected to using the command `ovs-vsctl port-del en3f0pf0sf*`. Otherwise, the port representor will still be connected to the bridge but would not be recognizable.

To run the application, use the following command to initialize the SFs during runtime:

```
*Executable_binary* -a auxiliary:mlx5_core.sf.* -a auxiliary:mlx5_core.sf.*
```

For example:

```
doca_<app_name> -a auxiliary:mlx5_core.sf.4 -a auxiliary:mlx5_core.sf.5 -- [application_flags]
```

19.10 NVIDIA TLS Offload Guide

This guide provides an overview and configuration steps of TLS hardware offloading via kernel-TLS, using hardware capabilities of NVIDIA® BlueField® DPU.

19.10.1 Introduction

Transport layer security (TLS) is a cryptographic protocol designed to provide communications security over a computer network. The protocol is widely used in applications such as email, instant messaging, and voice over IP (VoIP), but its use in securing HTTPS remains the most publicly visible.

The TLS protocol aims primarily to provide cryptography, including privacy (confidentiality), integrity, and authenticity using certificates, between two or more communicating computer applications. It runs in the application layer and is itself composed of two layers: the TLS record and the TLS handshake protocols.

TLS works over TCP and consists of 3 phases:

1. Handshake – establishment of a connection
2. Application – sending and receiving encrypted packets
3. Termination – connection termination

19.10.1.1 TLS Handshake

In the handshake phase, the client and server decide on which cipher suites they will use, and exchange keys and certificates according to the following flow:

1. Client hello, provides the server at a minimum with the following:
    - A key exchange algorithm, to determine how symmetric keys are exchanged
    - An authentication or digital signature algorithm, which dictates how server authentication and client authentication (if required) are implemented
    - A bulk encryption cipher, which is used to encrypt the data
    - A hash/MAC (message authentication code) function, which determines how data integrity checks are carried out
    - The version of the protocol it understands
    - The cipher suites it is capable of working with
    - A unique random number, which is important to guard against replay attacks
2. Server hello:
    - Selects a cipher suite
    - Generates its own random number
    - Assigns a session ID to the TLS connection
    - Sends enough information to complete a key exchange—most often, this means sending a certificate including an RSA public key
3. Client:
    - Responsible for completing the key exchange using the information the server provided

At this point, the connection is secured, both sides have agreed on an encryption algorithm, a MAC algorithm, and respective keys.

19.10.1.2 kTLS

The Linux kernel provides TLS offload infrastructure. kTLS (kernel TLS) offloads TLS handling from the user-space to the kernel-space.

kTLS has 3 modes of operation:

- SW – all operation is handled in kernel (i.e., handshake, encryption, decryption)
- HW-offload (the focus of this guide) – handshake and error handling are performed in software. Packets are encrypted/decrypted in hardware. In this case, there is an additional offload from the kernel to the hardware.
- HW-record – all operations are handled by the hardware (driver and firmware) including the handshake. It also handles its own TCP session. This option is currently not supported.

> ⚠ It is important to understand that Rx (receiving) and Tx (sending) can have two separate modes. For example, Rx can be dealt in SW mode but Tx in HW-offload mode (i.e., the hardware will only encrypt but not decrypt).

19.10.1.3 HW-offloading kTLS

In general, the TLS HW-offload performs best and provides optimal value on longer lived sessions, with relatively large packets. Scaling in terms of concurrent connections and connections per second is use-case dependent (e.g., the amount of active concurrent connections from the overall open concurrent connections is material).

It is necessary to learn the following terms before proceeding:

- The transport interface send (TIS) object is responsible for performing all transport-related operations of the transmit side. Messages from Send Queues (SQs) get segmented and transmitted by the TIS including all transport required implications. For example, in the case of a large send offload, the TIS is responsible for the segmentation. The NVIDIA® ConnectX® hardware uses a TIS object to save and access the TLS crypto information and state of an offloaded Tx kTLS connection.
- The transport interface receive (TIR) object is responsible for performing all transport-related operations on the receive side. TIR performs the packet processing and reassembly and is also responsible for demultiplexing the packets into different receive queues (RQs).
- Both TIS and TIR hold the data encryption key (DEK).

19.10.1.3.1 kTLS Offload Flow in High Level

> ⚠ The following flow does not include resync and errors.

1. Establishes a TLS connection with remote host (server or client) by handling a TLS handshake by kernel on current host.
2. Initializes the following state for each connection, Rx and Tx:
    - Crypto secrets (e.g., public key)
    - Crypto processing state

- Record metadata (e.g., record sequence number, offset)
- Expected TCP sequence number

Tx flow:

1. Packets belonging to device offloaded sockets arrive to the kernel and it does not encrypt them.
2. Kernel performs record framing and marks the packet with a connection identifier.
3. Kernel sends packets to the device driver for offloading.
4. Device checks that the sequence number matches the state in the TIS and performs encryption and authentication.

Rx flow:

1. When the connection is created, a HW steering rule is added to steer packets to their respective TIR.
2. Device receives the packet then validates and checks that sequence number of TCP matches the state in the TIR.
3. Performs decryption and authentication, and indicates in the CQE (completion queue entry).
4. Kernel understands that the packet is already decrypted so it does not decrypt it itself and passes it on to the user-space.

19.10.1.3.2 Resync and Error Handling

When the sequence number does not match expectations or if any other error occurs, the hardware gives control back to the SW which handles the problem.

See more about kTLS modes, resync, and error handling in the Linux Kernel documentation.

19.10.2 Prerequisites

All commands in this section should be performed on host (not on BlueField) unless stated otherwise.

19.10.2.1 Checking Hardware Support for Crypto Acceleration

To check if the BlueField or ConnectX have crypto acceleration, run the following command from host:

```
host> mst start # turn on mst driver
host> flint -d <device under /dev/mst/ directory> dc | grep Crypto
```

The output should include `Crypto Enabled`. For example:

```
host> flint -d  /dev/mst/mt41686_pciconf0 dc | grep Crypto
....
;;Description = NVIDIA BlueField-2 E-Series Eng. sample DPU; 200GbE single-port QSFP56; PCIe Gen4 x16; Secure Boot Disabled; Crypto Enabled; 16GB on-board DDR; 1GbE OOB management
....
```

19.10.2.2 Kernel Requirements

- Operating system must be either:
  - FreeBSD 13.0+.
  - A Linux distribution built on Linux kernel version 5.3 or later for Tx support and version 5.9 or later for Rx support. We recommend using the latest version when possible for the best available optimizations.

    > TIS Pool optimization is added to Linux kernel version 6.0. Instead of creating TIS per new connection, unused TIS from previous connection, will be recycled. This will improve Tx connection rate. No further installations required beyond installing the kernel itself.

- Check the current kernel version on the host. Run:

  ```
  host> uname -r
  ```

- The kernel must be configured to support TLS by setting the options `TLS_DEVICE` and `MLX5_TLS` to `y`. To check if TLS is configured, run:

  ```
  host> cat /boot/config-$(uname -r) | grep TLS
  ```

  Example output:

  ```
  host> cat /boot/config-5.4.0-121-generic | grep TLS
  ...
  CONFIG_TLS_DEVICE=y
  CONFIG_MLX5_TLS=y
  ...
  ```

  If the current kernel does not support one of the options, you can change the configurations and recompile, or build a new kernel.

  > Follow the build instructions provided with the kernel provider.

  Schematic flow for building a Linux kernel:

  a. Enter the Linux kernel directory downloaded (usually in `/usr/src/`):

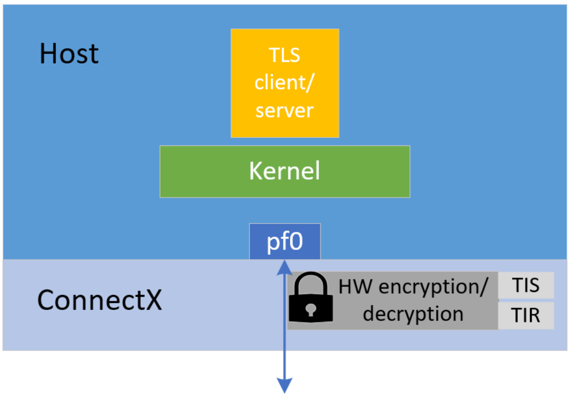
     ```
     host> make menuconfig # Set TLS_DEVICE=y and MLX5_TLS=y in options. Setting location in the menu
     can be found by pressing '/' and typing 'setting'.
     host> make -j <num-of-cores> && make -j <num-of-cores> modules_install && make -j <num of cores>
     install
     ```

  b. Update the grub to the new configured kernel then reboot.

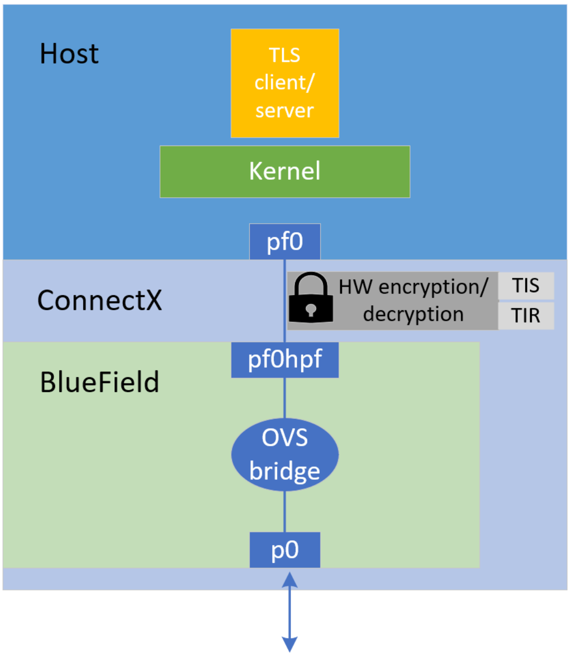
19.10.3 Configurations and Useful Commands

19.10.3.1 TLS Setup

Host + ConnectX Setup



Host + BlueField Setup



19.10.3.2 Finding NVIDIA Interfaces

```
host> mst start        # if mst driver is not loaded.
host> mst status -v
```

NVIDIA's netdev interfaces are found be under the `NET` column.

For example:

```
host> mst status -v
....
DEVICE_TYPE             MST                            PCI       RDMA            NET                       NUMA
BlueField2(rev:0)       /dev/mst/mt41686_pciconf0.1    b1:00.1   mlx5_1          net-ens5f1                 1

BlueField2(rev:0)       /dev/mst/mt41686_pciconf0      b1:00.0   mlx5_0          net-ens5f0                 1
```

In this example, the interfaces `ens5f1` and `ens5f0` are NVIDIA's netdev interfaces.

19.10.3.3 Configuring TLS Offload

- To check if the offload option is on or off, run:

  ```
  host> ethtool -k $iface | grep tls
  ```

  Example output:

```
tls-hw-tx-offload: on
tls-hw-rx-offload: off
tls-hw-record: off [fixed]
```

> ⚠ `tls-hw-record` is not required for the device as kTLS does not support "HW Record" mode.

- To turn Tx offload on or off:

```
host> ethtool -K $iface tls-hw-tx-offload <on | off>
```

- To turn Rx offload on or off:

```
host> ethtool -K $iface tls-hw-rx-offload <on | off>
```

19.10.3.4 Configuring OVS Bridge on BlueField

When the host is connected to a BlueField device, an OVS bridge must be configured on the BlueField so traffic passes bidirectionally from host to uplink. If no OVS bridge is configured, the host is isolated from the network (see diagram above).

> ⚠ On BlueField image version 3.7.0 or higher the default OVS configuration can be used without additional modifications.

To configure the OVS bridge on BlueField, run the following commands on BlueField:

```
dpu> for br in $(ovs-vsctl list-br); do ovs-vsctl del-br $br; done # erasing existing bridges
dpu> ovs-vsctl add-br ovs-br0 && ovs-vsctl add-port ovs-br0 p0 && ovs-vsctl add-port ovs-br0 pf0hpf
dpu> ovs-vsctl add-br ovs-br1 && ovs-vsctl add-port ovs-br1 p1 && ovs-vsctl add-port ovs-br1 pf1hpf
dpu> ovs-vsctl set Open_vSwitch . other_config:hw-offload=true && systemctl restart openvswitch-switch
```

Where `p0` / `p1` are the uplink interfaces and `pf0hpf` / `pf1hpf` are the interfaces facing the host.

19.10.4 Common Use Cases

19.10.4.1 OpenSSL

OpenSSL is an all-around cryptography library that offers open-source application of the TLS protocol. It is the main library for using kTLS and other applications since Nginx depends on it as their base library.

> ⚠ The kTLS and HW offloading do not depend on OpenSSL. Any program that can implement a TLS stack can be run instead. However, because of the vast use of OpenSSL, this guide addresses installation recommendations.

kTLS is supported only in OpenSSL version 3.0.0 or higher, and only on the supported kernel versions. The supported OpenSSL version is available for download from distro packages, or it can be downloaded and compiled from the OpenSSL GitHub.

Many modules depend on OpenSSL. Changing the default version may cause problems. Adding `--prefix=/var/tmp/ssl --openssldir=/var/tmp/ssl` in the `./Configure` command below may prevent the built OpenSSL from becoming the default one used by the system. Make sure the directory of the OpenSSL you build manually is not located in any paths listed in the PATH environment variable.

1. Check the version of the default OpenSSL:

   ```
   host> openssl version
   ```

2. Follow OpenSSL installation instructions from OpenSSL's supplied guides. During the configuration process, make sure to set the `enable-ktls` option before building it by running it from within the OpenSSL directory (works in version 3.0 and higher). For example:

   ```
   host> ./Configure linux-$(uname -p) enable-ktls --prefix=/var/tmp/ssl --openssldir=/var/tmp/ssl # Add "threads" as well for multithread support
   ```

3. Check if kTLS is enabled in OpenSSL by running the following command from within the OpenSSL directory, and check whether `ktls` is listed under `Enabled features`:

   ```
   host> perl configdata.pm --dump | less
   ```

If OpenSSL has been downloaded manually, the OpenSSL executable would be located in the `/<openssl-dir>/apps/` directory. For example, checking the version from within OpenSSL directory is done using the command `./apps/openssl version`.

Installing a new OpenSSL requires recompiling user tools that were configured over OpenSSL (e.g., Nginx).

In OpenSSL's master source code, there is a feature "Support for kTLS Zero-Copy sendfile() on Linux" (Zero-Copy commit). If the Zero-Copy option is set, `SSL_sendfile()` uses the Zero-Copy TX mode which means that the data itself is not copied from the user space to Kernel space. This gives a performance boost when used with kTLS hardware offload. Be aware that invalid TLS records may be transmitted if the file is changed while being sent.

19.10.4.2 Nginx

Nginx is a free and open-source software web server that can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache. Nginx can be configured to depend on OpenSSL library and therefore Nginx could have the great advantages of TLS HW-offload on ConnectX-6 Dx, ConnectX-7 or the DPU.

19.10.4.2.1 Prerequisites

Refer to the OpenSSL section for setting OpenSSL.

19.10.4.2.2 Configuration

1. Install dependencies. For Ubuntu distribution, for example:

```
host> apt install libpcre3 libpcre3-dev
```

2. Clone Nginx's repository and enter directory:

```
host> git clone https://github.com/nginx/nginx.git && cd nginx
```

3. Configure Nginx components to support kTLS:

```
host> ./auto/configure --with-openssl=/<insert_path_to_openssl_directory> --with-debug --with-
http_ssl_module --with-openssl-opt="enable-ktls -DOPENSSL_LINUX_TLS -g3"
```

4. Build Nginx:

```
host> make -j <num of cores> && sudo make -j <num-of-cores> install
```

> ⚠ If `make` fails with a `deprecated openssl functions` error, remove `-Werror` for `CFLAGS` in objs/Makefile and try again.

5. Add the following lines to the end of the `/usr/local/nginx/conf/nginx.conf` file (before the last closing bracket):

```
server {
    listen 443 ssl default_server reuseport;
    server_name localhost;
    root /tmp/nginx/docs/html/;

    include /etc/nginx/default.d/*.conf;
    ssl_certificate /usr/local/nginx/conf/cert.pem;
    ssl_certificate_key /usr/local/nginx/conf/key.pem;
    ssl_ciphers ECDHE-RSA-AES128-GCM-SHA256;
    ssl_protocols TLSv1.2;

 location / {
    index index.html;
 }

 error_page 404 /404.html;
    location = /40x.html {
 }

 error_page 500 502 503 504 /50x.html;
    location = /50x.html {
 }
}
```

6. Notice that the key and certificate of the Nginx server should be located in `/usr/local/nginx/conf/`. Therefore, after creating a key and certificate (as mentioned in section "Adding Certificate and Key") they should be copied to the aforementioned directory:

```
host> cp key.pem /usr/local/nginx/conf/ && cp cert.pem /usr/local/nginx/conf/
```

7. To run Nginx:

```
host> cd nginx && objs/nginx
```

This command starts Nginx Server in the background.

19.10.4.2.3 Stopping Nginx

```
host> pkill nginx
```

19.10.4.2.4 Wrk – Client

A simple client for requesting Nginx's server is "wrk". It can be installed by running the following:

```
host> git clone https://github.com/wg/wrk.git && cd wrk/ && make -j <num-of-cores>
```

19.10.4.2.5 Using Wrk

The following is an example of using the wrk client to request the page `index.html` from the Nginx server in address `4.4.4.4` (run within wrk's directory):

```
host> taskset -c 0 ./wrk -t1 -c10 -d30s https://4.4.4.4:443/index.html
```

> ⚠ Testing the kTLS offload (with or without hardware offload) is in the same manner as mentioned in section "Testing kTLS". TBD

19.10.5 Testing Offload via OpenSSL

This chapter demonstrates how to test the kTLS hardware offload.

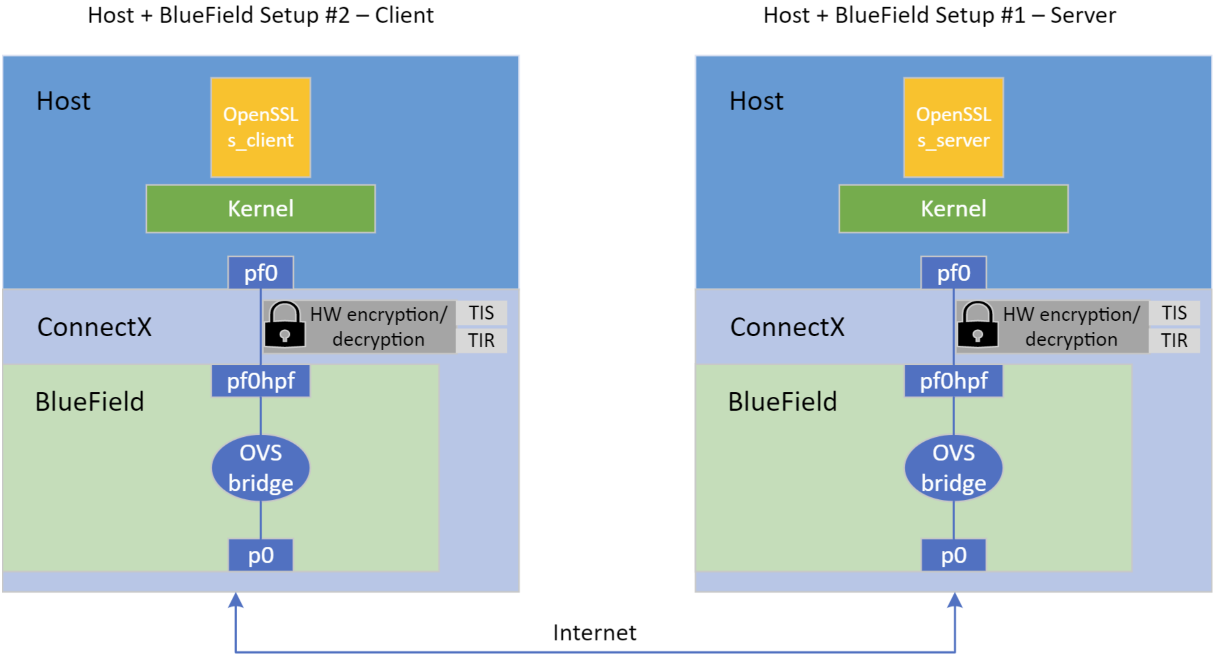
> ⚠ Make sure to refer to section "OpenSSL" before proceeding.

19.10.5.1 TLS Testing Setup

For testing purposes, a server and a client are required. The testing section only tests a single setup of a host and BlueField-2 or a host ConnectX which will participate either as a server or as a client. Setting a back-to-back setup of the same kind and installing the same OpenSSL version can help avoid misconfigurations. Nevertheless, it is required to have the same OpenSSL version on both the client and server.

Make sure the desired kTLS is configured as detailed in section "Configuring TLS Offload". To test hardware offload, make sure `tls-hw-tx-offload` and/or `tls-hw-rx-offload` are on. To test kTLS software mode, make sure to turn them off.

In addition, make sure both hosts (server and client) can communicate bidirectionally through ConnectX or BlueField. One can set the interface that supports the offload (on the host) with an IP, in same subnet. Make sure that when using BlueField, an OVS bridge is set on BlueField as shown in "Configuring OVS Bridge on BlueField".



19.10.5.2 Adding Certificate and Key

The server side should create a certificate and key. The client can also use a certificate, but it is not necessary for this test case. Run the following command in the installed OpenSSL directory and fill in all the requested details:

```
host> openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem -days 365 -nodes
```

The following files are created:

- `key.pem` - private-key file used to generate the CSR and, later, to secure and verify connections using the certificate
- `cert.pem` - certificate signing request (CSR) file used to order your SSL certificate and, later, to encrypt messages that only its corresponding private key can decrypt

⚠ The server side should be run before client side so that client's request are answered by server.

19.10.5.3 Running Server Side

The following example works on OpenSSL version 3.1.0:

```
host> openssl s_server -key key.pem -cert cert.pem -tls1_2 -cipher ECDHE-RSA-AES128-GCM-SHA256 -accept 443 -ktls
```

⚠ Notice the `-ktls` flag.

Refer to official OpenSSL documentation on `s_server` for more information.

In this example, the key and certificate are provided, the cipher suite and TLS version are configured, and the server listens to port 443 and is instructed to use kTLS.

19.10.5.4 Running Client Side

The following example works on OpenSSL version 3.1.0:

```
host> openssl s_client -connect 4.4.4.4:443 -tls1_2
```

Where 4.4.4.4 is the IP of the remote server.

Refer to official OpenSSL documentation on `s_client` for more information.

19.10.5.5 Testing kTLS

After the connection is established (handshake is done), a prompt will open and the user, both on the client and server side, can send a message to other side in a chat-like manner. Messages should appear on the other side once they are received.

The following example checks kTLS hardware offload on the tested setup by tracking Rx and Tx TLS on device counters:

```
host> ethtool -S $iface | grep -i 'tx_tls_encrypted\|rx_tls_decrypted' # ($iface is the interface that offloads)
```

To check kTLS over kernel counters:

```
host> cat /proc/net/tls_stat
```

Output example:

The comments are not part of the output and are added as explanation.

```
host> cat /proc/net/tls_stat
TlsCurrTxSw                             0               # Current Tx connections opened in SW mode
TlsCurrRxSw                             0               # Current Rx connections opened in SW mode
TlsCurrTxDevice                         0               # Current Tx connections opened in HW-offload mode
TlsCurrRxDevice                         0               # Current Rx connections opened in HW-offload mode
TlsTxSw                                 2323828         # Accumulated number of Tx connections opened in SW mode
TlsRxSw                                 1               # Accumulated number of Rx connections opened in SW mode
TlsTxDevice                             12203652        # Accumulated number of Tx connections opened in HW-offload
mode
TlsRxDevice                             0               # Accumulated number of Rx connections opened in HW-offload
mode
TlsDecryptError                         0               # Failed record decryption (e.g., due to incorrect
authentication tag)
TlsRxDeviceResync                       0               # Rx resyncs sent to HW's handling cryptography
TlsDecryptRetry                         0               # All Rx records re-decrypted due to TLS_RX_EXPECT_NO_PAD
misprediction
TlsRxNoPadViolation                     0               # Data Rx records re-decrypted due to TLS_RX_EXPECT_NO_PAD
misprediction
```

More information about the kernel counters can be found in the Statistics section of the Kernel TLS documentation.

19.10.6 Optimizations over kTLS

19.10.6.1 XLIO

The NVIDIA accelerated IO (XLIO) software library boosts the performance of TCP/IP applications based on Nginx (e.g., CDN, DoH) and storage solutions as part of SPDK. XLIO is a user-space software library that exposes standard socket APIs with kernel-bypass architecture, enabling a hardware-based direct copy between an application's user-space memory and the network interface. In particular, XLIO can boost the performance of applications that use the kTLS hardware offload as OpenSSL and Nginx. Read more about XLIO in the NVIDIA XLIO Documentation and XLIO TLS HW-offload over kTLS in the TLS HW Offload section.

Even though XLIO is a kernel-bypass library, the kernel must support kTLS for the bypass to work properly.

19.10.7 Performance Tuning Options

TLS offload performance is related to how fast data can be pumped though the offload engine. In the case of user space applications, certain system configurations can be tuned to optimize its performance.

The following are items that can be tuned for optimal performance, mainly focusing on dedicating the server's work to the NUMA, or non-uniform memory access, cores:

Non-uniform memory access (NUMA) cores are cores with a dedicated memory for each of them, granting cores fast access to their own memory and slower access to others'. This architecture is best for scenarios when it is not necessary to share memory between cores.

1. Add NUMA cores of the NIC to the `isolcpus` kernel boot arguments for each server so that the kernel scheduler does not interrupt the core's running user thread. The following are examples of adding commands:
   a. Identify the NIC NUMA node (see NUMA column):

   ```
   host> mst status -v
   DEVICE_TYPE          MST                      PCI        RDMA       NET                  NUMA
   ConnectX6DX(rev:0)   /dev/mst/mt4125_pciconf0 41:00.0    mlx5_0     net-enp65s0f0np0     1
   ```

   b. Identify the cores of the NIC NUMA node using the NUMA node number acquired from the previous output:

   ```
   host> lscpu | grep "NUMA node1"
   NUMA node1 CPU(s): 1,3,5,7,9,11,13,15,17,19,21,23
   ```

c. Add the NIC NUMA cores to a grub file (e.g., `/etc/default/grub`) by adding the line `GRUB_CMDLINE_LINUX_DEFAULT="isolcpus=<NUMA-cores-from-previous-output>"`. For example:

```
GRUB_CMDLINE_LINUX_DEFAULT="isolcpus=1,3,5,7,9,11,13,15,17,19,21,23"
```

d. Update grub:

```
host> sudo update-grub
```

e. Reboot and check that the configuration has been applied:

```
host> cat /proc/cmdline
BOOT_IMAGE=/vmlinuz-5.10.12 root=UUID=1879326c-711f-4f95-a974-d732af14ef04 ro department=general
user_notifier=dovd osi_string None BOOTIF=01-90-b1-1c-14-02-44 quiet splash
isolcpus=1,3,5,7,9,11,13,15,17,19,21,23
```

2. Disable `irqbalance` service:

> ⚠ Interrupt request, or IRQ, determines what hardware interrupts arrive to each core.

```
host> service irqbalance stop
```

3. Run `set_irq_affinity.sh` to redistribute IRQs to various cores.

> ⚠ The script is within MLNX_OFED's sources:
>
> a. You can find it in MLNX_OFED downloads.
> b. Under "Download" select the correct version and download the "SOURCES" `.tgz` file.
> c. Extract the `.tgz`.
> d. Under SOURCES, extract the `mlnx_tools`.
>
> You should find both files `set_irq_affinity.sh` and its helper file `common_irq_affinity.sh` under the `sbin` directory.

```
host> ./set_irq_affinity.sh <ConnectX_or_BlueField_network_interface>
```

4. Set the interface RSS to the number of cores to use:

```
host> ethtool -X <ConnectX_or_BlueField_network_interface> equal <number_of_isolcpus_cores>
```

5. Set the interface queues for number of cores to use:

```
host> ethtool -L <ConnectX_or_BlueField_network_interface> combined <number_of_isolcpus_cores>
```

6. Pin the application with `taskset` to the `isolcpus` cores used. For example:

```
host> taskset -c 1,3,5,7,9,11,13,15,17,19,21,23 openssl s_server -key key.pem -cert cert.pem -tls1_2
-cipher ECDHE-RSA-AES128-GCM-SHA256 -accept 443 -ktls
```

19.10.8 Additional Reading

- Linux kernel TLS documentation
- Linux kernel TLS offload documentation
- Autonomous NIC offloads research paper

19.11 NVIDIA DOCA Troubleshooting Guide

This guide provides troubleshooting information for common issues and misconfigurations encountered when using DOCA for NVIDIA® BlueField® DPU.

19.11.1 DOCA Infrastructure

19.11.1.1 RShim Troubleshooting and How-Tos

19.11.1.1.1 Another backend already attached

Several generations of BlueField DPUs are equipped with a USB interface in which RShim can be routed, via USB cable, to an external host running Linux and the RShim driver.

In this case, typically following a system reboot, the RShim over USB prevails and the DPU host reports RShim status as "`another backend already attached`". This is correct behavior, since there can only be one RShim backend active at any given time. However, this means that the DPU host does not own RShim access.

To reclaim RShim ownership safely:

1. Stop the RShim driver on the remote Linux. Run:

   ```
   systemctl stop rshim
   systemctl disable rshim
   ```

2. Restart RShim on the DPU host. Run:

   ```
   systemctl enable rshim
   systemctl start rshim
   ```

The "`another backend already attached`" scenario can also be attributed to the RShim backend being owned by the BMC in DPUs with integrated BMC. This is elaborated on further down on this page.

19.11.1.1.2 RShim driver not loading

Verify whether your DPU features an integrated BMC or not. Run:

```
# sudo sudo lspci -s $(sudo lspci -d 15b3: | head -1 | awk '{print $1}') -vvv | grep "Product Name"
```

Example output for DPU with integrated BMC:

```
Product Name: BlueField-2 DPU 25GbE Dual-Port SFP56, integrated BMC, Crypto and Secure Boot Enabled, 16GB on-board
DDR, 1GbE OOB management, Tall Bracket, FHHL
```

If your DPU has an integrated BMC, refer to RShim driver not loading on host with integrated BMC.

If your DPU does not have an integrated BMC, refer to RShim driver not loading on host on DPU without integrated BMC.

19.11.1.1.2.1 RShim driver not loading on DPU with integrated BMC

RShim driver not loading on host

1. Access the BMC via the RJ45 management port of the DPU.
2. Delete RShim on the BMC:

   ```
   systemctl stop rshim
   systemctl disable rshim
   ```
3. Enable RShim on the host:

   ```
   systemctl enable rshim
   systemctl start rshim
   ```
4. Restart RShim service. Run:

   ```
   sudo systemctl restart rshim
   ```

   If RShim service does not launch automatically, run:

   ```
   sudo systemctl status rshim
   ```

   This command is expected to display "`active (running)`".
5. Display the current setting. Run:

   ```
   # cat /dev/rshim<N>/misc | grep DEV_NAME
   DEV_NAME        pcie-0000:04:00.2
   ```

   This output indicates that the RShim service is ready to use.

RShim driver not loading on BMC

1. Verify that the RShim service is not running on host. Run:

   ```
   systemctl status rshim
   ```

   If the output is `active`, then it may be presumed that the host has ownership of the RShim.
2. Delete RShim on the host. Run:

   ```
   systemctl stop rshim
   systemctl disable rshim
   ```
3. Enable RShim on the BMC. Run:

   ```
   systemctl enable rshim
   systemctl start rshim
   ```

4. Display the current setting. Run:

   ```
   # cat /dev/rshim<N>/misc | grep DEV_NAME
   DEV_NAME        usb-1.0
   ```

   This output indicates that the RShim service is ready to use.

19.11.1.1.2.2 RShim driver not loading on host on DPU without integrated BMC

1. Download the suitable DEB/RPM for RShim (management interface for DPU from the host) driver.
2. Reinstall RShim package on the host.
   - For Ubuntu/Debian, run:

     ```
     sudo dpkg --force-all -i rshim-<version>.deb
     ```

   - For RHEL/CentOS, run:

     ```
     sudo rpm -Uhv rshim-<version>.rpm
     ```

3. Restart RShim service. Run:

   ```
   sudo systemctl restart rshim
   ```

   If RShim service does not launch automatically, run:

   ```
   sudo systemctl status rshim
   ```

   This command is expected to display "`active (running)`".
4. Display the current setting. Run:

   ```
   # cat /dev/rshim<N>/misc | grep DEV_NAME
   DEV_NAME        pcie-0000:04:00.2
   ```

   This output indicates that the RShim service is ready to use.

19.11.1.1.3 Change ownership of RShim from NIC BMC to host

1. Verify that your card has BMC. Run the following on the host:

   ```
   # sudo sudo lspci -s $(sudo lspci -d 15b3: | head -1 | awk '{print $1}') -vvv |grep "Product Name"
   Product Name: BlueField-2 DPU 25GbE Dual-Port SFP56, integrated BMC, Crypto and Secure Boot Enabled, 16GB
   on-board DDR, 1GbE OOB management, Tall Bracket, FHHL
   ```

   The product name is supposed to show "integrated BMC".
2. Access the BMC via the RJ45 management port of the DPU.
3. Delete RShim on the BMC:

   ```
   systemctl stop rshim
   systemctl disable rshim
   ```

4. Enable RShim on the host:

   ```
   systemctl enable rshim
   systemctl start rshim
   ```

5. Restart RShim service. Run:

   ```
   sudo systemctl restart rshim
   ```

   If RShim service does not launch automatically, run:

   ```
   sudo systemctl status rshim
   ```

   This command is expected to display "`active (running)`".
6. Display the current setting. Run:

   ```
   # cat /dev/rshim<N>/misc | grep DEV_NAME
   DEV_NAME        pcie-0000:04:00.2
   ```

   This output indicates that the RShim service is ready to use.

19.11.1.2 Connectivity Troubleshooting

19.11.1.2.1 Connection (ssh, screen console) to the DPU is lost

The UART cable in the Accessories Kit (OPN: MBF20-DKIT) can be used to connect to the DPU console and identify the stage at which BlueField is hanging.

Follow this procedure:

1. Connect the UART cable to a USB socket, and find it in your USB devices.

   ```
   sudo lsusb
   Bus 002 Device 003: ID 0403:6001 Future Technology Devices International, Ltd FT232 Serial (UART) IC
   ```

   > ⚠ For more information on the UART connectivity, please refer to the DPU's hardware user guide under Supported Interfaces > Interfaces Detailed Description > NC-SI Management Interface.

   > ⓘ It is good practice to connect the other end of the NC-SI cable to a different host than the one on which the BlueField DPU is installed.

2. Install the minicom application.

OS	Command
CentOS/RHEL	`sudo yum install minicom -y`
Ubuntu/Debian	`sudo apt-get install minicom`

3. Open the minicom application.

```
sudo minicom -s -c on
```

4. Go to "Serial port setup".
5. Enter "F" to change "Hardware Flow control" to NO.
6. Enter "A" and change to `/dev/ttyUSB0` and press Enter.
7. Press ESC.
8. Type on "Save setup as dfl".
9. Exit minicom by pressing Ctrl + a + z.

19.11.1.2.2 Driver not loading in host server

What this looks like in dmsg:

```
[275604.216789] mlx5_core 0000:af:00.1: 63.008 Gb/s available PCIe bandwidth, limited by 8 GT/s x8 link at
0000:ae:00.0 (capable of 126.024 Gb/s with 16 GT/s x8 link)
[275624.187596] mlx5_core 0000:af:00.1: wait_fw_init:316:(pid 943): Waiting for FW initialization, timeout abort in
100s
[275644.152994] mlx5_core 0000:af:00.1: wait_fw_init:316:(pid 943): Waiting for FW initialization, timeout abort in
79s
[275664.118404] mlx5_core 0000:af:00.1: wait_fw_init:316:(pid 943): Waiting for FW initialization, timeout abort in
59s
[275684.083806] mlx5_core 0000:af:00.1: wait_fw_init:316:(pid 943): Waiting for FW initialization, timeout abort in
39s
[275704.049211] mlx5_core 0000:af:00.1: wait_fw_init:316:(pid 943): Waiting for FW initialization, timeout abort in
19s
[275723.954752] mlx5_core 0000:af:00.1: mlx5_function_setup:1237:(pid 943): Firmware over 120000 MS in pre-
initializing state, aborting
[275723.968261] mlx5_core 0000:af:00.1: init_one:1813:(pid 943): mlx5_load_one failed with error code -16
[275723.978578] mlx5_core: probe of 0000:af:00.1 failed with error -16
```

The driver on the host server is dependent on the Arm side. If the driver on Arm is up, then the driver on the host server will also be up.

Please verify that:

- The driver is loaded in the BlueField DPU
- The Arm is booted into OS
- The Arm is not in UEFI Boot Menu
- The Arm is not hanged

Then:

1. Perform graceful shutdown.
2. Power cycle on the host server.
3. If the problem persists, reset nvconfig (`sudo mlxconfig -d /dev/mst/<device> -y reset`) and power cycle the host.

   > ⚠ If your DPU is VPI capable, please be aware that this configuration will reset the link type on the network ports to IB. To change the network port's link type to Ethernet, run:
   >
   > ```
   > sudo mlxconfig -d <device> s LINK_TYPE_P1=2 LINK_TYPE_P2=2
   > ```

4. If this problem persists, please make sure to install the latest bfb image and then restart the driver in host server. Please refer to this page for more information.

19.11.1.2.3 No connectivity between network interfaces of source host to destination device

Verify that the bridge is configured properly on the Arm side.

The following is an example for default configuration:

```
$ sudo ovs-vsctl show
f6740bfb-0312-4cd8-88c0-a9680430924f
    Bridge ovsbr1
        Port pf0sf0
            Interface pf0sf0
        Port p0
            Interface p0
        Port pf0hpf
            Interface pf0hpf
        Port ovsbr1
            Interface ovsbr1
                type: internal
    Bridge ovsbr2
        Port p1
            Interface p1
        Port pf1sf0
            Interface pf1sf0
        Port pf1hpf
            Interface pf1hpf
        Port ovsbr2
            Interface ovsbr2
                type: internal
    ovs_version: "2.14.1"
```

If no bridge configuration exists, refer to "Virtual Switch on DPU".

19.11.1.2.4 Uplink in Arm down while uplink in host server up

Please check that the cables are connected properly into the network ports of the DPU and the peer device.

19.11.1.3 Performance Degradation

Degradation in performance indicates that openvswitch may not be offloaded.

Verify offload state. Run:

```
# ovs-vsctl get Open_vSwitch . other_config:hw-offload
```

- If `hw-offload = true` - Fast Pass is configured (desired result)
- If `hw-offload = false` - Slow Pass is configured

If `hw-offload = false`:

- For RHEL/CentOS, run:

  ```
  # ovs-vsctl set Open_vSwitch . other_config:hw-offload=true;
  # systemctl restart openvswitch;
  # systemctl enable openvswitch;
  ```

- For Ubuntu/Debian, run:

  ```
  # ovs-vsctl set Open_vSwitch . other_config:hw-offload=true;
  # /etc/init.d/openvswitch-switch restart
  ```

19.11.1.4 SR-IOV Troubleshooting

19.11.1.4.1 Unable to create VFs

1. Please make sure that SR-IOV is enabled in BIOS.
2. Verify `SRIOV_EN` is true and `NUM_OF_VFS` bigger than 1. Run:

```
# mlxconfig -d /dev/mst/mt41686_pciconf0 -e q |grep -i "SRIOV_EN\|num_of_vf"
Configurations:           Default         Current         Next Boot
*        NUM_OF_VFS       16              16              16
*        SRIOV_EN         True(1)         True(1)         True(1)
```

3. Verify that `GRUB_CMDLINE_LINUX="iommu=pt intel_iommu=on pci=assign-busses"`.

19.11.1.4.2 No traffic between VF to external host

1. Please verify creation of representors for VFs inside the Bluefield DPU. Run:

```
# /opt/mellanox/iproute2/sbin/rdma link |grep -i up
...
link mlx5_0/2 state ACTIVE physical_state LINK_UP netdev pf0vf0
...
```

2. Make sure the representors of the VFs are added to the bridge. Run:

```
# ovs-vsctl add-port <bridage_name> pf0vf0
```

3. Verify VF configuration. Run:

```
$ ovs-vsctl show
bb993992-7930-4dd2-bc14-73514854b024
    Bridge ovsbr1
        Port pf0vf0
            Interface pf0vf0
                type: internal
        Port pf0hpf
            Interface pf0hpf
        Port pf0sf0
            Interface pf0sf0
        Port p0
            Interface p0
    Bridge ovsbr2
        Port ovsbr2
            Interface ovsbr2
                type: internal
        Port pf1sf0
            Interface pf1sf0
        Port p1
            Interface p1
        Port pf1hpf
            Interface pf1hpf
    ovs_version: "2.14.1"
```

19.11.1.5 eSwitch Troubleshooting

19.11.1.5.1 Unable to configure legacy mode

To set devlink to "Legacy" mode in BlueField, run:

```
# devlink dev eswitch set pci/0000:03:00.0 mode legacy
# devlink dev eswitch set pci/0000:03:00.1 mode legacy
```

Please verify that:

- No virtual functions are open. To verify if VFs are configured, run:

  ```
  # /opt/mellanox/iproute2/sbin/rdma link | grep -i up
  link mlx5_0/2 state ACTIVE physical_state LINK_UP netdev pf0vf0
  link mlx5_1/2 state ACTIVE physical_state LINK_UP netdev pf1vf0
  ```

  If any VFs are configured, destroy them by running:

  ```
  # echo 0 > /sys/class/infiniband/mlx5_0/device/mlx5_num_vfs
  # echo 0 > /sys/class/infiniband/mlx5_1/device/mlx5_num_vfs
  ```

- If any SFs are configured, delete them by running:

  ```
  /sbin/mlnx-sf -a delete --sfindex <SF-Index>
  ```

> ⚠ You may retrieve the `<SF-Index>` of the currently installed SFs by running:
>
> ```
> # mlnx-sf -a show
>
> SF Index: pci/0000:03:00.0/229408
>   Parent PCI dev: 0000:03:00.0
>   Representor netdev: en3f0pf0sf0
>   Function HWADDR: 02:61:f6:21:32:8c
>   Auxiliary device: mlx5_core.sf.2
>     netdev: enp3s0f0s0
>     RDMA dev: mlx5_2
>
> SF Index: pci/0000:03:00.1/294944
>   Parent PCI dev: 0000:03:00.1
>   Representor netdev: en3f1pf1sf0
>   Function HWADDR: 02:30:13:6a:2d:2c
>   Auxiliary device: mlx5_core.sf.3
>     netdev: enp3s0f1s0
>     RDMA dev: mlx5_3
> ```
>
> Pay attention to the SF Index values. For example:
>
> ```
> /sbin/mlnx-sf -a delete --sfindex pci/0000:03:00.0/229408
> /sbin/mlnx-sf -a delete --sfindex pci/0000:03:00.1/294944
> ```

If the error "`Error: mlx5_core: Can't change mode when flows are configured`" is encountered while trying to configure legacy mode, please make sure that

1. Any configured SFs are deleted (see above for commands).
2. Shut down the links of all interfaces, delete any `ip xfrm` rules, delete any configured OVS flows, and stop openvswitch service. Run:

   ```
   ip link set dev p0 down
   ip link set dev p1 down
   ip link set dev pf0hpf down
   ip link set dev pf1hpf down
   ip link set dev vxlan_sys_4789 down

   ip x s f ;
   ip x p f ;

   tc filter del dev p0 ingress
   tc filter del dev p1 ingress
   tc qdisc show dev p0
   tc qdisc show dev p1
   tc qdisc del dev p0 ingress
   tc qdisc del dev p1 ingress
   tc qdisc show dev p0
   tc qdisc show dev p1

   systemctl stop openvswitch-switch
   ```

19.11.1.5.2 DPU appears as two interfaces

What this looks like:

```
# sudo /opt/mellanox/iproute2/sbin/rdma link
link mlx5_0/1 state ACTIVE physical_state LINK_UP netdev p0
link mlx5_1/1 state ACTIVE physical_state LINK_UP netdev p1
```

- Check if you are working in legacy mode.

  ```
  # devlink dev eswitch show pci/0000:03:00.<0|1>
  ```

  If the following line is printed, this means that you are working in legacy mode:

  ```
  pci/0000:03:00.<0|1>: mode legacy inline-mode none encap enable
  ```

  Please configure the DPU to work in switchdev mode. Run:

  ```
  devlink dev eswitch set pci/0000:03:00.<0|1> mode switchdev
  ```

- Check if you are working in separated mode:

  ```
  # mlxconfig -d /dev/mst/mt41686_pciconf0 q | grep -i cpu
  * INTERNAL_CPU_MODEL SEPERATED_HOST(0)
  ```

  Please configure the DPU to work in embedded mode. Run:

  ```
  # mlxconfig -d /dev/mst/mt41686_pciconf0 s INTERNAL_CPU_MODEL=1
  ```

19.11.2 DOCA Applications

This chapter deals with troubleshooting issues related to DOCA applications.

19.11.2.1 EAL Initialization Failure

EAL initialization failure is a common error that may appear while running various DPDK-related applications.

19.11.2.1.1 Error

The error looks like this:

```
[DOCA][ERR][NUTILS]: EAL initialization failed
```

There may be many causes for this error. Some of them are as follows:

- The application requires huge pages and none were allocated
- The application requires root privileges to run and it was run without elevated privileges

19.11.2.1.2 Solution

The following solutions are respective to the possible causes listed above:

- Allocate huge pages. For example, run (on the host or the DPU, depending on where you are running the application):

```
sudo echo 2048 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

- Run the application using sudo (or as root):

```
sudo <run_command>
```

19.11.2.2 Ring Memory Issue

This is a common memory issue when running application on the host.

19.11.2.2.1 Error

The error looks as follows:

```
RING: Cannot reserve memory
[13:00:57:290147][DOCA][ERR][UFLTR::Core:156]: DPI init failed
```

The most common cause for this error is lack of memory (i.e., not enough huge pages per worker threads).

19.11.2.2.2 Solution

Possible solutions:

- Recommended: Increase the amount of allocated huge pages. Instructions for allocating huge pages can be found here.

> For an SFT application with 64 cores, it is recommended to increase the allocation from 2048 to 8192.

- Alternatively, one can also limit the number of cores used by the application:
  - `-c <core-mask>` - Set the hexadecimal bitmask of the cores to run on.
  - `-l <core-list>` - list of cores to run on.
- For example:

```
./doca_<app_name> -a 3b:00.3 -a 3b:00.4 -l 0-64 -- -l 60
```

19.11.2.3 DOCA Apps Using DPDK in Parallel Issue

When running two DOCA apps in parallel that use DPDK, the first app runs but the second one fails.

19.11.2.3.1 Error

The following error is received:

```
Failed to start URL Filter with output: EAL: Detected 16 lcore(s)
EAL: Detected 1 NUMA nodes
EAL: RTE Version: 'MLNX_DPDK 20.11.4.0.3' EAL: Detected shared linkage of DPDK
EAL: Cannot create lock on '/var/run/dpdk/rte/config'. Is another primary process running?
EAL: FATAL: Cannot init config
EAL: Cannot init config
[15:01:57:246339][DOCA][ERR][NUTILS]: EAL initialization failed
```

The cause of the error is that the second application is using `/var/run/dpdk/rte/config` when the first application is already using it.

19.11.2.3.2 Solution

To run two applications in parallel, the second application needs to be run with DPDK EAL option `--file-prefix <name>`.

In this example, after running the first application (without adding the `eal` option), to run the second with the EAL option. Run:

```
./doca_<app_name> --file-prefix second -a 0000:01:00.6,sft_en=1 -a 0000:01:00.7,sft_en=1 -v -c 0xff -- -l 60
```

19.11.2.4 Failure to Set Huge Pages

When trying to configure the huge pages from an unprivileged user account, a permission error is raised.

19.11.2.4.1 Error

Configuring the huge pages results in the following error:

```
$ sudo echo 600 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
-bash: /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages: Permission denied
```

19.11.2.4.2 Solution

Using `sudo` with `echo` works differently than users usually expect. Instead, the command should be as follows:

```
$ echo '600' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

19.11.3 DOCA Libraries

This chapter deals with troubleshooting issues related to DOCA libraries.

19.11.3.1 DOCA Flow Error

When trying to add new entry to the pipe, an error is received.

19.11.3.1.1 Error

The error happens after trying to add new entry function. The error message would look similar to the following:

```
mlx5_common: Failed to create TIR using DevX
mlx5_net: Port 0 cannot create DevX TIR.
[10:26:39:622581][DOCA][ERR][dpdk_engine]: create pipe entry fail on index:1, error=Port 0 create flow fail, type 1
 message: cannot get hash queue, type=8
```

The issue here seems to be caused by SF/ports configuration.

19.11.3.1.2 Solution

To fix the issue, apply the following commands on the DPU:

```
dpu# /opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode legacy
dpu# /opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode legacy
dpu# echo none > /sys/class/net/p0/compat/devlink/encap
dpu# echo none > /sys/class/net/p1/compat/devlink/encap
dpu# /opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode switchdev
dpu# /opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode switchdeV
```

19.11.4 DOCA SDK Compilation

This chapter deals with troubleshooting issues related to compiling DOCA-based programs to use the DOCA SDK (e.g., missing dependencies).

19.11.4.1 Meson Complains About Missing Dependencies

As part of DOCA's installation, a basic set of environment variables are defined so that projects (such as DOCA applications) could easily compile against the DOCA SDK, and to allow users easy access to the various DOCA tools. In addition, the set of DOCA applications sometimes rely on various 3rd party dependencies, some of which require specific environment variables so to be correctly found by the compilation environment (meson).

19.11.4.1.1 Error

There are multiple forms this error may appear in, such as:

- DOCA libraries are missing:

  ```
  Run-time dependency doca-common found: NO (tried pkgconfig)

  meson.build:230:0: ERROR: Dependency "doca-common" not found, tried pkgconfig
  ```

- DPDK definitions are missing:

  ```
  Dependency libdpdk found: NO (tried pkgconfig and cmake)
  meson.build:41:1: ERROR:  Dependency "libdpdk" not found, tried pkgconfig and cmake
  ```

- mpicc is missing for DPA All to All application:

```
====================
Skipped Applications
====================
 * dpa_all_to_all: Missing mpicc
```

19.11.4.1.2 Solution

All the dependencies mentioned above are installed as part of DOCA's installation, and yet it is recommended to check that the packages themselves were installed correctly. The packages that install each dependency define the environment variables needed by it, and apply these settings per user login session:

- If DOCA was just installed (on the host or DPU), user session restart is required to apply these definitions (i.e., log off and log in).
- It is important to compile DOCA using the same logged in user. Logging as `ubuntu` and using `sudo su`, or compiling using `sudo`, will not work.

If restarting the user session is not possible (e.g., automated non-interactive session), the following is a list of the needed environment variables:

All the following examples use the required environment variables for the DPU. For the host, the values should be adjusted accordingly (`aarch64` is for the DPU and `x86` is for the host): `aarch64-linux-gnu → x86_64-linux-gnu`.

It is recommended to define all of the following settings so as to not have to remember which DOCA application requires which module (whether DPDK, FlexIO, etc).

DOCA Tools:

- For Ubuntu:

```
export PATH=${PATH}:/opt/mellanox/doca/tools
```

- For CentOS:

```
export PATH=${PATH}:/opt/mellanox/doca/tools
```

DOCA Applications:

- For Ubuntu and CentOS

```
export PATH=${PATH}:/usr/mpi/gcc/openmpi-4.1.7a1/bin
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/mpi/gcc/openmpi-4.1.7a1/lib
```

DPDK:

- For Ubuntu:

```
export PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/opt/mellanox/dpdk/lib/aarch64-linux-gnu/pkgconfig
```

- For CentOS:

```
export PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/opt/mellanox/dpdk/lib64/pkgconfig
```

FlexIO:

- For Ubuntu:

```
export PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/opt/mellanox/flexio/lib/pkgconfig
```

- For CentOS:

```
export PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/opt/mellanox/flexio/lib/pkgconfig
```

CollectX:

- For Ubuntu and CentOS:

```
export PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/opt/mellanox/collectx/lib/aarch64-linux-gnu/pkgconfig
```

19.11.4.2 Meson Complains About Permissions

Our guides for compiling the reference samples and applications of DOCA's SDK are using the meson build system.

19.11.4.2.1 Error

A permission error is encountered when trying to reuse a build directory from a previous build:

```
ubuntu@localhost:/opt/mellanox/doca/samples/doca_flow/flow_acl$ meson /tmp/build
Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/mesonbuild/mesonmain.py", line 146, in run
    return options.run_func(options)
  File "/usr/lib/python3/dist-packages/mesonbuild/msetup.py", line 294, in run
    app.generate()
  File "/usr/lib/python3/dist-packages/mesonbuild/msetup.py", line 181, in generate
    mlog.initialize(env.get_log_dir(), self.options.fatal_warnings)
  File "/usr/lib/python3/dist-packages/mesonbuild/mlog.py", line 103, in initialize
    log_file = open(os.path.join(logdir, log_fname), 'w', encoding='utf-8')
PermissionError: [Errno 13] Permission denied: '/tmp/build/meson-logs/meson-log.txt'
```

19.11.4.2.2 Solution

Per the meson build instructions, the user can choose any write-accessible directory to be used as the build directory, using the following syntax:

```
meson <build-dir>
```

When reusing a build directory, it is best to ensure that the existing directory was created by a user with the same permissions, and only then do one of the following:

- Removing the old build directory:

```
rm -rf /tmp/build
```

- Reconfiguring the build directory:

```
meson --reconfigure /tmp/build
```

The above error is an indication that the build directory was created by a different user, and that our user doesn't have permissions to use it. In such cases, it is best to choose a different build directory, in a directory that our user has write-access to. For example:

```
meson /tmp/build2
```

19.11.4.3 Static Compilation on CentOS: Undefined References to C++

When statically compiling against the DOCA SDK on RHEL 7.x machines, there could be a conflict between the libstdc++ version available out-of-the-box and the one used when building DOCA's SDK libraries.

19.11.4.3.1 Error

There are multiple forms this error may appear in, such as:

```
$ cc test.o -o test_out `pkg-config --libs --static doca`
/opt/mellanox/doca/lib64/libdoca_common.a(doca_common_core_src_doca_dev.cpp.o): In function
`doca_devinfo_rep_list_create':
(.text.experimental+0x2193): undefined reference to `__cxa_throw_bad_array_new_length' /opt/mellanox/doca/lib64/
libdoca_common.a(doca_common_core_src_doca_dev.cpp.o): In function `doca_devinfo_rep_list_create':
(.text.experimental+0x2198): undefined reference to `__cxa_throw_bad_array_new_length' collect2: error: ld returned
1 exit status
```

19.11.4.3.2 Solution

Upgrading the `devtoolset` on the machine to the one used when building the DOCA SDK resolves the undefined references issue:

```
$ sudo yum install epel-release
$ sudo yum install centos-release-scl-rh
$ sudo yum install devtoolset-8
# This will enable the use of devtoolset-8 to the *current* bash session
$ source /opt/rh/devtoolset-8/enable
```

19.11.4.4 Static Compilation on CentOS: Unresolved Symbols

When statically compiling against the DOCA SDK on RHEL 7.x machines, a known issue in the default pkg-config version (0.27) causes a linking error.

19.11.4.4.1 Error

There are multiple forms this error may appear in. For example:

```
$ cc test.o -o test_out 'pkg-config --libs --static doca' ...
/opt/mellanox/dpdk/lib64/librte_net_mlx5.a(net_mlx5_mlx5_sft.c.o): In function 'mlx5_sft_start':
mlx5_sft.c:(.text+0x1827): undefined reference to 'mlx5_malloc' ...
```

19.11.4.4.2 Solution

Use an updated version of pkg-config or pkgconf instead when building applications (as is recommended in DPDK's compilation instructions).

19.11.5 Cross-compiling DOCA and CUDA

This chapter deals with troubleshooting issues related to DOCA-CUDA cross-compilation.

19.11.5.1 Application Build Error

When trying to build with meson, an architecture-related error is received.

19.11.5.1.1 Error

The error may happen when trying to build DOCA or DOCA-CUDA applications.

```
cc1: error: unknown value 'corei7' for -march
```

It indicates that some dependency (usually libdpdk) is not taken from the host machine (i.e., the machine the executable file should be running on). This dependency should be taken from the Arm dependencies directories (the path is specified in the cross file) but is skipped if the host's PKG_CONFIG_PATH environment variable is used instead.

19.11.5.1.2 Solution

Make sure that the cross file contains the following PKG_CONFIG related definitions:

```
[built-in options]
pkg_config_path = '' [properties]
pkg_config_libdir = … // Some content here
```

In addition, verify that pkg_config_libdir properly points to all pkgconfig-related directories under your cross-build root directory, and that the dependency reported in the error is not missing.

19.11.6 DOCA Services (Containers)

This section deals with troubleshooting issues related to DOCA-based containers.

19.11.6.1 YAML Syntax Error #1

When deploying the container using the respective YAML file, the pod fails to start.

19.11.6.1.1 Error

The error may happen after modifying a service's YAML file, or after copying an example YAML file from one of the guides.

```
$ crictl pods
POD ID              CREATED             STATE               NAME                NAMESPACE           ATTEMPT
RUNTIME
$ journalctl -u kubelet
...
Oct 06 12:10:08 dpu-name kubelet[3260]: E1006 12:10:08.552306    3260 file.go:108] "Unable to process watch event"
 err="can't process config file \"/etc/kubelet.d/file_name.yaml\": invalid pod: [metadata.name: Invalid value: \"-
dpu-name\": a lowercase RFC 1123 subdomain must consist of lower case alphanumeric characters, '-' or '.', and must
start and end with an alphanumeric character (e.g. 'example.com', regex used for validation is '[a-z0-9]([-a-
z0-9]*[a-z0-9])?(\\.[a-z0-9]([-a-z0-9]*[a-z0-9])?)*') spec.containers: Required value]"
...
```

This indicates that some of the fields in the YAML file fail to comply with RFC 1123.

19.11.6.1.2 Solution

Both the pod name and container name have a strict alphabet (RFC 1123) restrictions. This means that users can only use dash ("-") and not underscore ("_") as the latter is an illegal character and cannot be used in the pod/container name. However, for the container's image name, use underscore ("_") instead of dash ("-") to help differentiate the two.

19.11.6.2 YAML Syntax Error #2

When deploying the container using the respective YAML file, the pod fails to start.

19.11.6.2.1 Error

The error may happen after modifying a service's YAML file, or after copying an example YAML file from one of the guides.

> ⚠ This error can occur when there is a whitespace issue if the YAML file has been copied from one of the guides causing a formatting mistake. It is important to ensure that the space characters used in the files are indeed spaces (" ") and not some other whitespace character.

```
$ crictl pods
POD ID              CREATED             STATE               NAME                NAMESPACE           ATTEMPT
RUNTIME
$ journalctl -u kubelet
...
Oct 04 12:35:58 dpu-name kubelet[3046]: E1004 12:35:58.744406    3046 file.go:187] "Could not process manifest
file" err="/etc/kubelet.d/file_name.yaml: couldn't parse as pod(yaml: line 48: did not find expected '-'
indicator), please check config file" path="/etc/kubelet.d/file_name.yaml"
...
```

This indicates that there is a probable indentation issue in line 48 or in the line above it.

19.11.6.2.2 Solution

Go over the file and make sure that the file only uses spaces (" ") for indentations (2 per indent). Using any other number of spaces causes undefined behavior.

19.11.6.3 Missing Huge Pages

When deploying the container using the respective YAML file, the pod fails to start.

19.11.6.3.1 Error

```
$ crictl pods
POD ID              CREATED             STATE               NAME                NAMESPACE           ATTEMPT
RUNTIME
$ journalctl -u kubelet
...
Oct 04 12:39:41 dpu-name kubelet[3046]: I1004 12:39:41.643621    3046 predicate.go:103] "Failed to admit pod,
unexpected error while attempting to recover from admission failure" pod="default/file_name" err="preemption: error
finding a set of pods to preempt: no set of running pods found to reclaim resources: [(res: hugepages-2Mi, q:
1021313024), ]"
...
```

This error indicates that the service expected 1GB (1021313024 bytes) of huge pages of size 2MB per page, and could not find them.

19.11.6.3.2 Solution

1. Remove the YAML file of the service from the deployment directory (`/etc/kubelet.d`).
2. Allocate huge pages as described in the service's prerequisites steps:
   a. Make sure that the huge pages are allocated as required per the desired container.
   b. Both the amount and size of the pages are important and must match precisely.
3. Restart the container infrastructure daemons:

   ```
   sudo systemctl restart kubelet.service
   sudo systemctl restart containerd.service
   ```

4. Once the above operations are completed successfully, the container could be deployed (YAML can be copied to `/etc/kubelet.d`).

19.11.6.4 Failed to Reserve Sandbox Name

After rebooting the DPU, the respective pods start. However, the containers repeatedly fail to spawn and their "attempt" counter does not increment.

19.11.6.4.1 Error

```
$ crictl pods
POD ID              CREATED                  STATE               NAME
NAMESPACE           ATTEMPT             RUNTIME
bee147792a85b       Less than a second ago   Ready               doca-hbn-service-my-dpu                   default
0                   (default)
ea66ee46e75a5       Less than a second ago   Ready               doca-telemetry-service-my-dpu             default
0                   (default)

$ crictl ps -a
CONTAINER           IMAGE               CREATED                  STATE               NAME
ATTEMPT             POD ID              POD
6a35c025a3590       ce4c0cafd583e       Less than a second ago   Exited              init-sfs                   0
bee147792a85b       doca-hbn-service-my-dpu
9048f4c7b8f3c       095a5833a3f80       Less than a second ago   Running             doca-telemetry-service     0
ea66ee46e75a5       doca-telemetry-service-my-dpu
059d0aa8a3199       095a5833a3f80       Less than a second ago   Exited              init-telemetry-service     0
ea66ee46e75a5       doca-telemetry-service-my-dpu
bcfbe536271ea       ce4c0cafd583e       33 seconds ago           Running             init-sfs                   1
bee147792a85b       doca-hbn-service-my-dpu

$ journalctl -u containerd
...
"2023-11-28T08:43:42.408173348+02:00" level=error msg="RunPodSandbox for &PodSandboxMetadata{Name:doca-hbn-service-
my-dpu,Uid:823b1ad0e241a10475edde26e905856b,Namespace:default,Attempt:0,} failed, error" error="failed to reserve
sandbox name \"doca-hbn-service-my-dpu_default_823b1ad0e241a10475edde26e905856b_0\": name \"doca-hbn-service-my-
dpu_default_823b1ad0e241a10475edde26e905856b_0\" is reserved for
\"bee147792a85bc23a3629a9fcd0a5f388794f6b67ef552c959d4d5e49d04f5b2\""
...
```

This error indicates that there has been some collision with prior instances of the `doca-hbn-service` container, probably pre-reboot.

> This issue indicates irregularities in the time of the machine, and usually that the DPU's time pre-reboot was later than the time post-reboot. This leads to bugs in the recovery of the container infrastructure daemons. It is of utmost importance that the time of the system does not jump backwards.

19.11.6.4.2 Solution

1. Remove all YAML files from the deployment directory (`/etc/kubelet.d`).
2. Stop all pods:

   ```
   sudo crictl stopp $(crictl pods | tail -n +2 | awk '{ print $1 }')
   ```

3. Clear all containers:

   ```
   sudo ctr -n k8s.io container rm $(ctr -n k8s.io container ls | tail -n +2 | awk '{ print $1 }')
   ```

4. Make sure the system's time is correct, and adjust it if needed:

   ```
   date
   ```

5. Restart the container infrastructure daemons:

   ```
   sudo systemctl restart kubelet.service
   sudo systemctl restart containerd.service
   ```

6. Once the above operations are completed successfully, the container could be deployed (YAML can be copied to `/etc/kubelet.d`).

19.11.7 Collecting DOCA Logs for NVIDIA Inspection

To help NVIDIA Support investigate issues customers may encounter, NVIDIA strongly recommends collecting all relevant logs using the `doca-sosreport` tool. This tool includes plugins to gather logs from various NVIDIA products and more.

On the device customers are facing issues with, run the following command with superuser privileges:

```
sudo sos report
```

This creates a tar file in the `/tmp` directory. When opening a support ticket for NVIDIA Support, make sure to upload this tar file.

If there is private information that you wish not to include in the tar file, extract the file and edit or remove any sensitive information, then create a new tar package.

> For more options on running the tool, refer to the tool's readme.

19.11.8 NVIDIA BlueField Reset and Reboot Procedures

Unable to render include or excerpt-include. Could not retrieve page.

19.12 NVIDIA DOCA Virtual Functions User Guide

This guide provides an overview and configuration of virtual functions for NVIDIA® BlueField® and demonstrates a use case for running the DOCA applications over x86 host.

19.12.1 Introduction

Single root IO virtualization (SR-IOV) is a technology that allows a physical PCIe device to present itself multiple times through the PCIe bus. This technology enables multiple virtual instances of the device with separate resources. NVIDIA adapters are able to expose virtual instances or functions (VFs) for each port individually. These virtual functions can then be provisioned separately.

Each VF can be seen as an additional device connected to the physical interface or function (PF). It shares the same resources with the PF, and its number of ports equals those of the PF.

SR-IOV is commonly used in conjunction with an SR-IOV-enabled hypervisor to provide virtual machines direct hardware access to network resources, thereby increasing its performance.

There are several benefits to running applications on the host. For example, one may want to utilize a strong and high-resource host machine, or to start DOCA integration on the host before offloading it to the BlueField DPU.

The configuration in this document allows the entire application to run on the host's memory, while utilizing the HW accelerators on BlueField.

When VFs are enabled on the host, VF representors are visible on the Arm side which can be bridged to corresponding PF representors (e.g., the uplink representor and the host representor). This allows the application to only scan traffic forwarded to the VFs as configured by the user and to behave as a simple "bump-on-the-wire". DOCA installed on the host allows access to the hardware capabilities of the BlueField DPU without comprising features which use HW offload/steering elements embedded inside the eSwitch.

19.12.2 Prerequisites

To run all the reference applications over the host, you must install the host DOCA package. Refer to the NVIDIA DOCA Installation Guide for Linux for more information on host installation.
VFs must be configured as trusted for the hardware jump action to work as intended. The following steps configure "trusted" mode for VFs:

1. Delete all existing VFs
    a. To delete all VFs on a PF run the following on the host:

    ```
    $ echo 0 > /sys/class/net/<physical_function>/device/sriov_numvfs
    ```

    For example:

```
$ echo 0 > /sys/class/net/ens1f0/device/sriov_numvfs
```

2. Delete all existing SFs.

   > Refer to NVIDIA BlueField DPU Scalable Function User Guide for instructions on deleting SFs.

3. Stop the main driver on the host:

   ```
   /etc/init.d/openibd stop
   ```

4. Before creating the VFs, set them to "trusted" mode on the device by running the following commands on the DPU side.
   a. Setting VFs on port 0:

      ```
      $ mlxreg -d /dev/mst/mt41686_pciconf0 --reg_id 0xc007 --reg_len 0x40 --indexes
      "0x0.0:32=0x80000000" --yes --set "0x4.0:32=0x1"
      ```

   b. Setting VFs on port 1:

      ```
      $ mlxreg -d /dev/mst/mt41686_pciconf0.1 --reg_id 0xc007 --reg_len 0x40 --indexes
      "0x0.0:32=0x80000000" --yes --set "0x4.0:32=0x1"
      ```

   > These commands set trusted mode for all created VFs/SFs after their execution on the DPU.

   > Setting trusted mode should be performed once per reboot.

5. Restart the main driver on the host by running the following command:

   ```
   /etc/init.d/openibd restart
   ```

19.12.3 VF Creation

1. Make sure mst driver is running:

   ```
   host $ mst status
   ```

   If it is not loaded, run:

   ```
   host $ mst start
   ```

2. Enable SR-IOV. Run:

   ```
   host $ mlxconfig -y -d /dev/mst/mt41686_pciconf0 s SRIOV_EN=1
   ```

3. Set number of VFs. Run:

   ```
   host $ mlxconfig -y -d /dev/mst/mt41686_pciconf0 s NUM_OF_VFS=X
   ```

⚠ Perform a BlueField system reboot for the `mlxconfig` settings to take effect.

```
host $ echo X > /sys/class/net/<physical_function>/device/sriov_numvfs
```

For example:

```
host $ mlxconfig -y -d /dev/mst/mt41686_pciconf0 s NUM_OF_VFS=2
host $ reboot
host $ echo 2 > /sys/class/net/ens1f0/device/sriov_numvfs
```

After enabling VF, the representor appears on the DPU. The function itself is seen at the x86 side.

4. To verify that the VFs have been created. Run:

```
$ lspci | grep Virtual
b1:00.3 Ethernet controller: Mellanox Technologies ConnectX Family mlx5Gen Virtual Function (rev 01)
b1:00.4 Ethernet controller: Mellanox Technologies ConnectX Family mlx5Gen Virtual Function (rev 01)
b1:01.3 Ethernet controller: Mellanox Technologies ConnectX Family mlx5Gen Virtual Function (rev 01)
```

⚠ 2 new virtual Ethernet devices are created in this example.

19.12.4 Running DOCA Application on Host

⚠ Allocate the required number of VFs as explained previously.

⚠ Allocate any other resources as specified by the application (e.g., huge pages).

The following is the CLI example for running a reference application over the host using VF:

```
doca_<app_name> -a "pci address VF0" -a "pci address VF1" -c 0xff -- [application flags]
```

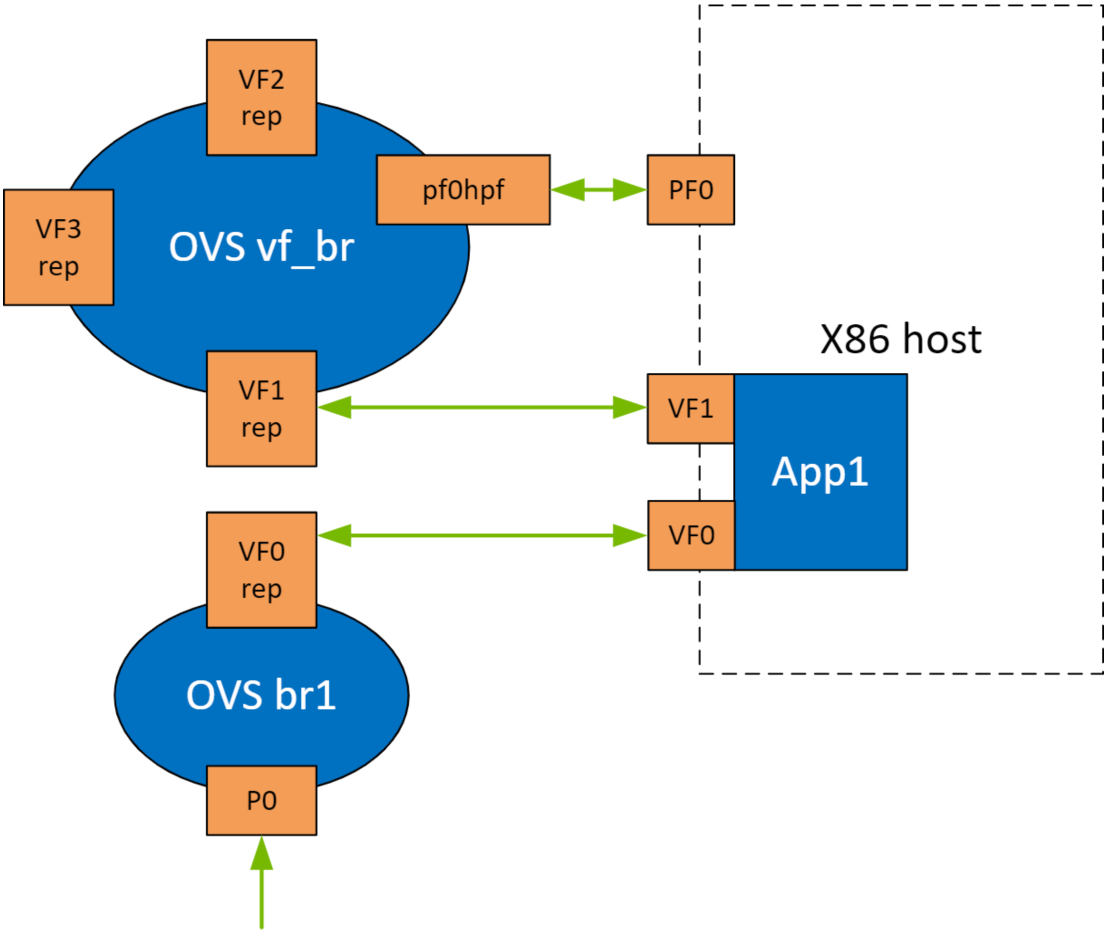
The following is an example with specific PCIe addresses for the VFs:

```
doca_<app_name> -a b1:00.3 -a b1:00.4 -c 0xff -- -l 60
```

⚠ By default, a DPDK application initializes all the cores of the device. This is usually unnecessary and may even cause unforeseeable issues. It is recommended to limit the number of cores, especially when using an AMD-based system, to 16 cores using the `-c` flag when running DPDK.

19.12.5 Topology Example

The following is a topology example for running the application over the host.



Configure the OVS on BlueField as follows:

```
Bridge ovsbr1
    Port ovsbr1
        Interface ovsbr1
            type: internal
    Port pf0hpf
        Interface pf0hpf
    Port pf0vf1
        Interface pf0vf1
Bridge vf_br
    Port p0
        Interface p0
    Port vf_br
        Interface vf_br
            type: internal
    Port pf0vf0
        Interface pf0vf0
```

When enabling a new VF over the host, VF representors are created on the Arm side. The first OVS bridge connects the uplink connection (`p0`) to the new VF representor (`pf0vf0`), and the second bridge connects the second VF representor (`pf0vf1`) to the host representors (`pf0phf`). On the host, the 2 PCIe addresses of the newly created function must be initialized when running the applications.

When traffic is received (e.g., from the uplink), the following occurs:

1. Traffic is received over `p0` .
2. Traffic is forwarded to `pf0vf0` .
3. Application "listens" to `pf0vf0` and `pf0vf1` and can, therefore, acquire the traffic from `pf0vf0` , inspect it, and forward to `pf0vf1` .

4. Traffic is forwarded from `pf0vf1` to `pf0hpf`.

19.12.6 VF Creation on Adapter Card

> ⚠ Supported only for NVIDIA® ConnectX®-6 Dx based adapter cards and higher.

The following steps are required only when running DOCA applications on an adapter card.

1. Set trust level for all VFs. Run:

   ```
   host# mlxreg -d /dev/mst/mt4125_pciconf0 --reg_name VHCA_TRUST_LEVEL --yes --set
   "all_vhca=0x1,trust_level=0x1" --indexes "vhca_id=0x0,all_vhca=0x0"
   ```

2. Create X VFs (X being the required number of VFs) and run the following to turn on trusted mode for the created VFs:

   ```
   echo ON | tee /sys/class/net/enp1s0f0np0/device/sriov/X/trust
   ```

   For example, if you are creating 2 VFs, the following commands should be used:

   ```
   echo ON | tee /sys/class/net/enp1s0f0np0/device/sriov/0/trust
   echo ON | tee /sys/class/net/enp1s0f0np0/device/sriov/1/trust
   ```

3. Create a VF representor using the following command, replace the PCIe address with the PCIe address of the created VF:

   ```
   echo 0000:17:00.2 > /sys/bus/pci/drivers/mlx5_core/unbind
   echo 0000:17:00.2 > /sys/bus/pci/drivers/mlx5_core/bind
   ```

20 Archives

This section contains the following pages:

- NVIDIA DOCA LTS Versions
- NVIDIA DOCA Documentation Archives

20.1 NVIDIA DOCA LTS Versions

Documentation for DOCA long term support (LTS) releases.

20.1.1 Introduction

DOCA LTS releases are stable and verified DOCA versions. LTS updates include bug fixes and security vulnerability fixes but not ongoing features and enhancements.

20.1.2 LTS Documentation

Follow these links to navigate to the relevant LTS release or specific update:

- DOCA 2.5.0 LTS base version
  - 2.5.2 LTS update
  - 2.5.1 LTS update
- DOCA 1.5.0 LTS base version
  - 1.5.3 LTS update
  - 1.5.2 LTS update
  - 1.5.1 LTS update

20.2 NVIDIA DOCA Documentation Archives

Archived documentation of previous DOCA software releases.

- DOCA v2.7.0 documentation
- DOCA v2.6.0 documentation
- DOCA v2.5.2 LTS documentation
- DOCA v2.5.1 LTS documentation
- DOCA v2.5.0 LTS documentation
- DOCA v2.2.1 documentation
- DOCA v2.2.0 documentation
- DOCA v2.0.2 documentation
- DOCA v1.5.3 LTS documentation
- DOCA v1.5.2 LTS documentation
- DOCA v1.5.1 LTS documentation
- DOCA v1.5.0 LTS documentation
- DOCA v1.4.0 documentation
- DOCA v1.3.0 documentation
- DOCA v1.2.1 documentation
- DOCA v1.2.0 documentation

- DOCA v1.1.1 documentation
- DOCA v1.1.0 documentation
- DOCA v1.0.0 documentation

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. Neither NVIDIA Corporation nor any of its direct or indirect subsidiaries and affiliates (collectively: “NVIDIA”) make any representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice. Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer (“Terms of Sale”). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks