

DOCA Documentation v2.5.3 (2023 LTS U3)

Table of contents

DOCA SDK v2.5.3 17

NVIDIA DOCA Overview 17

NVIDIA DOCA Release Notes 20

BlueField DPU and DOCA User Types 37

NVIDIA DOCA EULA 41

Quick Start 48

NVIDIA DOCA Developer Quick Start Guide 48

Installation and Setup 52

NVIDIA DOCA Profiles 52

NVIDIA DOCA Installation Guide for Linux 56

NVIDIA DOCA Developer Guide 106

DOCA Programming Guide 122

DOCA Programming Overview 122

Hardware Overview 123

DOCA SDK Architecture 124

DOCA Backward Compatibility Policy 129

DOCA Development Best Practices 132

Capability Checking 132

Debuggability 134

DOCA Libraries 138

DOCA Flow 139

DOCA Flow Connection Tracking 347

DOCA IPsec 357

DPA Subsystem 367
DPA Development 370
DOCA DPA 447
DOCA PCC 481
DOCA DMA 489
DOCA Comm Channel 503
DOCA Comm Channel – New 503
DOCA Comm Channel – Deprecated 529
DOCA RDMA 567
DOCA Ethernet 619
DOCA GPUNetIO 652
DOCA App Shield 708
DOCA Compress 733
DOCA SHA 751
DOCA Erasure Coding 778
DOCA AES-GCM 806
DOCA Rivermax 824
DOCA Telemetry Exporter 837
DOCA Telemetry 862
DOCA Utils 880
DOCA Arg Parser 880
DOCA Drivers 889
DOCA UCX 889
Features Overview and Configuration 910
InfiniBand Network 911
InfiniBand Interface 911
NVIDIA SM 912
QoS - Quality of Service 979

IP over InfiniBand (IPoIB) 983
Advanced Transport 997
Optimized Memory Access 999
NVIDIA PeerDirect 1005
CPU Overhead Distribution 1005
Out-of-Order (OOO) Data Placement 1006
IB Router 1007
MAD Congestion Control 1008
Ethernet Network 1010
Ethernet Interface 1011
Quality of Service (QoS) 1014
Ethtool 1030
Checksum Offload 1035
Ignore Frame Check Sequence (FCS) Errors 1036
RDMA over Converged Ethernet (RoCE) 1036
Flow Control 1051
Explicit Congestion Notification (ECN) 1060
RSS Support 1062
Time-Stamping 1064
Flow Steering 1077
Wake-on-LAN (WoL) 1082
Hardware Accelerated 802.1ad VLAN (Q-in-Q Tunneling) 1082
VLAN Stripping in Linux Verbs 1083
Offloaded Traffic Sniffer 1084
Dump Configuration 1085
Local Loopback Disable 1089
Kernel Transport Layer Security (kTLS) Offloads 1089
IPsec Crypto Offload 1092
IPsec Full Offload 1096
MACsec Full Offload 1104
Storage Protocols 1110
SRP - SCSI RDMA Protocol 1111
iSCSI Extensions for RDMA (iSER) 1126
Lustre 1128

NVME-oF - NVM Express over Fabrics 1129
Virtualization 1130
Single Root IO Virtualization (SR-IOV) 1130
SR-IOV Live Migration 1156
Enabling Paravirtualization 1168
VXLAN Hardware Stateless Offloads 1170
Q-in-Q Encapsulation per VF in Linux (VST) 1172
802.1Q Double-Tagging 1175
Scalable Functions 1177
Resiliency 1178
Reset Flow 1178
Docker Containers 1181
Docker Using SR-IOV 1182
Kubernetes Using SR-IOV 1182
Kubernetes with Shared HCA 1183
HPC-X 1183
Fast Driver Unload 1183
DOCA Applications 1184
NVIDIA DOCA Allreduce Application Guide 1192
NVIDIA DOCA App Shield Agent Application Guide 1215
NVIDIA DOCA DMA Copy Application Guide 1235
NVIDIA DOCA DPA All-to-all Application Guide 1250
NVIDIA DOCA DPA L2 Reflector Application Guide 1267
NVIDIA DOCA East-West Overlay Encryption Application 1279
NVIDIA DOCA Eth L2 Forwarding Application Guide 1307
NVIDIA DOCA File Compression Application Guide 1321
NVIDIA DOCA File Integrity Application Guide 1334
NVIDIA DOCA Firewall Application Guide 1347
NVIDIA DOCA GPU Packet Processing Application Guide 1362

NVIDIA DOCA IPsec Security Gateway Application Guide 1388
NVIDIA DOCA NAT Application Guide 1432
DOCA NVMe-oF RDMA Target Reference Application Guide 1448
NVIDIA DOCA PCC Application Guide 1462
NVIDIA DOCA Secure Channel Application Guide 1482
NVIDIA DOCA Simple Forward VNF Application Guide 1496
NVIDIA DOCA Switch Application Guide 1515
NVIDIA DOCA YARA Inspection Application Guide 1533
DOCA Tools 1554
NVIDIA DOCA Comm Channel Admin Tool 1556
NVIDIA DOCA PCC Counter Tool 1562
NVIDIA DOCA Socket Relay 1566
NVIDIA DOCA Ngague 1572
DOCA Services 1577
NVIDIA BlueField DPU Container Deployment Guide 1581
NVIDIA DOCA BlueMan Service Guide 1598
NVIDIA DOCA Firefly Service Guide 1606
NVIDIA DOCA Flow Inspector Service Guide 1658
NVIDIA DOCA HBN Service Guide 1676
HBN Service Release Notes 1750
NVIDIA DOCA Telemetry Service Guide 1760
OpenvSwitch Offload 1803
API References 1904
NVIDIA DOCA Driver APIs 1904

NVIDIA DOCA Library APIs 1904

Miscellaneous (Runtime) 1905

NVIDIA DOCA Glossary 1905

NVIDIA DOCA Crypto Acceleration 1910

NVIDIA DOCA Services Fluent Logger 1911

NVIDIA DOCA DPU CLI 1916

NVIDIA DOCA Emulated Devices 1921

NVIDIA DOCA gRPC Infrastructure User Guide 1956

NVIDIA BlueField DPU Modes of Operation 1969

NVIDIA DOCA Switching Support 1978

DPU Kernel Representors Model 1978

Virtio Acceleration through Hardware vDPA 1980

Bridge Offload 1987

DPU Functional Diagram 1989

Link Aggregation 1990

NVIDIA DOCA with OpenSSL 2001

NVIDIA BlueField DPU Scalable Function User Guide 2006

NVIDIA TLS Offload Guide 2020

NVIDIA DOCA Troubleshooting Guide 2046

NVIDIA DOCA Virtual Functions User Guide 2087

Archives 2096

NVIDIA DOCA LTS Versions 2096

NVIDIA DOCA Documentation Archives 2096

DOCA Documentation v2.5.3

DOCA Overview

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

Release Notes

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

User Types

This page provides a quick introduction to the NVIDIA® BlueField® family of DPUs, its DOCA software components, and the DPU user types.

NVIDIA DOCA EULA

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

Quick Start

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.

Installation and Setup

Profiles

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

Installation Guide for Linux

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

Developer Guide

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

DOCA Programming Guides

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

Applications

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

Allreduce

This page provides a DOCA Allreduce collective operation implementation on top of NVIDIA® BlueField® DPU® using UCX.

App Shield Agent

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

DMA Copy

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

DPA All-to-all

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

DPA L2 Reflector

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

East-west Overlay Encryption

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

File Compression

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

File Integrity

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

Firewall

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

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.

IPsec Security Gateway

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

NAT

This page provides a NAT implementation on top of NVIDIA® BlueField® DPU.

PCC

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

Secure Channel

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

Simple Forward VNF

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

Switch

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

YARA Inspection

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

Tools

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

DPA EU Management Tool

This page describes the DPA execution unit management tool, `dpaeumgmt` .

DPACC Compiler

This page describes DOCA DPACC compiler and instructions about DPA toolchain setup and usage.

FlexIO Build

This page describes the DOCA FlexIO Build tool.

PCC Counter Tool

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

Socket Relay

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

DOCA Services

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

Container Deployment

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

DOCA BlueMan Service

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

DOCA Firefly Service

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

DOCA Flow Inspector Service

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

DOCA HBN Service

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

DOCA Telemetry Service

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

API References

DOCA Driver APIs

This page contains DOCA driver APIs.

DOCA Libraries APIs

This page contains DOCA libraries APIs.

Miscellaneous

Glossary

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

Crypto Acceleration

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

DOCA Services Fluent Logger

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

DPU CLI

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

Emulated Devices

This page describes the ability of NVIDIA® BlueField® DPU to emulate and accelerate physical and virtual host functions.

gRPC Infrastructure

This page provides an overview and configuration instructions for gRPC-supported DOCA infrastructure, applications, and services for NVIDIA® BlueField® DPU.

Modes of Operation

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

OpenSSL

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

Scalable Functions (SFs)

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

Switching Support

This page provides information on the control of the NVIDIA® BlueField® DPU embedded switch and representation of the host network functions.

TLS Offload

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

Troubleshooting

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

Virtual Functions (VFs)

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

Archive

LTS Versions

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

Documentation Archives

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

Info

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

DOCA SDK v2.5.3

This section contains the following pages:

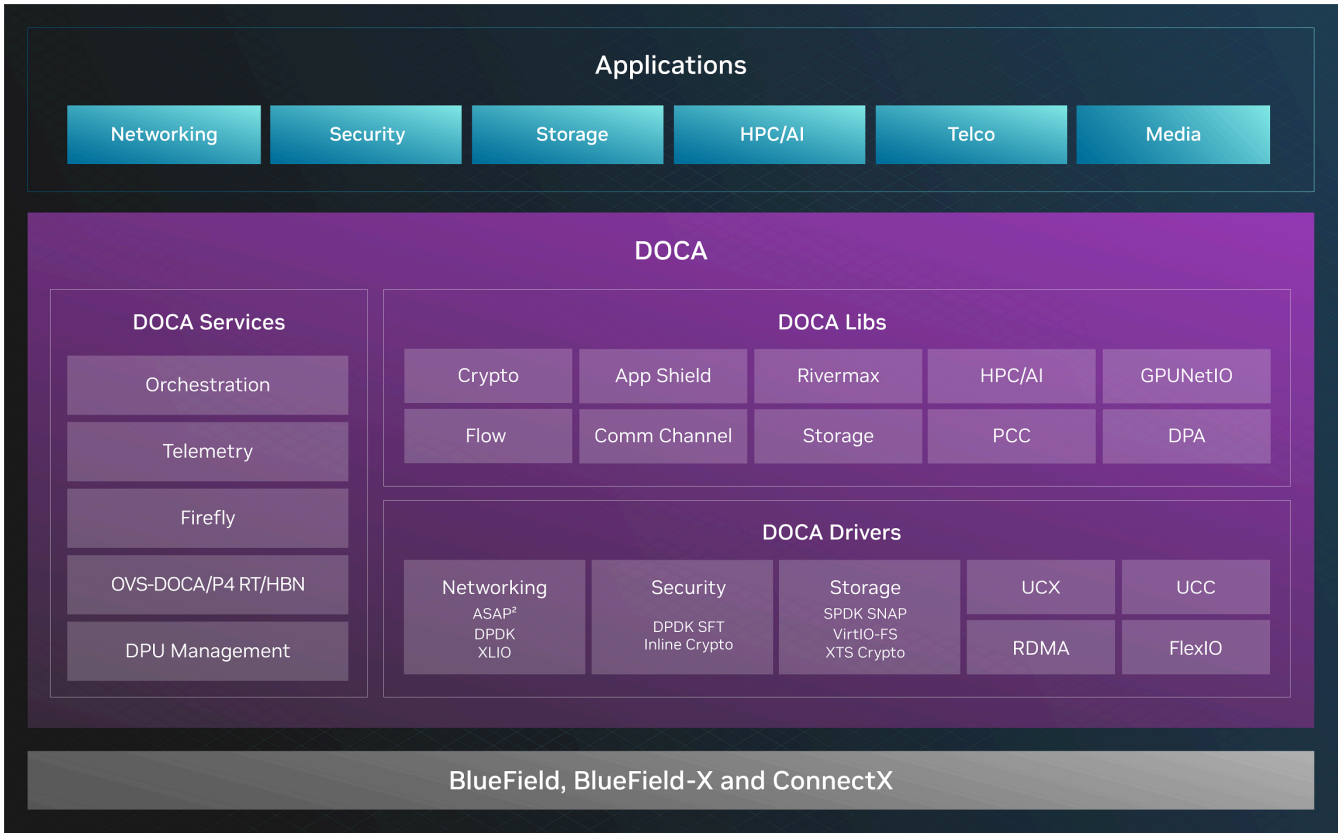
- NVIDIA DOCA Overview
- NVIDIA DOCA Release Notes
- BlueField DPU and DOCA User Types
- NVIDIA DOCA EULA

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 everything you need to develop DPU-accelerated applications.

Introduction

The NVIDIA DOCA™ SDK enables developers to rapidly create applications and services on top of NVIDIA® BlueField® data processing units (DPUs), leveraging industry-standard APIs. With DOCA, developers can deliver breakthrough networking, security, and storage performance by harnessing the power of NVIDIA's DPUs.



Installation

DOCA contains a runtime and development environment for both the host and as part of a BlueField image for the DPU. The full installation instructions for both can be found in the NVIDIA DOCA Installation Guide for Linux.

Whether DOCA has been installed on the host or on the DPU, 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.

API

The DOCA SDK is built around the different DOCA libraries designed to leverage the capabilities of the DPUs. 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 developers wishing to develop DOCA-based programs.

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

Please note that, as explained in the NVIDIA DOCA gRPC Infrastructure User Guide, some of DOCA's libraries also support a gRPC-based API. More information about these extended programming interfaces can be found in detail in the programming guides of the respective libraries.

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.

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.

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.

An example for one such DOCA tool is the `doca_dpi_compiler`, responsible for converting Suricata-based rules to their matching `.cdo` definition files which are then used by the DOCA DPI library.

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

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 the DPU, and sometimes also to the host.

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

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

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

NVIDIA DOCA Release Notes

NVIDIA DOCA release notes containing information on new features, bug fixes, and known issues.

Introduction

DOCA 2.5.3 is an LTS update to DOCA 2.5.2 which includes bug fixes.

Issues Fixed

This version introduces the following bug fixes.

Ref #	Issue Description
4087450	Description: Increased the RX lossless buffer size to delay transmission of pause/PFC frames when the NIC is congested.
	Reported in version: 2.5.0

Ref #	Issue Description
3992563	Description: Thermal sensor hangs due to incorrect initialization sequence.
	Reported in version: 2.5.0
3991098	Description: Thermal sensors may show abnormal temperature readings.
	Reported in version: 2.5.0
4073220	Description: When configured for NIC mode, BlueField-3 may boot into Linux after a reset.
	Reported in version: 2.5.0
4149393	Description: Added address validation in MLNX OEM CMD 0x0032 (get debug info) to be 4-bytes aligned.
	Reported in version: 2.5.0
4054931	Description: Added support for PCIe Gen 5 AFE configuration.
	Reported in version: 2.5.0
4163371	Description: Improved the Virtual QoS mechanism ability to identify high bandwidth bursty traffic pattern and apply the correct rate shapers.
	Reported in version: 2.5.0
4175173	Description: Reduced the bandwidth fluctuation induced by VQoS rate limiting in systems with bellow 350 QPs. **Info** In this release, this change is enabled by default, while in future versions it will be disabled by default and an additional NV configuration will be required to enable it.
	Reported in version: 2.5.0

Installation Notes

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

- Setting up DOCA SDK on your NVIDIA® BlueField® platform

- Supported hardware platforms

Embedded DOCA Libraries

Component	Version
doca-apps	2.5.1007-1
doca-grpc	2.5.1007-1
doca-libs	2.5.1007-1
ucx	1.16.0-1.2310213
gpunetio	2.5.1007-1

Embedded DOCA Firmware Components

Component	Version	Description
ATF	v2.2(release):4.5.1-1-g8c00262	Arm-trusted firmware is a reference implementation of secure world software for Arm architectures
UEFI	4.5.1-4-g3eb5e0b	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	32.39.3004	Firmware is used to run user programs on the BlueField-3 which allow hardware to run
BlueField-2 NIC firmware	24.39.3004	Firmware is used to run user programs on the BlueField-2 which allow hardware to run
BMC firmware	23.10-7	BlueField BMC firmware
BlueField-3 eROT (Glacier)	02.0152.0000	BlueField-3 eROT firmware
BlueField-2 eROT (CEC)	04.0f	BlueField-2 eROT firmware

Embedded DOCA Drivers

Component	Version	Description
collectx-clxapi	1.15.3	A library which exposes the CollectX API, which allows any 3rd party to easily use CollectX functionality in their own programs
doca-base (MLNX_OFED)	23.10-3.2.2.0	NVIDIA® MLNX_OFED is a single software stack that operates across all NVIDIA network adapter solutions
dpacc	1.6.0-8	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
dpcp	1.1.43-1.2310055	DPCP provides a unified flexible interface for programming IB devices using DevX
flexio	23.10.1873	FlexIO SDK exposes an API for managing the device and executing native code over the DPA processor
libvma	9.8.40-1	The NVIDIA® 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)
libxlio	3.20.8-1	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
MFT	4.26.1-6	NVIDIA ® MFT is a set of firmware management and debug tools for NVIDIA devices

Component	Version	Description
mlnx-dpdk	22.11.0-2310.3.0.23102130.2310.3.0	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).
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 DPUs and take the most advantage from hardware capabilities
mlnx-snap	3.8.0-3	BlueField SNAP for NVMe and virtio-blk enables hardware-accelerated virtualization of local storage
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
Rivermax	1:1.41.2	NVIDIA® Rivermax® is an optimized networking SDK for media and data streaming applications
RShim	2.0.20	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
spdk	23.01.5-17	SPDK provides a set of tools and libraries for writing high performance, scalable, user-mode storage applications
virtio-net-controller	1.7.13-1	Virtio-net-controller is a systemd service running on the DPU, with a user interface front-end to communicate with the background service

DOCA Packages

Device	Component	Version	Description
Host	DOCA SDK	2.5.3	Software development kit package for developing host software
	DOCA Runtime	2.5.3	Runtime libraries required to run DOCA-based software applications on host
	DOCA Tools	2.5.3	Tools for developers and administrators on host
	DOCA Extra	2.5.3	Contains helper scripts (doca-info, doca-kernel-support)
	DOCA OFED	2.5.3	Software stack which operates across all NVIDIA network adapter solutions
	Arm emulated (QEMU) development container	4.5.3	Linux-based BlueField Arm emulated container for developers
Target BlueField DPU (Arm)	BlueField BSP	4.5.3	BlueField image and firmware
	DOCA SDK	2.5.3	Software development kit packages for developing Arm software
	DOCA Runtime	2.5.3	Runtime libraries required to run DOCA-based software applications on Arm
	DOCA Tools	2.5.3	Tools for developers and administrators for Arm target

Supported Operating System Distributions

Note

Starting from DOCA 2.6.0, OSs with kernel versions lower than 4.18 will no longer be supported. DOCA 2.5.0 is the last version to support OS with a kernel lower than 4.18.

The default operating system of the BlueField DPU (Arm) is Ubuntu 22.04.

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

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

DOCA for Host	Kernel	Arch	doca-all	doca-cx	doca-ofed
CTYunOS3 23.01	5.10	aarch64	✔	✔	✔
RHEL/CentOS 8.2	4.18	x86	✔	✔	✔
RHEL/Rocky 8.6	4.18	x86	✔	✔	✔
Ubuntu 20.04	5.4	x86	✔	✔	✔
Ubuntu 22.04	5.15	x86 / aarch64	✔	✔	✔
Debian 10.8	4.19	x86	✔	✔	✔
Debian 10.13	5.10.135 / 5.4.210 / 5.15	x86	✔	✔	✔
Alinux 3.2	5.10	x86	✔	✔	✔
Oracle Linux 8.7	5.15	x86	✔	✔	✔
RHEL/Rocky Linux 9.1	5.14	x86	✔	✔	✔
BCLinux 21.10 SP2	4.19.90	x86 / aarch64			✔
CTYunOS2.0	4.19.90	x86 / aarch64			✔
Debian 10.9	4.19.0-16	x86			✔
Debian 11.3	5.10.0-13	x86 / aarch64			✔

DOCA for Host	Kernel	Arch	doca-all	doca-cx	doca-ofed
Debian 12.1	6.1.0-10	x86 / aarch64			✔
Debian 12.5	6.1.0-18	x86 / aarch64	✔	✔	✔
Kylin 10 SP2	4.19.90	x86 / aarch64			✔
Oracle Linux 8.6	5.4	x86			✔
openEuler 20.03 SP3	4.19.90	x86 / aarch64			✔
openEuler 22.03	5.10.0	x86 / aarch64			✔
RHEL/CentOS 8.0	4.18.0-80.el8	x86			✔
RHEL/CentOS 8.2	4.18.0-193.el8	aarch64			✔
RHEL/CentOS 8.4	4.18.0-305.el8	x86 / aarch64			✔
RHEL/Rocky 8.6	4.18.0-372.41.1.el8	aarch64			✔
RHEL/Rocky 8.8	4.18.0-477.10.1.el8_8	x86 / aarch64	✔	✔	✔
RHEL/Rocky 8.9	4.18.0-513.5.1.el8_9	x86 / aarch64			✔
RHEL/Rocky 8.10	4.18.0-553.el8_10	x86 / aarch64			✔
RHEL/Rocky 9.0	5.14.0-70.46.1.el9_0	x86 / aarch64			✔
RHEL/Rocky 9.1	5.14.0-162.19.1.el9_1	x86 / aarch64			✔
RHEL/Rocky 9.2	5.14.0-284.11.1.el9_2	x86 / aarch64			✔
RHEL/Rocky 9.3	5.14.0-362.8.1.el9_3	x86 / aarch64			✔
RHEL/Rocky 9.4	5.14.0-427.13.1.el9_4	x86 / aarch64			✔

DOCA for Host	Kernel	Arch	doca-all	doca-cx	doca-ofed
SLES 15 SP3	5.3.18-57	x86 / aarch64			✔
SLES 15 SP4	5.14.21-150400.22	x86 / aarch64			✔
SLES 15 SP5	5.14.21-150500.53	x86 / aarch64			✔
SLES 15 SP6	6.4.0-150600.21-default	x86 / aarch64			✔

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.

Info

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

Known Issues

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

Reference	Description
3928336	Description: In CTYunOS 23.01 , which includes MLNX_OFED drivers as part of the OS repository, the DOCA-HOST repository should be enabled before installation to avoid any installation issues.
	Workaround: Run: `yum --enablerepo=doca install`
	Keyword: CTYunOS; installation
	Reported in version: 2.5.2
3631257	Description: When installing DOCA for host on a SLES15.5 system that uses NVMe storage (e.g., NVMe disk) and a non-default kernel that requires rebuilding the kernel, the script `doca-kernel-support` builds and installs the `mlnx-nvme` drivers that override standard NVMe kernel drivers, which renders the system unbootable.
	Workaround: After installing `doca-extra`, edit the `doca-kernel-extra` script before running it to avoid it building `mlnx-nvme` modules. The option `--without-mlnx-nvme` must be added to the invocation of `install.pl` : `sed -i -e '/install\.pl/a\\t--without-mlnx-nvme \\' /opt/mellanox/doca/tools/doca-kernel-support`
	Keyword: NVMe; SLES
	Reported in version: 2.5.0
3678069	Description: If using DPUs 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

Reference	Description
3669736	Description: DOCA can run only one instance of a DOCA library per process on setups with a page size different than 4K (e.g., Rockey). Combination of different libraries cannot be run on the same process.
	Workaround: N/A
	Keyword: OpenEuler; Rocky; DOCA libs
	Reported in version: 2.5.0
3469692	Description: The overall number of IPsec sessions is limited to 64K per port.
	Workaround: N/A
	Keyword: IPsec
	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
3674295	Description: The script for the East-West Overlay Encryption application is not supported in DOCA 2.5.0. Users who wish to configure OVS IPsec with hardware offload must should follow the steps explained in the "Configuration Flow" section of the guide instead of running the application.
	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 future releases.
	Workaround: N/A
	Keyword: EU tool
	Reported in version: 2.5.0

Reference	Description
3618936	Description: When moving to DPU mode from NIC mode, it is necessary to reinstall the BFB and perform a graceful reboot to the DPU by shutting down the Arm cores before rebooting the host system.
	Workaround: N/A
	Keyword: NIC mode
	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
3603146	Description: Running `mlxfwreset` on BlueField-3 may cause the external host to crash when the RShim driver is running on that host.
	Workaround: Stop the RShim driver on the external host using `systemctl stop rshim` before performing `mlxfwreset`.
	Keyword: RShim; mlxfwreset
	Reported in version: 2.2.1
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 the DPU.
	Workaround: N/A
	Keyword: UEK; VF
	Reported in version: 2.2.1

Reference	Description
3568924	Description: `doca_rdma` does not support cross-subnet routing.
	Workaround: N/A
	Keyword: RDMA
	Reported in version: 2.2.1
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 the DPU, 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

Reference	Description
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
	Keywords: Linux; PHY; kernel
	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".
	Workaround: Add `intel_idle.max_cstate=1` entry to the kernel command line.
	Keywords: Longevity; mlxfwreset; DPU reboot
	Reported in version: 2.2.0
3529297	Description: Enhanced NIC mode is not supported on BlueField-2 DPUs.
	Workaround: N/A
	Keywords: Operation; mode
	Reported in version: 2.2.0
3538486	Description: When removing LAG configuration from the DPU, 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

Reference	Description
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
3516279	Description: The NVIDIA DOCA East-West Overlay Encryption Reference Application (and the underlying DPU OS Kernel driver IPsec functionality) is not supported. User space DOCA IPsec is not impacted.
	Workaround: N/A
	Keyword: IPsec
	Reported in version: 2.2.0
3444073	Description: `mlxfwreset` is not supported in this release.
	Workaround: Perform a graceful shutdown, then power cycle the host.
	Keyword: mlxfwreset; support
	Reported in version: 2.0.2

Reference	Description

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 DPU 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
3232444	Description: After live migration of virtio-net devices using the VFE driver, the `max_queues_size` output from the `virtnet list` may be wrong. This does not affect the actual value.
	Workaround: N/A
	Keyword: Virtio-net; live migration
	Reported in version: 2.0.2

Reference	Description
3441287	Description: Failure occurs when attempting to raise static LAG with `ifenslave_2.10ubuntu3` package.
	Workaround: Use `ifenslave_2.9ubuntu1`.
	Keywords: `ifenslave`; bonding
	Reported in version: 4.0.2
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
	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

Reference	Description
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

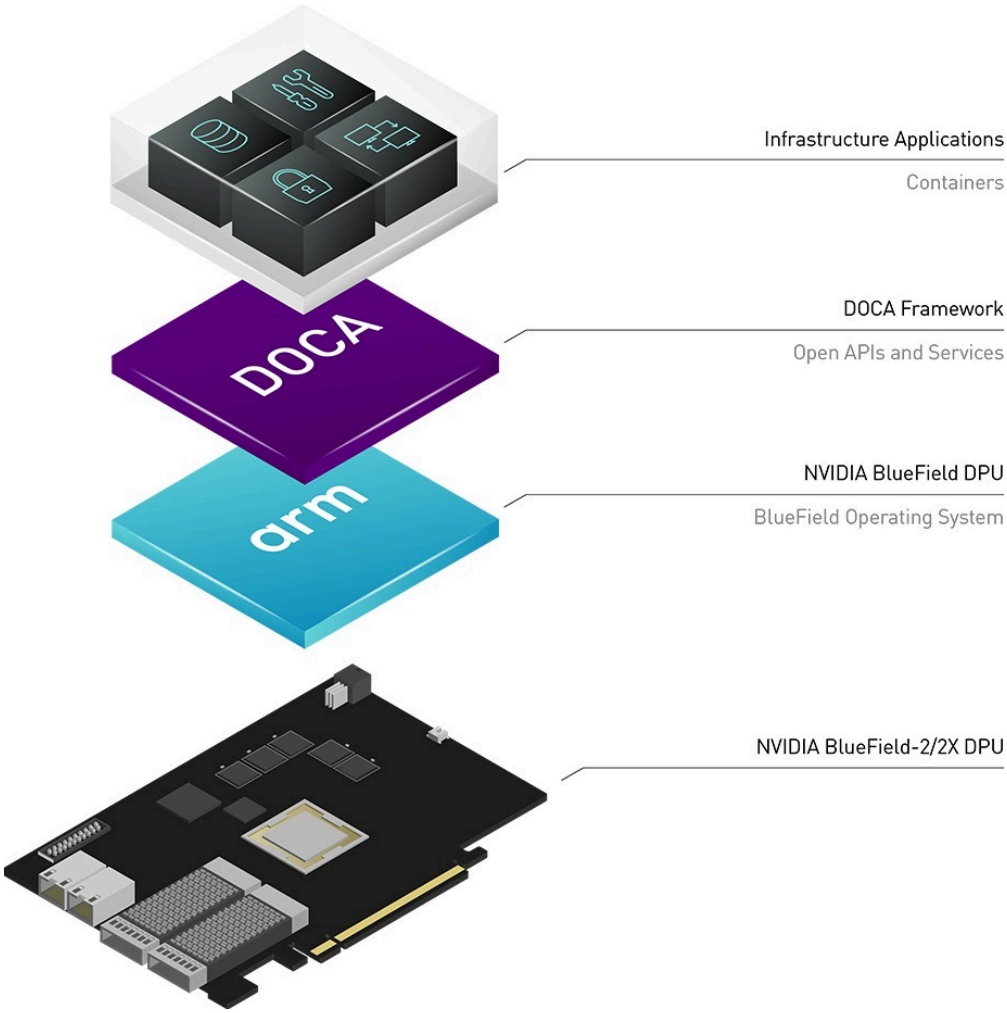
BlueField DPU and DOCA User Types

This guide provides a quick introduction to the NVIDIA® BlueField® family of DPUs, its DOCA software components, and the DPU user types.

Introduction

The BlueField family of data processing units (DPUs) 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 DPUs enable 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.

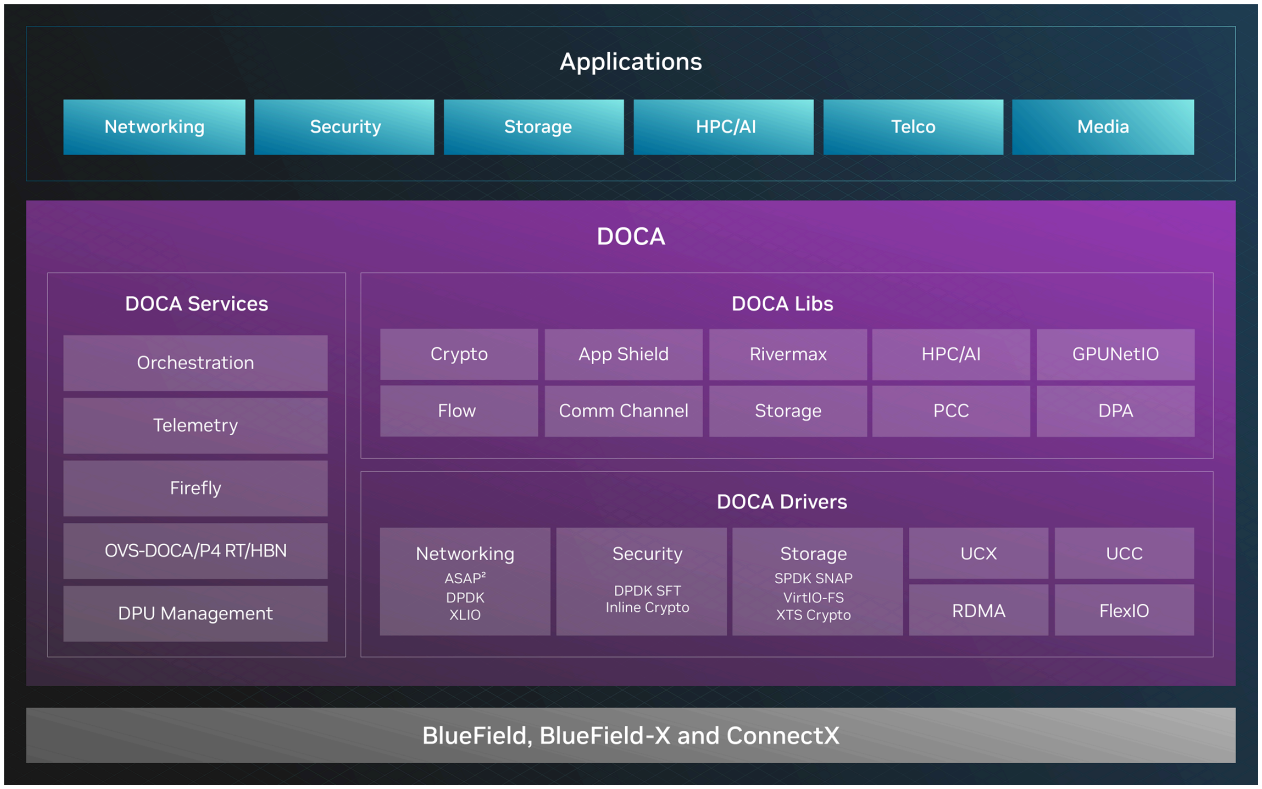
NVIDIA DOCA™ brings together a wide range of powerful APIs, libraries, and frameworks for programming and acceleration of the modern data center infrastructure. Like NVIDIA® CUDA® for GPUs, DOCA is a consistent and essential resource across all existing and future generations of DPU products.



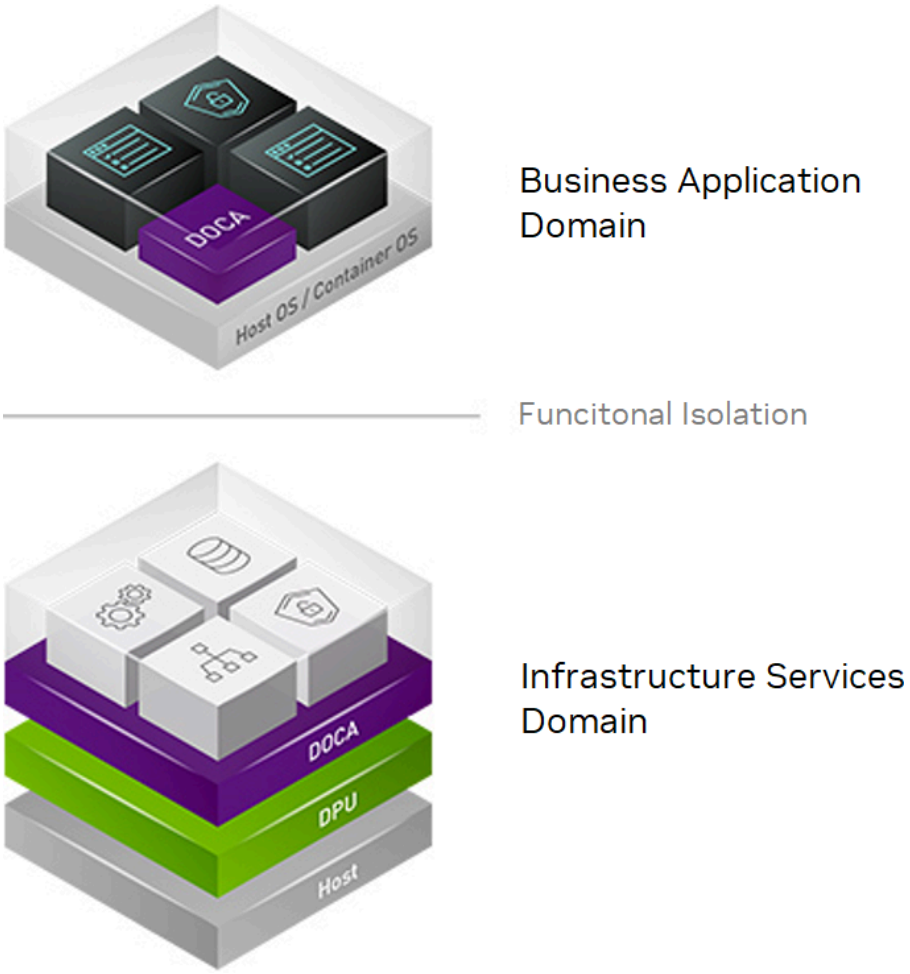
DOCA Components

DOCA software consists of an SDK and a runtime environment.

- **DOCA SDK** 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 SDK 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 DPUs in bulk across the data center.



BlueField DPU User Types

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 DPU.

Common operations performed by the BlueField admin:

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

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

DOCA Developer

A DOCA developer creates the services and applications that run on top of the DPU 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

NVIDIA DOCA EULA

NVIDIA DOCA SDK end-user license agreement.

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:

    1. Install and use the SOFTWARE,

    2. Modify and create derivative works of sample or reference source code delivered in the SOFTWARE, and

    3. 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:

    1. An application must have material additional functionality, beyond the included portions of the SOFTWARE.

    2. 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."

    3. 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.

    4. 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:

    1. 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.

    2. 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.

    3. 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.

    4. 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.

    5. 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.

    6. You may not bypass, disable, or circumvent any technical limitation, encryption, security, digital rights management or authentication mechanism in the SOFTWARE.

    7. 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.

    8. 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.

    9. 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.

10. 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

1. 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.

2. 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

Quick Start

This section contains the following pages:

- NVIDIA DOCA Developer Quick Start Guide

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.

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 DPU products.

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

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

Install DPU

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

Register to NVIDIA DOCA Developer Zone

As part of the installation process, you must provide your NVIDIA account information. Get your NVIDIA Developer Zone access here.

Install DOCA Software Package

A detailed step-by-step process for installing the required development software on both the host and DPU 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® DPU.

Access DPU

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 DPU OS according to the instructions on "Host-side Interface Configuration" in the *NVIDIA BlueField DPU BSP* document.

2. Log into the DPU's Ubuntu-based OS by running the following command from the host:

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

   Use the DPU password you defined during the installation process.

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

Run Reference DOCA Application

DOCA package assets (e.g., references, tools) are located on the DPU 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 the DPU over the Comm Channel interface:

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

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

More Information

To learn more about NVIDIA BlueField DPUs, refer to the NVIDIA BlueField DPU Hardware Manuals.

Info

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

Installation and Setup

This section contains the following page:

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

NVIDIA DOCA Profiles

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

Introduction

NVIDIA DOCA™ can be used by a variety of customers who have different workloads and requirements. The DOCA package includes drivers, libraries, and tools to support NVIDIA® BlueField® DPU and NVIDIA® ConnectX® SmartNIC, Ethernet and InfiniBand, and kernel and user-space components. Depending on their specific needs, customers may choose not to install the full DOCA package 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 allows installation of DOCA profiles, which are a subset of the full DOCA installation. DOCA profiles are validated and tested installation packages. The following are the available DOCA profiles:

- doca-all
- doca-cx
- doca-ofed

doca-all

doca-all profile contains all DOCA libraries and drivers.

The doca-all profile supports the following devices:

- BlueField-3
- BlueField-2
- ConnectX-7
- ConnectX-6 Dx
- ConnectX-6 Lx
- ConnectX-6

doca-cx

The DOCA for ConnectX profile is intended for users who have ConnectX devices connected to their host server and wish to benefit from DOCA's framework and the enhanced functionality it grants.

The content of the doca-cx package is as follows:

- MLNX_OFED
- MLNX-DPDK
- DOCA Flow
- DOCA IPsec

The doca-cx profile supports the following devices:

- ConnectX-7
- ConnectX-6 Dx
- ConnectX-6 Lx
- ConnectX-6

doca-ofed

This profile is intended for users who wish to have the exact same user experience and content as MLNX_OFED but with DOCA Package. doca-ofed installs the MLNX_OFED

drivers and tools but not other DOCA components.

The content of the doca-ofed package is as follows:

- MLNX_OFED drivers
- MLNX_OFED tools

The doca-ofed profile supports the following devices:

- ConnectX-7
- ConnectX-6 Dx
- ConnectX-6 Lx
- ConnectX-6

DOCA Profile Installation

Follow the instructions under Manual BlueField Image Installation in the "NVIDIA DOCA Installation Guide for Linux".

Supported Host OS per DOCA Profile

DOCA for Host	Kernel	Arch	doca-all	doca-cx	doca-ofed
CTYunOS3 23.01	5.10	x86	✔	✔	✔
RHEL/CentOS 7.6	3.10	x86	✔	✔	✔
	4.14	aarch64	✔	✔	✔
RHEL/CentOS 8.2	4.18	x86	✔	✔	✔
RHEL / Rocky Linux 8.6	4.18	x86	✔	✔	✔
Ubuntu 18.04	4.15	x86	✔	✔	✔
Ubuntu 20.04	5.4	x86	✔	✔	✔
Ubuntu 22.04	5.15	X86	✔	✔	✔
	5.15	aarch64	✔	✔	✔
Debian 10.8	4.19	x86	✔	✔	✔

DOCA for Host	Kernel	Arch	doca-all	doca-cx	doca-ofed
Debian 10.13	5.10.135 / 5.4.210	x86	✔	✔	✔
Allinux 3.2	5.10	X86	✔	✔	✔
Oracle Linux 8.7	5.10	x86	✔	✔	✔
RHEL/Rocky Linux 9.1	5.14.0-162.19.1.el9	x86	✔	✔	✔
Debian10.9	4.19.0-16	x86			✔
BCLinux 21.10 SP2	4.19.90	x86/aarch64			✔
Kylin 10 SP2	4.19.90	x86/aarch64			✔
openEuler 20.03 SP3	4.19.90	x86/aarch64			✔
openEuler 22.03	5.10.0	x86/aarch64			✔
UOS 20 1040d	4.19.0	x86/aarch64			✔
RHEL/CentOS 7.7	3.10.0-1062.el7	x86			✔
RHEL/CentOS 7.8	3.10.0-1127.el7	x86			✔
RHEL/CentOS 7.9	3.10.0-1160.el7	x86			✔
RHEL/CentOS 8.1	4.18.0-147.el8	x86/aarch64			✔
RHEL/CentOS 8.3	4.18.0-240.el8	x86/aarch64			✔
RHEL/CentOS 8.4	4.18.0-305.el8	x86/aarch64			✔
RHEL/CentOS 8.5	4.18.0-305.el8	x86/aarch64			✔
RHEL/CentOS 8.8	4.18.0-477.10.1.el8_8	x86/aarch64			✔
RHEL/CentOS 8.9	4.18.0-513.5.1.el8_9	x86/aarch64			✔
RHEL/CentOS 9.0	5.14.0-70.46.1.el9_0	x86/aarch64			✔
RHEL/CentOS 9.2	5.14.0-284.11.1.el9_2	x86/aarch64			✔
RHEL/CentOS 9.3	5.14.0-362.8.1.el9_3	x86			✔
sles12sp4		x86 / aarch64			✔

DOCA for Host	Kernel	Arch	doca-all	doca-cx	doca-ofed
sles12sp5		x86 / aarch64			✔
sles15sp2		x86 / aarch64			✔
sles15sp3		x86 / aarch64			✔
sles15sp4		x86 / aarch64			✔
sles15sp5		x86 / aarch64			✔

NVIDIA DOCA Installation Guide for Linux

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

Introduction

Installation of the NVIDIA® BlueField® DPU software requires following the following step-by-step procedure.

Supported Platforms

Supported BlueField DPUs

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

Supported ConnectX NICs

The following NVIDIA® ConnectX® NICs are supported with DOCA on the host:

- ConnectX-7 devices
- ConnectX-6 Dx devices
- ConnectX-6 Lx devices
- ConnectX-6 devices

Hardware Prerequisites

This quick start guide assumes that an NVIDIA® BlueField® DPU has been installed in a server according to the instructions detailed in your DPU's hardware user guide .

DOCA Packages

Supported Operating System Distributions

The default operating system of the BlueField DPU (Arm) is Ubuntu 22.04.

The supported operating systems on the host machine are the following:

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

DOCA for Host	Kernel	Arch	doca-all	doca-cx	doca-ofed

CTYunOS 23.01	5.10	x86	✔	✔	✔
		aarch64			✔
CTYunOS 2.0	4.19	x86 / aarch64			✔
RHEL/CentOS 7.6	3.10	x86	✔	✔	✔
	4.14	aarch64	✔	✔	✔
RHEL/CentOS 8.2	4.18	x86	✔	✔	✔
RHEL / Rocky Linux 8.6	4.18	x86	✔	✔	✔
Ubuntu 18.04	4.15	x86	✔	✔	✔
Ubuntu 20.04	5.4	x86	✔	✔	✔
Ubuntu 22.04	5.15	X86	✔	✔	✔
	5.15	aarch64	✔	✔	✔
Debian 10.8	4.19	x86	✔	✔	✔
Debian 10.13	5.10.135 / 5.4.210	x86	✔	✔	✔
Allinux 3.2	5.10	X86	✔	✔	✔
Oracle Linux 8.7	5.10	x86	✔	✔	✔
RHEL/Rocky Linux 9.1	5.14.0-162.19.1.el9	x86	✔	✔	✔
Debian10.9	4.19.0-16	x86			✔
BCLinux 21.10 SP2	4.19.90	x86/aarch64			✔
Kylin 10 SP2	4.19.90	x86/aarch64			✔
openEuler 20.03 SP3	4.19.90	x86/aarch64			✔
openEuler 22.03	5.10.0	x86/aarch64			✔
UOS 20 1040d	4.19.0	x86/aarch64			✔
RHEL/CentOS 7.7	3.10.0-1062.el7	x86			✔
RHEL/CentOS 7.8	3.10.0-1127.el7	x86			✔
RHEL/CentOS 7.9	3.10.0-1160.el7	x86			✔
RHEL/CentOS 8.1	4.18.0-147.el8	x86/aarch64			✔
RHEL/CentOS 8.3	4.18.0-240.el8	x86/aarch64			✔
RHEL/CentOS 8.4	4.18.0-305.el8	x86/aarch64			✔
RHEL/CentOS 8.5	4.18.0-305.el8	x86/aarch64			✔
RHEL/CentOS 8.8	4.18.0-477.10.1.el8_8	x86/aarch64			✔
RHEL/CentOS 8.9	4.18.0-513.5.1.el8_9	x86/aarch64			✔
RHEL/CentOS 9.0	5.14.0-70.46.1.el9_0	x86/aarch64			✔
RHEL/CentOS 9.2	5.14.0-284.11.1.el9_2	x86/aarch64			✔
RHEL/CentOS 9.3	5.14.0-362.8.1.el9_3	x86			✔
sles12sp4		x86 / aarch64			✔
sles12sp5		x86 / aarch64			✔
sles15sp2		x86 / aarch64			✔
sles15sp3		x86 / aarch64			✔

sles15sp4		x86 / aarch64			✔
sles15sp5		x86 / aarch64			✔

BlueField DPU Image Installation

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

Installation Files

Device	Component	OS	Arch	Link
Host	These files contain the following components suitable for their respective OS version. • DOCA SDK v2.5.2 • DOCA Runtime v2.5.2 • DOCA Tools v2.5.2 • DOCA Extra v2.5.2 • DOCA OFED v2.5.2	Alinux 3.2	x86	doca-host-repo-alinux32-2.5.2-0.0.6.2.5.2003.1.al8.23.10.3.2.2.0.x86_6
		BCLinux 21.10 SP2	aarch64	doca-host-repo-bclinux2110sp2-2.5.2-0.0.6.23.10.3.2.2.0.oe1.bclinux.aarch64
			x86	doca-host-repo-bclinux2110sp2-2.5.2-0.0.6.23.10.3.2.2.0.oe1.bclinux.x86_64.r
		CTyunOS 2.0	aarch64	doca-host-repo-ctyunos20-2.5.2-0.0.6.23.10.3.2.2.0.ctl2.aarch64.rpm
			x86	doca-host-repo-ctyunos20-2.5.2-0.0.6.23.10.3.2.2.0.ctl2.x86_64.rpm
		CTyunOS 23.01	aarch64	doca-host-repo-ctyunos2301-2.5.2-0.0.6.23.10.3.2.2.0.ctl3.aarch64.rpm
			x86	doca-host-repo-ctyunos2301-2.5.2-0.0.6.2.5.2003.1.ctl3.23.10.3.2.2.0.x86_
		Debian 10.13	x86	doca-host-repo-debian1013_2.5.2-0.0.6.2.5.2003.1.23.10.3.2.2.0_amd64.d
		Debian 10.8	x86	doca-host-repo-debian108_2.5.2-0.0.6.2.5.2003.1.23.10.3.2.2.0_amd64.d
		Debian 10.9	x86	doca-host-repo-debian109_2.5.2-0.0.6.23.10.3.2.2.0_amd64.deb
		Debian 12.5	aarch64	doca-host-repo-debian125_2.5.2-0.0.6.2.5.2003.1.23.10.3.2.2.0_arm64.d
			x86	doca-host-repo-debian125_2.5.2-0.0.6.2.5.2003.1.23.10.3.2.2.0_amd64.d
		Kylin 1.0	aarch64	doca-host-repo-kylin10sp2-2.5.2-0.0.6.23.10.3.2.2.0.ky10.aarch64.rpm
			x86	doca-host-repo-kylin10sp2-2.5.2-0.0.6.23.10.3.2.2.0.ky10.x86_64.rpm
		Oracle Linux 7.9	x86	doca-host-repo-ol79-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		Oracle Linux 8.4	x86	doca-host-repo-ol84-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		Oracle Linux 8.6	x86	doca-host-repo-ol86-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm

Device	Component	OS	Arch	Link
		Oracle Linux 8.7	x86	doca-host-repo-ol87-2.5.2-0.0.6.2.5.2003.1.el8.23.10.3.2.2.0.x86_6
		Oracle Linux 9.0	x86	doca-host-repo-ol90-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		openEuler 20.03 SP3	aarch64	doca-host-repo-openeuler2003sp3-2.5 0.0.6.23.10.3.2.2.0.aarch64.rpm
			x86	doca-host-repo-openeuler2003sp3-2.5 0.0.6.23.10.3.2.2.0.x86_64.rpm
		openEuler 22.03	aarch64	doca-host-repo-openeuler2203-2.5.2-0.0.6.23.10.3.2.2.0.aarch64.rpm
			x86	doca-host-repo-openeuler2203-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		RHEL/CentOS 7.2	x86	doca-host-repo-rhel72-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		RHEL/CentOS 7.4	x86	doca-host-repo-rhel74-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		RHEL/CentOS 7.6	aarch64	doca-host-repo-rhel76-2.5.2-0.0.6.2.5.2003.1.el7a.23.10.3.2.2.0.aarc
			x86	doca-host-repo-rhel76-2.5.2-0.0.6.2.5.2003.1.el7.23.10.3.2.2.0.x86_6
		RHEL/CentOS 7.7	x86	doca-host-repo-rhel77-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		RHEL/CentOS 7.8	x86	doca-host-repo-rhel78-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		RHEL/CentOS 7.9	x86	doca-host-repo-rhel79-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		RHEL/CentOS 8.0	x86	doca-host-repo-rhel80-2.5.2-0.0.6.2.5.2003.1.el8.23.10.3.2.2.0.x86_6
		RHEL/CentOS 8.1	aarch64	doca-host-repo-rhel81-2.5.2-0.0.6.23.10.3.2.2.0.aarch64.rpm
			x86	doca-host-repo-rhel81-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		RHEL/CentOS 8.2	x86	doca-host-repo-rhel82-2.5.2-0.0.6.2.5.2003.1.el8.23.10.3.2.2.0.x86_6

Device	Component	OS	Arch	Link
		RHEL/CentOS 8.3	aarch64	doca-host-repo-rhel83-2.5.2-0.0.6.23.10.3.2.2.0.aarch64.rpm
			x86	doca-host-repo-rhel83-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		RHEL/CentOS 8.4	aarch64	doca-host-repo-rhel84-2.5.2-0.0.6.23.10.3.2.2.0.aarch64.rpm
			x86	doca-host-repo-rhel84-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		RHEL/CentOS 8.5	aarch64	doca-host-repo-rhel85-2.5.2-0.0.6.23.10.3.2.2.0.aarch64.rpm
			x86	doca-host-repo-rhel85-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		RHEL/Rocky 8.6	aarch64	doca-host-repo-rhel86-2.5.2-0.0.6.2.5.2003.1.el8.23.10.3.2.2.0.aarch
			x86	doca-host-repo-rhel86-2.5.2-0.0.6.2.5.2003.1.el8.23.10.3.2.2.0.x86_6
		RHEL/Rocky 8.8	aarch64	doca-host-repo-rhel88-2.5.2-0.0.6.2.5.2003.1.el8.23.10.3.2.2.0.aarch
			x86	doca-host-repo-rhel88-2.5.2-0.0.6.2.5.2003.1.el8.23.10.3.2.2.0.x86_6
		RHEL/Rocky 8.9	aarch64	doca-host-repo-rhel89-2.5.2-0.0.6.23.10.3.2.2.0.aarch64.rpm
			x86	doca-host-repo-rhel89-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		RHEL/Rocky 8.10	aarch64	doca-host-repo-rhel810-2.5.2-0.0.6.23.10.3.2.2.0.aarch64.rpm
			x86	doca-host-repo-rhel810-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		RHEL/Rocky 9.0	aarch64	doca-host-repo-rhel90-2.5.2-0.0.6.23.10.3.2.2.0.aarch64.rpm
			x86	doca-host-repo-rhel90-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm

Device	Component	OS	Arch	Link
		RHEL/Rocky 9.1	aarch64	doca-host-repo-rhel91-2.5.2-0.0.6.2.5.2003.1.el9.23.10.3.2.2.0.aarch
			x86	doca-host-repo-rhel91-2.5.2-0.0.6.2.5.2003.1.el9.23.10.3.2.2.0.x86_6
		RHEL/Rocky 9.2	aarch64	doca-host-repo-rhel92-2.5.2-0.0.6.23.10.3.2.2.0.aarch64.rpm
			x86	doca-host-repo-rhel92-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		RHEL/Rocky 9.3	aarch64	doca-host-repo-rhel93-2.5.2-0.0.6.23.10.3.2.2.0.aarch64.rpm
			x86	doca-host-repo-rhel93-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		RHEL/Rocky 9.4	aarch64	doca-host-repo-rhel94-2.5.2-0.0.6.23.10.3.2.2.0.aarch64.rpm
			x86	doca-host-repo-rhel94-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		SLES 12 SP4	aarch64	doca-host-repo-sles12sp4-2.5.2-0.0.6.23.10.3.2.2.0.aarch64.rpm
			x86	doca-host-repo-sles12sp4-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		SLES 12 SP5	aarch64	doca-host-repo-sles12sp5-2.5.2-0.0.6.23.10.3.2.2.0.aarch64.rpm
			x86	doca-host-repo-sles12sp5-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		SLES 15 SP2	aarch64	doca-host-repo-sles15sp2-2.5.2-0.0.6.23.10.3.2.2.0.aarch64.rpm
			x86	doca-host-repo-sles15sp2-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		SLES 15 SP3	aarch64	doca-host-repo-sles15sp3-2.5.2-0.0.6.23.10.3.2.2.0.aarch64.rpm
			x86	doca-host-repo-sles15sp3-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm

Device	Component	OS	Arch	Link
		SLES 15 SP4	aarch64	doca-host-repo-sles15sp4-2.5.2-0.0.6.23.10.3.2.2.0.aarch64.rpm
			x86	doca-host-repo-sles15sp4-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		SLES 15 SP5	aarch64	doca-host-repo-sles15sp5-2.5.2-0.0.6.23.10.3.2.2.0.aarch64.rpm
			x86	doca-host-repo-sles15sp5-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		SLES 15 SP6	x86	doca-host-repo-sles15sp6-2.5.2-0.0.6.23.10.3.2.2.0.x86_64.rpm
		Ubuntu 18.04	x86	doca-host-repo-ubuntu1804_2.5.2-0.0.6.2.5.2003.1.23.10.3.2.2.0_amd64.d
		Ubuntu 20.04	x86	doca-host-repo-ubuntu2004_2.5.2-0.0.6.2.5.2003.1.23.10.3.2.2.0_amd64.d
		Ubuntu 22.04	aarch64	doca-host-repo-ubuntu2204_2.5.2-0.0.6.2.5.2003.1.23.10.3.2.2.0_arm64.d
			x86	doca-host-repo-ubuntu2204_2.5.2-0.0.6.2.5.2003.1.23.10.3.2.2.0_amd64.d
		UOS 20 1040d	aarch64	doca-host-repo-uos201040_2.5.2-0.0.6.23.10.3.2.2.0_arm64.deb
			x86	doca-host-repo-uos201040_2.5.2-0.0.6.23.10.3.2.2.0_amd64.deb
Target BlueField DPU (Arm)	BlueField Software v 4.5.2	Ubuntu 22.04	aarch64	doca_2.5.2_bsp_4.5.2_ubuntu_22.04-206-lts.prod.bfb
	DOCA SDK v2.5.2	Ubuntu 22.04	aarch64	doca-dpu-repo-ubuntu2204-local_2.5.1.23.10.3.2.2.bf.4.5.2.13183_arm64.deb
	DOCA Runtime v2.5.2			
	DOCA Tools v2.5.2			

Uninstalling Software from Host

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

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

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

Then perform the following steps:

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 )
```

Installing Prerequisites on Host for Target DPU

Install `doca-tools` to manage and flash the BlueField DPU.

OS	Procedure

Ubuntu/Debian	1. Download the DOCA host repo package from the "Installation Files" section.
	2. Unpack the deb repo. Run:
	`host# sudo dpkg -i doca-host-repo-ubuntu<version>_amd64.deb`
	3. Perform apt update. Run:
	`host# sudo apt-get update`
	4. Run `apt install` for DOCA Tools:
	`host# sudo apt install doca-tools`

OS	Procedure

CentOS/RHEL 7.x	1. Download the DOCA host repo package from the "Installation Files" section.
	2. Unpack the RPM repo. Run:
	`host# sudo rpm -Uvh doca-host-repo-rhel<version>.x86_64.rpm`
	3. Enable new yum repos. Run:
	`host# sudo yum makecache`
	4. Run `yum install` to install DOCA Tools:
	`host# sudo yum install doca-tools`

CentOS/RHEL 8.x or Rocky 8.6	1. Download the DOCA host repo package from the "Installation Files" section.
	2. Unpack the RPM repo. Run:
	`host# sudo rpm -Uvh doca-host-repo-rhel<version>.x86_64.rpm`
	3. Enable new dnf repos. Run:
	`host# sudo dnf makecache`
	4. Run `dnf install` to install DOCA Tools:
	`host# sudo dnf install doca-tools`

Note

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

Determining DPU Device ID

It is is important to learn your DPU's `device-id` for performing some of the software installations or upgrades in this guide.

To determine the device ID of the DPUs on your setup, run:

```
host# mst start
host# mst status -v
```

Example output:

```
MST modules:
------------
    MST PCI module is not loaded
    MST PCI configuration module loaded
PCI devices:
------------
DEVICE_TYPE             MST                           PCI
RDMA            NET                         NUMA
BlueField2(rev:1)       /dev/mst/mt41686_pciconf0.1   3b:00.1
mlx5_1          net-ens1f1                  0

BlueField2(rev:1)       /dev/mst/mt41686_pciconf0     3b:00.0
mlx5_0          net-ens1f0                  0

BlueField3(rev:1)       /dev/mst/mt41692_pciconf0.1   e2:00.1
mlx5_1          net-ens7f1np1               4

BlueField3(rev:1)       /dev/mst/mt41692_pciconf0     e2:00.0
mlx5_0          net-ens7f0np0               4
```

The device IDs for the BlueField-2 and BlueField-3 DPUs in this example are `/dev/mst/mt41686_pciconf0` and `/dev/mst/mt41692_pciconf0` respectively.

Installing Software on Host

Note

Skip this section if you intend to update only the BlueField software (`*.bfb`).

1. Follow the instructions under "Installing Prerequisites on Host for Target DPU".

2. Install DOCA local repo package for host:

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

OS	Profile	Device	Instructions

Ubuntu/ Debian	**doca-all**	**BlueField**	1. Download the DOCA SDK and DOCA Runti Files" for the host.
			2. Unpack the deb repo. Run:
			`host# dpkg -i <repo_file>`
			3. Perform apt update. Run:
			`host# apt-get update`
			4. If the kernel version on your host is not su Operating System Distributions"), refer to
			5. Run apt install for DOCA SDK, DOCA runtir
			`host# sudo apt install -y doc`

	doca-cx	**ConnectX**	1. Download the DOCA SDK and DOCA Runti Files" for the host.
			2. Unpack the deb repo. Run:
			`host# dpkg -i <repo_file>`
			3. Perform apt update. Run:
			`host# apt-get update`
			4. If the kernel version on your host is not su Operating System Distributions"), refer to
			5. Run apt install for DOCA SDK, DOCA runtir
			`host# sudo apt install -y doc cx-tools`

OS	Profile	Device	Instructions

	doca-ofed	**All**	1. Download the DOCA SDK and DOCA Runti
			Files" for the host.
			2. Unpack the deb repo. Run:
			`host# sudo dpkg -i doca-host-r`
			3. Perform apt update. Run:
			`host# sudo apt-get update`
			4. If the kernel version on your host is not su
			Operating System Distributions"), refer to
			5. Install `doca-ofed`. Run:
			`host# sudo apt install -y doca`

OS	Profile	Device	Instructions

CentOS/RHEL/ Alinux/Rocky/ Oracle Linux	**doca-all**	**BlueField**	**Note**
			RHEL users need a valid subscription to i
			Open RedHat Account".
			
			1. Install the following software dependencie
			• For RH 8.x, run:
			`host# yum -y install https`
			latest-8.noarch.rpm
			• For Alinux 3.x (Alibaba Cloud Linux), r
			`host# yum -y install https`
			latest-8.noarch.rpm
			• For Rocky 8.x, run:
			`host# sudo dnf makecache`
			`host# sudo dnf install ep`
			`host# sudo dnf install -y`
			`host# sudo dnf config-man`
			`host# sudo dnf clean dbca`
			• For CentOS 8.x, run:
			`host# sudo dnf install ep`
			`host# sudo dnf config-man`
			• For CentOS 7.x, run:

OS	Profile	Device	Instructions
			``` host# sudo yum install -y release-latest-7.noarch.rpm host# sudo yum-config-man http://mirror.centos.org/centos/7/os/x host# sudo rpm --import htt GPG-KEY-CentOS-7 host# sudo yum-config-man http://mirror.centos.org/centos/7/extr host# sudo yum-config-man setopt=mirror.centos.org_ libnl3*' host# yum makecache ``` 2. Download the DOCA SDK and DOCA Runti Files" for the host . 3. Unpack the rpm repo. Run: ``` host# rpm -Uvh <repo_file>.rpr ``` 4. Perform yum update. Run: ``` host# sudo yum makecache ``` 5. If the kernel version on your host is not su Operating System Distributions"), refer to 6. Run yum install for DOCA SDK, DOCA runt ``` host# sudo yum install -y doc ```

OS	Profile	Device	Instructions

	doca-cx	**ConnectX**	1. Install the following software dependencie
			▪ For RH 8.x, run:
			`host# yum -y install https`
			latest-8.noarch.rpm
			▪ For Alinux 3.x (Alibaba Cloud Linux), r
			`host# yum -y install https`
			latest-8.noarch.rpm
			▪ For Rocky 8.x, run:
			`host# sudo dnf makecache`
			`host# sudo dnf install ep`
			`host# sudo dnf install -y`
			`host# sudo dnf config-man`
			`host# sudo dnf clean dbca`
			▪ For CentOS 8.x, run:
			`host# sudo dnf install ep`
			`host# sudo dnf config-man`
			▪ For CentOS 7.x, run:

OS	Profile	Device	Instructions
			host# sudo yum install -y release-latest-7.noarch.rpm host# sudo yum-config-man http://mirror.centos.org/centos/7/os/x host# sudo rpm --import htt GPG-KEY-CentOS-7 host# sudo yum-config-man http://mirror.centos.org/centos/7/extr host# sudo yum-config-man setopt=mirror.centos.org_ libnl3*' host# yum makecache ▪ For Oracle Linux 8.7, run: host# yum config-manager ol8_codeready_builder 2. Download the DOCA SDK and DOCA Runti Files" for the host . 3. Unpack the rpm repo. Run: host# rpm -Uvh <repo_file>.rpr 4. Perform yum update. Run: host# sudo yum makecache 5. If the kernel version on your host is not su Operating System Distributions"), refer to 6. Run yum install for DOCA SDK, DOCA runt host# sudo yum install -y doc cx-tools

OS	Profile	Device	Instructions

	doca-ofed	**All**	1. Download the DOCA SDK and DOCA Runti Files" for the host.
			2. Unpack the RPM repo. Run:
			`host# sudo rpm -Uvh doca-host-`
			3. Install doca-extra. Run:
			`host# sudo yum/dnf install -y`
			4. If the kernel version on your host is not su Operating System Distributions"), refer to
			5. Install `doca-ofed`. Run:
			`host# sudo yum install -y doca`

3. Initialize MST. Run:

```
host# sudo mst start
```

4. Reset the `nvconfig` parameters to their default values:

```
host# sudo mlxconfig -d <device-id> -y reset

Reset configuration for device <device-id>? (y/n) [n] : y
Applying... Done!
-I- Please reboot machine to load new configurations.
```

> **Note**
>
> To learn the DPU's device ID, refer to section "Determining DPU Device ID".

5. Skip this step if your BlueField DPU is Ethernet only. Please refer to Supported Platforms to learn your DPU type.

   If you have a VPI DPU, the default link type of the ports will be configured to IB. To verify your link type, run:

```
host# sudo mst start
host# sudo mlxconfig -d <device-id> -e q | grep -i link_type
Configurations:                                Default
Current         Next Boot
*        LINK_TYPE_P1                          IB(1)
ETH(2)          IB(1)
*        LINK_TYPE_P2                          IB(1)
ETH(2)          IB(1)
```

> **Note**
>
> If your DPU is Ethernet capable only, then the `sudo mlxconfig -d <device>` command will not provide an output.

   If the current link type is set to IB, run the following command to change it to Ethernet:

```
host# sudo mlxconfig -d <device-id> s LINK_TYPE_P1=2
LINK_TYPE_P2=2
```

6. Verify that RShim is active.

```
host# sudo systemctl status rshim
```

This command is expected to display `active (running)`. If RShim service does not launch automatically, run:

```
host# sudo systemctl enable rshim
host# sudo systemctl start rshim
```

7. Assign a dynamic IP to `tmfifo_net0` interface (RShim host interface).

> **Note**
>
> Skip this step if you are installing the DOCA image on multiple DPUs.

```
host# ifconfig tmfifo_net0 192.168.100.1 netmask 255.255.255.252 up
```

DOCA Extra Package

If the kernel version on 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:

    1. Run:

    ```
    host# sudo apt install -y doca-tools doca-extra
    ```

    2. Execute the `doca-kernel-support` script which downloads the appropriate missing packages from your kernel to support DOCA:

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

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

Installing Software on DPU

Users have two options for installing DOCA on the DPU:

- Upgrading the full DOCA image on the DPU (recommended) – this option overwrites the entire boot partition
- Upgrading DOCA local repo package on the DPU – this option upgrades DOCA components without overwriting the boot partition. Use this option to preserve configurations or files on the DPU itself.

Installing Full DOCA Image on DPU via Host

Warning

This step overwrites the entire boot partition.

Note

This installation sets up the OVS bridge.

Note

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

Option 1 – No Pre-defined Password

Note

To set the password in advance, proceed to Option 2.

BFB installation is executed as follows:

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

Where `rshimN` is `rshim0` if you only have one DPU. You may run the following command to verify:

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

Option 2 – Set Pre-defined Password

Ubuntu users can provide a unique password that will be applied at the end of the BlueField software image 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# sudo vim bf.cfg
   ubuntu_PASSWORD='$1$3B0RIrfX$TlHry93NFUJzg3Nya00rE1'
   ```

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

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

> **Note**
>
> If `--config` is not used, then upon first login to the BlueField device, users will be asked to update their password.

The following is an example of Ubuntu installation assuming the `pv` Linux tool has been installed (to view the installation progress).

```
host# sudo bfb-install --rshim rshim0 --bfb DOCA_<version>-
aarch64.bfb --config bf.cfg
Pushing bfb
1.08GiB 0:00:57 [19.5MiB/s] [      <=>     ]
Collecting BlueField booting status. Press Ctrl+C to stop…
INFO[BL2]: start
INFO[BL2]: DDR POST passed
INFO[BL2]: UEFI loaded
INFO[BL31]: start
INFO[BL31]: runtime
INFO[UEFI]: eMMC init
INFO[UEFI]: eMMC probed
INFO[UEFI]: PCIe enum start
INFO[UEFI]: PCIe enum end
INFO[MISC]: Ubuntu installation started
INFO[MISC]: Installation finished
INFO[MISC]: Rebooting...
```

Installing Full DOCA Image on Multiple DPUs

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

```
host# ./multi-bfb-install --bfb <bfb-file> --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 7.6 – the script creates a configuration file `/etc/sysconfig/network-scripts/ifcfg-br_tmfifo`
- 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`.

Note

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 DPU and configures a new IP, 192.168.100.x, as described earlier.

Installing DOCA Local Repo Package on DPU

Note

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.

Note

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

1. Download the DOCA SDK, DOCA Runtime, and DOCA Tools 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, DOCA Tools, and DOCA SDK:

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

Upgrading Firmware

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

To upgrade firmware:

1. SSH to your BlueField device via 192.168.100.2 (preconfigured).

> **Note**
>
> If multiple DPUs are installed, the tmfifo IP interface does not have to be 192.168.100.2. The last octate changes and depends on the RShim number.

The default credentials for Ubuntu are as follows:

Username	Password
ubuntu	`ubuntu` or a unique password that you set in `bf.cfg`

For example:

```
host# ssh ubuntu@192.168.100.2
Password: <configured-password>
```

2. Upgrade firmware in BlueField DPU. 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:

    1. Run the following command on the BlueField DPU and host:

    ```
    dpu# sudo mst start
    ```

    2. Otherwise, trigger reset by running the following:

```
dpu# sudo mlxfwreset -d /dev/mst/mt41686_pciconf0 --sync
1 -y reset
```

If the `mlxfwreset` command fails, perform a graceful reboot and host power cycle.

> **Note**
>
> The entire DPU will experience reset.

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.

Upgrading BlueField DPU Using Standard Linux Tools

This dpu-upgrade procedure enables upgrading DOCA components using standard Linux tools (e.g., `apt update` and `yum update`). This process utilizes native package manager repositories to upgrade DPUs without the need for a full installation, and has the following benefits :

- Only updates components that include modifications
    - Configurable – user can select specific components (e.g., UEFI-ATF, NIC-FW)
- Includes upgrade of:
    - DOCA drivers and libraries
    - DOCA reference applications
    - BSP (UEFI/ATF) upgrade while maintaining the configuration
    - NIC firmware upgrade while maintaining the configuration
- Does not:
    - Impact user binaries
    - Upgrade non-Ubuntu OS kernels
    - Upgrade DPU BMC firmware
- After completion of DPU upgrade:
    - If NIC firmware was not updated, perform DPU Arm reset (software reset / reboot DPU)

- If NIC firmware was updated, perform firmware reset (`mlxfwreset`) or perform a graceful shutdown and power cycle

OS	Action	Instructions
Ubuntu/ Debian	Remove `mlxbf-bootimages` package	`<dpu> $ apt remove --purge mlxbf-bootima`
	Install the the GPG key	`<dpu> $ apt update` `<dpu> $ apt install gnupg2`
	Export the desired distribution	Export `DOCA_REPO` with the relevant URL. The following `<dpu> $ export DOCA_REPO="https://linux.mellanox.com/pu arm64"` • Ubuntu 22.04 – https://linux.mellanox.com/public/re • Ubuntu 20.04 – https://linux.mellanox.com/public/re • Debian 12 – https://linux.mellanox.com/public/repo/
	Add GPG key to APT trusted keyring	`<dpu> $ curl $DOCA_REPO/GPG-KEY-Mellanox /etc/apt/trusted.gpg.d/GPG-KEY-Mellanox.`
	Add DOCA online repository	`<dpu> $ echo "deb [signed-by=/etc/apt/tr $DOCA_REPO ./" > /etc/apt/sources.list.d`
	Update index	`<dpu> $ apt update`

OS	Action	Instructions

	Upgrade UEFI/ATF firmware	Run:
		`<dpu> $ apt install mlxbf-bootimages-sig`
		Then i nitiate upgrade for UEFI/ATF firmware:
		`<dpu> $ bfrec`

	Upgrade BlueField DPU NIC firmware	Run:
		`<dpu> $ apt install mlnx-fw-updater-sign`
		Note
		This immediately starts NIC firmware upgrade.
		To prevent automatic upgrade, run:
		`<dpu> $ export RUN_FW_UPDATER=no`

	Upgrade system	`<dpu> $ apt upgrade`

OS	Action	Instructions
	Apply the new changes, NIC firmware, and UEFI/ATF	`<dpu> $ mlxfwreset -d /dev/mst/mt*_pcico` **Note** If `mlxfwreset` is not supported, graceful shutd required for the NIC firmware upgrade to take eff

OS	Action	Instructions

CentOS/RHEL/ Anolis/Rocky	Remove `mlxbf-bootimages`, librerswan, and openvswitch-ipsec packages	`<dpu> $ yum -y remove mlxbf-bootimages*`
		`<dpu> $ yum remove libreswan openvswitch`
		`<dpu> $ yum makecache`

	Export the desired distribution	Export `DOCA_REPO` with the relevant URL. The following `<dpu> $ export DOCA_REPO="https://linux.mellanox.com/pu arm64/"` • AnolisOS 8.6 – https://linux.mellanox.com/public/rep • OpenEuler 20.03 sp1 – https://linux.mellanox.com/p arm64/ • CentOS 7.6 with 4.19 kernel – https://linux.mellano arm64/ • CentOS 7.6 with 5.10 kernel – https://linux.mellano arm64/ • CentOS 7.6 with 5.4 kernel – https://linux.mellanox. • Rocky Linux 8.6 – https://linux.mellanox.com/public/

	Add DOCA online repository	`echo "[doca]`
		`name=DOCA Online Repo`
		`baseurl=$DOCA_REPO`
		`enabled=1`
		`gpgcheck=0`
		`priority=10`
		`cost=10" > /etc/yum.repos.d/doca.repo`
		A file is created under `/etc/yum.repos.d/doca.repo`

	Update index	`<dpu> $ yum makecache`

OS	Action	Instructions

	Upgrade UEFI/ATF firmware	Run:
		`<dpu> $ yum install mlxbf-bootimages-sig`
		Then i nitiate the upgrade for UEFI/ATF firmware:
		`<dpu> $ bfrec`

	Upgrade BlueField DPU NIC firmware	The following command updates the firmware package a to the NIC:
		`<dpu> $ yum install mlnx-fw-updater-sign`
		Note
		To prevent automatic flashing of the firmware to
		`<dpu> $ export RUN_FW_UPDATER=no`
		00000194-6a9d-d115-addd-fb9f21880003 0000 fb9f21880007

	Upgrade system	`<dpu> $ yum upgrade --nobest`

OS	Action	Instructions
	Apply the new changes, NIC firmware, and UEFI/ATF	`<dpu> $ mlxfwreset -d /dev/mst/mt*_pcico` **Note** If `mlxfwreset` is not supported, a graceful shut required for the NIC firmware upgrade to take eff

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.

Note

For a customized BlueField OS image to boot on the UEFI secure-boot-enabled DPU (default DPU 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.

Setting Up Build Environment for Developers

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

Additional SDKs for DOCA

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

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

> **Note**
>
> 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.

Configuring Operation Mode

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

- Standard mode (default) – the BlueField DPU and the GPU operate separately
- BlueField-X mode – the GPU is exposed to the DPU 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]
```

Note

To learn the DPU's device ID, refer to section "Determining DPU 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
```

Note

To learn the DPU's device ID, refer to section "Determining DPU Device ID".

Graceful reboot and power cycle are required for configuration to take effect. To power cycle the host run:

```
host# ipmitool power cycle
```

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.

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

> **Note**
>
> 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.

    1. Download CUDA samples repo. Run:

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

    2. Build and run vectorAdd CUDA sample. Run:

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

Note

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

Note

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
```

GPUDirect RDMA

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

Installing Rivermax on DPU

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.

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. D ownload 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`.

Installing Rivermax Driver

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

```
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
```

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.

Info

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

NVIDIA DOCA Developer Guide

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

Introduction

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

For steps to install DOCA on NVIDIA® BlueField® DPU, refer to the NVIDIA DOCA Installation Guide for Linux.

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

- BlueField DPU is accessible and can be used during the development and testing process
    - Working within a development container
- BlueField DPU 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 former case, leveraging the DPU during the development and testing process.

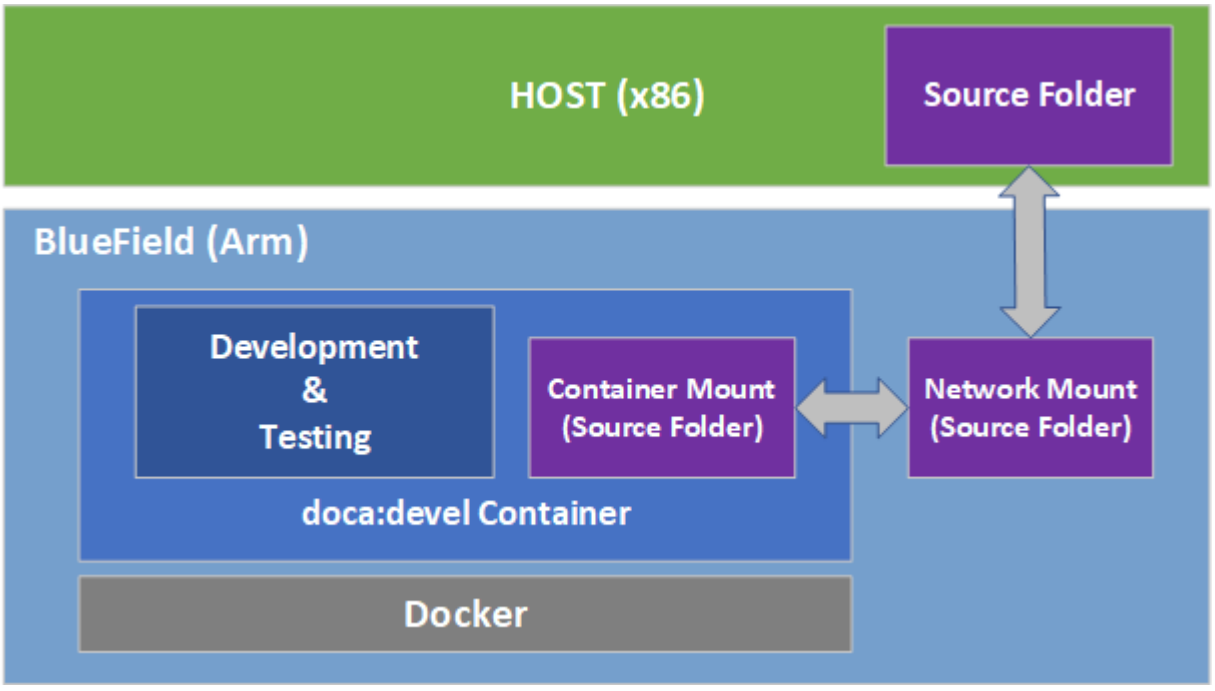
This guide recommends using DOCA's development container during the development process, whether it is on the DPU or on the host. Deploying development containers allows multiple developers to work simultaneously on the same device (host or DPU) in an isolated manner and even across multiple different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

Developing Using BlueField DPU

Setup

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



The recommended approach for working using DOCA's development container on top of the DPU, 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:

    1. Visit the NGC page of the DOCA base image.
    2. Under the "Tags" menu, select the desired development tag.
    3. 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`

> **Note**
>
> 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 the DPU
- `-v /dev/hugepages:/dev/hugepages` ensures that allocated huge pages are accessible to the container

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 the DPU (refer to NVIDIA DOCA DPU CLI for more information) and to the container.

> **Note**
>
> 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.

Testing

The container is marked as "privileged", hence it can directly access the HW capabilities of the BlueField DPU. 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 the DPU and running it there.

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:

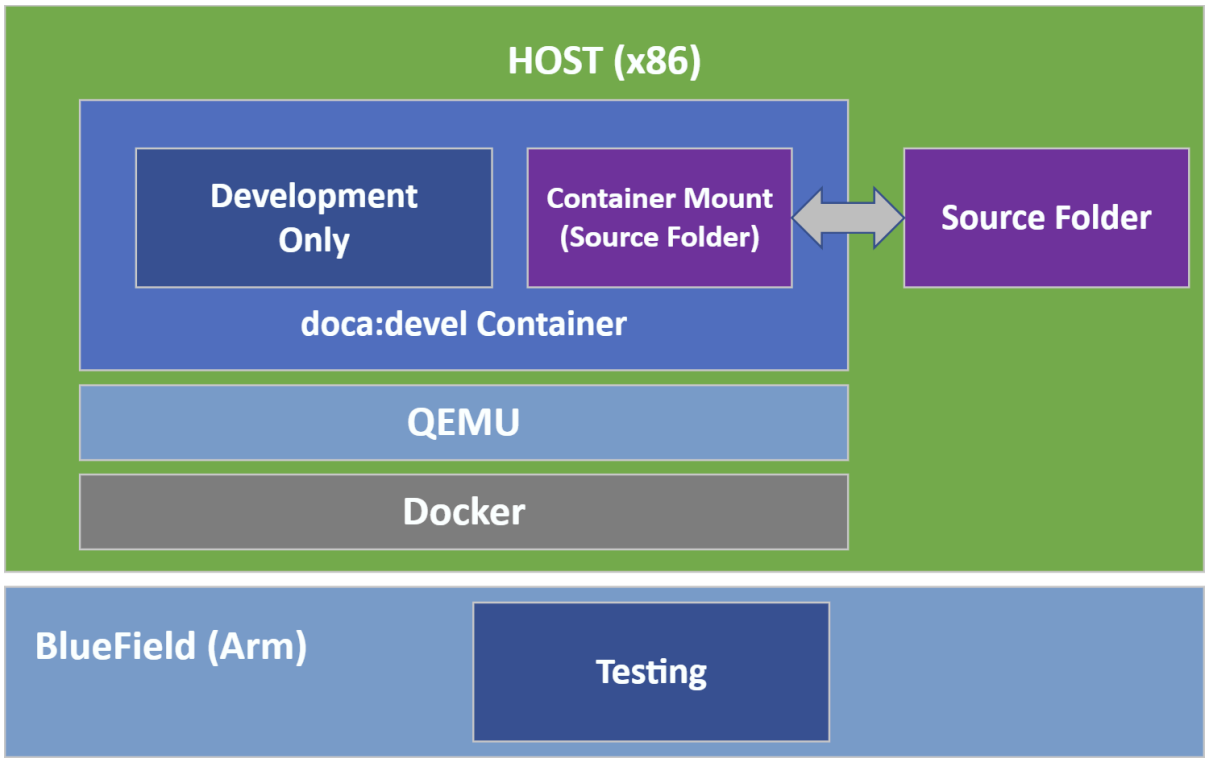
- `doca:base-rt` – slim DOCA runtime environment
- `doca:full-rt` – full DOCA runtime environment similar to the BlueField image

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 the DPU.

Developing Without BlueField DPU

If the development process needs to be done without access to a BlueField DPU, 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.



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.

> **Note**
>
> 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`

Host OS	Command
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
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 DPU side.

Note

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 the DPU). 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
```

Development

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

Testing

While the compilation can be performed on top of the container, testing the compiled software must be done on top of a BlueField DPU. This is because the QEMU environment emulates an aarch64 architecture, but it does not emulate the hardware devices present on the BlueField DPU. 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 DPU.

Note

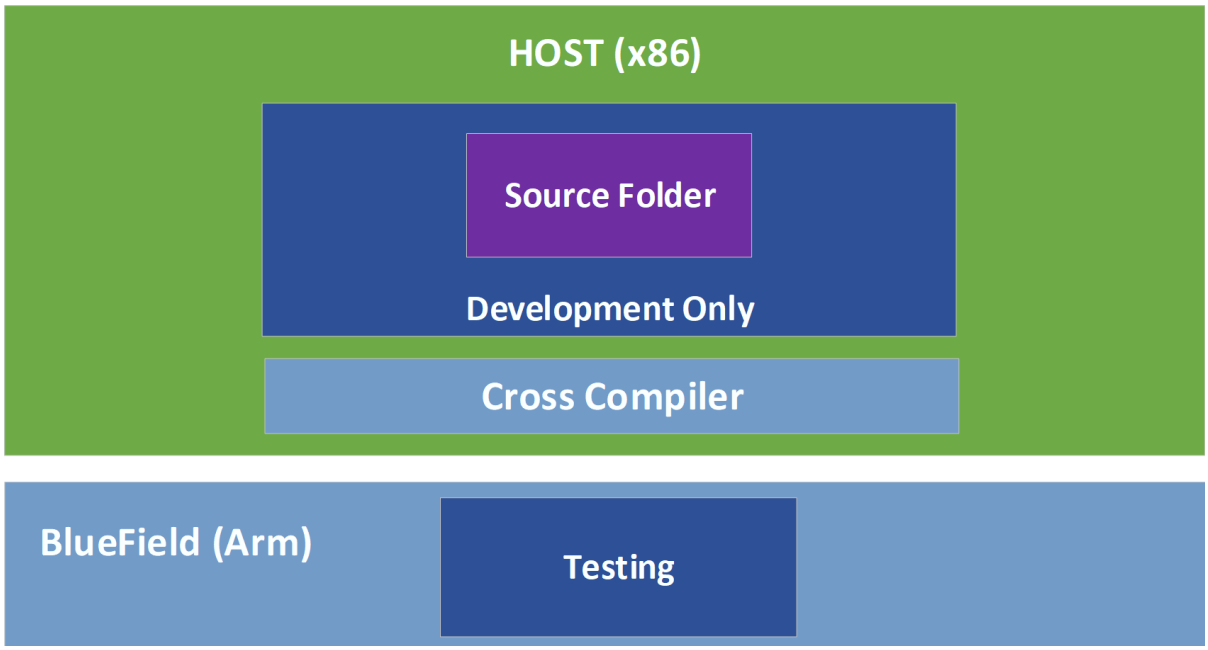
Make sure that the DOCA version used for compilation is the same as the version installed on the DPU used for testing.

Publishing

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

Cross-compilation from Host

In a typical setup, developers prefer to work on a familiar host since compilation is often significantly faster there. Therefore, developers may work on their host while cross-compiling their project to the DPU's Arm architecture.



Setup

1. Install Docker and QEMU your host. See steps 1-4 under section Setup.

2. Download the `doca-cross` component as described in the NVIDIA DOCA Installation Guide for Linux and unpack it under the `/root` directory.

   Inside this directory one can find:

   - `arm64_armv8_linux_gcc` – cross file containing specific information about the cross compiler and the host machine
   - `DOCA_cross.sh` – script which handles all the required dependencies and pre-installations steps
   - A .txt file used by the script

3. To load the development container, refer to section "Docker Deployment" of the NVIDIA BlueField DPU Container Deployment Guide.

   > **Note**
   >
   > It is important to ensure that the same DOCA version is used in the development container and the DOCA metapackages installed on the host.

4. Start running the container using the container's tag while mapping the `doca-cross` directory to the container's `/doca_devel` directory:

   ```
   sudo docker run -v /root/doca-cross/:/doca_devel --privileged
   -it <imagename/ID>
   ```

   Now the shell will be redirected to be within the container.

5. Run the preparation script to copy all the Arm dependencies required for DOCA's cross compilation. The script will be in the mapped directory named `doca_devel`.

```
(container) /# cd doca_devel/
(container) /doca_devel# ./DOCA_cross.sh
```

6. Exit the container and run the same script from the host side:

```
(host) /root/doca-cross# ./DOCA_cross.sh
```

The `/root/doca-cross` directory is now fully configured and prepared for cross-compilation against DOCA.

7. Update the environment variables to point at the Linaro cross-compiler:

```
export PATH=${PATH}:/opt/gcc-linaro/<linaro_version_dir>/aarch64-linux-gnu/bin:/opt/gcc-linaro/<linaro_version_dir>/bin
```

Everything is set up and the cross-compilation can now be used.

> **Note**
>
> Make sure to update the command according to the Linaro version installed by the script in the previous step.
> `<linaro_version_dir>` can be found under `/opt/gcc-linaro/`.

> **Note**
>
> Cross-compilation requires Meson version ⩾0.61.2 to be installed on the host. This is already provided as part of DOCA's installation.

DOCA and CUDA Setup

1. To cross-compile DOCA and CUDA applications, you must install CUDA Toolkit 12.1:
    1. The first toolkit installation is for x86 architecture. Select `x86_64`.
    2. The second toolkit installation is for Arm. Select `arm64-sbsa` and then cross.
    3. Select your host operating system, architecture, OS distribution, and version and select the installation type. It is recommended to use the deb (local) type.
2. Execute the following exports:

```
export CPATH=/usr/local/cuda/targets/sbsa-
linux/include:$CPATH
export LD_LIBRARY_PATH=/usr/local/cuda/targets/sbsa-
linux/lib:$LD_LIBRARY_PATH
export PATH=/usr/local/cuda/bin:/usr/local/cuda-11.6/bin:$PATH
```

3. Verify the `meson` version is at least 0.61.2 as provided with DOCA's installation.

Everything is set up and the cross-compilation can now be used.

Development

It is recommended to develop normally while remembering to compile using the cross-compilation configuration file `arm64_armv8_linux_gcc` which can be found under the `doca-cross` directory.

The following is an example procedure for cross-compiling DOCA applications from the host and to the Arm architecture:

1. Enable the meson cross-compilation option in `/opt/mellanox/doca/applications/meson_options.txt` by setting `enable_cross_compilation_to_dpu` to `true`.

2. Cross-compile the DOCA applications:

   ```
   /opt/mellanox/doca/applications # meson cross-build --cross-file /root/doca-cross/arm64_armv8_linux_gcc
   /opt/mellanox/doca/applications # ninja -C cross-build
   ```

   The cross-compiled binaries are created under the `cross-build` directory.

3. Cross-compile the DOCA and CUDA application:

   1. Set flag for GPU-enabled cross-compilation, `enable_gpu_support`, in `/opt/mellanox/doca/applications/meson_options.txt` to `true`.

   2. Run the compilation command as follows:

      ```
      /opt/mellanox/doca/applications # meson cross-build --cross-file /root/doca-cross/arm64_armv8_linux_gcc -Dcuda_ccbindir=aarch64-linux-gnu-g++
      /opt/mellanox/doca/applications # ninja -C cross-build
      ```

      The cross-compiled binaries are created under the `cross-build` directory.

Note

Due to the system's use of the PKG_CONFIG_PATH environment variable, it is crucial that the cross file include the following:

```
[built-in options]
pkg_config_path = "
```

This definition, already provided as part of the supplied cross file, guarantees that meson does not accidently use the build system's environment variable during the cross build.

Testing

While the compilation can be performed on top of the host, testing the compiled software must be done on top of a BlueField-2 DPU. This is because the tested program is not able to access the devices needed for its successful execution, which mandates that the testing is performed on top of a physical DPU.

Note

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

Publishing

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

Info

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

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.

- 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.

Info

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

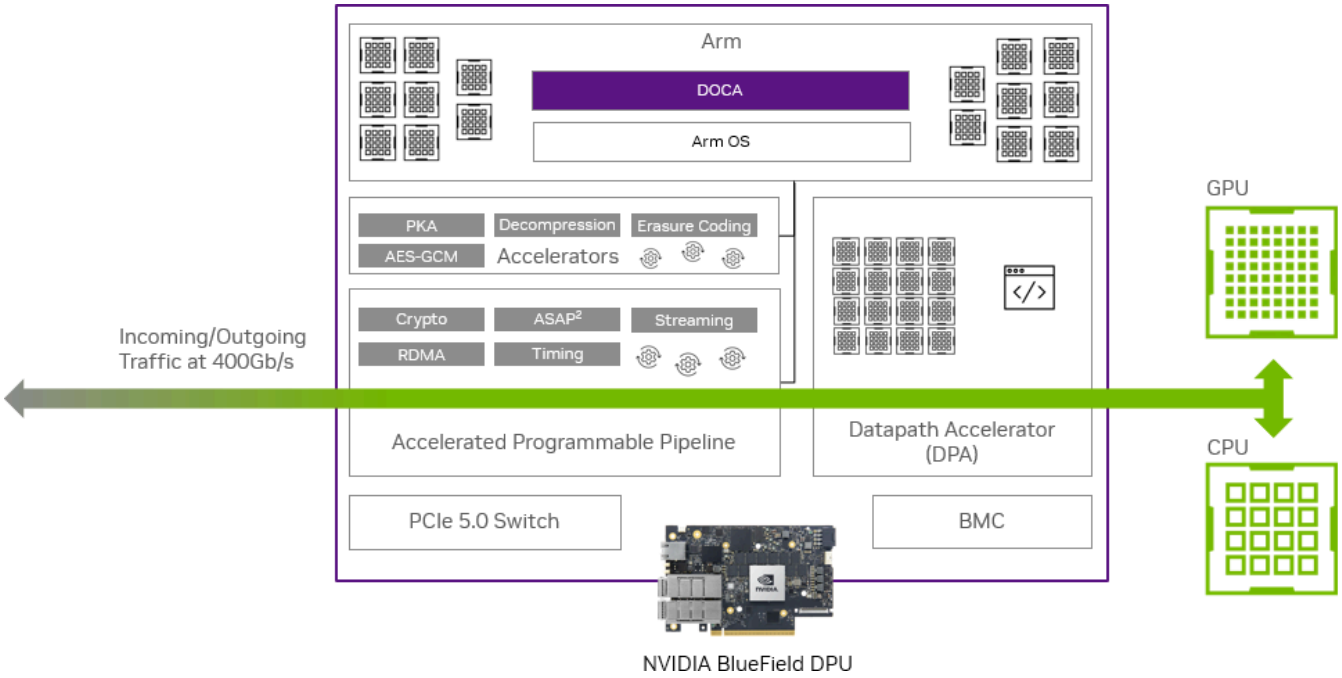
DOCA Programming Overview

This section contains the following pages:

- Hardware Overview
- DOCA SDK Architecture

Hardware Overview

DOCA is the software framewor k 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

DOCA SDK Architecture

DOCA provides libraries for networking and data processing programmability that leverage NVIDIA® BlueField® DPU 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.

Device Subsystem

The DOCA SDK allows applications to offload resource intensive tasks to HW, such as encryption, and compression.

The SDK also allows applications to offload network related tasks, such as packet acquisition, and RDMA send.

As such DPUs/NICs provide dedicated HW processing units for executing such tasks.

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

DOCA Device subsystem provides means to:

- Discover available hardware acceleration units provided by DPUs/NICs

- Query capabilities and properties of available hardware acceleration units

- Open device to enable libraries to allocate and share resources necessary for HW acceleration

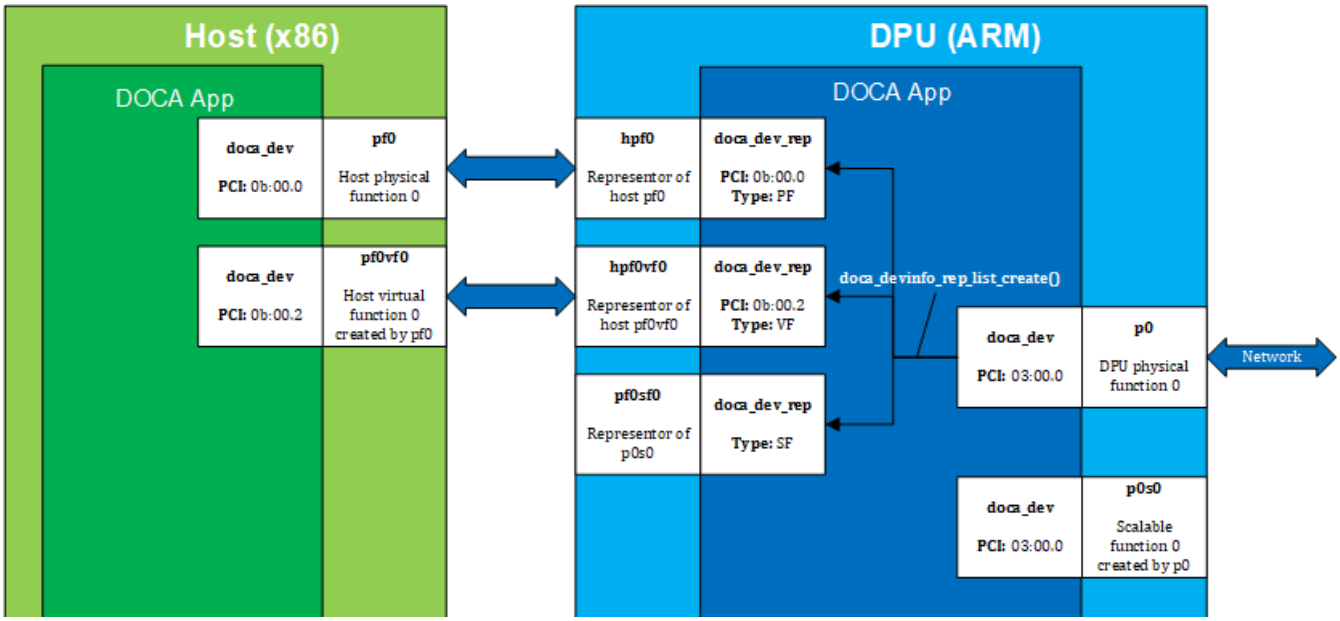
On a given system there can be multiple available devices. An application can choose a device based on the following characteristics:

- Topology - E.g., PCIe address
- Capabilities - E.g., Encryption support

DOCA Core supports two DOCA Device types:

- Local device – this is an actual device exposed in the local system (DPU or host) and can perform DOCA library processing jobs. This can be a PCI 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 usually on the host (except for SFs) and the representor is always on the DPU side (a proxy on the DPU for the host-side device).

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



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

The diagram shows a DPU (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 DPU 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 the DPU.

For more details about DOCA Device subsystem, see section "DOCA Device".

Memory Management Subsystem

HW 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.

In order to achieve maximum performance, the SDK uses zero-copy technology to pass data to the HW. To allow zero-copy, the application must register the memory that will 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 that will be used 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:

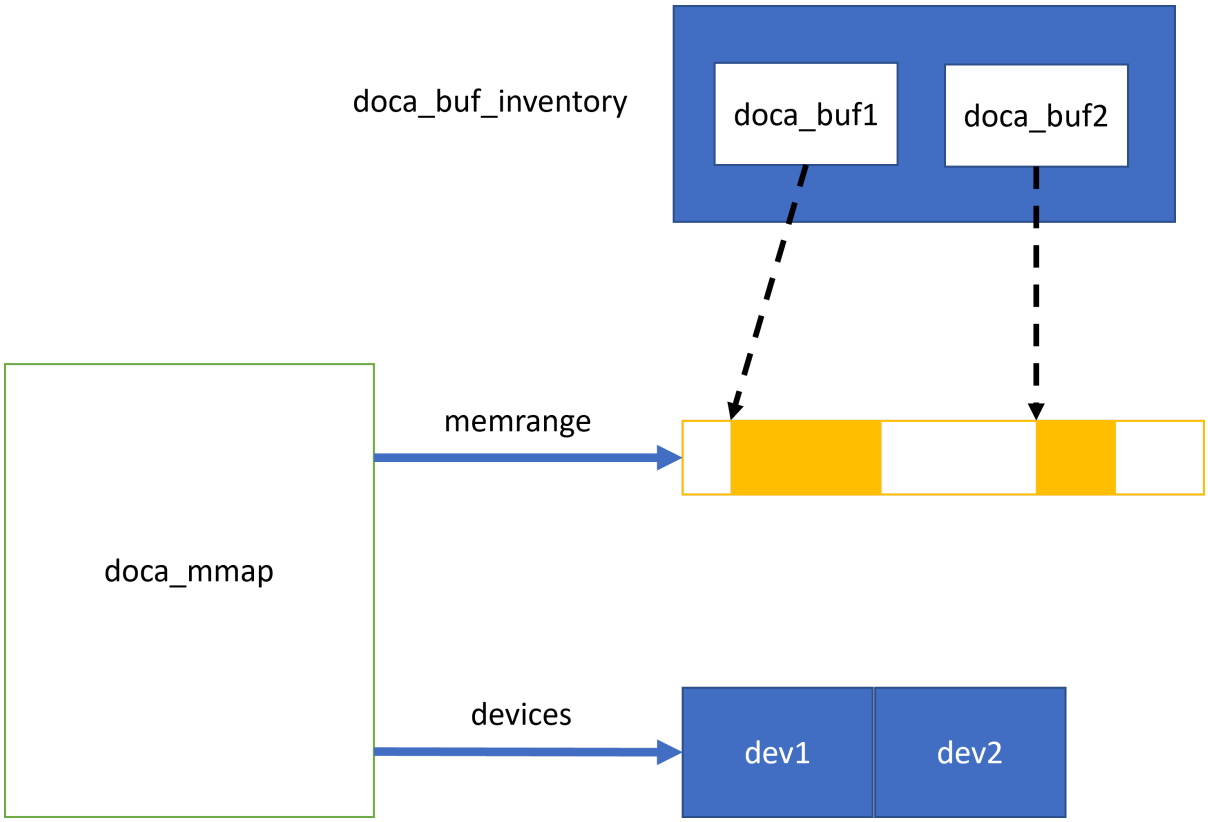
- Allows allocating data buffers that cover subranges within the registered memory.
- Allows memory pool semantics over registered memory.

DOCA memory has the following main components:

- `doca_buf` – Describes a data buffer, and is used as input/output to various HW processing tasks within DOCA libraries.
- `doca_mmap` – Describes registered memory, that 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 Two DOCA Devices, dev1 and dev2.

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

Execution Model

DOCA SDK introduces libraries that utilize HW 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 HW, and instead lets the application focus on what matters. This type of library is referred to as a context.

Since a context utilizes a HW processing unit, it needs a device in order to operate. This device will also determine which buffers are accessible by that context.

Contexts provide HW processing operation APIs in the form of tasks and events.

Task:

- Application prepares the task arguments.
- Application submits the task, this will issue a request to the relevant HW processing unit.
- Application receives a completion in the form of a callback **once** the HW processing is completed.

Event:

- Application registers to the event. This will inform HW to report whenever the event occurs.
- Application receives a completion in form of a callback **every time** HW identifies that the event occurred.

Since HW processing is asynchronous in nature. DOCA provides an object that allows waiting on processing operations (tasks & events). This object is referred to as Progress Engine or PE.

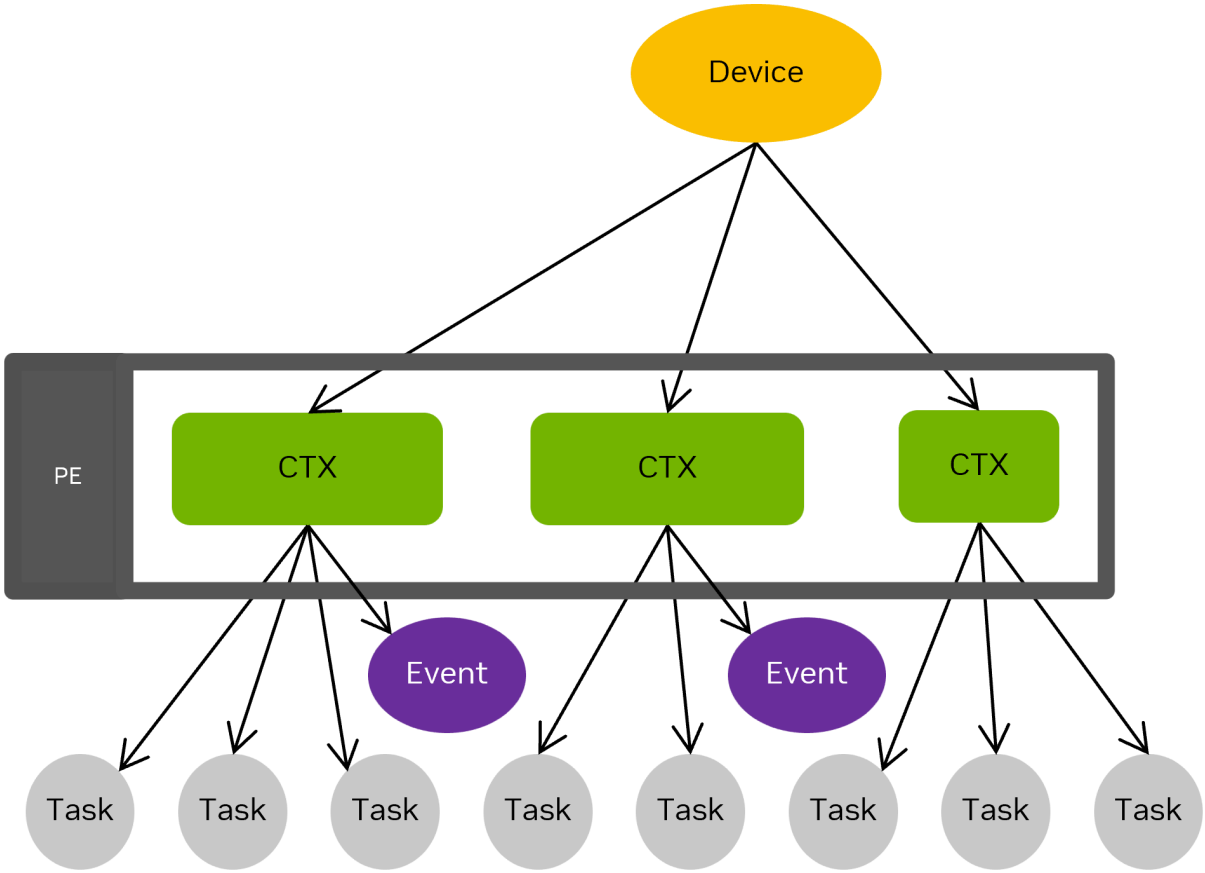
The PE allows waiting on completions using the following methods:

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

Once completion has occurred, whether caused by Task or Event, the relevant callback will be invoked as part of PE method.

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

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



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

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

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.

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

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 major release
4. Removed – an API marked as `DOCA_STABLE` or `DOCA_DEPRECATED` 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.

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.

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.

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).

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).

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.

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

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. T o 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.

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);
```

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);
```

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);
```

Debuggability

DOCA logging infrastructure allows enabling printing DOCA SDK library error messages, and printing debug and error messages from applications.

To work with the DOCA logging mechanism, the header `doca_log.h` must be included in every source code using it.

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 */
};
```

Note

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.

Logging Backends

DOCA 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
- buffer – 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.

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_buffer_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.

> **Note**
>
> Messages may change between different versions of DOCA. Users cannot rely on any message exitance or formatting.

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_buffer()`
- `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 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 any 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 any 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.

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.

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 Flow
- DOCA IPsec
- DPA Subsystem

- DOCA DMA
- DOCA Comm Channel
- 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

Info

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

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.

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).

1. MAC/VLAN/ETHERTYPE
2. IPv4/IPv6
3. TCP/UDP/ICMP
4. GRE/VXLAN/GTP-U
5. Metadata

The execution pipe can include packet modification actions such as the following:

- Modify MAC address
- Modify IP address
- Modify L4 (ports, TCP sequences and acknowledgments)
- Strip tunnel
- Add tunnel
- Set metadata

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

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
```

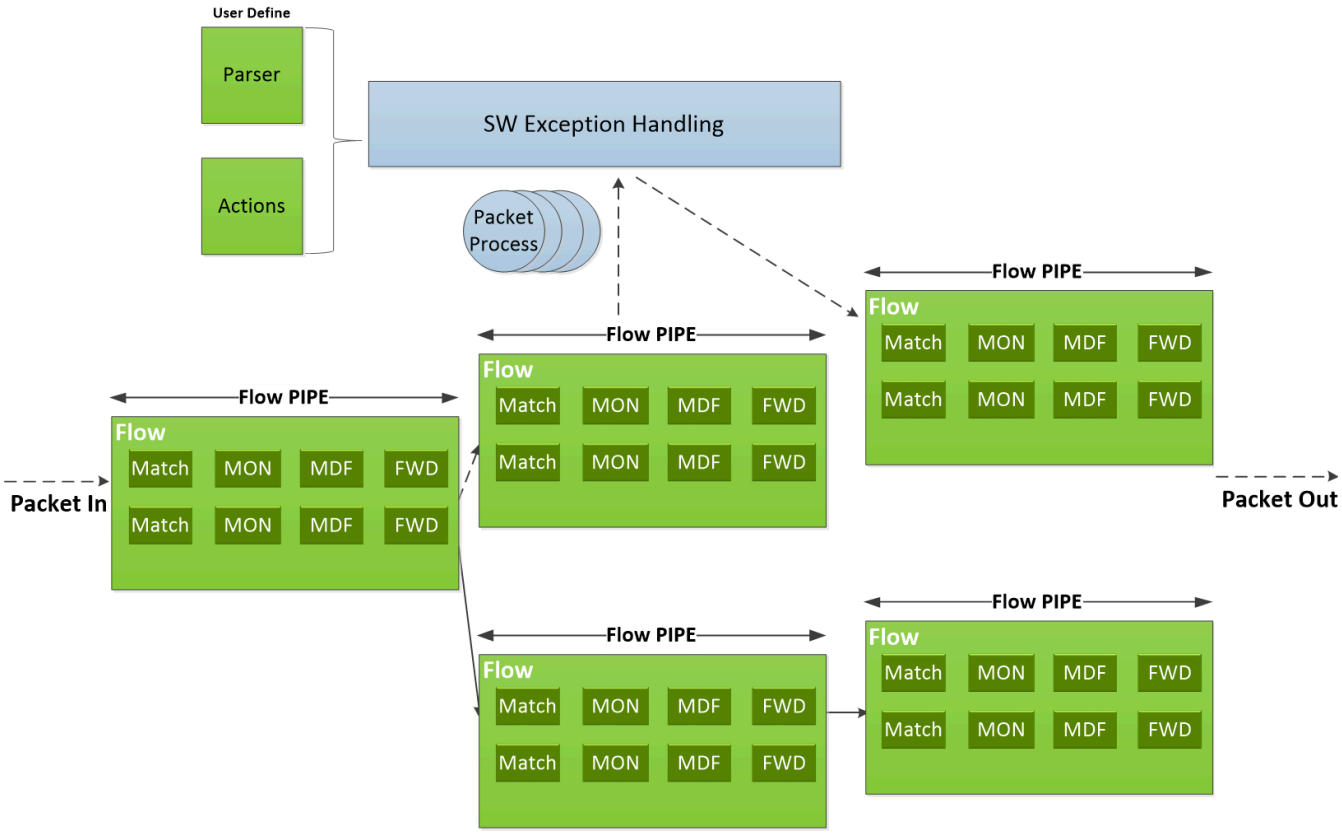
Note

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. In such cases, the guiding principal is to allocate 2GB of RAM, and instead of allocating 1024 pages, one should allocate the matching amount (4 pages):

```
sudo echo 4 > /proc/sys/vm/nr_hugepages
```

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

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

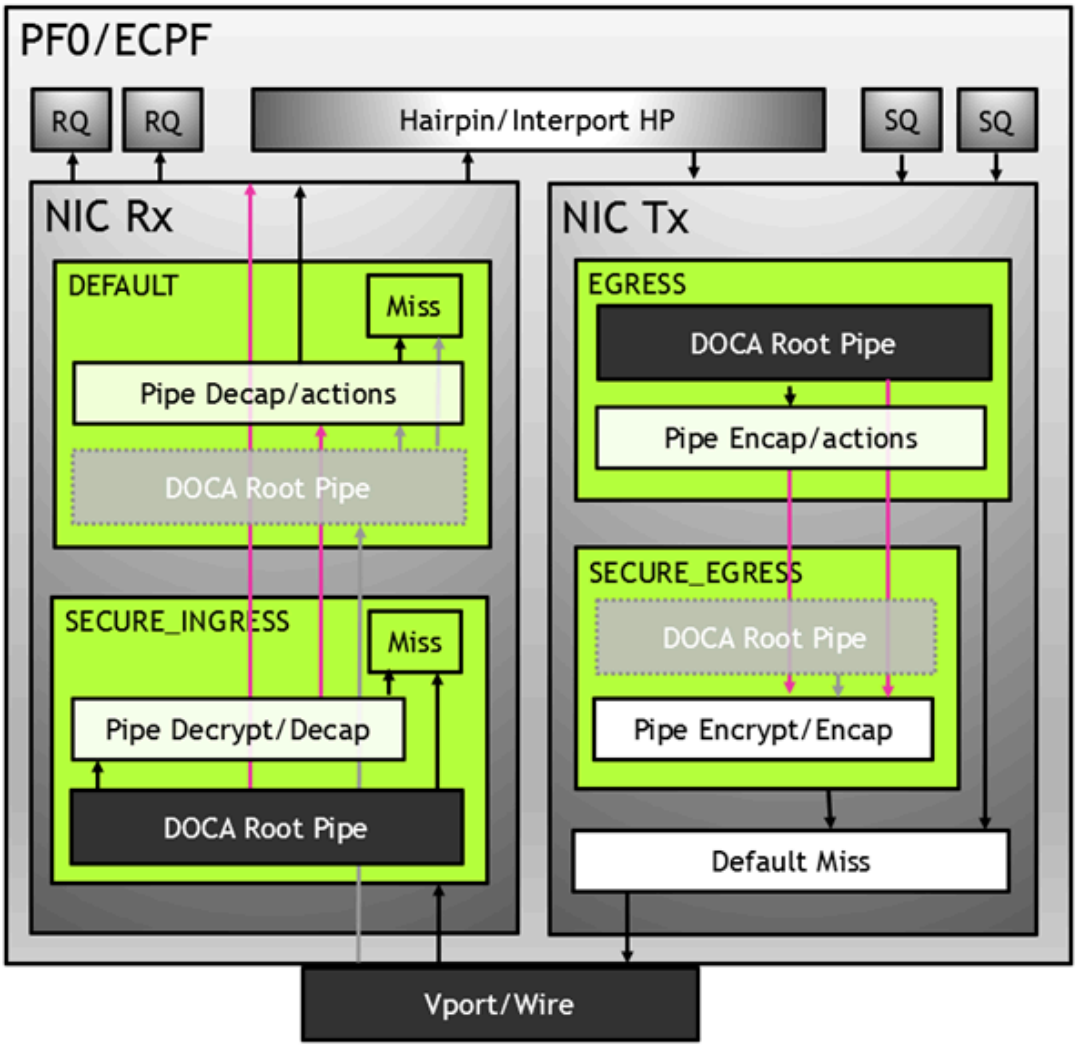
List of Steering Domains

DOCA Flow provides the following set of predefined steering domains:

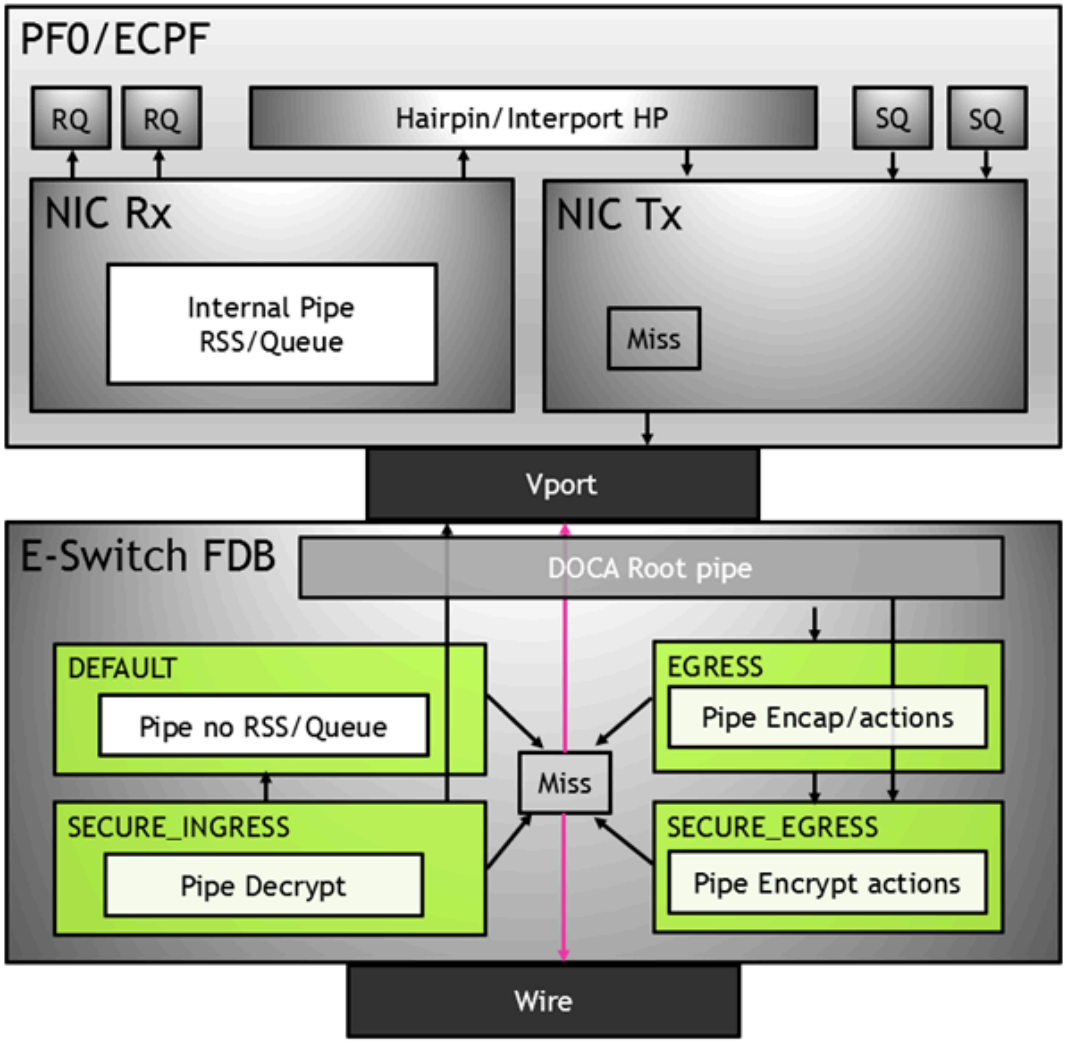
Domain	Description
`DOCA_FLOW_PIPE_DOMAIN_DEFAULT`	• Default domain for actions on ingress traffic • Encapsulated and secure actions are not allowed here • The next milestone is queue or pipe in the EGRESS domain • Miss action is: Drop
`DOCA_FLOW_PIPE_DOMAIN_SECURE_INGRESS`	• For secure actions on ingress 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

Domain	Description
`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
`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

Domains in VNF Mode



Domains in Switch Mode



API

> **Info**
>
> You can find more detailed information on DOCA Flow API in NVIDIA DOCA Libraries API Reference Manual.

Note

The pkg-config (`*.pc` file) for the DOCA Flow library is included in DOCA's regular definitions (i.e., `doca`).

The following sections provide additional details about the library API.

doca_flow_cfg

This structure is required input for the the DOCA Flow global initialization function - `doca_flow_init`.

```
struct doca_flow_cfg {
        uint64_t flags;
        uint16_t queues;
        struct doca_flow_resources resource;
        uint8_t nr_acl_collisions;
        const char *mode_args;
        uint32_t
nr_shared_resources[DOCA_FLOW_SHARED_RESOURCE_MAX];
        unit32_t queue_depth;
        doca_flow_entry_process_cb cb;
        doca_flow_shared_resource_unbind_cb unbind_cb;
};
```

- `flag` – configuration flags
- `queues` – the number of hardware acceleration control queues. It is expected that the same core always uses the same `queue_id`. In cases where multiple cores access the API using the same `queue_id`, it is up to the application to use locks between different cores/threads.

- `resource` – resource quota. This field includes the flow resource quota defined in the following structs:
  - `uint32_t nb_counters` – number of regular (non-shared) counters to configure
  - `uint32_b nb_meters` – number of regular (non-shared) traffic meters to configure
- `nr_acl_collisions` – number of collisions for the ACL module. Default value is 3. Maximum value is 8.
- `mode_args` – sets the DOCA Flow architecture mode.
- `nr_shared_resources` – total shared resource per type. See "Shared Counter Resource" section for more information.
  - Index `DOCA_FLOW_SHARED_RESOURCE_METER` – number of meters that can be shared among flows
  - Index `DOCA_FLOW_SHARED_RESOURCE_COUNT` – number of counters that can be shared among flows
  - Index `DOCA_FLOW_SHARED_RESOURCE_RSS` – number of RSS that can be shared among flows
  - Index `DOCA_FLOW_SHARED_RESOURCE_CRYPTO` – number of crypto actions that can be shared among flows
- `queue_depth` – n umber of flow rule operations a queue can hold. This value is preconfigured at port start (`queue_size`). Default value is 128 (configuring `0` sets default)
- `cb` – callback function for entry to be called during `doca_flow_entries_process` to complete entry operation (add, update, delete, and aged)
- `unbind_cb` – callback for unbinding of a shared resource

doca_flow_entry_process_cb

```
typedef void (*doca_flow_entry_process_cb)(struct
doca_flow_pipe_entry *entry,
                                           uint16_t pipe_queue,
enum doca_flow_entry_status status,
                                           enum
doca_flow_entry_op op, void *user_ctx);
```

- `entry [in]` – p ointer to pipe entry
- `pipe_queue [in]` – q ueue identifier
- `status [in]` – entry processing status (see `doca_flow_entry_status`)
- `op [in]` – entry's operation, defined in the following enum:
  - `DOCA_FLOW_ENTRY_OP_ADD` – Add entry operation
  - `DOCA_FLOW_ENTRY_OP_DEL` – Delete entry operation
  - `DOCA_FLOW_ENTRY_OP_UPD` – Update entry operation
  - `DOCA_FLOW_ENTRY_OP_AGED` – Aged entry operation
- `user_ctx [in]` – user context as provided to doca_flow_pipe_add_entry

> **Note**
>
> User context is set once to the value provided to `doca_flow_pipe_add_entry` (or to any `doca_flow_pipe_*_add_entry` variant) as the `usr_ctx` parameter, and is then reused in subsequent callback invocation for all operations. This user context must remain available for all potential invocations of the callback depending on it, as it is memorized as part of the entry and provided each time.

shared_resource_unbind_cb

```
typedef void (*shared_resource_unbind_cb)(enum
engine_shared_resource_type type,
                                          uint32_t
shared_resource_id,
                                          struct engine_bindable
*bindable);
```

- `type [in]` – engine shared resource type. Supported types are: `meter`, `counter`, `rss`, `crypto`, `mirror`.
- `shared_resource_id [in]` – shared resource; unique ID
- `bindable [in]` – pointer to bindable object (e.g., port, pipe)

doca_flow_port_cfg

This struct is required input for the DOCA Flow port initialization function - `doca_flow_port_start`.

```
struct doca_flow_port_cfg {
        uint16_t port_id;
        enum doca_flow_port_type type;
        const char *devargs;
        uint16_t priv_data_size;
        void *dev;
};
```

The struct `doca_flow_port_cfg` contains the following elements:

- `port_id` – DPDK port ID.
- `type` – depends on underlying API. This field includes the following port type:
  - `DOCA_FLOW_PORT_DPDK_BY_ID` – DPDK port by mapping ID
- `devargs` – a string containing the exact configuration needed according to the type

> **Info**
>
> For usage information of the type and devargs fields, refer to section "Start Point".

- `priv_data_size` – per port, users may define private data where application-specific info can be stored
- `dev` – port's `doca_dev`; used to create internal hairpin resource for switch mode if it is set

doca_flow_pipe_cfg

This is a pipe configuration that contains the user-defined template for the packet process.

```
struct doca_flow_pipe_attr {
    const char *name;
    enum doca_flow_pipe_type type;
        enum doca_flow_pipe_domain domain;
    bool is_root;
    uint32_t nb_flows;
    bool enable_strict_matching;
    uint8_t nb_actions;
    uint8_t nb_ordered_lists;
        enum doca_flow_direction_info dir_info;
    bool miss_counter;
};
```

- `name` – a string containing the name of the pipeline.
- `type` – type of pipe (`enum doca_flow_pipe_type`). This field includes the following pipe types:
  - `DOCA_FLOW_PIPE_BASIC` – flow pipe
  - `DOCA_FLOW_PIPE_CONTROL` – control pipe
  - `DOCA_FLOW_PIPE_LPM` – LPM pipe
  - `DOCA_FLOW_PIPE_ACL` – ACL pipe
  - `DOCA_FLOW_PIPE_ORDERED_LIST` – ordered list pipe
  - `DOCA_FLOW_PIPE_HASH` – hash pipe
- `domain` – pipe steering domain:
  - `DOCA_FLOW_PIPE_DOMAIN_DEFAULT` – default pipe domain for actions on ingress traffic

    - `DOCA_FLOW_PIPE_DOMAIN_SECURE_INGRESS` – pipe domain for secure actions on ingress traffic
    - `DOCA_FLOW_PIPE_DOMAIN_EGRESS` – pipe domain for actions on egress traffic
    - `DOCA_FLOW_PIPE_DOMAIN_SECURE_EGRESS` – pipe domain for actions on egress traffic
- `is_root` – determines whether or not the pipeline is root. If true, then the pipe is a root pipe executed on packet arrival

> **Note**
>
> Only one root pipe is allowed per port of any type.

- `enable_strict_matching` – if true, relaxed matching (enabled by default) is disabled for this pipe
- `nb_flows` – maximum number of flow rules. Default is 8k if not set
- `nb_actions` – maximum number of DOCA Flow action array. Default is 1 if not set

> **Note**
>
> nb_actions is mutually exclusive with `nb_ordered_lists`.

- `nb_ordered_lists` – Number of ordered lists in the array. Default is 0.

> **Note**
>
> nb_ordered_lists is mutually exclusive with `nb_actions`.

- `dir_info` – pipe direction information:
  - `DOCA_FLOW_DIRECTION_BIDIRECTIONAL` – default for traffic in both directions
  - `DOCA_FLOW_DIRECTION_NETWORK_TO_HOST` – network to host traffic
  - `DOCA_FLOW_DIRECTION_HOST_TO_NETWORK` – host to network traffic

> **Note**
>
> dir_info is supported in Switch Mode only.

> **Note**
>
> dir_info is optional. It can provide potential optimization at the driver layer. The driver may ignore it silently.

- `miss_counter` – determines whether or not to add a miss counter to the pipe. If true, then the pipe will have a miss counter and the user can query it using doca_flow_query_pipe_miss.

> **Note**
>
> Miss counter may affect performance and should be avoided if not required by the application.

```
struct doca_flow_ordered_list {
        uint32_t idx;
        uint32_t size;
        const void **elements;
        enum doca_flow_ordered_list_element_type *types;
};
```

- `idx` – list index among the lists of the pipe.
  - At pipe creation, it must match the list position in the array of lists
  - At entry insertion, it determines which list to use
- `size` – number of elements in the list.
- `elements` – an array of DOCA flow structure pointers, depending on the `types`.
- `types` – types of DOCA Flow structures each of the elements is pointing to. This field includes the following ordered list element types:
  - `DOCA_FLOW_ORDERED_LIST_ELEMENT_ACTIONS` – ordered list element is struct `doca_flow_actions`. The next element is struct `doca_flow_action_descs` which is associated with the current element
  - `DOCA_FLOW_ORDERED_LIST_ELEMENT_ACTION_DESCS` – ordered list element is struct `doca_flow_action_descs`. If the previous element type is ACTIONS, the current element is associated with it. Otherwise, the current element is ordered with regards to the previous one

- `DOCA_FLOW_ORDERED_LIST_ELEMENT_MONITOR` – ordered list element is struct `doca_flow_monitor`

```
struct doca_flow_pipe_cfg {
        struct doca_flow_pipe_attr attr;
        struct doca_flow_port *port;
        struct doca_flow_match *match;
        struct doca_flow_match *match_mask;
        struct doca_flow_actions **actions;
        struct doca_flow_actions **actions_masks;
        struct doca_flow_action_descs **action_descs;
        struct doca_flow_monitor *monitor;
        struct doca_flow_ordered_list **ordered_lists;
};
```

- `attr` – attributes for the pipeline
- `port` – port for the pipeline
- `match` – matcher for the pipeline except for `DOCA_FLOW_PIPE_HASH`
- `match_mask` – match mask for the pipeline. Only for `DOCA_FLOW_PIPE_BASIC`, `DOCA_FLOW_PIPE_CONTROL`, `DOCA_FLOW_PIPE_HASH`, and `DOCA_FLOW_PIPE_ORDERED_LIST`
- `actions` – action references array for the pipeline. Only for `DOCA_FLOW_PIPE_BASIC` and `DOCA_FLOW_PIPE_HASH`
- `actions_masks` – actions' masks array for the pipeline. Only for `DOCA_FLOW_PIPE_BASIC` and `DOCA_FLOW_PIPE_HASH`
- `action_descs` – action descriptors array. Only for `DOCA_FLOW_PIPE_BASIC`, `DOCA_FLOW_PIPE_CONTROL`, and `DOCA_FLOW_PIPE_HASH`
- `monitor` – monitor for the pipeline. Only for `DOCA_FLOW_PIPE_BASIC`, `DOCA_FLOW_PIPE_CONTROL`, and `DOCA_FLOW_PIPE_HASH`

- `ordered_list` – array of ordered list types. Only for `DOCA_FLOW_PIPE_ORDERED_LIST`

doca_flow_parser_geneve_opt_cfg

This is a parser configuration that contains the user-defined template for a single GENEVE TLV option.

```
struct doca_flow_parser_geneve_opt_cfg {
        enum doca_flow_parser_geneve_opt_mode
match_on_class_mode;
        doca_be16_t option_class;
        uint8_t option_type;
        uint8_t option_len;
        doca_be32_t
data_mask[DOCA_FLOW_GENEVE_DATA_OPTION_LEN_MAX];
};
```

- `match_on_class_mode` – role of `option_class` in this option (`enum doca_flow_parser_geneve_opt_mode`). This field includes the following class modes:
  - `DOCA_FLOW_PARSER_GENEVE_OPT_MODE_IGNORE` – class is ignored, its value is neither part of the option identifier nor changeable per pipe/entry
  - `DOCA_FLOW_PARSER_GENEVE_OPT_MODE_FIXED` – class is fixed (the class defines the option along with the type)
  - `DOCA_FLOW_PARSER_GENEVE_OPT_MODE_MATCHABLE` – class is the field of this option; different values can be matched for the same option (defined by type only)
- `option_class` – option class ID (must be set when class mode is fixed)
- `option_type` – option type

- `option_len` – length of the option data (in 4-byte granularity)
- `data_mask` – mask for indicating which dwords (DWs) should be configured on this option

> **Note**
>
> This is not a bit mask. Each DW can contain either `0xffffffff` for configure or `0x0` for ignore. Other values are not valid.

doca_flow_meta

There is a maximum `DOCA_FLOW_META_MAX`-byte scratch area which exists throughout the pipeline.

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.

> **Note**
>
> In a real application, it is encouraged to create a union of `doca_flow_meta` defining the application's scratch fields to use as metadata.

```
struct doca_flow_meta {
        union {
                uint32_t pkt_meta; /**< Shared with
application via packet. */
                struct {
                        uint32_t lag_port :2; /**< Bits of
LAG member port. */
                        uint32_t type :2; /**< 0: traffic 1:
SYN 2: RST 3: FIN. */
                        uint32_t zone :28; /**< Zone ID for
CT processing. */
                } ct;
        };
        uint32_t u32[DOCA_FLOW_META_MAX / 4 - 1]; /**<
Programmable user data. */
        uint32_t mark; /**< Mark id. */
};
```

- `pkt_meta` – Metadata can be received along with the packet
- `u32[]` – Scratch are `u32[]`
  - `u32[0]` contains the IPsec syndrome in the lower 8 bits if the packet passes the pipe with IPsec crypto action configured in full offload mode:
    - `0` – signifies a successful IPsec operation on the packet
    - `1` – bad replay. Ingress packet sequence number is beyond anti-reply window boundaries.
  - `u32[1]` contains the IPsec packet sequence number (lower 32 bits) if the packet passes the pipe with IPsec crypto action configured in full offload mode
- `mark` – O ptional parameter that may be communicated to the software. If it is set and the packet arrives to the software, the value can be examined using the software API.

- When DPDK is used, MARK is placed on the struct `rte_mbuf` (see "Action: MARK" section in official DPDK documentation)
- When the Kernel is used, MARK is placed on the struct `sk_buff` 's MARK field

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
orderd_list	[0, 1, 2, 3]
LPM	[0, 1, 2, 3]
mirror	[0, 1, 2, 3]
ACL	[0, 1, 2, 3]
Fwd from ingress to egress	[0, 1, 2, 3]

doca_flow_parser_meta

This structure contains all metadata information which hardware extracts from the packet.

These fields contain read-only hardware data which can be used to match on.

```
struct doca_flow_parser_meta {
        uint32_t port_meta;
        uint16_t random;
        uint8_t ipsec_syndrome;
        enum doca_flow_meter_color meter_color;
        enum doca_flow_l2_meta outer_l2_type;
        enum doca_flow_l3_meta outer_l3_type;
        enum doca_flow_l4_meta outer_l4_type;
        enum doca_flow_l2_meta inner_l2_type;
        enum doca_flow_l3_meta inner_l3_type;
        enum doca_flow_l4_meta inner_l4_type;
        uint8_t outer_ip_fragmented;
        uint8_t inner_ip_fragmented;
        uint8_t outer_l3_ok;
        uint8_t outer_ip4_checksum_ok;
        uint8_t outer_l4_ok;
        uint8_t inner_l3_ok;
        uint8_t inner_ip4_checksum_ok;
        uint8_t inner_l4_ok;
};
```

- `port_meta` – Programmable source vport.
- `random` – Random value to match regardless to packet data/headers content. Application should not assume that this value is kept during the lifetime of the packet. It holds a different random value for each matching.

> **Note**
>
> When random matching is used for sampling, the number of entries in the pipe must be 1 (`doca_flow_pipe_attr.nb_flows` = 1).

- `ipsec_syndrome` – IPsec decrypt/authentication syndrome
- `meter_color` – Meter colors (`enum doca_flow_meter_color`). Valid colors:
  - `DOCA_FLOW_METER_COLOR_GREEN` – Meter marking packet color as green
  - `DOCA_FLOW_METER_COLOR_YELLOW` – Meter marking packet color as yellow
  - `DOCA_FLOW_METER_COLOR_RED` – Meter marking packet color as red.
- `outer_l2_type` – Outer L2 packet type (`enum doca_flow_l2_meta`). Valid L2 types:
  - `DOCA_FLOW_L2_META_NO_VLAN` – No VLAN present
  - `DOCA_FLOW_L2_META_SINGLE_VLAN` – Single VLAN present
  - `DOCA_FLOW_L2_META_MULTI_VLAN` – Multiple VLAN present
- `outer_l3_type` – Outer L3 packet type (`enum doca_flow_l3_meta`). Valid L3 types:
  - `DOCA_FLOW_L3_META_NONE` – L3 type is none of the below
  - `DOCA_FLOW_L3_META_IPV4` – L3 type is IPv4
  - `DOCA_FLOW_L3_META_IPV6` – L3 type is IPv6
- `outer_l4_type` – Outer L4 packet type (`enum doca_flow_l4_meta`). Valid L4 types:
  - `DOCA_FLOW_L4_META_NONE` – L4 type is none of the below
  - `DOCA_FLOW_L4_META_TCP` – L4 type is TCP
  - `DOCA_FLOW_L4_META_UDP` – L4 type is UDP
  - `DOCA_FLOW_L4_META_ICMP` – L4 type is ICMP
  - `DOCA_FLOW_L4_META_ESP` – L4 type is ESP

- `inner_l2_type` – Inner L2 packet type (`enum doca_flow_l2_meta`). Valid L2 types:
  - `DOCA_FLOW_L2_META_NO_VLAN` – No VLAN present
  - `DOCA_FLOW_L2_META_SINGLE_VLAN` – Single VLAN present
  - `DOCA_FLOW_L2_META_MULTI_VLAN` – Multiple VLAN present
- `inner_l3_type` – Inner L3 packet type (`enum doca_flow_l3_meta`). Valid L3 types:
  - `DOCA_FLOW_L3_META_NONE` – L3 type is none of the below
  - `DOCA_FLOW_L3_META_IPV4` – L3 type is IPv4
  - `DOCA_FLOW_L3_META_IPV6` – L3 type is IPv6
- `inner_l4_type` – Inner L4 packet type (`enum doca_flow_l4_meta`). Valid L4 types:
  - `DOCA_FLOW_L4_META_NONE` – L4 type is none of the below
  - `DOCA_FLOW_L4_META_TCP` – L4 type is TCP
  - `DOCA_FLOW_L4_META_UDP` – L4 type is UDP
  - `DOCA_FLOW_L4_META_ICMP` – L4 type is ICMP
  - `DOCA_FLOW_L4_META_ESP` – L4 type is ESP
- `outer_ip_fragmented` – Whether outer IP packet is fragmented
- `inner_ip_fragmented` – Whether inner IP packet is fragmented
- `outer_l3_ok` – Whether outer network layer is valid regardless to IPv4 checksum
- `outer_ip4_checksum_ok` – Whether outer IPv4 checksum is valid, packets without outer IPv4 header are taken as invalid checksum

- `outer_l4_ok` – Whether outer transport layer is valid including L4 checksum
- `inner_l3_ok` – Whether inner network layer is valid regardless to IPv4 checksum
- `inner_ip4_checksum_ok` – Whether inner IPv4 checksum is valid, packets without inner IPv4 header are taken as invalid checksum
- `inner_l4_ok` – Whether inner transport layer is valid including L4 checksum

> **Note**
>
> Matching on either `outer_l4_ok=1` or `inner_l4_ok=1` means that all L4 checks (length, checksum, etc.) are ok.
>
> Matching on either `outer_l4_ok=0` or `inner_l4_ok=0` means that all L4 checks are **not** ok.
>
> It is not possible to match using these fields for cases where a part of the checks is okay and a part is not ok.

doca_flow_header_format

This structure defines each layer of the packet header format.

```
struct doca_flow_header_format {
        struct doca_flow_header_eth eth;
        uint16_t l2_valid_headers;
        struct doca_flow_header_eth_vlan
eth_vlan[DOCA_FLOW_VLAN_MAX];
        enum doca_flow_l3_type l3_type;
        union {
                struct doca_flow_header_ip4 ip4;
                struct doca_flow_header_ip6 ip6;
        };
        enum doca_flow_l4_type_ext l4_type_ext;
        union {
                struct doca_flow_header_icmp icmp;
                struct doca_flow_header_udp udp;
                struct doca_flow_header_tcp tcp;
                struct doca_flow_header_l4_port transport;
        };
};
```

- `eth` – Ethernet header format, including source and destination MAC address and the Ethernet layer type. If a VLAN header is present then `eth.type` represents the type following the last VLAN tag.
- `l2_valid_headers` – bitwise OR one of the following options: `DOCA_FLOW_L2_VALID_HEADER_VLAN_0`, `DOCA_FLOW_L2_VALID_HEADER_VLAN_1`
- `eth_vlan` – VLAN tag control information for each VLAN header.
- `l3_type` – Layer 3 type, indicates the next layer is IPv4 or IPv6.
- `ip4` – IPv4 header format including source and destination IP address, type of service (dscp and ecn), and the next protocol.

- `ip6` – IPv6 header format including source and destination IP address, traffic class (dscp and ecn), and the next protocol.
- `l4_type_ext` – The next layer type after the layer 3.
- `icmp` – ICMP header format.
- `udp` – UDP header format.
- `tcp` – TCP header format.
- `transport` – header format for source and destination ports; used for defining match or actions with relaxed matching while not caring about the L4 protocol (whether TCP or UDP). This is used only if the `l4_type_ext` is `DOCA_FLOW_L4_TYPE_EXT_TRANSPORT`.

doca_flow_tun

This structure defines tunnel headers.

```
struct doca_flow_tun {
        enum doca_flow_tun_type type;
        union {
                struct {
                        doca_be32_t vxlan_tun_id;
                };
                struct {
                        bool key_present;
                        doca_be16_t protocol;
                        doca_be32_t gre_key;
                };
                struct {
                        doca_be32_t gtp_teid;
                };
                struct {
                        doca_be32_t esp_spi;
                        doca_be32_t esp_sn;
                };
                struct {
                        struct doca_flow_header_mpls
mpls[DOCA_FLOW_MPLS_LABELS_MAX];
                };
                struct {
                        struct doca_flow_header_geneve geneve;
                        union doca_flow_geneve_option
geneve_options[DOCA_FLOW_GENEVE_OPT_LEN_MAX];
                };
        };
};
```

- `type` – type of tunnel (`enum doca_flow_tun_type`). Valid tunnel types:
  - `DOCA_FLOW_TUN_VXLAN` – VXLAN tunnel

    - `DOCA_FLOW_TUN_GRE` – GRE tunnel with option KEY (optional)
    - `DOCA_FLOW_TUN_GTP` – GTP tunnel
    - `DOCA_FLOW_TUN_ESP` – ESP tunnel
    - `DOCA_FLOW_TUN_MPLS_O_UDP` – MPLS tunnel (supports up to 5 headers)
    - `DOCA_FLOW_TUN_GENEVE` – GENEVE header format including option length, VNI, next protocol, and options.
- `vxlan_tun_vni` – VNI (24) + reserved (8)
- `key_present` – GRE option KEY is present
- `protocol` – GRE next protocol
- `gre_key` – GRE key option, match on this field only when `key_present` is true
- `gtp_teid` – GTP TEID
- `esp_spi` – IPsec session parameter index
- `esp_sn` – IPsec sequence number
- `mpls` – List of MPLS header format
- `geneve` – GENEVE header format
- `geneve_options` – List DWs describing GENEVE TLV options

The following table details which tunnel types support which operation on the tunnel header:

Tunnel Type	Match[a]	L2 encap[b]	L2 decap[b]	L3 encap[b]	L3 decap[b]	Modif
`DOCA_FLOW_TUN_VXLAN`	✔	✔	✔	☐	☐	✔
`DOCA_FLOW_TUN_GRE`	✔	✔	✔	✔	✔	☐

Tunnel Type	Match[a]	L2 encap[b]	L2 decap[b]	L3 encap[b]	L3 decap[b]	Modif
`DOCA_FLOW_TUN_GTP`	✔	☐	☐	✔	✔	✔
`DOCA_FLOW_TUN_ESP`	✔	☐	☐	☐	☐	☐
`DOCA_FLOW_TUN_MPLS_O_UDP`	✔	☐	☐	✔	✔	☐
`DOCA_FLOW_TUN_GENEVE`	✔	✔	✔	✔	✔	✔

Note

[a] Support for matching on this tunnel header, configured in the `tun` field in `struct doca_flow_match`.

[b] Decapsulation/encapsulation of the tunnel header is to be enabled in `struct doca_flow_actions` using the Boolean fields `decap` and `has_encap`. An action descriptor with the type `DOCA_FLOW_ACTION_DECAP_ENCAP` must be provided if decap/encap is enabled. The action descriptor's Boolean field `is_l2` determines the decapsulation/encapsulation tunnel type. It is the user's responsibility to determine whether a tunnel is type L2 or L3. If the user sets settings unaligned with the packets coming, anomalous behavior may occur.

[c] Support for modifying this tunnel header, configured in the `tun` field in `struct doca_flow_actions`.

[d] Support for copying fields to/from this tunnel header, configured in `struct doca_flow_action_descs`.

DOCA Flow Tunnel GENEVE

The `DOCA_FLOW_TUN_GENEVE` type includes the basic header for GENEVE and an array for GENEVE TLV options.

Info

The GENEVE TLV options must be configured before in parser creation (doca_flow_parser_geneve_opt_create).

doca_flow_header_geneve

This structure defines GENEVE protocol header.

```
struct doca_flow_header_geneve {
        uint8_t ver_opt_len;
        uint8_t o_c;
        doca_be16_t next_proto;
        doca_be32_t vni;
};
```

- `ver_opt_len` – version (2) + options length (6). The length is expressed in 4-byte multiples, excluding the GENEVE header.
- `o_c` – OAM packet (1) + critical options present (1) + reserved (6).
- `next_proto` – GENEVE next protocol. When GENEVE has options, it describes the protocol after the options.
- `gre_key` – GENEVE VNI (24) + reserved (8).

doca_flow_geneve_option

This object describes a single DW (4-bytes) from the GENEVE option header. It describes either the first DW in the option including class, type, and length, or any other data DW.

```
union doca_flow_geneve_option {
    struct {
        doca_be16_t class_id;
        uint8_t type;
        uint8_t length;
    };
    doca_be32_t data;
};
```

- `class_id` – Option class ID
- `type` – Option type
- `length` – Reserved (3) + option data length (5). The length is expressed in 4-byte multiples, excluding the option header.
- `data` – 4 bytes of option data

GENEVE Matching Notes

- Option type and length must be provided for each option at pipe creation time in a match structure
- When class mode is `DOCA_FLOW_PARSER_GENEVE_OPT_MODE_FIXED`, option class must also be provided for each option at pipe creation time in a match structure
- Option length field cannot be matched
- Type field is the option identifier, it must be provided as a specific value upon pipe creation
- Option data is taken as changeable when all data is filled with `0xffffffff` including DWs which were not configured at parser creation time
- In the `match_mask` structure, the DWs which have not been configured at parser creation time must be zero

GENEVE Encapsulation Notes

- The encapsulation size is constant per `doca_flow_actions` structure. The size is determined by the `tun.geneve.ver_opt_len` field at pipe creation time. The default is 0 (no options).
- The total encapsulation size is limited by 128 bytes. The `tun.geneve.ver_opt_len` field should follow this limitation according to the requested outer header sizes:

Header Included in Outer	Maximal `tun.geneve.ver_opt_len` Value
ETH + VLAN + IPV4 + UDP + GENEVE	18
ETH + VLAN + IPV6 + UDP + GENEVE	13
ETH + IPV4 + UDP + GENEVE	19
ETH + IPV6 + UDP + GENEVE	14

- Options in encapsulation data do not have to be configured at parser creation time

GENEVE Decapsulation Notes

The options to decapsulate do not have to be configured at parser creation time.

doca_flow_match

This structure is a match configuration that contains the user-defined fields that should be matched on the pipe.

```
struct doca_flow_match {
        uint32_t flags;
        struct doca_flow_meta meta;
        struct doca_flow_parser_meta parser_meta;
        struct doca_flow_header_format outer;
        struct doca_flow_tun tun;
        struct doca_flow_header_format inner;
};
```

- `flags` – Match items which are no value needed.
- `meta` – Programmable metadata.
- `parser_meta` – Read-only metadata.
- `outer` – Outer packet header format.
- `tun` – Tunnel info
- `inner` – Inner packet header format.

doca_flow_actions

This structure is a flow actions configuration.

```
struct doca_flow_actions {
    uint8_t action_idx;
    uint32_t flags;
    bool decap;
        bool pop;
    struct doca_flow_meta meta;
    struct doca_flow_header_format outer;
        struct doca_flow_tun tun;
    bool has_encap;
    struct doca_flow_encap_action encap;
        bool has_push;
        struct doca_flow_push_action push;
        bool has_crypto_encap;
        struct doca_flow_crypto_encap_action crypto_encap;
        struct doca_flow_crypto_action crypto;
};
```

- `action_idx` – Index according to place provided on creation.
- `flags` – Action flags.
- `decap` – Decap while it is set to true. If set to true, struct `doca_flow_action_descs` must be provided and an action descriptor of type `DOCA_FLOW_ACTION_DECAP_ENCAP` must be a part of it.
- `pop` – Pop header while it is set to true.
- `meta` – Modify meta value.
- `outer` – Modify outer header.
- `tun` – Modify tunnel header.
- `has_encap` – Encap while it is set to true. If set to true, struct `doca_flow_action_descs` must be provided and an action descriptor of type

`DOCA_FLOW_ACTION_DECAP_ENCAP` must be a part of it.

- `encap` – Encap data information.
- `has_push` – Push header while it is set to true.
- `push` – Push header data information.
- `has_crypto_encap` – Perform packet reformat for crypto protocols while it is set to true. If set to true, the structure `doca_flow_crypto_encap_action` provides a description for the header and trailer to be inserted or removed.
- `crypto_encap` – Crypto protocols header and trailer data information.
- `crypto` – Contains crypto action information.

doca_flow_encap_action

This structure is an encapsulation action configuration.

```
struct doca_flow_encap_action {
    struct doca_flow_header_format outer;
    struct doca_flow_tun tun;
};
```

- `outer` – L2/3/4 layers of the outer tunnel header.
  - L2 - src/dst MAC addresses, ether type, VLAN
  - L3 - IPv4/6 src/dst IP addresses, TTL/hop_limit, dscp_ecn
  - L4 - the UDP dst port is determined by the tunnel type
- `tun` – The specific fields of the used tunnel protocol. Supported tunnel types: GRE, GTP, VXLAN, MPLS, GENEVE.

doca_flow_push_action

This structure is a push action configuration.

```
struct doca_flow_push_action {
        enum doca_flow_push_action_type type;
        union {
                struct doca_flow_header_eth_vlan vlan;
    };
};
```

- `type` – Push action type.
- `vlan` – VLAN data.

The `type` field includes the following push action types:

- `DOCA_FLOW_PUSH_ACTION_VLAN` – push VLAN.

doca_flow_crypto_encap_action

This structure is a crypto protocol packet reformat action configuration.

```
struct doca_flow_crypto_encap_action {
        enum doca_flow_crypto_encap_action_type action_type;
        enum doca_flow_crypto_encap_net_type net_type;
        uint16_t icv_size;
        uint16_t data_size;
        uint8_t encap_data[DOCA_FLOW_CRYPTO_HEADER_LEN_MAX];
};
```

- `action_type` – Reformat action type.

- `net_type` – Protocol mode, network header type.
- `icv_size` – Integrity check value size, in bytes; defines the trailer size.
- `data_size` – Header size in bytes to be inserted from `encap_data`.
- `encap_data` – Header data to be inserted.

The `action_type` field includes the following crypto encap action types:

- `DOCA_FLOW_CRYPTO_REFORMAT_NONE` – No crypto encap action performed.
- `DOCA_FLOW_CRYPTO_REFORMAT_ENCAP` – Add/insert the crypto header and trailer to the packet. `data_size` and `encap_data` should provide the data for the headers being inserted.
- `DOCA_FLOW_CRYPTO_REFORMAT_DECAP` – Remove the crypto header and trailer from the packet. `data_size` and `encap_data` should provide the data for the headers being inserted for tunnel mode.

The `net_type` field includes the following protocol/network header types:

- `DOCA_FLOW_CRYPTO_HEADER_NONE` – No header type specified.
- `DOCA_FLOW_CRYPTO_HEADER_ESP_TUNNEL` – ESP tunnel mode header type. On encap, the full tunnel header data for new L2+L3+ESP should be provided. On decap, the new L2 header data should be provided.
- `DOCA_FLOW_CRYPTO_HEADER_ESP_OVER_IP` – ESP transport over IP mode header type. On encap, the data for ESP header being inserted should be provided.
- `DOCA_FLOW_CRYPTO_HEADER_UDP_ESP_OVER_IP` – UDP+ESP transport over IP mode header type. On encap, the data for UDP+ ESP headers being inserted should be provided.
- `DOCA_FLOW_CRYPTO_HEADER_ESP_OVER_LAN` – ESP transport over UDP/TCP mode header type. On encap, the data for ESP header being inserted should be provided.

The `icv_size` field can be either 8, 12 or 16 for ESP.

The `data_size` field should not exceed `DOCA_FLOW_CRYPTO_HEADER_LEN_MAX`; can be zero for decap in non-tunnel modes.

doca_flow_crypto_action

This structure is a crypto action configuration to perform packet data encryption and decryption.

```
struct doca_flow_crypto_action {
        enum doca_flow_crypto_action_type action_type;
        enum doca_flow_crypto_protocol_type proto_type;
        union {
                struct {
                        bool sn_en;
                } esp;
        };
        uint32_t crypto_id;
};
```

- `action_type` – Crypto action type.
- `proto_type` – Protocol type.
- `esp.sn_en` – Enable ESP sequence number generation/checking.
- `crypto_id` – Shared crypto resource ID.

The `action_type` field includes the following crypto action types:

- `DOCA_FLOW_CRYPTO_ACTION_NONE` – No crypto action specified.
- `DOCA_FLOW_CRYPTO_ACTION_ENCRYPT` – Encrypt packet data according to the chosen protocol.

- `DOCA_FLOW_CRYPTO_ACTION_DECRYPT` – Decrypt packet data according to the chosen protocol.

The `proto_type` field includes the following protocols supported:

- `DOCA_FLOW_CRYPTO_PROTOCOL_NONE` – No crypto protocol specified.
- `DOCA_FLOW_CRYPTO_PROTOCOL_ESP` – IPsec ESP protocol.

doca_flow_action_descs

This structure describes operations executed on packets matched by the pipe.

> **Info**
>
> Detailed compatibility matrix and usage can be found in section "Summary of Action Types".

```
struct doca_flow_action_descs {
        uint8_t nb_action_desc;
        struct doca_flow_action_desc *desc_array;
};
```

- `nb_action_desc` – Maximum number of action descriptor array (i.e., number of descriptor array elements)
- `desc_array` – Action descriptor array pointer

struct doca_flow_action_desc

```
struct doca_flow_action_desc {
        enum doca_flow_action_type type;
        union {
                struct {
                        struct doca_flow_action_desc_field src;
                        struct doca_flow_action_desc_field dst;
                        unit32_t width;
                } field_op;
                struct {
                        bool is_l2;
                } decap_encap;
        };
};
```

- `type` – Action type.
- `field_op` – Field to copy/add source and destination descriptor. Add always applies from field bit 0 .
- `decap_encap` – Descriptor of decap or encap action.

The `type` field includes the following forwarding modification types:

- `DOCA_FLOW_ACTION_AUTO` – modification type derived from pipe action
- `DOCA_FLOW_ACTION_ADD` – add from field value or packet field. Supports meta scratch, `ipv4_ttl`, `ipv6_hop`, `tcp_seq`, and `tcp_ack`.
- `DOCA_FLOW_ACTION_COPY` – copy field
- `DOCA_FLOW_ACTION_DECAP_ENCAP` – configures decap/encap for L2 or L3 level. Descriptor of this type is mandatory when `decap` or `has_encap` is `true`.

doca_flow_action_desc_field

This struct is the action descriptor's field configuration.

```
struct doca_flow_action_desc_field {
        const char *field_string;
        /**< Field selection by string. */
        uint32_t bit_offset;
        /**< Field bit offset. */
};
```

- `field_string` – Field string. Describes which packet field is selected in string format.
- `bit_offset` – Bit offset in the field.

> **Note**
>
> The complete supported field string could be found in the "Field String Support" appendix.

doca_flow_monitor

This structure is a monitor configuration.

```
struct doca_flow_monitor {
        enum doca_flow_resource_type meter_type;
        /**< Type of meter configuration. */
        union {
                struct {
                        enum doca_flow_meter_limit_type
limit_type;
                        /**< Meter rate limit type: bytes /
packets per second */
                        uint64_t cir;
                        /**< Committed Information Rate
(bytes/second). */
                        uint64_t cbs;
                        /**< Committed Burst Size (bytes). */
                } non_shared_meter;
                struct {
                        uint32_t shared_meter_id;
                        /**< shared meter id */
                        enum doca_flow_meter_color
meter_init_color;
                        /**< meter initial color */
                } shared_meter;
        };

        enum doca_flow_resource_type counter_type;
        /**< Type of counter configuration. */
        union {
                struct {
                        uint32_t shared_counter_id;
                        /**< shared counter id */
                } shared_counter;
        };

        uint32_t shared_mirror_id;
        /**< shared mirror id. */
```

```
        bool aging_enabled;
        /**< Specify if aging is enabled */
        uint32_t aging_sec;
        /**< aging time in seconds.*/
};
```

- `meter_type` – Defines the type of meter. Meters can be shared, non-shared, or not used at all.
- `non_shared_meter` – non-shared meter params
  - `limit_type` – bytes versus packets measurement
  - `cir` – committed information rate of non-shared meter
  - `cbs` – committed burst size of non-shared meter
- `shared_meter` – shared meter params
  - `shared_meter_id` – meter ID that can be shared among multiple pipes
  - `meter_init_colr` – the initial color assigned to a packet entering the meter
- `counter_type` – defines the type of counter. Counters can be shared, or not used at all.
  - `shared_counter_id` – counter ID that can be shared among multiple pipes
- `shared_mirror_id` – mirror ID that can be shared among multiple pipes
- `aging_enabled` – set to true to enable aging
- `aging_sec` – number of seconds from the last hit after which an entry is aged out

`doca_flow_resource_type` is defined as follows.:

```
enum doca_flow_resource_type {
        DOCA_FLOW_RESOURCE_TYPE_NONE,
        DOCA_FLOW_RESOURCE_TYPE_SHARED,
        DOCA_FLOW_RESOURCE_TYPE_NON_SHARED
};
```

T(c) is the number of available tokens. For each packet where b equals the number of bytes, if t(c)-b⩾0 the packet can continue, and tokens are consumed so that t(c)=t(c)-b. If t(c)-b<0, the packet is dropped.

CIR is the maximum bandwidth at which packets continue being confirmed. Packets surpassing this bandwidth are dropped. CBS is the maximum number of bytes allowed to exceed the CIR to be still CIR confirmed. Confirmed packets are handled based on the `fwd` parameter.

The number of `<cir,cbs>` pair different combinations is limited to 128.

Metering packets can be individual (i.e., per entry) or shared among multiple entries:

- For the individual use case, set `meter_type` to `DOCA_FLOW_RESOURCE_TYPE_NON_SHARED`
- For the shared use case, set `meter_type` to `DOCA_FLOW_RESOURCE_TYPE_SHARED` and `shared_meter_id` to the meter identifier

Counting packets can be individual (i.e., per entry) or shared among multiple entries:

- For the individual use case, set counter type to `DOCA_FLOW_RESOURCE_TYPE_SHARED` and `shared_counter_id` to the counter identifier
- For the shared use case, use a non-zero `shared_counter_id`

Mirroring packets can only be used as shared with a non-zero `shared_mirror_id`.

doca_flow_fwd

This structure is a forward configuration which directs where the packet goes next.

```
struct doca_flow_fwd {
        enum doca_flow_fwd_type type;
        union {
                struct {
                        uint32_t rss_outer_flags;
                        uint32_t rss_inner_flags;
                        unit32_t *rss_queues;
                        int num_of_queues;
                };
                struct {
                        unit16_t port_id;
                };
                struct {
                           struct doca_flow_pipe *next_pipe;
        };
        struct {
                     struct doca_flow_pipe *pipe;
                 uint32_t idx;
            } ordered_list_pipe;
            struct {
                         struct doca_flow_target *target;
            };
        };
};
```

- `type` – indicates the forwarding type
- `rss_outer_flags` – RSS offload types on the outer-most layer (tunnel or non-tunnel).
- `rss_inner_flags` – RSS offload types on the inner layer of a tunneled packet.

- `rss_queues` – RSS queues array
- `num_of_queues` – number of queues
- `port_id` – destination port ID
- `next_pipe` – next pipe pointer
- `ordered_list_pipe.pipe` – ordered list pipe to select an entry from
- `ordered_list_pipe.idx` – index of the ordered list pipe entry
- `target` - target pointer

The `type` field includes the forwarding action types defined in the following enum:

- `DOCA_FLOW_FWD_RSS` – forwards packets to RSS
- `DOCA_FLOW_FWD_PORT` – forwards packets to port
- `DOCA_FLOW_FWD_PIPE` – forwards packets to another pipe
- `DOCA_FLOW_FWD_DROP` – drops packets
- `DOCA_FLOW_FWD_ORDERED_LIST_PIPE` - forwards packet to a specific entry in an ordered list pipe
- `DOCA_FLOW_FWD_TARGET` – forwards packets to a target

The `rss_outer_flags` and `rss_inner_flags` fields must be configured exclusively (either outer or inner).

Each outer/inner field is a bitwise OR of the RSS fields defined in the following enum:

- `DOCA_FLOW_RSS_IPV4` – RSS by IPv4 header
- `DOCA_FLOW_RSS_IPV6` – RSS by IPv6 header
- `DOCA_FLOW_RSS_UDP` – RSS by UDP header

- `DOCA_FLOW_RSS_TCP` – RSS by TCP header

When specifying an RSS L4 type (`DOCA_FLOW_RSS_TCP` or `DOCA_FLOW_RSS_UDP`) it must have a bitwise OR with RSS L3 types (`DOCA_FLOW_RSS_IPV4` or `DOCA_FLOW_RSS_IPV6`).

doca_flow_query

This struct is a flow query result.

```
struct doca_flow_query {
        uint64_t total_bytes;
        uint64_t total_pkts;
};
```

The struct `doca_flow_query` contains the following elements:

- `total_bytes` – total bytes that hit this flow.
- `total_ptks` – total packets that hit this flow.

doca_flow_init

This function is the global initialization function for DOCA Flow.

```
doca_error_t doca_flow_init(const struct doca_flow_cfg *cfg);
```

- `cfg [in]` – Pointer to flow config structure.
- Returns – `DOCA_SUCCESS` on success. Error code in case of failure:

- `DOCA_ERROR_INVALID_VALUE` – Received invalid input
- `DOCA_ERROR_NO_MEMORY` – Memory allocation failed
- `DOCA_ERROR_NOT_SUPPORTED` – Unsupported pipe type
- `DOCA_ERROR_UNKNOWN` – Otherwise

Note

`doca_flow_init` must be invoked first before any other function in this API. This is a one-time call used for DOCA Flow initialization and global configurations.

doca_flow_destroy

This function is the global destroy function for DOCA Flow.

```
void doca_flow_destroy(void);
```

Note

`doca_flow_destroy` must be invoked last to stop using DOCA Flow.

doca_flow_port_start

This function starts a port with its given configuration. It creates one port in the DOCA Flow layer, allocates all resources used by this port, and creates the default offload flow rules to redirect packets into software queues .

```
doca_error_t doca_flow_port_start(const struct doca_flow_port_cfg
*cfg,
                                  struct doca_flow_port **port);
```

- `cfg [in]` – Pointer to DOCA Flow config structure.
- `port [out]` – Pointer to DOCA Flow port handler on success.
- Returns – `DOCA_SUCCESS` on success. Error code in case of failure:
  - `DOCA_ERROR_INVALID_VALUE` – Received invalid input
  - `DOCA_ERROR_NO_MEMORY` – Memory allocation failed
  - `DOCA_ERROR_NOT_SUPPORTED` – Unsupported pipe type
  - `DOCA_ERROR_UNKNOWN` – Otherwise

Note

`doca_flow_port_start` modifies the state of the underlying DPDK port implementing the DOCA port. The DPDK port is stopped, then the flow configuration is applied, calling `rte_flow_configure` before starting the port again.

Note

`doca_flow_port_start` must be called before any other DOCA Flow API to avoid conflicts.

Note

In `switch` mode, the representor port must be stopped before switch port is stopped.

doca_flow_port_stop

This function releases all resources used by a DOCA flow port and removes the port's default offload flow rules.

```
doca_error_t doca_flow_port_stop(struct doca_flow_port *port);
```

- `port [in]` – Pointer to DOCA Flow port handler.

doca_flow_port_pair

This function pairs two DOCA ports. After successfully pairing the two ports, traffic received on either port is transmitted via the other port by default.

For a pair of non-representor ports, this operation is required before port-based forwarding flows can be created. It is optional, however, if either port is a representor.

Note

The two paired ports have no order.

Note

A port cannot be paired with itself.

```
doca_error_t *doca_flow_port_pair(struct doca_flow_port *port,
                                  struct doca_flow_port
*pair_port);
```

- `port [in]` – Pointer to the DOCA Flow port structure.
- `pair_port [in]` – Pointer to another DOCA Flow port structure.
- Returns – `DOCA_SUCCESS` on success. Error code in case of failure:
  - `DOCA_ERROR_INVALID_VALUE` – Received invalid input.
  - `DOCA_ERROR_NO_MEMORY` – Memory allocation failed.
  - `DOCA_ERROR_UNKNOWN` – Otherwise.

doca_flow_pipe_create

This function creates a new pipeline to match and offload specific packets. The pipeline configuration is defined in the `doca_flow_pipe_cfg`. The API creates a new pipe, but does not start the hardware offload.

When `cfg` type is `DOCA_FLOW_PIPE_CONTROL`, the function creates a special type of pipe that can have dynamic matches and forwards with priority.

```
doca_error_t
doca_flow_pipe_create(const struct doca_flow_pipe_cfg *cfg,
                      const struct doca_flow_fwd *fwd,
                      const struct doca_flow_fwd *fwd_miss,
                      struct doca_flow_pipe **pipe);
```

- `cfg [in]` – Pointer to flow pipe config structure.
- `fwd [in]` – Pointer to flow forward config structure.
- `fwd_miss [in]` – Pointer to flow forward miss config structure. NULL for no `fwd_miss`.

> **Note**
>
> When `fwd_miss` configuration is provided for basic and hash pipes , they are executed on miss. For any other pipe type, the configuration is ignored.

- `pipe [out]` – Pointer to pipe handler on success.
- Returns – `DOCA_SUCCESS` on success. Error code in case of failure:
  - `DOCA_ERROR_INVALID_VALUE` – Received invalid input
  - `DOCA_ERROR_NOT_SUPPORTED` – Unsupported pipe type
  - `DOCA_ERROR_DRIVER` – Driver error

doca_flow_pipe_add_entry

This function adds a new entry to a pipe. When a packet matches a single pipe, it starts hardware offload. The pipe defines which fields to match. This API does the actual hardware offload, with the information from the fields of the input packets.

```
doca_error_t
doca_flow_pipe_add_entry(uint16_t pipe_queue,
                         struct doca_flow_pipe *pipe,
                         const struct doca_flow_match *match,
                         const struct doca_flow_actions *actions,
                         const struct doca_flow_monitor *monitor,
                         const struct doca_flow_fwd *fwd,
                         unit32_t flags,
                         void *usr_ctx,
                         struct doca_flow_pipe_entry **entry);
```

- `pipe_queue [in]` – Queue identifier
- `priority [in]` – Priority value
- `pipe [in]` – Pointer to flow pipe
- `match [in]` – Pointer to flow match. Indicates specific packet match information.
- `actions [in]` – Pointer to modify actions. Indicates specific modify information.
- `monitor [in]` – Pointer to monitor actions
- `fwd [in]` – Pointer to flow forward actions
- `flags [in]` – can be set as `DOCA_FLOW_WAIT_FOR_BATCH` or `DOCA_FLOW_NO_WAIT`
    - `DOCA_FLOW_WAIT_FOR_BATCH` – this entry waits to be pushed to hardware

    - `DOCA_FLOW_NO_WAIT` – this entry is pushed to hardware immediately
- `usr_ctx [in]` – Pointer to user context (see note at doca_flow_entry_process_cb)
- `entry [out]` – Pointer to pipe entry handler on success
- Returns – `DOCA_SUCCESS` on success. Error code in case of failure:
    - `DOCA_ERROR_INVALID_VALUE` – Received invalid input
    - `DOCA_ERROR_DRIVER` – Driver error

> **Note**
>
> Some sanity checks may be omitted to avoid extra delays during flow insertion. For example, when forwarding to a pipe, the `next_pipe` field of struct `doca_flow_fwd` must contain a valid pointer. DOCA does not detect misconfigurations like these unless `doca_flow_init()` is configured to with the `verify` option.

doca_flow_pipe_update_entry

This function updates an entry with a new set of actions.

```
doca_error_t
doca_flow_pipe_update_entry(uint16_t pipe_queue,
                            struct doca_flow_pipe *pipe,
                            const struct doca_flow_actions *actions,
                            const struct doca_flow_monitor *mon,
                            const struct doca_flow_fwd *fwd,
                            const enum doca_flow_flags_type flags,
                            struct doca_flow_pipe_entry *entry);
```

- `pipe_queue [in]` – Queue identifier.
- `pipe [in]` – Pointer to flow pipe.
- `actions [in]` – Pointer to modify actions. Indicates specific modify information.
- `mon [in]` – Pointer to monitor actions.
- `fwd [in]` – Pointer to flow forward actions.
- `flags [in]` – can be set as `DOCA_FLOW_WAIT_FOR_BATCH` or `DOCA_FLOW_NO_WAIT`.
  - `DOCA_FLOW_WAIT_FOR_BATCH` – this entry waits to be pushed to hardware.
  - `DOCA_FLOW_NO_WAIT` – this entry is pushed to hardware immediately.
- `entry [in]` – Pointer to pipe entry to update.
- Returns – `DOCA_SUCCESS` on success. Error code in case of failure:
  - `DOCA_ERROR_INVALID_VALUE` – Received invalid input
  - `DOCA_ERROR_DRIVER` – Driver error

doca_flow_pipe_control_add_entry

This function adds a new entry to a control pipe. When a packet matches a single pipe, it starts hardware offload. The pipe defines which fields to match. This API does the actual hardware offload with the information from the fields of the input packets.

```
doca_error_t
doca_flow_pipe_control_add_entry(uint16_t pipe_queue,
                         uint32_t priority,
                         struct doca_flow_pipe *pipe,
                         const struct doca_flow_match *match,
                         const struct doca_flow_match
*match_mask,
                         const struct doca_flow_actions *actions,
                         const struct doca_flow_actions
*actions_mask,
                         const struct doca_flow_action_descs
*action_descs,
                         const struct doca_flow_monitor *monitor,
                         const struct doca_flow_fwd *fwd,
                         struct doca_flow_pipe_entry **entry);
```

- `pipe_queue [in]` – Queue identifier.
- `priority [in]` – Priority value.
- `pipe [in]` – Pointer to flow pipe.
- `match [in]` – Pointer to flow match. Indicates specific packet match information.
- `match_mask [in]` – Pointer to flow match mask information.
- `actions [in]` – Pointer to modify actions. Indicates specific modify information.
- `actions_mask [in]` – Pointer to modify actions' mask. Indicates specific modify mask information.

- `action_descs [in]` – Pointer to action descriptors.
- `monitor [in]` – Pointer to monitor actions.
- `fwd [in]` – Pointer to flow forward actions.
- `entry [out]` – Pointer to pipe entry handler on success.
- Returns – `DOCA_SUCCESS` on success. Error code in case of failure:
  - `DOCA_ERROR_INVALID_VALUE` – Received invalid input
  - `DOCA_ERROR_DRIVER` – Driver error

doca_flow_pipe_lpm_add_entry

This function adds a new entry to an LPM pipe. This API does the actual hardware offload all entries when `flags` is set to `DOCA_FLOW_NO_WAIT`.

```
doca_error_t
doca_flow_pipe_lpm_add_entry(uint16_t pipe_queue,
                        struct doca_flow_pipe *pipe,
                        const struct doca_flow_match *match,
                        const struct doca_flow_match
*match_mask,
                                                const struct
doca_flow_actions *actions,
                                                const struct
doca_flow_monitor *monitor,
                        const struct doca_flow_fwd *fwd,
                                                unit32_t flags,
                                                void *usr_ctx,
                        struct doca_flow_pipe_entry **entry);
```

- `pipe_queue [in]` – Queue identifier.
- `pipe [in]` – Pointer to flow pipe.
- `match [in]` – Pointer to flow match. Indicates specific packet match information.
- `match_mask [in]` – Pointer to flow match mask information.
- `actions [in]` – Pointer to modify actions. Indicates specific modify information.
- `monitor [in]` – Pointer to monitor actions.
- `fwd [in]` – Pointer to flow FWD actions.
- `flags [in]` – Can be set as `DOCA_FLOW_WAIT_FOR_BATCH` or `DOCA_FLOW_NO_WAIT`.
  - `DOCA_FLOW_WAIT_FOR_BATCH` – LPM collects this flow entry
  - `DOCA_FLOW_NO_WAIT` – LPM adds this entry, builds the LPM software tree, and pushes all entries to hardware immediately
- `usr_ctx [in]` – Pointer to user context (see note at doca_flow_entry_process_cb)
- `entry [out]` – Pointer to pipe entry handler on success.
- Returns – `DOCA_SUCCESS` on success. Error code in case of failure:
  - `DOCA_ERROR_INVALID_VALUE` – Received invalid input.
  - `DOCA_ERROR_DRIVER` – Driver error.

doca_flow_pipe_lpm_update_entry

This function updates an LPM entry with a new set of actions.

```
doca_error_t
doca_flow_pipe_lpm_update_entry(uint16_t pipe_queue,
                        struct doca_flow_pipe *pipe,
                                                const struct
doca_flow_actions *actions,
                                                const struct
doca_flow_monitor *monitor,
                        const struct doca_flow_fwd *fwd,
                                                const enum
doca_flow_flags_type flags,
                        struct doca_flow_pipe_entry *entry);
```

- `pipe_queue [in]` – Queue identifier.
- `pipe [in]` – Pointer to flow pipe.
- `actions [in]` – Pointer to modify actions. Indicates specific modify information.
- `monitor [in]` – Pointer to monitor actions.
- `fwd [in]` – Pointer to flow FWD actions.
- `flags [in]` – Can be set as `DOCA_FLOW_WAIT_FOR_BATCH` or `DOCA_FLOW_NO_WAIT`.
  - `DOCA_FLOW_WAIT_FOR_BATCH` – LPM collects this flow entry
  - `DOCA_FLOW_NO_WAIT` – LPM updates this entry and pushes all entries to hardware immediately
- `entry [in]` – Pointer to pipe entry to update.
- Returns – `DOCA_SUCCESS` on success. Error code in case of failure:
  - `DOCA_ERROR_INVALID_VALUE` – Received invalid input.

- `DOCA_ERROR_DRIVER` – Driver error.

doca_flow_pipe_acl_add_entry

This function adds a new entry to an ACL pipe. This API performs the actual hardware offload for all entries when `flags` is set to `DOCA_FLOW_NO_WAIT`.

```
doca_error_t
doca_flow_pipe_acl_add_entry(uint16_t pipe_queue,
                             struct doca_flow_pipe *pipe,
                             const struct doca_flow_match *match,
                             const struct doca_flow_match
*match_mask,
                                                   uint8_t
priority,
                             const struct doca_flow_fwd *fwd,
                                                   unit32_t flags,
                                                   void *usr_ctx,
                             struct doca_flow_pipe_entry **entry);
```

- `pipe_queue [in]` – Queue identifier.
- `pipe [in]` – Pointer to flow pipe.
- `match [in]` – Pointer to flow match. Indicates specific packet match information.
- `match_mask [in]` – Pointer to flow match mask information.
- `priority [in]` – Priority value.
- `fwd [in]` – Pointer to flow FWD actions.
- `flags [in]` – Can be set as `DOCA_FLOW_WAIT_FOR_BATCH` or `DOCA_FLOW_NO_WAIT`.

    - `DOCA_FLOW_WAIT_FOR_BATCH` – ACL collects this flow entry
    - `DOCA_FLOW_NO_WAIT` – ACL adds this entry, builds the ACL software model, and pushes all entries to hardware immediately
- `usr_ctx [in]` – Pointer to user context (see note at doca_flow_entry_process_cb)
- `entry [out]` – Pointer to pipe entry handler on success.
- Returns – `DOCA_SUCCESS` on success. Error code in case of failure:
    - `DOCA_ERROR_INVALID_VALUE` – Received invalid input
    - `DOCA_ERROR_DRIVER` – Driver error

doca_flow_pipe_ordered_list_add_entry

This function adds a new entry to an order list pipe. When a packet matches a single pipe, it starts hardware offload. The pipe defines which fields to match. This API does the actual hardware offload, with the information from the fields of the input packets.

```
doca_error_t
doca_flow_pipe_ordered_list_add_entry(uint16_t pipe_queue,
                                                    struct
doca_flow_pipe *pipe,
                                                    uint32_t idx,
                                                    const struct
doca_flow_ordered_list *ordered_list,
                                                    const struct
doca_flow_fwd *fwd,
                                                    enum
doca_flow_flags_type flags,
                                                    void *user_ctx,
                                                    struct
doca_flow_pipe_entry **entry);
```

- `pipe_queue [in]` – Queue identifier.
- `pipe [in]` – Pointer to flow pipe.
- `idx [in]` – A unique entry index. It is the user's responsibility to ensure uniqueness.
- `ordered_list [in]` – Pointer to an ordered list structure with pointers to struct `doca_flow_actions` and struct `doca_flow_monitor` at the same indices as they were at the pipe creation time. If the configuration contained an element of struct `doca_flow_action_descs`, the corresponding array element is ignored and can be NULL.
- `fwd [in]` – Pointer to flow FWD actions.
- `flags [in]` – Can be set as `DOCA_FLOW_WAIT_FOR_BATCH` or `DOCA_FLOW_NO_WAIT`.
  - `DOCA_FLOW_WAIT_FOR_BATCH` – this entry waits to be pushed to hardware
  - `DOCA_FLOW_NO_WAIT` – this entry is pushed to hardware immediately

- `user_ctx [in]` – Pointer to user context (see note at doca_flow_entry_process_cb).
- `entry [out]` – Pointer to pipe entry handler to fill.
- Returns – `DOCA_SUCCESS` in case of success. Error code in case of failure:
  - `DOCA_ERROR_INVALID_VALUE` – Received invalid input.
  - `DOCA_ERROR_NO_MEMORY` – Memory allocation failed.
  - `DOCA_ERROR_DRIVER` – Driver error.

doca_flow_pipe_hash_add_entry

This function adds a new entry to a hash pipe. When a packet matches a single pipe, it starts hardware offload. The pipe defines which fields to match. This API does the actual hardware offload with the information from the fields of the input packets.

```
doca_error_t
doca_flow_pipe_hash_add_entry(uint16_t pipe_queue,
                              struct doca_flow_pipe *pipe,
                              uint32_t entry_index,
                              const struct doca_flow_actions
*actions,
                              const struct doca_flow_monitor
*monitor,
                              const struct doca_flow_fwd *fwd,
                              const enum doca_flow_flags_type
flags,
                              void *usr_ctx,
                              struct doca_flow_pipe_entry
**entry);
```

- `pipe_queue [in]` – Queue identifier.
- `pipe [in]` – Pointer to flow pipe.
- `entry_index [in]` – A unique entry index. If the index is not unique, the function returns error.
- `actions [in]` – Pointer to modify actions. Indicates specific modify information.
- `monitor [in]` – Pointer to monitor actions.
- `fwd [in]` – Pointer to flow FWD actions.
- `flags [in]` – Can be set as `DOCA_FLOW_WAIT_FOR_BATCH` or `DOCA_FLOW_NO_WAIT`
  - `DOCA_FLOW_WAIT_FOR_BATCH` – This entry waits to be pushed to hardware
  - `DOCA_FLOW_NO_WAIT` – This entry is pushed to hardware immediately
- `usr_ctx [in]` – Pointer to user context (see note at doca_flow_entry_process_cb)
- `entry [out]` – Pointer to pipe entry handler to fill.
- Returns – `DOCA_SUCCESS` in case of success. Error code in case of failure:
  - `DOCA_ERROR_INVALID_VALUE` – Received invalid input.
  - `DOCA_ERROR_DRIVER` – Driver error.

doca_flow_pipe_resize

This function resizes a pipe (currently only type `DOCA_FLOW_PIPE_CONTROL` is supported if, on creation, its configuration had `is_resizable == true`). The new congestion level is the percentage by which the new pipe size is determined. For example, if there are 800 entries in the pipe and congestion level is 50%, the new size of the pipe would be 1600 entries rounded up to the nearest a power of 2. The `nr_entries_changed_cb` and `entry_relocation_cb` are optional callbacks. The

first callback informs on the exact new number of entries in the pipe. The second callback informs on entries that have been relocated from the smaller to the resized pipe.

```
doca_error_t
doca_flow_pipe_resize(struct doca_flow_pipe *pipe,
                      uint8_t new_congestion_level,
                      doca_flow_pipe_resize_nr_entries_changed_cb
nr_entries_changed_cb,
                      doca_flow_pipe_resize_entry_relocate_cb
entry_relocation_cb);
```

- `pipe [in]` – Pointer to pipe to be resized
- `new_congestion_level [in]` – Percentage to calculate the new new pipe size based on the current number of entries. The final size may be rounded up to the nearest power of 2.
- `nr_entries_changed_cb [in]` – Pointer to callback when the new pipe size is set.
- `entry_relocation_cb [in]` – Pointer to callback for each entry in the pipe that will be part of the resized pipe.

> **Note**
>
> Callbacks are optional. Either both are set or none.

- Returns – `DOCA_SUCCESS` on success. Error code in case of failure.

doca_flow_pipe_process_cb

This typedef callback defines the received notification API for once congestion is reached or once the resize operation is completed.

```
enum doca_flow_pipe_op {
        DOCA_FLOW_PIPE_OP_CONGESTION_REACHED,
        DOCA_FLOW_PIPE_OP_RESIZED,
                ...
};
enum doca_flow_pipe_status {
        DOCA_FLOW_PIPE_STATUS_SUCCESS = 1,
        DOCA_FLOW_PIPE_STATUS_ERROR,
};
typedef void (*doca_flow_pipe_process_cb)(struct doca_flow_pipe
*pipe,
                                          enum
doca_flow_pipe_status status,
                                          enum doca_flow_pipe_op
op, void *user_ctx);
```

- `pipe [in]` – Pointer to the pipe whose process is reported.
- `status [in]` – Process completion status (i.e., success or error).
- `op [in]` – Operation reported in process, such as `CONGESTION_REACHED`, `RESIZED`.
- `user_ctx [in]` – Pointer to user pipe context as configured on pipe creation.

> **Note**
>
> Callbacks are optional. Either both are set or none.

- Returns – `DOCA_SUCCESS` on success. Error code in case of failure.

doca_flow_entries_process

This function processes entries in the queue. The application must invoke this function to complete flow rule offloading and to receive the flow rule's operation status.

If `doca_flow_pipe_resize()` is called, this function must be invoked as well to complete the resize process (i.e., until `DOCA_FLOW_OP_PIPE_RESIZED` is received as part of `doca_flow_pipe_process_cb()` callback).

```
doca_error_t
doca_flow_entries_process(struct doca_flow_port *port,
                          uint16_t pipe_queue,
                          uint64_t timeout,
                          uint32_t max_processed_entries);
```

- `port [in]` – Pointer to the flow port structure.
- `pipe_queue [in]` – Queue identifier.
- `timeout [in]` – Timeout value in microseconds.
- `max_processed_entries [in]` – Pointer to the flow pipe.
- Returns – `DOCA_SUCCESS` on success. Error code in case of failure:
    - `DOCA_ERROR_DRIVER` – Driver error.

doca_flow_entry_status

This function gets the status of pipe entry.

```
enum doca_flow_entry_status
doca_flow_entry_get_status(struct doca_flow_entry *entry);
```

- `entry [in]` – Pointer to the flow pipe entry to query.
- Returns – Entry's status, defined in the following enum:
  - `DOCA_FLOW_ENTRY_STATUS_IN_PROCESS` – the operation is in progress
  - `DOCA_FLOW_ENTRY_STATUS_SUCCESS` – the operation completed successfully
  - `DOCA_FLOW_ENTRY_STATUS_ERROR` – the operation failed

doca_flow_entry_query

This function queries packet statistics about a specific pipe entry.

> **Note**
>
> The pipe must have been created with the `DOCA_FLOW_MONITOR_COUNT` flag or the query will return an error.

```
doca_error_t doca_flow_query_entry(struct doca_flow_pipe_entry
*entry, struct doca_flow_query *query_stats);
```

- `entry [in]` – Pointer to the flow pipe entry to query
- `query_stats [out]` – Pointer to the data retrieved by the query

- Returns – `DOCA_SUCCESS` on success. Error code in case of failure:
  - `DOCA_ERROR_INVALID_VALUE` – Received invalid input
  - `DOCA_ERROR_UNKNOWN` – Otherwise

doca_flow_query_pipe_miss

This function queries packet statistics about a specific pipe miss flow.

> **Note**
>
> The pipe must have been created with the `miss_counter` set to true or the query will return an error.

```
doca_error_t doca_flow_query_pipe_miss(struct doca_flow_pipe
*pipe, struct doca_flow_query *query_stats);
```

- `pipe [in]` – Pointer to the flow pipe to query.
- `query_stats [out]` – Pointer to the data retrieved by the query.
- Returns – `DOCA_SUCCESS` on success. Error code in case of failure:
  - `DOCA_ERROR_INVALID_VALUE` – Received invalid input
  - `DOCA_ERROR_UNKNOWN` – Otherwise

doca_flow_aging_handle

This function handles the aging of all the pipes of a given port. It goes over all flows and releases aged flows from being tracked. The user gets a notification in the callback about the aged entries. Since the number of flows can be very large, it can take a significant amount of time to go over all flows, so this function is limited by a time quota. This means it might return without handling all flows which requires the user to call it again.

Note

The pipe needs to have been created with the `DOCA_FLOW_MONITOR_COUNT` flag or the query will return an error.

```
int doca_flow_aging_handle(struct doca_flow_port *port,
                           uint16_t queue,
                           uint64_t quota);
```

- `queue [in]` – Queue identifier.
- `quota [in]` – Max time quota in microseconds for this function to handle aging.
- Returns
  - >0 – the number of aged flows filled in entries array.
  - 0 – no aged entries in current call.
  - -1 – full cycle is done.

doca_flow_mpls_label_encode

This function prepares an MPLS label header in big-endian. Input variables are provided in CPU-endian.

```
doca_error_t
doca_flow_mpls_label_encode(uint32_t label, uint8_t
traffic_class, uint8_t ttl, bool bottom_of_stack,
                            struct doca_flow_header_mpls *mpls);
```

- `label [in]` – Label value (20 bits).
- `traffic_class [in]` – Traffic class (3 bits).
- `ttl [in]` – Time to live (8 bits).
- `bottom_of_stack [in]` – Whether this MPLS is bottom-of-stack.
- `mpls [out]` – Pointer to MPLS structure to fill.
- Returns – `DOCA_SUCCESS` on success. Error code in case of failure:
  - `DOCA_ERROR_INVALID_VALUE` – Received invalid input.

doca_flow_mpls_label_decode

This function decodes an MPLS label header. Output variables are returned in CPU-endian.

```
doca_error_t
doca_flow_mpls_label_decode(const struct doca_flow_header_mpls
*mpls, uint32_t *label,
                            uint8_t *traffic_class, uint8_t *ttl,
bool *bottom_of_stack);
```

- `mpls [in]` – Pointer to MPLS structure to decode.
- `label [out]` – Pointer to fill MPLS label value.

- `traffic_class [out]` – Pointer to fill MPLS traffic class value.
- `ttl [out]` – Pointer to fill MPLS TTL value.
- `bottom_of_stack [out]` – Pointer to fill whether this MPLS is bottom-of-stack.
- Returns – `DOCA_SUCCESS` on success. Error code in case of failure:
  - `DOCA_ERROR_INVALID_VALUE` – Received invalid input

doca_flow_parser_geneve_opt_create

Note

This function must be called before creation of any pipe using GENEVE option.

Note

This operation is only supported when `FLEX_PARSER_PROFILE_ENABLE = 8`. To do that, run:

```
mlxconfig -d <device-id> set
FLEX_PARSER_PROFILE_ENABLE=8
```

This function prepares a GENEVE TLV parser for a selected port.

This API is port oriented, but the configuration is done once for all ports under the same physical device. Each port should call this API before using GENEVE options, but it must use the same options in the same order in the list.

Each physical device has 7 DWs for GENEVE TLV options. Each non-zero element in the `data_mask` array consumes one DW, and choosing a matchable mode per class consumes additional one.

> **Info**
>
> Calling this API for a second port under the same physical device does not consume more DWs as it uses same configuration.

```
doca_error_t
doca_flow_parser_geneve_opt_create(const struct doca_flow_port
*port,
                                   const struct
doca_flow_parser_geneve_opt_cfg tlv_list[i],
                                   uint8_t nb_options, struct
doca_flow_parser **parser);
```

- `port [in]` – Pointer to DOCA Flow port.
- `tlv_list [in]` – An array to create GENEVE TLV parser for. The index option in this array is used as an option identifier in the action descriptor string.
- `nb_options [in]` – The number of options in the TLV array.
- `parser [out]` – Pointer to parser handler to fill on success.
- Returns – `DOCA_SUCCESS` on success. Error code in case of failure:
  - `DOCA_ERROR_INVALID_VALUE` – Received invalid input
  - `DOCA_ERROR_NO_MEMORY` – Memory allocation failed
  - `DOCA_ERROR_NOT_SUPPORTED` – Unsupported configuration

- `DOCA_ERROR_ALREADY_EXIST` – Physical device already has parser, by either same or another port
- `DOCA_ERROR_UNKNOWN` – Otherwise

doca_flow_parser_geneve_opt_destroy

> **Note**
>
> This function must be called after the last use of the GENEVE option and before port closing.

This function destroys GENEVE TLV parser.

```
doca_error_t
doca_flow_parser_geneve_opt_destroy(struct doca_flow_parser
*parser);
```

- `parser [in]` – Pointer to parser to be destroyed.
- Returns – `DOCA_SUCCESS` on success. Error code in case of failure:
  - `DOCA_ERROR_INVALID_VALUE` – Received invalid input.
  - `DOCA_ERROR_IN_USE` – One of the options is being used by a pipe.
  - `DOCA_ERROR_DRIVER` – There is no valid GENEVE TLV parser in this handle.
  - `DOCA_ERROR_UNKNOWN` – Otherwise.

doca_flow_get_target

This function gets a target handler.

```
doca_error_t
doca_flow_get_target(enum doca_flow_target_type, struct
doca_flow_target **target);
```

- `type [in]` – Target type.
- `target [out]` – Pointer to target handler.
- Returns – `DOCA_SUCCESS` on success. Error code in case of failure:
  - `DOCA_ERROR_INVALID_VALUE` – Received invalid input.
  - `DOCA_ERROR_NOT_SUPPORTED` – Unsupported type.

The `type` field includes the following target type:

- `DOCA_FLOW_TARGET_KERNEL`

doca_flow_port_switch_get

```
doca_flow_port_switch_get(const struct doca_flow_port *port);
```

- `port [in]` – The port for which to get the associated switch port
- Returns - the switch port or NULL if none exists

doca_flow_pipe_calc_hash

```
doca_flow_pipe_calc_hash(struct doca_flow_pipe *pipe, const struct
doca_flow_match *match, uint32_t *hash);
```

- `pipe [in]` – Pointer to flow pipe.
- `match [in]` – Pointer to flow match. Indicates specific packet match information.
- `hash [out]` – Holds the calculation result for a given packet.

Limitations:

- Can only be used by hash pipe
- Must have only fixed actions
- Only supports full masked items. Partial masks are considered full mask.

Shared Counter Resource

A shared counter can be used in multiple pipe entries. The following are the steps for configuring and using shared counters.

On doca_flow_init()

Specify the total number of shared counters to be used, `nb_shared_counters`.

This call implicitly defines the shared counters IDs in the range of 0-`nb_shared_counters-1`.

```
.nr_shared_resources = {
    [DOCA_FLOW_SHARED_RESOURCE_COUNT] = nb_shared_counters
},
```

On doca_flow_shared_resource_cfg()

This call can be skipped for shared counters.

On doca_flow_shared_resource_bind()

This call binds a bulk of shared counters IDs to a specific pipe or port.

```
doca_error_t
doca_flow_shared_resources_bind(enum
doca_flow_shared_resource_type type, uint32_t *res_array,
                                uint32_t res_array_len, void
*bindable_obj);
```

- `res_array [in]` – Array of shared counters IDs to be bound.
- `res_array_len [in]` – Array length.
- `bindable_obj` – Pointer to either a pipe or port.

This call allocates the counter's objects. A counter ID specified in this array can only be used later by the corresponding bindable object (pipe or port).

The following example binds counter IDs 2, 4, and 7 to a pipe. The counters' IDs must be within the range 0-`nb_shared_coutners-1`.

```
uint32_t shared_counters_ids[] = {2, 4, 7};
struct doca_flow_pipe *pipe = ...

doca_flow_shared_resources_bind(
        DOCA_FLOW_SHARED_RESOURCE_COUNT,
        shared_counters_ids, 3, pipe, &error);
```

On doca_flow_pipe_add_entry() or Pipe Configuration (struct doca_flow_pipe_cfg)

The shared counter ID is included in the monitor parameter. It must be bound in advance to the pipe object.

```
struct doca_flow_monitor {
        ...
        uint32_t shared_counter_id;
        /**< shared counter id */
        ...
}
```

Packets matching the pipe entry are counted on the `shared_counter_id`. In pipe configuration, the `shared_counter_id` can be changeable (all FFs) and then the pipe entry holds the specific shared counter ID.

In switch mode, verifying counter domain is skipped.

Querying Bulk of Shared Counter IDs

Use this API:

```
int
      doca_flow_shared_resources_query(enum
doca_flow_shared_resource_type type,
      uint32_t *res_array,
      struct doca_flow_shared_resource_result
*query_results_array,
      uint32_t array_len,
      struct doca_flow_error *error);
```

- `res_array [in]` – Array of shared counters IDs to be queried.
- `res_array_len [in]` – Array length.
- `query_results_array [out]` – Query results array. The user must have allocated it prior to calling this API.

The `type` parameter is `DOCA_FLOW_SHARED_RESOURCE_COUNT`.

On doca_flow_pipe_destroy() or doca_flow_port_stop()

All bound resource IDs of this pipe or port are destroyed.

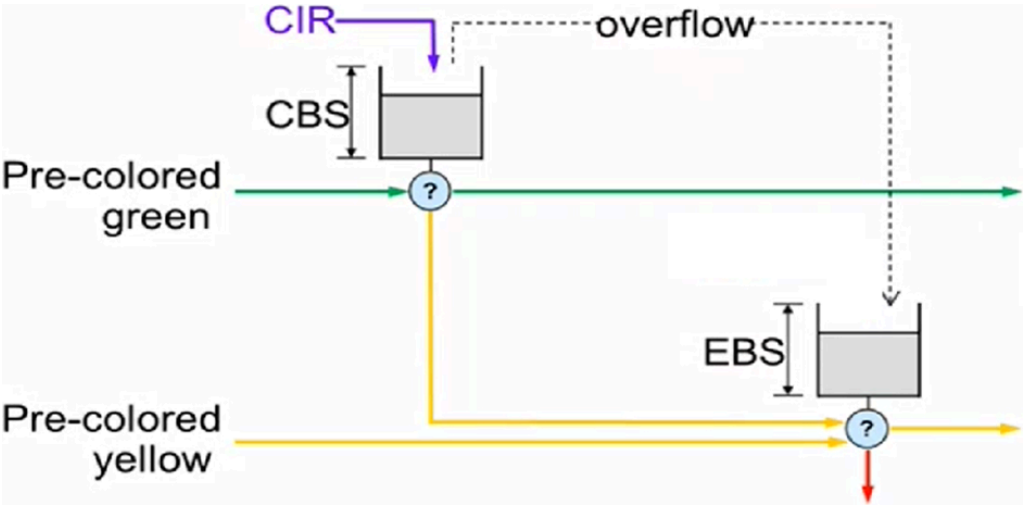
Shared Meter Resource

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`.

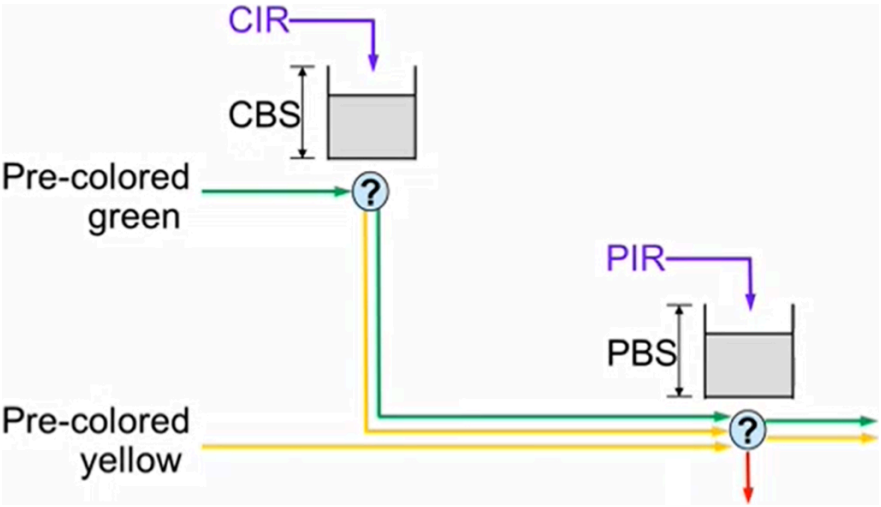
DOCA Flow supports three marking algorithms based on RFCs: 2697, 2698, and 4115.

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.

On doca_flow_init()

Specify the total number of shared meters to be used, `nb_shared_meters`.

The following call is an example how to initialize both shared counters and meter ranges. This call implicitly defines the shared counter IDs in the range of 0-`nb_shared_counters-1` and the shared meter IDs in the range of 0-`nb_shared_meters-1`.

```
struct doca_flow_cfg cfg = {
                  .queues = queues,
                  ...
                  .nr_shared_resources = {nb_shared_meters,
nb_shared_counters, ...},
}
doca_flow_init(&cfg, &error);
```

On doca_flow_shared_resource_cfg()

This call binds a specific meter ID with its committed information rate (CIR) and committed burst size (CBS).

```
struct doca_flow_resource_meter_cfg {
                ...
        uint64_t cir;
        /**< Committed Information Rate (bytes/second). */
        uint64_t cbs;
        /**< Committed Burst Size (bytes). */
                ...
};

struct doca_flow_shared_resource_cfg {
        union {
                struct doca_flow_resource_meter_cfg meter_cfg;
                        ...
        };
};

int
doca_flow_shared_resource_cfg(enum doca_flow_shared_resource_type
type, uint32_t id,
                              struct
doca_flow_shared_resource_cfg *cfg,
                              struct doca_flow_error *error);
```

The following example configures the shared meter ID 5 with a CIR of 0x1000 bytes per second and a CBS of 0x600 bytes:

```
struct doca_flow_shared_resource_cfg shared_cfg = { 0 };
shared_cfg.meter_cfg.cir = 0x1000;
shared_cfg.meter_cfg.cbs = 0x600;
doca_flow_shared_resource_cfg(DOCA_FLOW_SHARED_RESOURCE_METER,
0x5, &shared_cfg, &error);
```

The last meter configuration example sets only the CIR and CBS fields (using RFC 2697 algorithm by default).

The following is the full meter configuration struct:

```
enum doca_flow_meter_algorithm_type {
        DOCA_FLOW_METER_ALGORITHM_TYPE_RFC2697,
        /**< Single Rate Three Color Marker - IETF RFC 2697. */
        DOCA_FLOW_METER_ALGORITHM_TYPE_RFC2698,
        /**< Two Rate Three Color Marker - IETF RFC 2698. */
        DOCA_FLOW_METER_ALGORITHM_TYPE_RFC4115,
        /**< Two Rate Three Color Marker - IETF RFC 4115. */
};

enum doca_flow_meter_limit_type {
        DOCA_FLOW_METER_LIMIT_TYPE_BYTES = 0,
        /**< Meter parameters per bytes */
        DOCA_FLOW_METER_LIMIT_TYPE_PACKETS,
        /**< Meter parameters packets */
};

struct doca_flow_resource_meter_cfg {
        enum doca_flow_meter_limit_type limit_type;
        /**< Meter rate limit type: bytes / packets per second */
        enum doca_flow_meter_algorithm_type alg;
        /**< Meter algorithm by RFCs */
        uint64_t cir;
        /**< Committed Information Rate (bytes or packets per
second). */
        uint64_t cbs;
        /**< Committed Burst Size (bytes or packets). */
        union {
                struct {
                        uint64_t ebs;
                        /** Excess Burst Size (EBS) (bytes or
packets). */
                } rfc2697;
                struct {
                        uint64_t pir;
```

```
                        /**< Peak Information Rate (bytes or
packets per seconds). */
                        uint64_t pbs;
                        /**< Peak Burst Size (bytes or packets).
*/
                } rfc2698;
                struct {
                        uint64_t eir;
                        /**< Excess Information Rate (bytes or
packets per seconds). */
                        uint64_t ebs;
                        /**< Excess Burst Size (EBS) (bytes or
packets). */
                } rfc4115;
        };
};
```

- `limit_type` – Bytes versus packets measurement.
- `alg` – The meter marking RFC algorithm: 2697, 2698, or 4115.
- `cir` – Committed information rate for shared meter.
- `cbs` – Committed burst size of shared meter.
- `Pir` – Peak information rate of shared meter.
- `Pbs` – Peak burst size of shared meter.
- `Eir` – Excess information rate of shared meter.
- `Ebs` – Excess burst size of shared meter.

On doca_flow_shared_resource_bind()

This call binds a bulk of shared meter IDs to a specific pipe or port.

```
doca_error_t
doca_flow_shared_resources_bind(enum
doca_flow_shared_resource_type type, uint32_t *res_array,
                                uint32_t res_array_len, void
*bindable_obj);
```

- `res_array [in]` – array of shared meter IDs to be bound
- `res_array_len [in]` – array length
- `bindable_obj` – pointer to either a pipe or port

This call allocates the meter's objects. A meter ID specified in this array can only be used later by the corresponding bindable object (pipe or port).

The following example binds meter IDs 5 and 14 to a pipe. The meter IDs must be within the range 0-`nb_shared_meters-1`.

```
uint32_t shared_meters_ids[] = {5, 14};
struct doca_flow_pipe *pipe = ...

doca_flow_shared_resources_bind(
        DOCA_FLOW_SHARED_RESOURCE_METER,
        shared_meters_ids, 2, pipe, &error);
```

On doca_flow_pipe_add_entry() or Pipe Configuration (struct doca_flow_pipe_cfg)

The shared meter ID is included in the monitor parameter. It must be bound in advance to the pipe object.

```
struct doca_flow_monitor {
        ...
        uint32_t shared_meter_id;
        /**< shared meter id */
        ...
}
```

Packets matching the pipe entry are metered based on the `cir` and the `cbs` parameters related to the `shared_meter_id`. In the pipe configuration, the `shared_meter_id` can be changeable (all FFs) and then the pipe entry must hold the specific shared meter ID for that entry.

In switch mode, verifying meter domain is skipped.

Querying Bulk of Shared Meter IDs

There is no direct API to query a shared meter ID. To count the number of packets before a meter object, add a counter (shared or single) and use an API to query it. For an example, see section "Querying Bulk of Shared Counter IDs".

On doca_flow_pipe_destroy() or doca_flow_port_stop()

All bound resource IDs of this pipe or port are destroyed.

Shared RSS Resource

A shared RSS can be used in multiple pipe entries.

On doca_flow_init()

Specify the total number of shared RSS to be used, `nb_shared_rss`.

This call implicitly defines the shared RSS IDs in the range between 0 to `nb_shared_rss`-1.

```
struct doca_flow_cfg cfg;

cfg.nr_shared_resources[DOCA_FLOW_SHARED_RESOURCE_RSS] =
nb_shared_rss;
doca_flow_init(&cfg, &error);
```

On doca_flow_shared_resource_cfg()

This call configures shared RSS resource.

```
struct doca_flow_shared_resource_cfg res_cfg;

for (uint8_t i = 0; i < nb_shared_rss; i++) {
        res_cfg.rss_cfg.nr_queues = nr_queues;
        res_cfg.rss_cfg.flags = flags;
        res_cfg.rss_cfg.queues_array = queues_array;

doca_flow_shared_resource_cfg(DOCA_FLOW_SHARED_RESOURCE_RSS, i,
&rss_cfg, &error);
}
```

On doca_flow_shared_resource_bind()

This call binds a bulk of shared RSS to a specific port.

```
uint32_t shared_rss_ids[] = {2, 4, 7};
struct doca_flow_port *port;

doca_flow_shared_resources_bind(
        DOCA_FLOW_SHARED_RESOURCE_RSS,
        shared_rss_ids, 3, port);
```

On doca_flow_pipe_add_entry()

On `doca_flow_pipe_create`, the user can input NULL as fwd. On `doca_flow_pipe_add_entry`, the user can input preconfigured shared RSS as `fwd` by specifying the `shared_rss_id`.

```
struct doca_flow_fwd;

fwd.shared_rss_id = 2;
fwd.type = DOCA_FLOW_FWD_RSS;
doca_flow_pipe_add_entry(queue, pipe, match, action, mon, &fwd,
flag, usr_ctx, &error);
```

On doca_flow_port_stop()

All bound `shared_rss` resource IDs of this port are destroyed.

Shared Crypto Resource

A shared crypto resource can be used in multiple pipe entries and is intended to perform crypto offloads (i.e., encrypt or decrypt packet data operations).

The following subsections expand on the steps for configuring and using shared crypto actions.

On doca_flow_init()

Specify the total number of shared crypto operations to be used, `nb_shared_crypto`.

This call implicitly defines the shared counter IDs in the range of 0-`nb_shared_crypto` -1.

```
struct doca_flow_cfg cfg = {
                  .queues = queues,
                  ...
                  .nr_shared_resources = {0, 0, 0,
nb_shared_crypto},
}
doca_flow_init(&cfg, &error);
```

On doca_flow_shared_resource_cfg()

This call provides the specific crypto ID with its configuration.

```
struct doca_flow_resource_crypto_cfg {
        enum doca_flow_crypto_protocol_type proto_type;
        void* security_ctx;
};

struct doca_flow_shared_resource_cfg {
        union {
                struct doca_flow_crypto_cfg crypto_cfg;
                        ...
        };
};

int
doca_flow_shared_resource_cfg(enum doca_flow_shared_resource_type
type, uint32_t id,
                              struct
doca_flow_shared_resource_cfg *cfg,
                              struct doca_flow_error *error);
```

On doca_flow_shared_resource_bind()

This call binds a bulk of shared crypto IDs to a specific pipe or port.

```
doca_error_t
doca_flow_shared_resources_bind(enum
doca_flow_shared_resource_type type, uint32_t *res_array,
                                uint32_t res_array_len, void
*bindable_obj);
```

- `res_array [in]` – Array of shared crypto IDs to be bound.
- `res_array_len [in]` – Array length.

- `bindable_obj` – Pointer to either a pipe or port.

This call allocates the crypto's objects. A crypto ID specified in this array can only be used later by the corresponding bindable object (pipe or port).

The following example binds crypto IDs 2, 5, and 7 to a pipe. The cryptos' IDs must be within the range 0-`nb_shared_crypto`-1.

```
uint32_t shared_crypto_ids[] = {2, 5, 7};
struct doca_flow_pipe *pipe = ...

doca_flow_shared_resources_bind(
        DOCA_FLOW_SHARED_RESOURCE_CRYPTO,
        shared_crypto_ids, 3, pipe, &error);
```

On doca_flow_pipe_add_entry() or Pipe Configuration (struct doca_flow_pipe_cfg)

The shared crypto ID is included in the action parameter. It must be bound in advance to the pipe object.

```
struct doca_flow_actions {
        ...
        struct {
            enum doca_flow_crypto_protocol_type proto_type;
            /**< Crypto shared action type */
            uint32_t crypto_id;
            /**< Crypto shared action id */
        } security;
}
```

Crypto operations are performed over the packets matching the pipe entry according to the `crypto_id` configuration. Afterwards, the flow continues from the point specified in

the forward part of the pipe configuration. In pipe configuration, the `crypto_id` should reference the shared crypto object ID to fetch the security pool information required for pipe creation. `crypto_id` is always considered changeable (regardless of all FFs) and the pipe entry holds the specific shared crypto ID.

The IPsec shared crypto resource can provide extra information in the `u32[]` fields of `doca_flow_meta` and the application can establish a match on these fields in the next pipes.

> **Note**
>
> Extra information is provided only for actions with an IPsec object configured in full offload mode.

On doca_flow_pipe_destroy() or doca_flow_port_stop()

All bound crypto resource IDs of this pipe or port are destroyed.

Shared Mirror Resource

A shared mirror can be used in multiple pipe entries (hardware steering mode support only). The following are the steps for configuring and using shared mirrors.

On doca_flow_init()

Specify the total number of shared mirrors to be used, `nb_shared_mirrors`.

The following call is an example for how to initialize both shared counters and mirror ranges. This call implicitly defines the shared counter IDs in the range of 0-`nb_shared_counters-1` and the shared mirror IDs in the range of 0-`nb_shared_mirrors-1`.

```
struct doca_flow_cfg cfg = {
                  .queues = queues,
                  ...
                  .nr_shared_resources = {nb_shared_mirrors,
nb_shared_counters},
}
doca_flow_init(&cfg, &error);
```

On doca_flow_shared_resource_cfg()

This call binds a specific mirror ID with its mirror packet destination and original packet destination.

```
struct doca_flow_mirror_target {
       bool has_encap;
       /**< Encap mirrored packets. */
       struct doca_flow_encap_action encap;
       /**< Encap data. */
       struct doca_flow_fwd fwd;
       /**< Mirror target, must be filled. */
};

struct doca_flow_resource_mirror_cfg {
        int nr_targets;
        /**< Mirror target number. */
        struct doca_flow_mirror_target *target;
        /**< Mirror target pointer. */
        struct doca_flow_fwd fwd;
        /**< Original packet dst, can be filled optional. */
};

struct doca_flow_shared_resource_cfg {
        union {
                struct doca_flow_resource_mirror_cfg mirror_cfg;
                        ...
        };
};

int
doca_flow_shared_resource_cfg(enum doca_flow_shared_resource_type
type, uint32_t id,
                              struct
doca_flow_shared_resource_cfg *cfg,
                              struct doca_flow_error *error);
```

The following example configures the shared mirror ID 5 with mirroring the packet to the second hairpin port:

```
struct doca_flow_shared_resource_cfg shared_cfg = { 0 };
target.fwd.type = DOCA_FLOW_FWD_PORT;
target.fwd.port_id = 0;
shared_cfg.mirror_cfg.nr_targets = 1;
shared_cfg.mirror_cfg.target = &target;
doca_flow_shared_resource_cfg(DOCA_FLOW_SHARED_RESOURCE_MIRROR,
0x5, &shared_cfg, &error);
```

On doca_flow_shared_resource_bind()

This call binds a bulk of shared mirror IDs to a specific pipe or port.

```
doca_error_t
doca_flow_shared_resources_bind(enum
doca_flow_shared_resource_type type, uint32_t *res_array,
                                uint32_t res_array_len, void
*bindable_obj);
```

- `res_array [in]` – array of shared mirror IDs to be bound
- `res_array_len [in]` – array length
- `bindable_obj` – pointer to either a pipe or port

This call allocates the mirror's objects. A mirror ID specified in this array can only be used later by the corresponding bindable object (i.e., pipe or port).

Note

Mirror can only be used with a `BASIC` pipe.

The following example binds mirror IDs 5 and 14 to a pipe. The mirror IDs must be within the range 0-`nb_shared_mirrors-1`.

```
uint32_t shared_mirrors_ids[] = {5, 14};
struct doca_flow_pipe *pipe = ...

doca_flow_shared_resources_bind(
        DOCA_FLOW_SHARED_RESOURCE_MIRROR,
        shared_mirrors_ids, 2, pipe, &error);
```

On doca_flow_pipe_add_entry() or Pipe Configuration (struct doca_flow_pipe_cfg)

The shared mirror ID is included in the monitor parameter. It must be bound in advance to the pipe object.

```
struct doca_flow_monitor {
        ...
        uint32_t shared_mirror_id;
        /**< shared mirror id */
        ...
}
```

Packets matching the pipe entry are mirrored to the targets related to the `shared_mirror_id`. In the pipe configuration, the `shared_mirror_id` can be changeable (all FFs) and then the pipe entry must hold the specific shared mirror ID for that entry.

Note

Mirror is not allowed to be used on NIC Tx (`"hws,VNF,Tx"`).

Note

Mirror can only be used with a `BASIC` pipe.

Querying Bulk of Shared Mirror IDs

Query is not supported with mirror.

On doca_flow_pipe_destroy() or doca_flow_port_stop()

All bound resource IDs of this pipe or port are destroyed.

Flow Life Cycle

Initialization Flow

Before using any DOCA Flow function, it is mandatory to call DOCA Flow initialization, `doca_flow_init()`, which initializes all resources used by DOCA Flow.

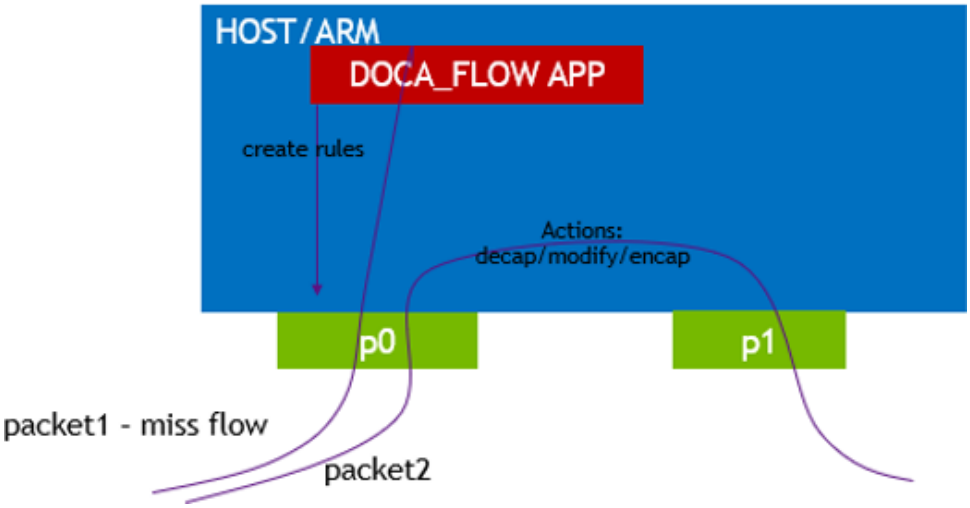
Pipe Mode

This mode (`mode_args`) defines the basic traffic in DOCA. It creates some miss rules when the DOCA port initialized. Currently, DOCA supports 3 types:

- `vnf`

  The packet arrives from one side of the application, is processed, and sent from the other side. The miss packet by default goes to the RSS of all queues.

  The following diagram shows the basic traffic flow in `vnf` mode. Packet1 firstly misses to host RSS queues. 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.

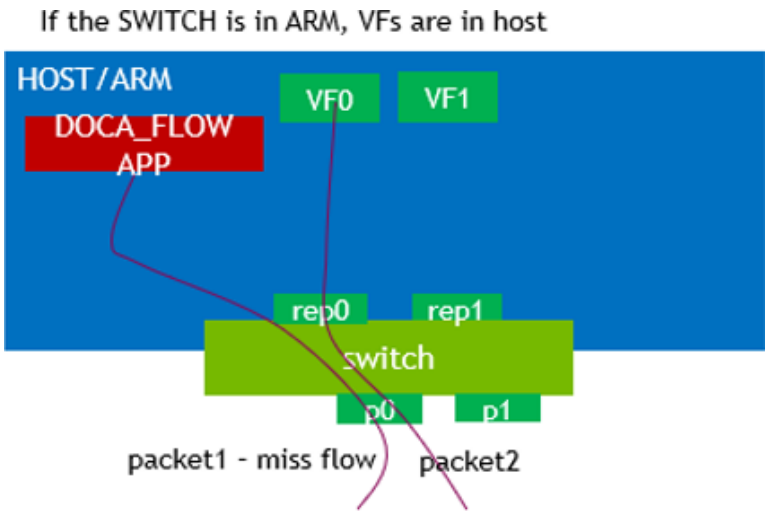
  

- `switch`

  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 which representor goes, and then sets the rule. Packets hit this rule and go to representor0.

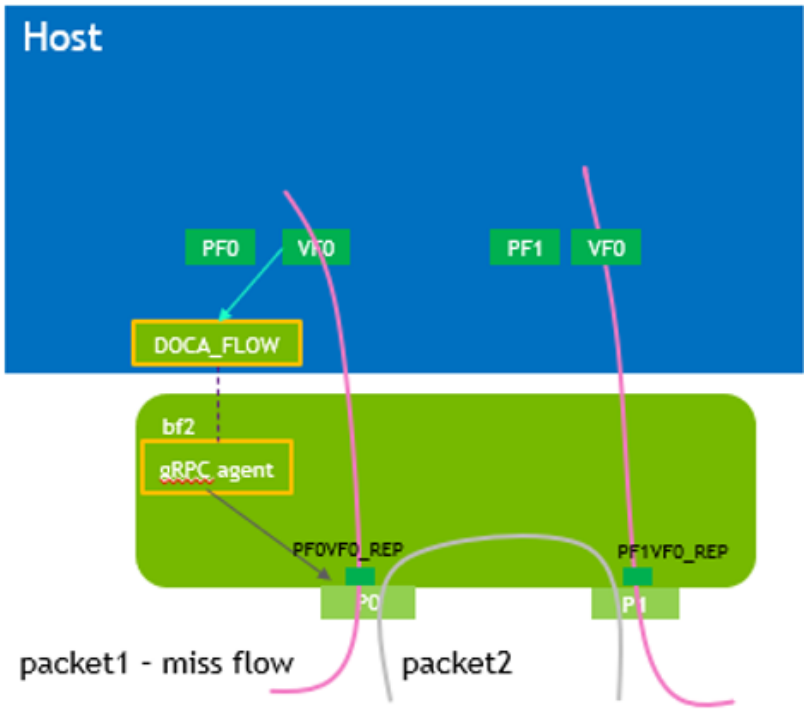
  If `dev` is provided in struct `doca_flow_port_cfg`, then pipe fwd cross-domain from ingress to egress is supported.



- `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.



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

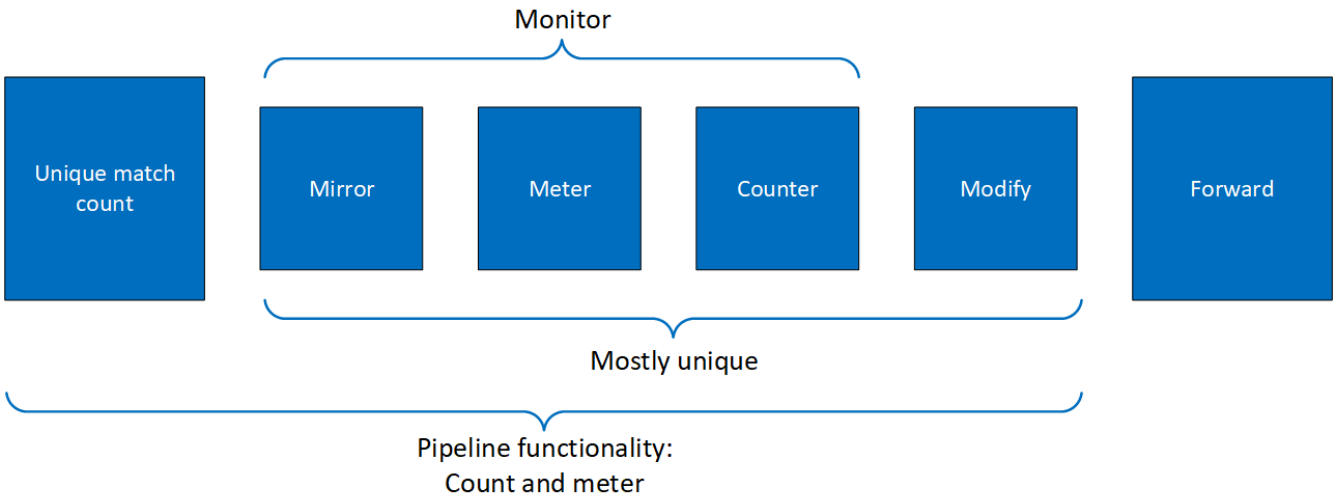
API. Mapping the DPDK port to the DOCA port is required to synchronize application ports with hardware ports.

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

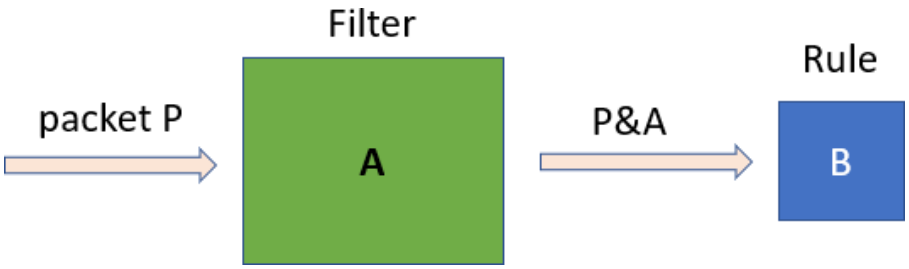
The following diagram illustrates a pipe structure.



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).

Matching

This section explains the concept of matching. Conceptually, the following logic is followed:



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 Match" and "Explicit Match".

Setting Pipe Match

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 – 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.

> **Note**
>
> 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.match_mask` pointer. 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:

> **Note**
>
> 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);
}
```

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.

> **Tip**
>
> 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);
```

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 use d 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`. T hat 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

Note

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
```

Setting Pipe Actions

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;
  ```

- Boolean fields – Boolean values, has_encap and decap are considered as constant values. It is not allowed to generate actions with `has_encap=true` and to then have an entry without an encap.

For example:

```
static void
create_decap_inner_modify_actions(struct doca_flow_actions
*actions)
{
        actions->decap = true;
        /* After decap, inner become outer. */
        actions->outer.ip4.dst_ip = 0xffffffff;
}
```

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 = {0};
    /* input configurations for actions_0 and actions_1 */
    actions_arr[0] = &actions_0;
    actions_arr[1] = &actions_1;
    pipe_cfg.attr.nb_actions = 2;
    pipe_cfg.actions = actions_arr;
}
```

Summary of Action Types

Pipe Creation				**Entry Creation**	**E**
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	F n
		`0`	`mask` != 0	N/A	A a
		`val` != 0 && `val` != 0xFF	`mask` != 0	N/A	A t
		`val` = 0xFF	`mask` = 0	N/A	A e
		`val` = 0xFF	`mask` != 0	Define `val` per entry	A r

Pipe Creation				Entry Creation	E
`DOCA_FLOW_ACTION_ADD` Add field value or from `src`	Define only the `dst` field and width	`val` != 0	N/A	N/A	A e
		`val` == 0	N/A	Define `val` per entry	A
	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	A fi e

Pipe Creation				Entry Creation	E
`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	C fi

Pipe Creation				Entry Creation	E
`DOCA_FLOW_ACTION_DECAP_ENCAP` Perform decap/encap for L2 or L3 tunnel	`is_l2 == false`	`decap == true`	N/A	N/A	T A p w h U `a` t t
		`has_encap == true`			T B o h f
	`is_l2 == true`	`decap == true`			T A a p `a` c t ir
		`has_encap == true`			T B a p

Summary of Fields

Field	Match	Modification	Add	Copy
meta.pkt_meta	x	x		x
meta.u32	x	x		x
Packet outer fields	x (field list)	x (field list)	TTL	Between meta[1]
Packet tunnel	x			To meta
Packet inner fields	x (field list)			To meta[1]

Note

[1] Copy from meta to IP is not supported.

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->flags |= DOCA_FLOW_MONITOR_AGING;
        monitor->aging_sec = 10;
}
```

Refer to Pipe Entry Aged Query for more information.

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 `port_id` is provided in struct `doca_flow_port_cfg`.

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_cfg->port_id;
```

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;
```

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.

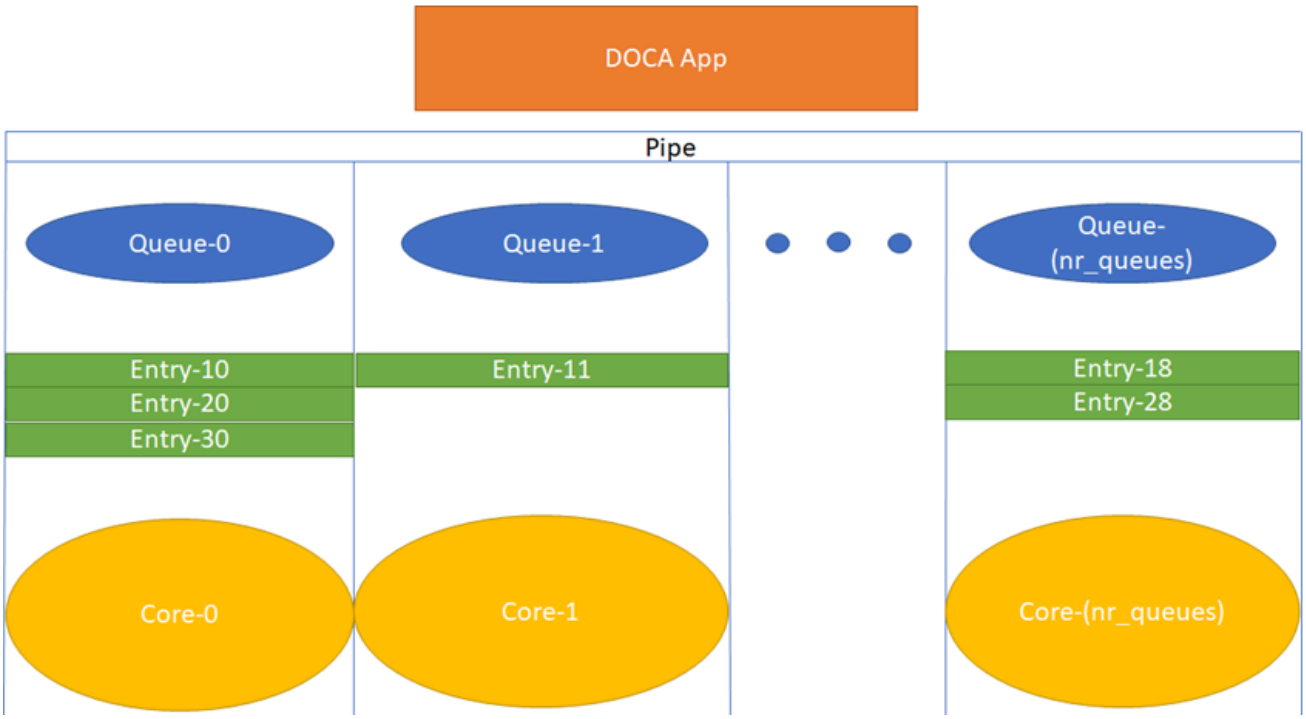
There is no priority between pipes or entries. The way that priority can be implemented is to match the highest priority first, and if a miss occurs, to jump to the next PIPE. There can be more than one PIPE on a root as long the pipes are not overlapping. If entries overlap, the priority is set according 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.

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.

Upon success, a handle is returned. If a failure occurs, a NULL value is returned, and an error message is filled. The application can keep this handle and call remove on the entry using its handle.

```
int doca_flow_pipe_rm_entry(uint16_t pipe_queue, void *usr_ctx,
struct doca_flow_pipe_entry *entry);
```

Pipe Entry Counting

By default, no counter is added. If defined in monitor, a unique counter is added per entry.

> **Note**
>
> 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.

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`.

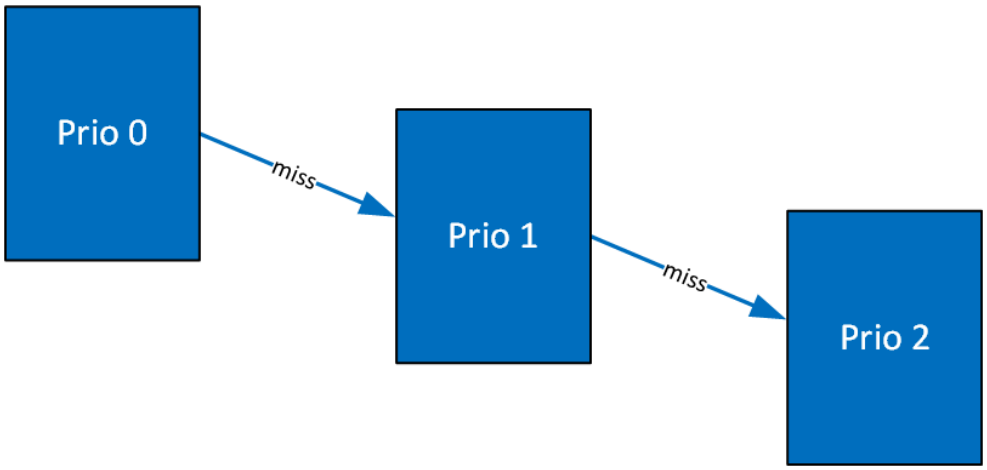
Users can also add two kinds of entries to the pipe, the first one with `Action_0` and the second with `Action_1`. This is done by assigning 0 in the `action_idx` field in `struct doca_flow_actions` when creating the first entry and 1 when creating the second one.

Miss Pipe and Control Pipe

> **Note**
>
> 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` is configured then if a packet does not match the specific pipe, steering should jump to `next_pipe` in `fwd_miss`.

Note

`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. When a user calls `doca_flow_create_control_pipe()`, the new control pipe is created without any configuration except for the port. Then the user can add different matches with different 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.

doca_flow_pipe_lpm

`doca_flow_pipe_lpm` uses longest prefix match (LPM) matching. LPM matching is limited to a single field of the `doca_flow_match` (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.

The actions and FWD of the DOCA Flow LPM pipe works the same as the basic DOCA Flow pipe.

> **Note**
>
> 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.

Note

An LPM pipe cannot be a root pipe. You must create a pipe as root and forward the packets to the LPM pipe.

Note

For monitoring, an LPM pipe only supports non-shared counters and does not support other capabilities of `doca_flow_monitor`.

doca_flow_pipe_acl

`doca_flow_pipe_acl` uses a ccess-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.

Note

An ACL pipe can be a root pipe.

Note

An ACL pipe can be in ingress and egress domain.

Note

An ACL pipe must be accessed on a single queue. Different ACL pipes may be accessed on different queues.

Note

Adding an entry to the ACL pipe after sending an entry with flag `DOCA_FLOW_NO_WAIT` is not supported.

Note

Removing an entry from an ACL pipe is not supported.

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.

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.

Note

The `nb_flows` in `doca_flow_pipe_attr` should be a power of 2.

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`, the member `mode_args` represents DOCA applications. If it is defined with `hws` (e.g., `"vnf,hws"`, `"switch,hws"`, `"remmote_vnf,hws"`) then hardware steering mode is enabled.

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 defined 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 define 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.

When `mode_args` contains the string `verify` (e.g., `"vnf,hws,verify"`), extra checks are asserted when adding an entry to a pipe using `doca_flow_add_entry`.

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.

To activate isolated mode, two configurations are required:

1. DOCA configuration: Update the string member `mode_args` (`struct doca_flow_cfg`) which represents the DOCA application mode and add "isolated" (separated by comma) to the other mode arguments. For example: `"vnf,hws,isolated"`, `"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 = ...
        ...
    },
    ...
};
```

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.

> **Info**
>
> For this release, pipe resizing it is only operational in Control Pipes.

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

> **Info**
>
> 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)

> **Info**
>
> 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

> **Note**
>
> 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 `doca_flow_init()`:

   ```
   .pipe_process_cb = <pipe-process-callback>
   ```

   This will inform on `OP_CONGESTION_REACHED` and `OP_RESIZED` operations when applicable.

2. Set the following pipe attributes on pipe creation:

   ```
   .is_resizable = true,
   .congestion_level_threshold = <CONGESTION_PERCENTAGE>,
   .user_ctx = <pipe-user-context>
   ```

3. Start adding entries:

```
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. 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.

7. Once all entries are transferred, an `OP_RESIZED` operation callback is received, at which point calling `doca_flow_entries_process()` can be stopped. Also, at this

point a new `OP_CONGESTION_REACHED` operation callback can be received again.

Hairpin Configuration

In switch mode, if `dev` is provided in struct `doca_flow_port_cfg`, 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

Teardown

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.

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.

> **⚠ Warning**
>
> During `doca_flow_pipe_destroy()` execution, the application must avoid adding/removing entries or checking for aged entries of any other pipes.

Port Teardown

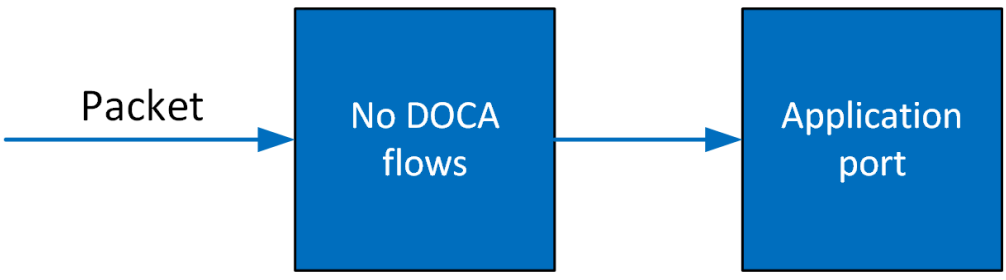
When the port is not used anymore, the user should call the port stop function, `doca_flow_port_stop()`. This stops the DOCA port, disables the traffic, destroys the port and frees all resources of the port.

Flow Teardown

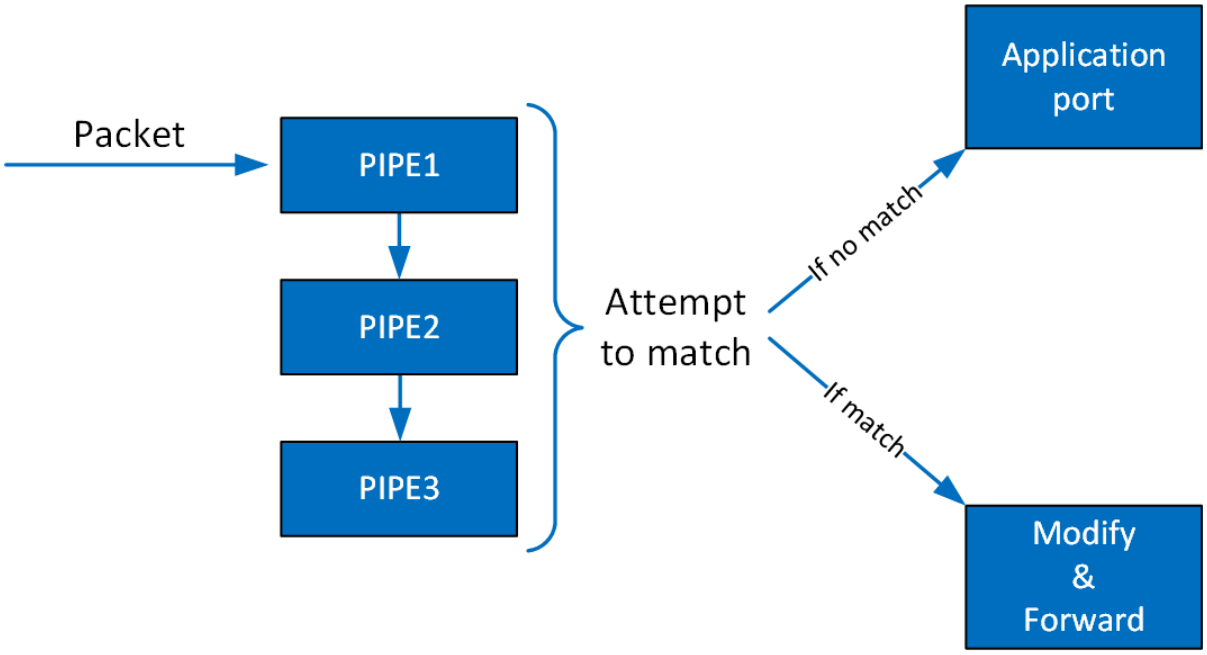
When the DOCA Flow is not used anymore, the user should call the flow destroy function, `doca_flow_destroy()`. This releases all the resources used by DOCA Flow.

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.

DOCA Flow gRPC

This chapter describes gRPC support for DOCA Flow. The DOCA Flow gRPC-based API allows users on the host to leverage the hardware offload capabilities of the BlueField DPU using gRPCs from the host itself.

DOCA Flow gRPC server implementation is based on gRPC's `async` API to maximize the performance offered to the gRPC client on the host. In addition, the gRPC support in the DOCA Flow library provides a client interface which gives the user the ability to send/receive messages to/from the client application in C.

This section is divided into the following parts:

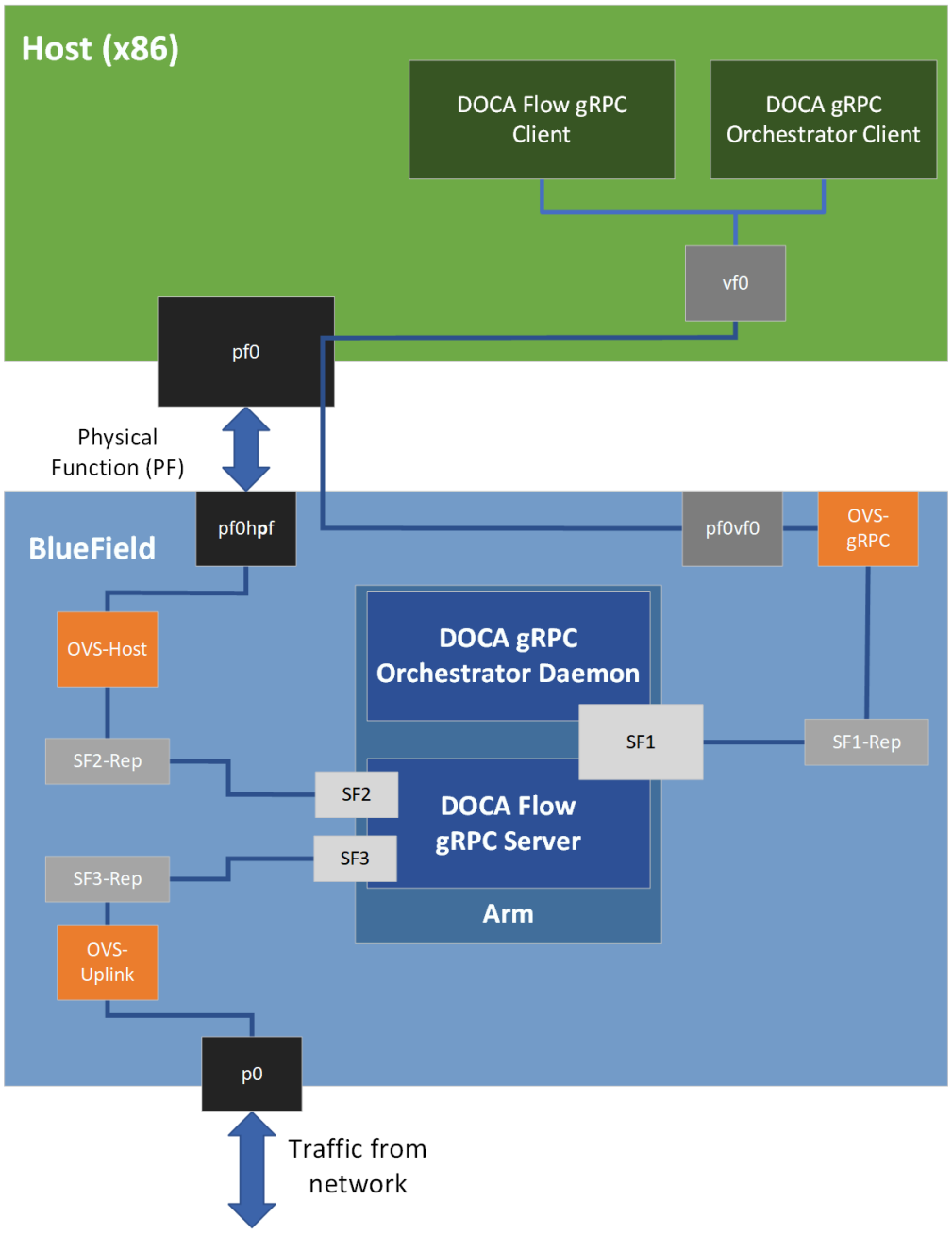
- proto-buff – this section details the messages defined in the proto-buff
- Client interface – this section details the API for communicating with the server

- Usage – this section explains how to use the client interface to develop your own client application based on DOCA Flow gRPC support

Refer to NVIDIA DOCA gRPC Infrastructure User Guide for more information about DOCA gRPC support.

> **Note**
>
> The pkg-config (`*.pc` file) for the DOCA Flow library is included in DOCA's regular definitions (i.e., `doca-grpc`).

The following figure illustrates the DOCA Flow gRPC server-client communication when running in VNF mode.



Running the DOCA Flow gRPC Server

Installation

Refer to the NVIDIA DOCA Installation Guide for Linux for instructions on installing BlueField-related software.

Prerequisites

1. Refer to NVIDIA DOCA gRPC Infrastructure User Guide for information on configuring the required steps for DOCA's gRPC infrastructure.

> **Info**
>
> Refer to the NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation or execution of the DOCA gRPC infrastructure.

2. The DOCA Flow gRPC server requires the user to allocate huge pages:

```
echo '2048' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

> **Note**
>
> 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. In such cases, the guiding principal is to allocate 4GB of RAM, and instead of allocating 2048 pages, one should allocate the matching amount (8 pages):
>
> ```
> sudo echo 8 > /proc/sys/vm/nr_hugepages
> ```

DOCA Flow gRPC Server Execution

Server usage instructions:

```
Usage: doca_flow_grpc [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=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                         Parse all command flags
from an input json file
  -g, --grpc-address ip_address[:port]    Set the IP address for
the grpc server

Program Flags:
  -hqs, --nb_hairpin_qs                     Set number of hairpin
queues
  -iso, --isolated_mode                     Set isolated mode
  -hbd, --hybrid_dev                        set hybrid device
configuration
  -ddev, --doca_dev                         set use of doca device
```

For additional information, refer to section "Command Line Flags".

The following is a CLI example deploying the gRPC server on the DPU using the gRPC orchestrator client on the host:

```
/opt/mellanox/doca/infrastructure/doca_grpc/orchestrator/doca_grpc.
192.168.103.2 create doca_flow_grpc -a
auxiliary:mlx5_core.sf.2,dv_flow_en=2 -a
auxiliary:mlx5_core.sf.3,dv_flow_en=2 -- -l 60 --nb_hairpin_qs 2
```

Command Line Flags

Flag Type	Short Flag	Long Flag/JSON Key	Description
General flags	`h`	`help`	Prints a help synopsis
	`v`	`version`	Prints 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 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
	`g`	`grpc-address`	Set the IP address for the gRPC server

Flag Type	Short Flag	Long Flag/JSON Key	Description
Program flags	`hqs`	`nb_hairpin_qs`	Set the number of hairpin queues
	`iso`	`isolated_mode`	Activate isolated mode
	`hbd`	`hybrid_dev`	Set hybrid device configuration
	`ddev`	`doca_dev`	Set use of DOCA Device

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

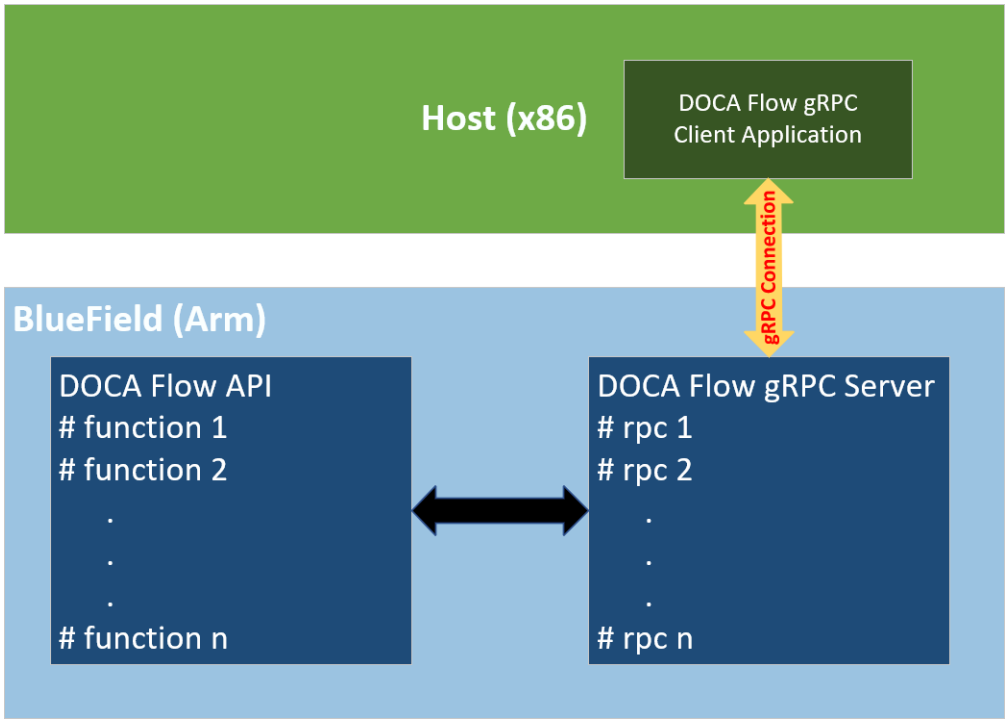
Proto-Buff

As with every gRPC proto-buff, DOCA Flow gRPC proto-buff defines the services it introduces, and the messages used for the communication between the client and the server. Each proto-buff DOCA Flow method:

- Represents exactly one function in DOCA Flow API
- Has its request message, depending on the type of the service
- Has the same response message (`DocaFlowResponse`)

In addition, DOCA Flow gRPC proto-buff defines several of messages that are used for defining request messages, the response message, or other messages.

Each message defined in the proto-buff represents either a struct or an enum defined by DOCA Flow API. The following figure illustrates how DOCA Flow gRPC server represents the DOCA Flow API.



The proto-buff path for DOCA Flow gRPC is `/opt/mellanox/doca/infrastructure/doca_grpc/doca_flow/doca_flow.proto`.

Response Message

All services have the same response message. `DocaFlowResponse` contains all types of results that the services may return to the client.

```
/** General DOCA Flow response message */
message DocaFlowResponse{
      bool success = 1;      /* True in case of success */
      DocaFlowError error = 2; /* Otherwise, this field contains
the error information */
      /* in case of success, one or more of the following may be
used */
        uint32 port_id = 3;
        uint64 pipe_id = 4;
        uint64 entry_id = 5;
        string port_pipes_dump = 6;
        DocaFlowQueryRes query_stats = 7;
        bytes priv_data = 8;
        DocaFlowHandleAgingRes handle_aging_res = 9;
            uint64 nb_entries_processed = 10;
            DocaFlowEntryStatus status = 11;
}
```

DocaFlowCfg

The `DocaFlowCfg` message represents the `doca_flow_cfg` struct.

DocaFlowPortCfg

The `DocaFlowPortCfg` message represents the `doca_flow_port_cfg` struct.

DocaFlowPipeCfg

The `DocaFlowPipeCfg` message represents the `doca_flow_pipe_cfg` struct.

DocaFlowMeta

The `DocaFlowMeta` message represents the `doca_flow_meta` struct.

The `DocaFlowMatch` message contains fields of types `DocaFlowIPAddress` and `DocaFlowTun`. These types are messages which are also defined in the `doca_flow.proto` file and represent `doca_flow_ip_address` and `doca_flow_tun` respectively.

DocaFlowMatch

The `DocaFlowMatch` message represents the `doca_flow_match` struct.

The `DocaFlowMatch` message contains fields of types `DocaFlowIPAddress` and `DocaFlowTun`. These types are messages which are also defined in the `doca_flow.proto` file and represent `doca_flow_ip_address` and `doca_flow_tun` respectively.

DocaFlowActions

The `DocaFlowActions` message represents the `doca_flow_actions` struct.

DocaFlowActionDesc

The `DocaFlowActionDesc` message represents the `doca_flow_action_desc` struct.

The `DocaFlowActionDesc` message contains fields of type `DocaFlowActionDescField` which are also defined in the `doca_flow.proto` file and represent `doca_flow_action_desc_field`.

DocaFlowMonitor

The `DocaFlowMonitor` message represents the `doca_flow_monitor` struct.

DocaFlowFwd

The `DocaFlowFwd` message represents the `doca_flow_fwd` struct.

DocaFlowQueryStats

The `DocaFlowQueryStats` message represents the `doca_flow_query` struct.

DocaFlowHandleAgingRes

The `DocaFlowHandleAgingRes` message contains all the parameters needed to save the result of an aging handler.

DocaFlowInit

DOCA Flow initialization gRPC:

```
rpc DocaFlowInit(DocaFlowCfg) returns (DocaFlowResponse);
```

If successful, the `success` field in the response message is set to `true`. Otherwise, the `error` field is populated with the error information.

DocaFlowPortStart

The service for starting the DOCA flow ports:

```
rpc DocaFlowPortStart(DocaFlowPortCfg) returns
(DocaFlowResponse);
```

If successful, the success field in the `DocaFlowResponse` is set to `true`. Otherwise, the `error` field is populated with the error information.

DocaFlowPortPair

The `DocaFlowPortPairRequest` message contains all the necessary information for port pairing:

```
message DocaFlowPortPairRequest {
        uint32 port_id = 1;                     /* port
identefier of doca flow port. */
        uint32 pair_port_id = 2;        /* port identefier to the
pair port. */
}
```

Once all the parameters are defined, a "port pair" service can be called:

The service for DOCA Flow port pair is as follows:

```
rpc DocaFlowPortPair(DocaFlowPortPairRequest) returns
(DocaFlowResponse);
```

If successful, the `success` field in the `DocaFlowResponse` is set to `true`. Otherwise, the `error` field is populated with the error information.

DocaFlowPipeCreate

The `DocaFlowPipeCreateRequest` message contains all the necessary information for pipe creation as the DOCA Flow API suggests:

```
message DocaFlowPipeCreateRequest {
        DocaFlowPipeCfg cfg = 1;           /* the pipe
configurations */
        DocaFlowFwd fwd = 2;               /* the pipe's
FORWARDING component  */
        DocaFlowFwd fwd_miss = 3;          /* The FORWARDING miss
component */
}
```

Once all the parameters are defined, a "create pipe" service can be called:

```
rpc DocaFlowPipeCreate (DocaFlowPipeCreateRequest) returns
(DocaFlowResponse);
```

If successful, the `success` field in `DocaFlowResponse` is set to `true`, and the `pipe_id` field is populated with the ID of the added entry. This ID should be given when adding entries to the pipe. Otherwise, the `error` field is filled accordingly.

DocaFlowPipeAddEntry

The `DocaFlowPipeAddEntryRequest` message contains all the necessary information for adding an entry to the pipe:

```
message DocaFlowPipeAddEntryRequest{
        uint32 pipe_queue = 2;              /* the pipe queue */
        uint64 pipe_id = 3;                 /* the pipe ID to add
the entry to */
        DocaFlowMatch match = 4;            /* matcher for the
entry */
        DocaFlowActions actions = 5;        /* actions for the
entry */
        DocaFlowMonitor monitor = 6;        /* monitor for the
entry */
        DocaFlowFwd fwd = 7;                /* The entry's
FORWARDING component */
        uint32 flags = 1;                                /* whether
the flow entry is pushed to HW immediately or not */
}
```

Once all the parameters are defined, an "add entry to pipe" service can be called:

```
rpc DocaFlowPipeAddEntry(DocaFlowPipeAddEntryRequest) returns
(DocaFlowResponse);
```

If successful, the success field in `DocaFlowResponse` is set to true, and the `entry_id` field is populated with the ID of the added entry. This ID should be given when adding entries to the pipe. Otherwise, the error field is filled accordingly.

DocaFlowPipeControlAddEntry

The `DocaFlowPipeControlAddEntryRequest` message contains the required arguments for adding entries to the control pipe:

```
message DocaFlowPipeControlAddEntryRequest {
        uint32 priority = 2;                /* he priority of the
added entry to the filter pipe */
        uint32 pipe_queue = 3;              /* the pipe queue */
        uint64 pipe_id = 4;                 /* the pipe ID to add
the entry to */
        DocaFlowMatch match = 5;            /* matcher for the
entry */
        DocaFlowMatch match_mask = 6;       /* matcher mask for
the entry */
        DocaFlowFwd fwd = 7;                /* The entry's
FORWARDING component */
}
```

Once all the parameters are defined, an "add entry to pipe" service can be called:

```
rpc
DocaFlowPipeControlAddEntry(DocaFlowPipeControlAddEntryRequest)
returns (DocaFlowResponse);
```

If successful, the success field in `DocaFlowResponse` is set to `true`, and the `entry_id` field is populated with the ID of the added entry. This ID should be given when adding entries to the pipe. Otherwise, the error field is filled accordingly.

DocaFlowPipeLpmAddEntry

The `DocaFlowPipeLpmAddEntryRequest` message contains the required arguments for adding entries to the LPM pipe:

```
message DocaFlowPipeLpmAddEntryRequest{
        uint32 pipe_queue = 1;              /* the pipe queue */
        uint64 pipe_id = 2;                 /* the pipe ID to add
the entry to */
        DocaFlowMatch match = 3;            /* matcher for the
entry */
        DocaFlowMatch match_mask = 4;       /* matcher mask for
the entry */
        DocaFlowActions actions = 5;        /* actions for the
entry */
        DocaFlowMonitor monitor = 6;        /* monitor for the
entry */
        DocaFlowFwd fwd = 7;                /* The entry's
FORWARDING component */
        uint32 flag = 8;                    /* whether the flow
entry will be pushed to HW immediately or not */
}
```

Once all the parameters are defined, an "add entry to LPM pipe" service can be called:

```
rpc DocaFlowPipeLpmAddEntry(DocaFlowPipeLpmAddEntryRequest)
returns (DocaFlowResponse);
```

If successful, the `success` field in `DocaFlowResponse` is set to `true`, and the `entry_id` field is populated with the ID of the added entry. This ID should be given when adding entries to the pipe. Otherwise, the `error` field is filled accordingly.

DocaFlowEntriesProcess

The `DocaFlowEntriesProcessRequest` message contains the required arguments for processing the entries in the queue.

```
message DocaFlowEntriesProcessRequest{
        uint32 port_id = 1;          /* the port ID of the
entries to process. */
        uint32 pipe_queue = 2;       /* the pipe queue of the
entries to process. */
        /* max time in micro seconds for the actual API to
process entries. */
        uint64 timeout = 3;
                /* An upper bound for the required number of
entries to process. */
        uint32 max_processed_entries = 4;
}
```

Once all the parameters are defined, the "entries process" service can be called:

```
rpc DocaFlowEntriesProcess(DocaFlowEntriesProcessRequest) returns
(DocaFlowResponse);
```

If successful, the success field in `DocaFlowResponse` is set to `true`, and the `nb_entries_processed` field is populated with the ID of the number of processed entries.

DocaFlowEntyGetStatus

The `DocaFlowEntryGetStatusRequest` message contains the required arguments for fetching the status of a given entry.

```
message DocaFlowEntryGetStatusRequest{
        /* the entry identifier of the requested entry's status.
*/
        uint64 entry_id = 1;
}
```

Once all the parameters are defined, the "entry get status" service can be called:

```
rpc DocaFlowEntriesProcess(DocaFlowEntriesProcessRequest) returns
(DocaFlowResponse);
```

If successful, the success field in `DocaFlowResponse` is set to true, and the status field is populated with the status of the requested entry. This field's type is `DocaFlowEntryStatus`, which is an enum defined in the proto-buff, and represents the enum `doca_flow_entry_status`, defined in the DOCA Flow header.

DocaFlowQuery

The `DocaFlowQueryRequest` message contains the required arguments for querying a given entry.

```
message DocaFlowQueryRequest{
        uint64 entry_id = 3;              /* the entry id. */
}
```

Once all the parameters are defined, the "query" service can be called:

```
rpc DocaFlowQuery(DocaFlowQueryRequest) returns
(DocaFlowResponse);
```

If successful, the success field in `DocaFlowResponse` is set to `true`, and the `query_stats` field is populated with the query result of the requested entry. This field's type is `DocaFlowQueryStats` which represents the `doca_flow_query` struct.

DocaFlowAgingHandle

The `DocaFlowAgingHandleRequest` message contains the required arguments for handling aging by DOCA Flow.

```
message DocaFlowAgingHandleRequest{
        uint32 port_id = 1;                /* the port id handle
aging to. */
        uint32 queue = 2;                  /* the queue
identifier */
        uint64 quota = 3;                  /* the max time quota
in micro seconds for this function to handle aging. */
}
```

Once all the parameters are defined, the "handle aging" service can be called:

```
rpc DocaFlowAgingHandle(DocaFlowAgingHandleRequest) returns
(DocaFlowResponse);
```

If successful, the `success` field in `DocaFlowResponse` is set to `true`, and the `handle_aging_res` is populated with the aging handler result. This field's type is `DocaFlowHandleAgingRes`.

DocaFlowSharedResourceCfg

The `DocaFlowSharedResourceCfgRequest` message contains the required arguments for configuring a shared resource by DOCA Flow.

```
message DocaFlowSharedResourceCfgRequest {
        DocaFlowSharedResourceType type = 1;                /*
Shared resource type */
        uint32 id = 2;
/* Shared resource id */
        SharedResourceCfg cfg= 3;
/* Shared resource configuration */
}
```

Once all the parameters are defined, the "config shared resource" service can be called:

```
rpc DocaFlowSharedResourceCfg(DocaFlowSharedResourceCfgRequest )
returns (DocaFlowResponse);
```

If successful, the `success` field in `DocaFlowResponse` is set to `true`.

DocaFlowSharedResourcesBind

The `DocaFlowSharedResourcesBindRequest` message contains the required arguments for configuring a shared resource by DOCA Flow.

```
message DocaFlowSharedResourcesBindRequest {
        DocaFlowSharedResourceType type = 1;                /*
Shared resource type */
        repeated uint32 resource_arr = 2;
/* Repeated shared resource IDs */
        /* id of allowed bindable object, use 0 to bind globally
*/
        oneof bindable_obj_id {
                uint64 port_id = 3;
/* Used if the bindable object is port */
                uint64 pipe_id = 4;
/* Used if the bindable object is pipe */
        }
}
```

Once all the parameters are defined, the "bind shared resources" service can be called:

```
rpc
DocaFlowSharedResourcesBind(DocaFlowSharedResourcesBindRequest )
returns (DocaFlowResponse);
```

If successful, the `success` field in `DocaFlowResponse` is set to `true`.

DocaFlowSharedResourcesQuery

The `DocaFlowSharedResourcesQueryRequest` message contains the required arguments for configuring a shared resource by DOCA Flow.

```
message DocaFlowSharedResourcesQueryRequest {
        DocaFlowSharedResourceType type = 1;    /* Shared object
type */
        repeated uint32 res_array = 2;          /*  Array of
shared objects IDs to query */
}
```

Once all the parameters are defined, the "query shared resources" service can be called:

```
rpc
DocaFlowSharedResourcesBind(DocaFlowSharedResourcesBindRequest )
returns (DocaFlowResponse);
```

If successful, the `success` field in `DocaFlowResponse` is set to `true` and the `query_result` is populated with the query shared resources result. This field's type is `DocaFlowQueryStats` which represents the `doca_flow_query` struct.

DOCA Flow gRPC Client API

This section describes the recommended way for C developers to utilize gRPC support for DOCA Flow API. Refer to the DOCA Flow gRPC API in NVIDIA DOCA Library APIs for the library API reference.

The following sections provide additional details about the library API.

The DOCA installation includes `libdoca_flow_grpc` which is a library that provides a C API wrapper to the C++ gRPC, while mimicking the regular DOCA Flow API, for ease of use, and allowing smooth transition to the Arm.

This library API is exposed in `doca_flow_grpc_client.h` and is essentially the same as `doca_flow.h`, with the notation differences detailed in the following subsections. In general, the client interface API usage is almost identical to the regular API (i.e., DOCA

Flow API). The arguments of each function in DOCA Flow API, are almost identical to the arguments of each function defined in the client API, except that each pointer is replaced with an ID representing the pointer.

For example, when creating a pipe or adding an entry, the original API returns a pointer to the created pipe or the added entry. However, when adding an entry or creating a pipe using the client interface, an ID representing the added entry or the created pipe is returned to the client application instead of the pointer.

doca_flow_grpc_response

`doca_flow_grpc_response` is a general response struct that holds information regarding the function result. Each API returns this struct. If an error occurs, the `error` field is populated with the error's information, and the `success` field is set to `false`. Otherwise, the `success` field is set to true and one of the other fields may hold a return value depending on the called function.

For example, when calling `doca_flow_grpc_create_pipe()` the `pipe_id` field is populated with the ID of the created pipe in case of success.

```
struct doca_flow_grpc_response {
      bool success;
      struct doca_flow_error error;
      uint64_t pipe_id;
      uint64_t entry_id;
      uint32_t aging_res;
      uint64_t nb_entries_processed;
      enum doca_flow_entry_status entry_status;
};
```

- `success` – In case of success, the value should be true.
- `error` – In case of error, this struct should contain the error information.
- `pipe_id` – Pipe ID of the created pipe.
- `entry_id` – Entry ID of the created entry.

- `aging_res` – Return value from handle aging.
- `nb_entries_processed` – Return value from entries process.
- `entry_status` – Return value from entry get status.

doca_flow_grpc_pipe_cfg

`doca_flow_grpc_pipe_cfg` is a pipeline configuration wrapper.

```
struct doca_flow_grpc_pipe_cfg {
        struct doca_flow_pipe_cfg cfg;
        uint16_t port_id;
};
```

- `cfg` – Pipe configuration containing the user-defined template for the packet process.
- `port_id` – Port ID for the pipeline.

doca_flow_grpc_fwd

`doca_flow_grpc_fwd` is a forwarding configuration wrapper.

```
struct doca_flow_grpc_fwd {
        struct doca_flow_fwd fwd;
        uint64_t next_pipe_id;
};
```

- `fwd` – Forward configuration which directs where the packet goes next.

- `next_pipe_id` – When using DOCA_FLOW_FWD_PIPE, this field contains the next pipe's ID.

doca_flow_grpc_client_create

This function initializes a channel to DOCA Flow gRPC server.

This must be invoked first before any other function in this API. This is a one-time call.

```
void doca_flow_grpc_client_create(char *grpc_address);
```

- `grpc_address [in]` – String representing the server IP.

DOCA Flow gRPC Usage

A DOCA flow gRPC based server is implemented using the `async` API of gRPC. This is because the `async` API gives the server the ability to expose DOCA flow's concurrency support. Therefore, it is very important to use the client interface API for communicating with the DOCA Flow gRPC server because it hides all gRPC-related details from the users, which eases the use of the server, and exposes to the client applications the efficiency of DOCA Flow, in terms of flow insertion rates.

The following phases demonstrate a basic flow of client applications:

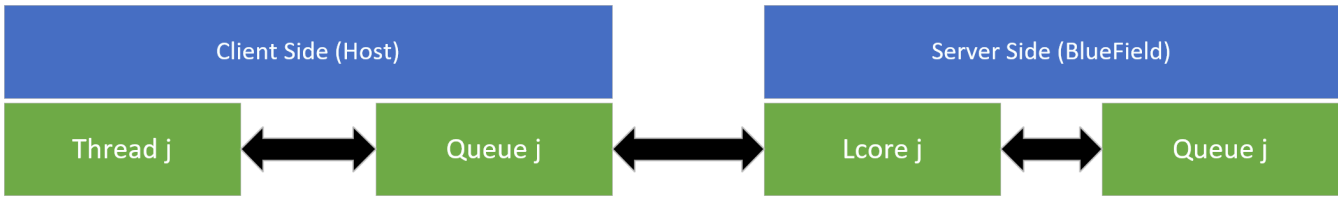
- Init Phase – client interface and environment initializations
- Flow life cycle – this phase is the same phase described in chapter Flow Life Cycle

It is important to emphasize that the number of threads for adding entries should be the same as the number of queues used when starting the server and initializing the environment (DPDK) and DOCA Flow API. This is to prevent bottlenecks on the server side.

If a client application starts the server on BlueField with N cores (through EAL arguments), this means that environment and DOCA Flow initialization should be done with N queues. As a result, the server launches N lcores, each one responsible for exactly one queue that

is accessed only by it. Therefore, the client application should launch N threads as well, each being responsible for adding entries to a specific queue which is accessed by it only as well.

The following illustration demonstrates the relation between thread "j" on the client side and lcore "j" on the server side:



DOCA Flow Samples

This section provides DOCA Flow sample implementation on top of the BlueField DPU.

Sample Prerequisites

The DOCA Flow samples are based on DPDK libraries. Therefore, the user is required to provide DPDK flags, and allocate huge pages.

```
sudo echo 2048 > /sys/kernel/mm/hugepages/hugepages-
2048kB/nr_hugepages
```

Note

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. In such cases, the guiding principal is to allocate 4GB of RAM, and instead of allocating 2048 pages, one should allocate the matching amount (8 pages):

```
sudo echo 8 > /proc/sys/vm/nr_hugepages
```

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
```

> **Note**
>
> 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=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
```

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 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 samples with `switch,hws` mode:

```
/tmp/build/doca_<sample_name> -a
03:00.0,representor=sf[2-3],dv_flow_en=2 -- -l 60
```

Note

When running on the DPU with `switch,hws` mode , it is not necessary to configure the OVS.

Note

When running on the DPU using the command above, sub-functions must be enabled according to the NVIDIA BlueField DPU Scalable Function User Guide.

Note

When running on the host, virtual functions must be used according to the instructions in the NVIDIA DOCA Virtual Functions User Guide.

Samples

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:
    1. Building an ACL pipe that matches changeable:
        1. Source IPv4 address
        2. Destination IPv4 address
        3. Source port
        4. Destination port
    2. Adding four example 5-tuple entries:
        1. 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)

2. 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)
3. 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)
4. 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)

3. 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`.

    1. Ingress domain: ACL is created as root pipe
    2. 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`

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:

    1. Building a pipe with changeable 5-tuple match and forward port action.

    2. 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`
- `/opt/mellanox/doca/samples/doca_flow/flow_aging/flow_aging_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_aging/meson.build`

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:

    1. Building VXLAN pipe with match on VNI field, decap action, action descriptor for decap, and forwarding the matched packets to the second port.

    2. 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.

    3. 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.

4. 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 6635, forward to MPLS pipe
    - If tunnel type and L4 type is GRE, forward to GRE pipe

Note

When any tunnel is decapped, it is user responsibility to identify if it is an L2 or L3 tunnel within the action descriptor. 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, DOCA Flow automatically encaps the inner packet with an empty ETH header. To make the ETH header valid, the user must modify the L2 source/destination MAC addresses and VLAN may optionally be modified. For example:

```
actions.decap = true;
/* append eth header after decap GRE tunnel */
SET_MAC_ADDR(actions.outer.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.outer.eth.dst_mac,
dst_mac[0], dst_mac[1], dst_mac[2], dst_mac[3],
dst_mac[4], dst_mac[5]);
actions.outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;

/* identify that the tunnel is of L3 type */
desc_array[0].type = DOCA_FLOW_ACTION_DECAP_ENCAP;
desc_array[0].decap_encap.is_l2 = false;
```

For a VXLAN tunnel, since VXLAN is a L2 tunnel, the user must indicate that to the DOCA Flow using an action descriptor so it does not execute any automatic actions:

```
actions.decap = true;

/* identify that the tunnel is of L2 type */
desc_array[0].type = DOCA_FLOW_ACTION_DECAP_ENCAP;
desc_array[0].decap_encap.is_l2 = true;
```

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_control_pipe/flow_contr`
- `/opt/mellanox/doca/samples/doca_flow/flow_control_pipe/flow_contr`
- `/opt/mellanox/doca/samples/doca_flow/flow_control_pipe/meson.buil`

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 `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 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_`
- `/opt/mellanox/doca/samples/doca_flow/flow_copy_to_meta/flow_copy_`
- `/opt/mellanox/doca/samples/doca_flow/flow_copy_to_meta/meson.buil`

Flow Drop

This sample illustrates how to build a pipe with 5-tuple match, forward action drop, and forward miss action to 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:

    1. Building a hairpin pipe with an entry that matches all traffic and forwarding traffic to the second port.

    2. 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.

    3. Adding example 5-tuple entry to the drop pipe with counter as monitor for 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_s`
- `/opt/mellanox/doca/samples/doca_flow/flow_drop/flow_drop_sample_m`
- `/opt/mellanox/doca/samples/doca_flow/flow_drop/meson.build`

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:
    1. Building ingress pipe with changeable 5-tuple match, copy to `pkt_meta` action, and forward port action.
    2. 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.
    3. Adding example 5-tuple and encapsulation values entries to the pipes.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_geneve_encap/flow_genev`
- `/opt/mellanox/doca/samples/doca_flow/flow_geneve_encap/flow_genev`

- `/opt/mellanox/doca/samples/doca_flow/flow_geneve_encap/meson.buil`

Flow GENEVE Options

This sample illustrates how to prepare a GENEVE options parser, match on configured options, and decap GENEVE tunnel.

> **Note**
>
> 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:
    1. Building GENEVE options parser, same input for all ports.
    2. Building match pipe with GENEVE VNI and options match and forwards decap pipe.
    3. Building decap pipe with more GENEVE options match, and 2 different decapsulation actions:
        - L2 decap
        - L3 decap with changeable mac addresses
    4. 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/mellanox/doca/samples/doca_flow/flow_geneve_opt/flow_geneve_`
- `/opt/mellanox/doca/samples/doca_flow/flow_geneve_opt/meson.build`

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:
    1. Building a pipe with changeable 5-tuple match and forwarding port action.
    2. Adding example 5-tuple entry to the pipe.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_hairpin_vnf/flow_hairpi`
- `/opt/mellanox/doca/samples/doca_flow/flow_hairpin_vnf/flow_hairpi`
- `/opt/mellanox/doca/samples/doca_flow/flow_hairpin_vnf/meson.build`

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()`.

> **Note**
>
> The test requires one PF with two representors (either VFs or SFs). `fdb_def_rule_en=0,vport_match=1` should be added to the DPDK devargs.

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:
    1. Building ingress and egress pipes with changeable 5-tuple match and forwarding port action.
    2. Adding example 5-tuple entry to the pipe.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_switch_to_wire/flow_swi`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_to_wire/flow_swi`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_to_wire/meson.bu`

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:
    1. Building a hash pipe while indicating which fields to use to calculate the hash in the `struct match_mask`.
    2. 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_pip`
- `/opt/mellanox/doca/samples/doca_flow/flow_hash_pipe/flow_hash_pip`
- `/opt/mellanox/doca/samples/doca_flow/flow_hash_pipe/meson.build`

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:

    1. 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.

    2. 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.

    3. 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).

    4. Adding one entry to the `RSS_TCP_IP` pipe, that matches a packet with a specific source and destination port and IPv4 addresses.

    5. 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.

    6. 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).

    7. 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_`
- `/opt/mellanox/doca/samples/doca_flow/flow_loopback/flow_loopback_`
- `/opt/mellanox/doca/samples/doca_flow/flow_loopback/meson.build`

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`

Flow Modify Header

This sample illustrates how to use DOCA Flow actions to decrease TTL by 1 and modify the destination MAC address.

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 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.

    2. Adding an entry with an example destination IP and `mod_dst_mac` value.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_modify_header/flow_modi`
- `/opt/mellanox/doca/samples/doca_flow/flow_modify_header/flow_modi`
- `/opt/mellanox/doca/samples/doca_flow/flow_modify_header/meson.bui`

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:

    1. Building a pipe with monitor meter flag and changeable 5-tuple match. The pipe forwards the matched packets to the second port.

    2. Adding an entry with an example CIR and CBS values.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_monitor_meter/flow_moni`
- `/opt/mellanox/doca/samples/doca_flow/flow_monitor_meter/flow_moni`

- `/opt/mellanox/doca/samples/doca_flow/flow_monitor_meter/meson.bui`

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:
    1. 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
    2. Adding two entries to the pipe with different source IP match:
        1. The first entry with an example modify source MAC address.
        2. The second with a modify source IP address.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_multi_actions/flow_mult`
- `/opt/mellanox/doca/samples/doca_flow/flow_multi_actions/flow_mult`
- `/opt/mellanox/doca/samples/doca_flow/flow_multi_actions/meson.bui`

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:
    1. Building a pipe with changeable source IP match and sending NULL in the forward.
    2. 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_fw`
- `/opt/mellanox/doca/samples/doca_flow/flow_multi_fwd/flow_multi_fw`
- `/opt/mellanox/doca/samples/doca_flow/flow_multi_fwd/meson.build`

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:

1. Building a root pipe with changeable 5-tuple match and forwarding to an ordered list pipe with a changeable index.

2. Adding two entries to the pipe with an example value sent to a different index in the ordered list pipe.

3. 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_order`
- `/opt/mellanox/doca/samples/doca_flow/flow_ordered_list/flow_order`
- `/opt/mellanox/doca/samples/doca_flow/flow_ordered_list/meson.buil`

Flow Random

This sample shows how to use `match.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:

    1. Building a root pipe with changeable 5-tuple match and forwarding to specific use-case pipe according to changeable source IP address.

2. Adding two entries to the pipe with an example value sent to a different index in the ordered list pipe.

3. Building a "sampling" pipe with a single entry and preparing the entry to sample 12.5% of traffic.

4. 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_samp`
- `/opt/mellanox/doca/samples/doca_flow/flow_random/flow_random_main`
- `/opt/mellanox/doca/samples/doca_flow/flow_random/meson.build`

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:

    1. Building a pipe with a changeable 5-tuple match, forwarding to RSS queue with index 0, and setting changeable packet meta data.

    2. 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_`
- `/opt/mellanox/doca/samples/doca_flow/flow_rss_meta/flow_rss_meta_`
- `/opt/mellanox/doca/samples/doca_flow/flow_rss_meta/meson.build`

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_`
- `/opt/mellanox/doca/samples/doca_flow/flow_set_meta/flow_set_meta_`
- `/opt/mellanox/doca/samples/doca_flow/flow_set_meta/meson.build`

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:
    1. Binding the shared counter to the port.
    2. Building a pipe with changeable 5-tuple match with UDP protocol, changeable shared counter ID and forwarding the matched packets to the second port.
    3. Adding an entry with an example 5-tuple match and shared counter with ID= `port_id`.
    4. Building a pipe with changeable 5-tuple match with TCP protocol, changeable shared counter ID and forwarding the matched packets to the second port.
    5. Adding an entry with an example 5-tuple match and shared counter with ID= `port_id`.
    6. 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_sha`
- `/opt/mellanox/doca/samples/doca_flow/flow_shared_counter/flow_sha`
- `/opt/mellanox/doca/samples/doca_flow/flow_shared_counter/meson.bu`

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:
    1. Config a shared meter with specific cir and cbs values.
    2. Binding the shared meter to the port.
    3. 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.
    4. Adding an entry with an example 5-tuple match and shared meter with ID= `port_id`.
    5. 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.
    6. Adding an entry with an example 5-tuple match and shared meter with ID= `port_id`.
    7. 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_share`
- `/opt/mellanox/doca/samples/doca_flow/flow_shared_meter/flow_share`

- `/opt/mellanox/doca/samples/doca_flow/flow_shared_meter/meson.buil`

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()`.

> **Note**
>
> 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:
    1. Building a basic pipe while indicating which fields to match on using `struct doca_flow_match match`.
    2. 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_samp`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch/flow_switch_main`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch/meson.build`

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.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_switch_single/flow_swit`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_single/flow_swit`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_single/meson.bui`

Flow VXLAN 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 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 5-tuple match, encap action, and forward port action.

2. 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_`
- `/opt/mellanox/doca/samples/doca_flow/flow_vxlan_encap/flow_vxlan_`
- `/opt/mellanox/doca/samples/doca_flow/flow_vxlan_encap/meson.build`

Flow gRPC Counter

This sample shows how to use DOCA Flow gRPC library to create a pipe and entry with a counter and to query the entry stats.

The sample logic includes:

1. Creating gRPC environment.

2. Initializing DOCA Flow.

3. Starting two DOCA Flow ports.

4. On each port:

    1. Building a pipe with changeable 5-tuple match.

    2. Adding example 5-tuple and monitoring with counter flag.

    3. Waiting 5 seconds and querying the entries (total bytes and total packets).

References:

- `/opt/mellanox/doca/samples/doca_flow/grpc_flow_counter/grpc_flow_`
- `/opt/mellanox/doca/samples/doca_flow/grpc_flow_counter/grpc_flow_`

- `/opt/mellanox/doca/samples/doca_flow/grpc_flow_counter/meson.buil`

Flow Shared Mirror

This sample shows how to use the DOCA Flow shared mirror.

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. Configuring a shared mirror with a clone destination hairpin to the second port.
    2. Binding the shared mirror to the port.
    3. 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.
    4. Adding an entry with an example 5-tuple match and shared mirror with ID= `port_id+1`.
    5. 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.
    6. Adding an entry with an example 5-tuple match and shared mirror with ID= `port_id+1`.
    7. 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
    8. 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_shar`
- `/opt/mellanox/doca/samples/doca_flow/flow_shared_mirror/flow_shar`
- `/opt/mellanox/doca/samples/doca_flow/flow_shared_mirror/meson.bui`

Flow Pipe Resize

This sample shows how the DOCA Flow pipe resize feature behaves as pipe size increases.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws,cpds"` in the `doca_flow_cfg` struct.

   > **Info**
   >
   > By default, a control pipe's internal tables have a default size of 64 entries. Using the CPDS (control pipe dynamic size) mode, each table's initial size matches the control pipe size.

2. Starting a PF port with two representors of subfunction (SF) or virtual functions (VF). For example:

   ```
   ./doca_flow_pipe_resize -a 03:00.0,representor=sf[0-
   1],dv_flow_en=2 -- -l 60
   ```

3. Starting with a control 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:

   1. Receiving a `CONGESTION_REACHED` callback whenever the number of current entries exceeds a threshold level of 80%.

2. 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.

3. 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);
```

4. Start calling `doca_flow_entries_process()` in a loop on each thread ID to trigger the entry relocations.

> **Info**
>
> The loop should continue as long as the resize process was not ended.

5. 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)
```

6. Receiving a `PIPE_RESIZED` callback upon completion of the resize process. At this point, calling `doca_flow_entries_process()` should stop.

> **Info**
>
> The resize cycles described above repeats five times increasing the pipe sizes in these steps: 10 -> 16 -> 32 -> 64 -> 128.

> **Info**
>
> The pipe control entries define a match on increasing destination IP address. The fwd action send packet to the other port.

7. 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_r`
- `/opt/mellanox/doca/samples/doca_flow/flow_pipe_resize/flow_pipe_r`
- `/opt/mellanox/doca/samples/doca_flow/flow_pipe_resize/meson.build`

Field String Support Appendix

Supported Field String

String Field	Path in Match Structure	Set	Add	Dst Copy	Src Copy
meta.data	meta.pkt_meta meta.u32[i]	✔	✔	✔	✔

String Field	Path in Match Structure	Set	Add	Dst Copy	Src Copy
meta.mark	meta.mark	☐	☐	☐	☐
meta.flags	flags	☐	☐	☐	☐
parser_meta.hash	None. See section "Copy Hash Result" for details.	☐	☐	☐	✔
parser_meta.port	parser_meta.port_meta	☐	☐	☐	☐
parser_meta.ipsec_syndrome	parser_meta.ipsec_syndrome	☐	☐	☐	☐
parser_meta.random	parser_meta.random	☐	☐	☐	☐
parser_meta.meter_color	parser_meta.meter_color	☐	☐	☐	☐
parser_meta.outer_l2_type	parser_meta.outer_l2_type	☐	☐	☐	☐
parser_meta.outer_l3_type	parser_meta.outer_l3_type	☐	☐	☐	☐
parser_meta.outer_l4_type	parser_meta.outer_l4_type	☐	☐	☐	☐
parser_meta.inner_l2_type	parser_meta.inner_l2_type	☐	☐	☐	☐
parser_meta.inner_l3_type	parser_meta.inner_l3_type	☐	☐	☐	☐
parser_meta.inner_l4_type	parser_meta.inner_l4_type	☐	☐	☐	☐
parser_meta.outer_ip_fragmented	parser_meta.outer_ip_fragmented	☐	☐	☐	☐
parser_meta.inner_ip_fragmented	parser_meta.inner_ip_fragmented	☐	☐	☐	☐
parser_meta.outer_l3_ok	parser_meta.outer_l3_ok	☐	☐	☐	☐
parser_meta.outer_l4_ok	parser_meta.outer_l4_ok	☐	☐	☐	☐
parser_meta.inner_l3_ok	parser_meta.inner_l3_ok	☐	☐	☐	☐
parser_meta.inner_l4_ok	parser_meta.inner_l4_ok	☐	☐	☐	☐
outer.eth.dst_mac	outer.eth.dst_mac	✔	☐	✔	✔
outer.eth.src_mac	outer.eth.src_mac	✔	☐	✔	✔
outer.eth.type	outer.eth.type	✔	☐	✔	✔
outer.eth_vlan0.tci	outer.eth_vlan[0].tci	✔	☐	✔	✔
outer.eth_vlan1.tci	outer.eth_vlan[1].tci	☐	☐	☐	☐
outer.ipv4.src_ip	outer.ip4.src_ip	✔	☐	✔	✔
outer.ipv4.dst_ip	outer.ip4.dst_ip	✔	☐	✔	✔

String Field	Path in Match Structure	Set	Add	Dst Copy	Src Copy
outer.ipv4.dscp_ecn	outer.ip4.dscp_ecn	✔	☐	✔	✔
outer.ipv4.next_proto	outer.ip4.next_proto	✔	☐	✔	✔
outer.ipv4.ttl	outer.ip4.ttl	✔	✔	✔	✔
outer.ipv4.version_ihl	outer.ip4.version_ihl	☐	☐	☐	☐
outer.ipv6.src_ip	outer.ip6.src_ip	✔	☐	✔	✔
outer.ipv6.dst_ip	outer.ip6.dst_ip	✔	☐	✔	✔
outer.ipv6.dscp_ecn	outer.ip6.dscp_ecn	✔	☐	✔	✔
outer.ipv6.next_proto	outer.ip6.next_proto	✔	☐	✔	✔
outer.ipv6.hop_limit	outer.ip6.hop_limit	✔	✔	✔	✔
outer.udp.src_port	outer.udp.l4_port.src_port	✔	☐	✔	✔
outer.udp.dst_port	outer.udp.l4_port.dst_port	✔	☐	✔	✔
outer.tcp.src_port	outer.tcp.l4_port.src_port	✔	☐	✔	✔
outer.tcp.dst_port	outer.tcp.l4_port.dst_port	✔	☐	✔	✔
outer.tcp.flags	outer.tcp.flags	☐	☐	☐	☐
outer.icmp4.type	outer.icmp.type	☐	☐	☐	☐
outer.icmp4.code	outer.icmp.code	☐	☐	☐	☐
outer.icmp4.ident	outer.icmp.ident	☐	☐	☐	☐
outer.icmp6.type	outer.icmp.type	☐	☐	☐	☐
outer.icmp6.code	outer.icmp.code	☐	☐	☐	☐
tunnel.gre.protocol	tun.protocol	☐	☐	☐	☐
tunnel.gre_key.value	tun.gre_key	☐	☐	☐	☐
tunnel.vxlan.vni	tun.vxlan_tun_id	✔	☐	✔	✔
tunnel.gtp.teid	tun.gtp_teid	✔	☐	✔	✔
tunnel.esp.spi	tun.esp_spi	☐	☐	☐	☐
tunnel.esp.sn	tun.esp_sn	☐	☐	☐	☐
tunnel.mpls[0].label	tun.mpls[0].label	☐	☐	☐	✔
tunnel.mpls[1].label	tun.mpls[1].label	☐	☐	☐	✔

String Field	Path in Match Structure	Set	Add	Dst Copy	Src Copy
tunnel.mpls[2].label	tun.mpls[2].label	☐	☐	☐	✔
tunnel.mpls[3].label	tun.mpls[3].label	☐	☐	☐	✔
tunnel.mpls[4].label	tun.mpls[4].label	☐	☐	☐	✔
tunnel.geneve.ver_opt_len	tun.geneve.ver_opt_len	☐	☐	☐	☐
tunnel.geneve.o_c	tun.geneve.o_c	☐	☐	☐	☐
tunnel.geneve.next_proto	tun.geneve.next_proto	☐	☐	☐	☐
tunnel.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	☐	☐	☐	☐
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.ipv6.src_ip	inner.ip6.src_ip	☐	☐	☐	☐
inner.ipv6.dst_ip	inner.ip6.dst_ip	☐	☐	☐	☐
inner.ipv6.dscp_ecn	inner.ip6.dscp_ecn	☐	☐	☐	☐
inner.ipv6.next_proto	inner.ip6.next_proto	☐	☐	☐	☐
inner.ipv6.hop_limit	inner.ip6.hop_limit	☐	☐	☐	☐
inner.udp.src_port	inner.udp.l4_port.src_port	☐	☐	☐	☐

String Field	Path in Match Structure	Set	Add	Dst Copy	Src Copy
inner.udp.dst_port	inner.udp.l4_port.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	□	□	□	□
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	□	□	□	□

Supported Non-field String

Users can modify fields which are not included in `doca_flow_match` structure.

Copy Hash Result

Users can copy the the matcher hash calculation into other fields using the `"parser_meta.hash"` string.

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`.

DOCA Flow Connection Tracking

This guide provides an overview and configuration instructions for DOCA Flow CT API.

Introduction

DOCA Flow Connection-Tracking AKA, ConnTrack (CT) is 5-tuple table usually supporting the following :

- Track 5-tuple sessions (or 6-tuple when having a zone).
- Zone based - virtual tables.
- Aging, remove idle connections.
- Sets Meta-Data for a connection.
- Bi-directional packet handling.
- High connections per second (CPS) rate.

The CT module makes it simple and efficient to track connections by leveraging hardware resources. The module supports both autonomous and managed mode.

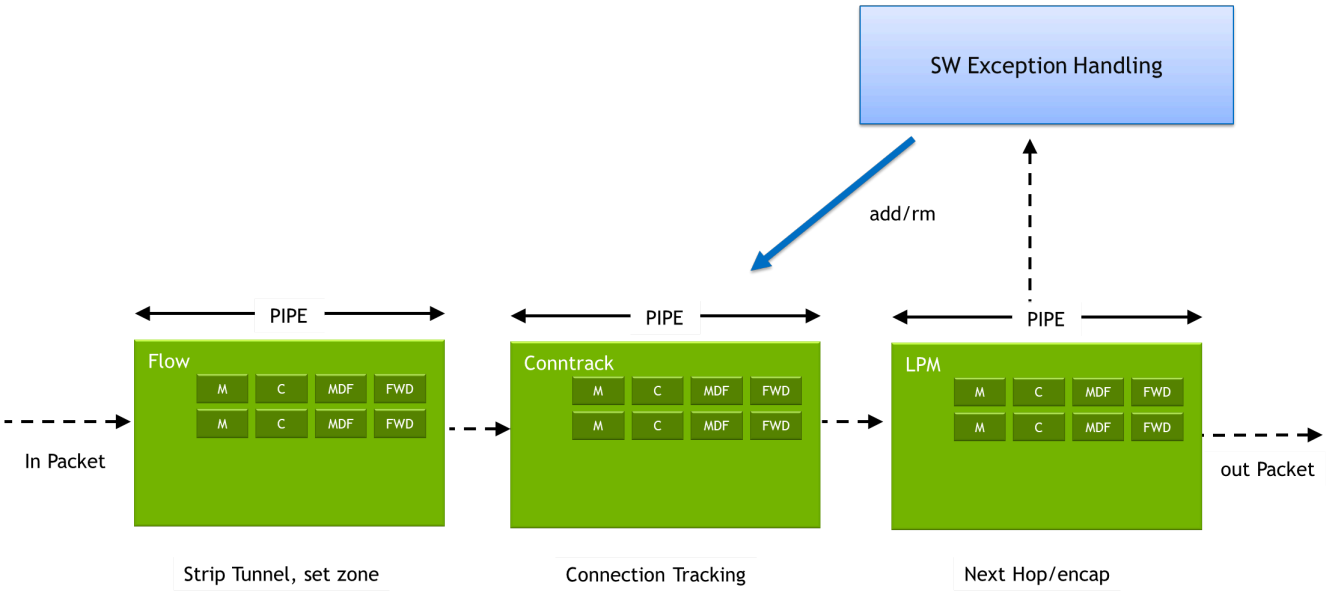
Architecture

DOCA Flow CT pipe handles non-encapsulated TCP and UDP packets. CT pipe only support forward next pipe or miss next pipe, all packets match known connection 6-tuples are forwarded to CT pipe forward pipe, no-matching packets are forwarded to miss pipe. The user application must handle packets accordingly.

DOCA Flow CT API is fundamentally built around four major parts:

- CT module manipulation – configure CT module resources.
- CT connection entry manipulation - add, remove or update connection entries.

- Callbacks - handle async entry processing result.
- Pipe and entry statistics.



Aging

Timeout/aging time is a time in seconds that sets the maximum allowed time for a session to be kept without a packet seen. If the time has passed and no packet was seen the session is considered as terminated.

To support Aging, a dedicated Aging thread is started to poll and check counters for all connections,

Autonomous mode

Multiple CT workers process connections in parallel, connection's lifecycle is controlled by packet state and aging.

CT workers establish and close connections automatically based on connection packet flags.

Packet Meta is defined as following:

uint32_t src : 1 ; /**< Source port in multi-port E-Switch mode */

uint32_t hairpin : 1 ; /**< Subject to forward using hairpin. */

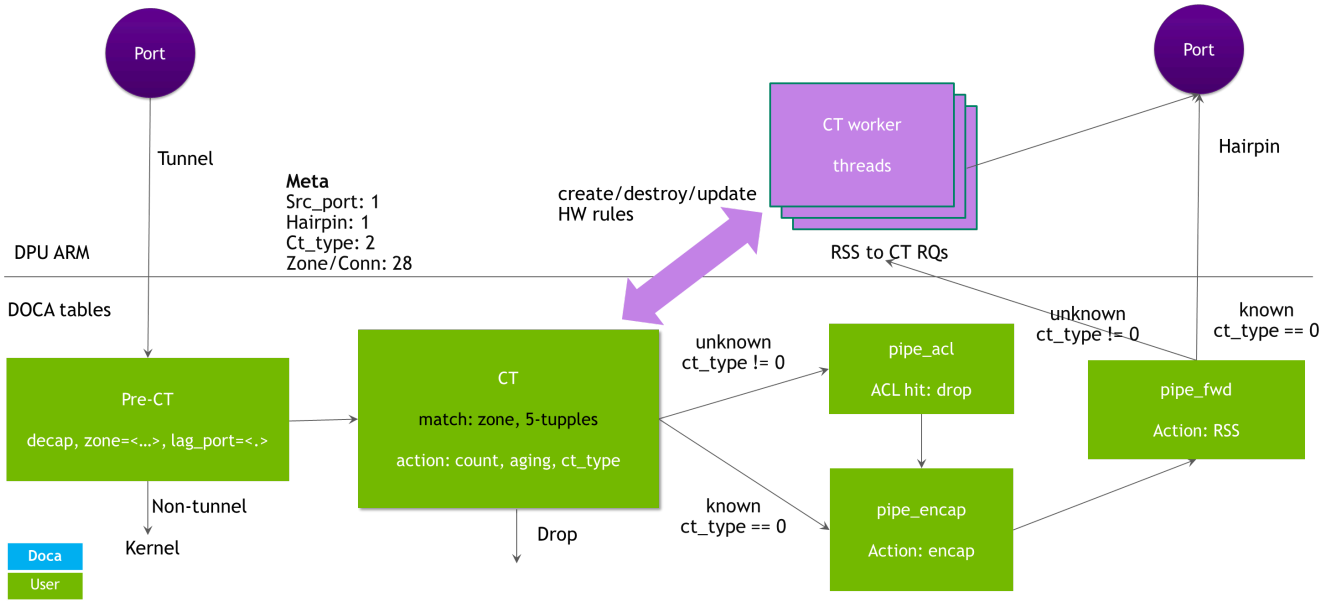
uint32_t type : 2 ; /**< CT packet type: New, End or Update */

uint32_t data : 28 ; /**< Zone set by user or reserved after CT pipe. */

CT table matches Packet Meta and 5-tuples, change type of meta:

- NONE (known) if packet match any connection rule.
- NEW if new TCP or UDP connection.
- END if TCP connection closed.

Packet Meta src and hairpin are used for forwarding pipe and worker to deliver packet.



Managed mode

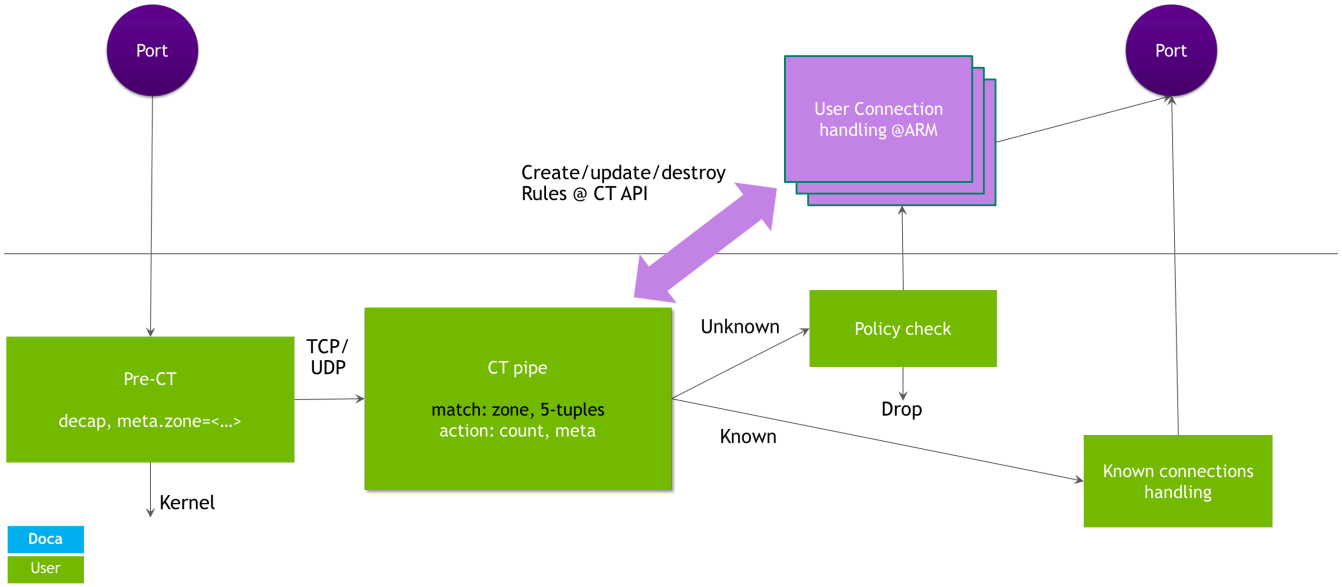
The user manages thread, parsing and process connection state packets, create or destroy connection by calling DOCA Flow CT management APIs.

CT aging module notify aged connections by calling callbacks.

The user can create connection rules with different pattern, meta or counter for each packet direction.

The user is responsible for defining meta and mask to match and modify.

The user can create one rule of a connection firstly, then create the other rule with API doca_flow_ct_entry_add_dir().



Prerequisites

To enable DOCA Flow CT on the DPU, perform the following:

1. Enable "iommu.passthrough" in Linux boot commands or disable SMMU from DPU BIOS
2. Create huge pages.
3. Configure DPU firmware with "LAG_RESOURCE_ALLOCATION=1"
4. Put DPU into E-Switch mode if single port or multi-port E-Switch mode to support two PF ports.

Please reference to unit_test/README.txt for detail steps.

Actions

DOCA Flow CT supports actions based on meta and NAT operations.

Each action can be defined as shared or non-shared.

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 the user responsibility to track shared actions and to remove them when they become irrelevant.

Shared actions are defined using control queue (see struct doca_flow_ct_cfg).

Non-shared actions

Actions provided with their data during entry create/update.

These actions are completely managed by DOCA Flow CT and can't be reused in multiple entries.

Actions templates

The user should provide actions templates during Doca Flow CT pipe creation (as for any other pipe),

Note that only templates for meta and NAT will be 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)

Feature enable

Enable user actions by setting the following:

1. User actions templates (during CT pipe creation).

2. Number of user actions (during DOCA Flow CT init).

API

For the library API reference, refer to DOCA FLOW and CT API documentation in the NVIDIA DOCA Libraries APIs.

Note

The pkg-config (`*.pc` file) for the Flow CT library is include d in DOCA's regular definitions : `doca`.

The following sections provide additional details about the library API.

enum doca_flow_ct_flags

DOCA Flow CT configuration optional flags.

Fields:

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 will dump 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 feature.

DOCA_FLOW_CT_FLAG_SW_PKT_PARSING = 1u << 3 ,

Enable CT worker software packet parsing to support VLAN, IPv6 options or special tunnel types.

DOCA_FLOW_CT_FLAG_MANAGED = 1u << 4 ,

Enable managed mode which user application is responsible to manage packet handling and call CT API to manipulate CT connection entries.

DOCA_FLOW_CT_FLAG_ASYMMETRIC = 1u << 5 ,

Allows different 6-tuple table definition for origin and reply direction, 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 origin and reply direction. 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.

struct direction_cfg

Managed mode configuration for origin or reply direction.

Fields:

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.

struct doca_flow_ct_cfg

DOCA Flow CT configuration

Fields:

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 ;

Max number of user actions supported (shared & non shared).

uint32_t nb_arm_sessions [DOCA_FLOW_CT_SESSION_MAX];

Max CT connection number of IPv4 and IPv6.

uint32_t flags ;

CT configuration flags.

struct doca_dev * doca_dev ;

DOCA device.

void * ib_dev ;

Deprecated.

void * ib_pd ;

Deprecated.

uint16_t tcp_timeout_s ;

TCP aging timeout in second. Autonomous mode only.

uint16_t tcp_session_del_s ;

Second to delay when closing a connection to allow final ACK packet pass.

uint16_t udp_timeout_s ;

UDP aging timeout in second. Autonomous mode only.

uint16_t aging_core ;

CPU core ID for CT aging thread to bind.

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.

union {

struct direction_cfg direction [2];

Managed mode configuration for origin and reply direction.

Below fields are dedicate for autonomous mode:

struct {

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_flow_ct_actions

DOCA Flow CT action structure is used in the following cases:

1. For defining shared actions. In this case action data is provided by the user. Action handle is returned by DOCA Flow CT.
2. For defining entry with actions. The structure can be filled in two options:
    1. With action handle of a previously created shared action.
    2. With non-shared action data.

Fields:

enum doca_flow_resource_type resource_type ;

Shared/ non-shared action.

union {

uint32_t action_handle ;

Used when creating an entry with a shared action.

struct { Used when creating an entry with non-shared action or when creating a shared action.

uint32_t action_idx ;

Actions template index.

struct doca_flow_meta meta;

Modify meta.

struct doca_flow_header_l4_port l4_port;

UDP or TCP source and destination port.

union {

struct doca_flow_ct_ip4 ip4;

Source and destination IPv4 addresses.

struct doca_flow_ct_ip6 ip6;

Source and destination IPv6 addresses.

};

} data ;

};

DOCA IPsec

This guide provides developers with instructions on building and developing applications that require IPsec objects for DOCA Flow encryption and decryption.

Introduction

> **Note**
>
> The DOCA IPsec library is supported at beta level.

IPsec is used to set up encrypted connections between different devices. It helps secure data sent over public networks. DOCA IPsec provides an API to create the security association (SA) object required for flow encryption and decryption hardware acceleration.

Using DOCA IPsec can accelerate various IPsec actions in hardware such as:

- IPsec Crypto Offload – accelerate ciphering (encryption and decryption) only
- IPsec Full Offload – accelerate all IPsec actions (i.e., encap/decap, ciphering, and replay protection so sequence numbers are done in the NIC hardware)
- Partial – doing some IPsec actions in hardware and others in software
- Transparent IPsec – a variant of IPsec full offload, data-path endpoint is not aware of IPsec (only available in NVIDIA® BlueField®)

This document is intended for software developers wishing to accelerate their application's flow encryption and decryption operations.

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

For more information about flow encryption and decryption, please refer to the DOCA Flow.

Environment

Note

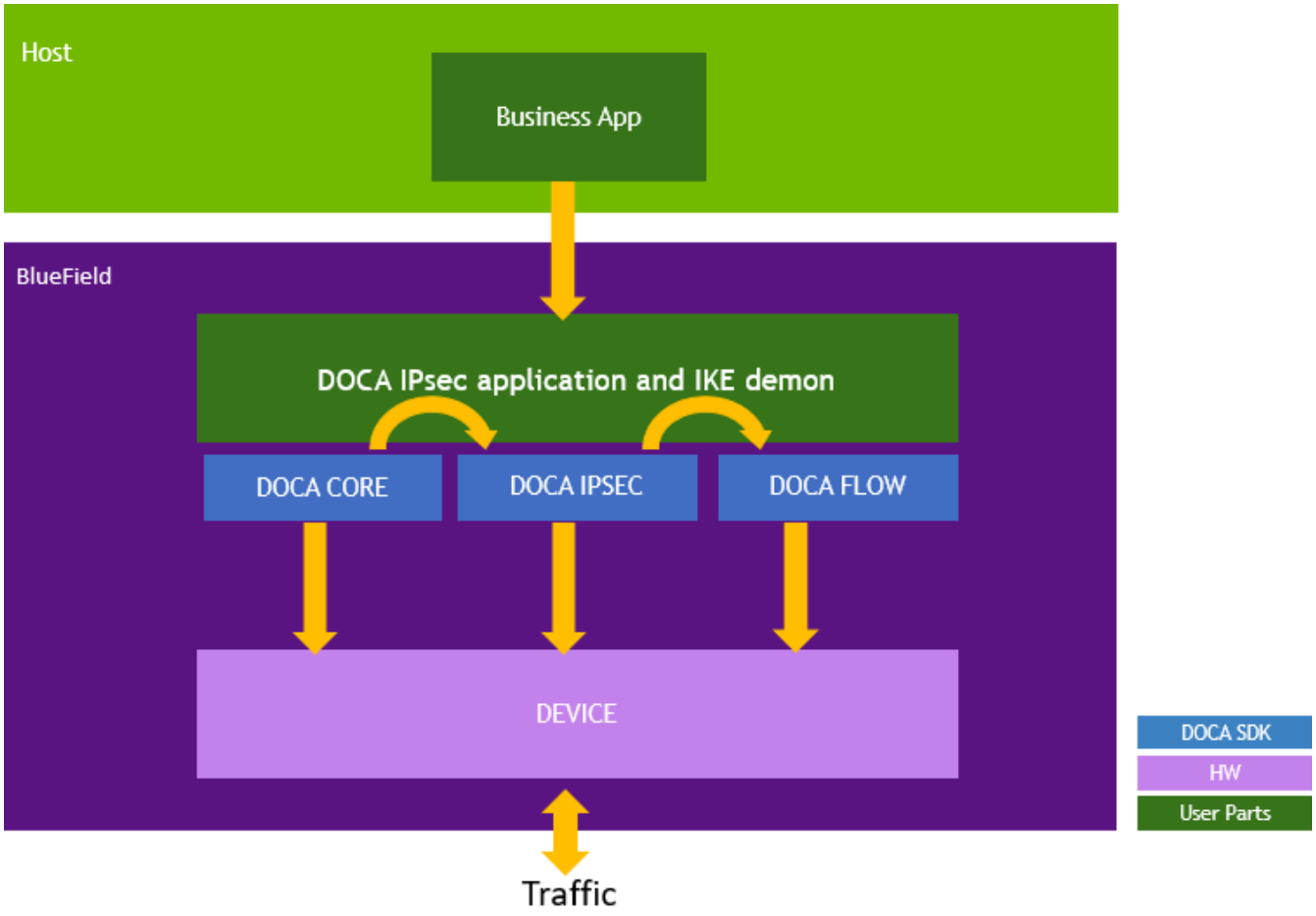
If your target application utilizes 100Gb/s or higher bandwidth, where a substantial part of the bandwidth is allocated for IPsec traffic, please refer to the *NVIDIA BlueField-2 DPUs Product Release Notes or NVIDIA ConnectX-6 Dx Adapters Product Release Notes* to learn about a potential bandwidth limitation. To access the relevant product release notes, please contact your NVIDIA sales representative.

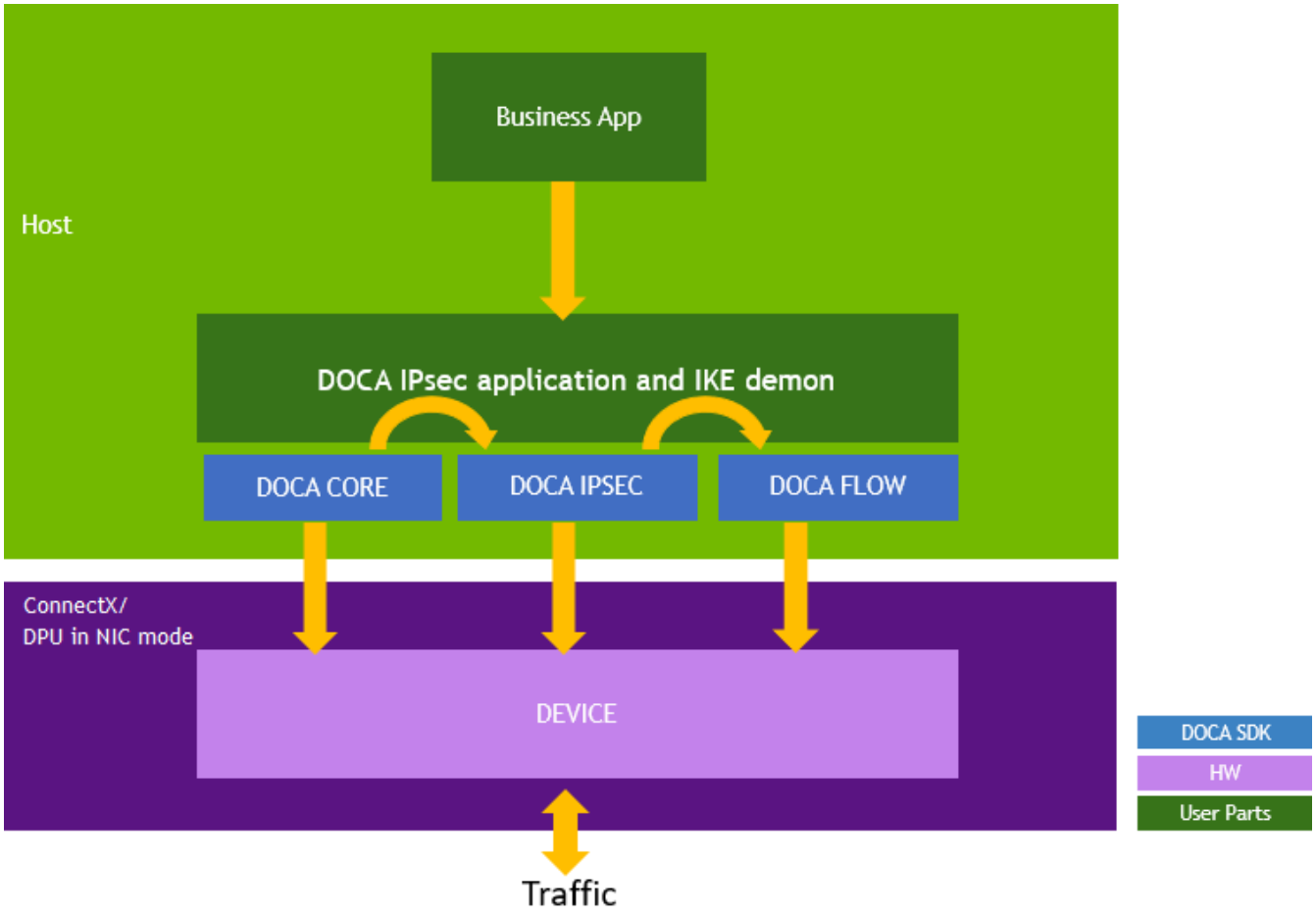
Note

NVIDIA® ConnectX®-6 Dx only supports Full Offload: Encrypted Overlay (where a Hypervisor controls IPsec offload) in a Linux OS with NVIDIA drivers.

Architecture

DOCA IPsec is a DOCA Context as defined by DOCA Core. It leverages DOCA Core architecture to expose asynchronous tasks/events that are offloaded to hardware.


Host
Business App
BlueField
DOCA IPsec application and IKE demon
DOCA CORE
DOCA IPSEC
DOCA FLOW
DEVICE
Traffic
DOCA SDK
HW
User Parts



Configuration Phase

To start using a library, make sure 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.

Configurations

The context can be configured to match the application use case.

To find if a configuration is supported, or what the allowed min/max value for it is, please refer to section "Device Support".

Mandatory Configurations

The following configurations are mandatory and must be set by the application before attempting to start the context:

- At least 1 task/event type must be configured. See configuration of Tasks and Events for information.
- A device with appropriate support must be provided

Device Support

DOCA IPsec needs a device to operate. To pick a device, refer to DOCA Core Device Discovery.

DOCA IPsec-based applications can run either on the host machine or on the BlueField DPU target, or ConnectX for some functionality.

Confirm that the BlueField DPU is crypto-enabled:

```
mlxfwmanager | grep Crypto # expected "Crypto Enabled;"
```

Note

Make sure IPsec is supported using the `doca_ipsec_cap_task_sa_create_is_supported` function. IPsec is supported in the DPU or on the host when the DPU is in NIC mode only.

If a hardware sequence number or hardware antireplay are used, IPsec support is enabled using `doca_ipsec_sequence_number_get_supported` and `doca_ipsec_antireplay_get_supported` respectively.

Execution Phase

This section describes execution on CPU using DOCA Core Progress Engine.

Tasks

DOCA IPsec exposes asynchronous tasks that leverage the DPU hardware according to the DOCA Core architecture. See DOCA Core Task for details.

SA Create Task

This task allows creating an SA object.

Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_ipsec_task_sa_create_set_conf`	`doca_ipsec_cap_task_sa`

Input

Common input as described in DOCA Core Task.

Name	Description	Notes
SA attributes	A pointer to the intended SA attributes	-

Output

SA object in addition to the c ommon output as described in DOCA Core Task.

Task Successful Completion

After the task completes successfully, an SA object is created.

Task Failed Completion

If the task fails midway:

- The task fails to allocate
- The task is not submitted
- The task fails to progress

Limitations

In addition to all the limitations described in DOCA Core Task, depending on how you set the offload (Crypto/Full), the acceleration varies:

- IPsec Crypto Offload – accelerate ciphering (encryption and decryption) only
- IPsec Full Offload – accelerate all IPsec actions (e.g., encap/decap, ciphering, and replay protection so sequence numbers are done in the NIC hardware)

SA Destroy Task

This task allows destroying an SA object.

Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_ipsec_task_sa_destroy_set_conf`	-

Input

Common input as described in DOCA Core Task.

Name	Description	Notes
SA object	-	-

Output

Common output as described in DOCA Core Task.

Task Successful Completion

After the task completes successfully, the SA object is destroyed.

Task Failed Completion

If the task fails midway, an SA object is not destroyed.

State Machine

The DOCA IPsec 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.

Idle

In this state, it is expected that application will either:

- Destroy the context
- Start 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 then freed

Starting

In this state, it is expected that application will c all progress to allow transition to next state.

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

Running

In this state, it is expected that application will:

- Allocate and submit tasks
- Call progress to complete tasks and/or receive events

Allowed operations:

- Allocating a 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

Previous State	Transition Action
Starting	Call progress until the context state transitions

Stopping

In this state, it is expected that application will free any completed tasks.

Allowed operations:

- Allocating previously configured task
- Submitting an allocated task
- Calling 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

DOCA IPsec Application Examples

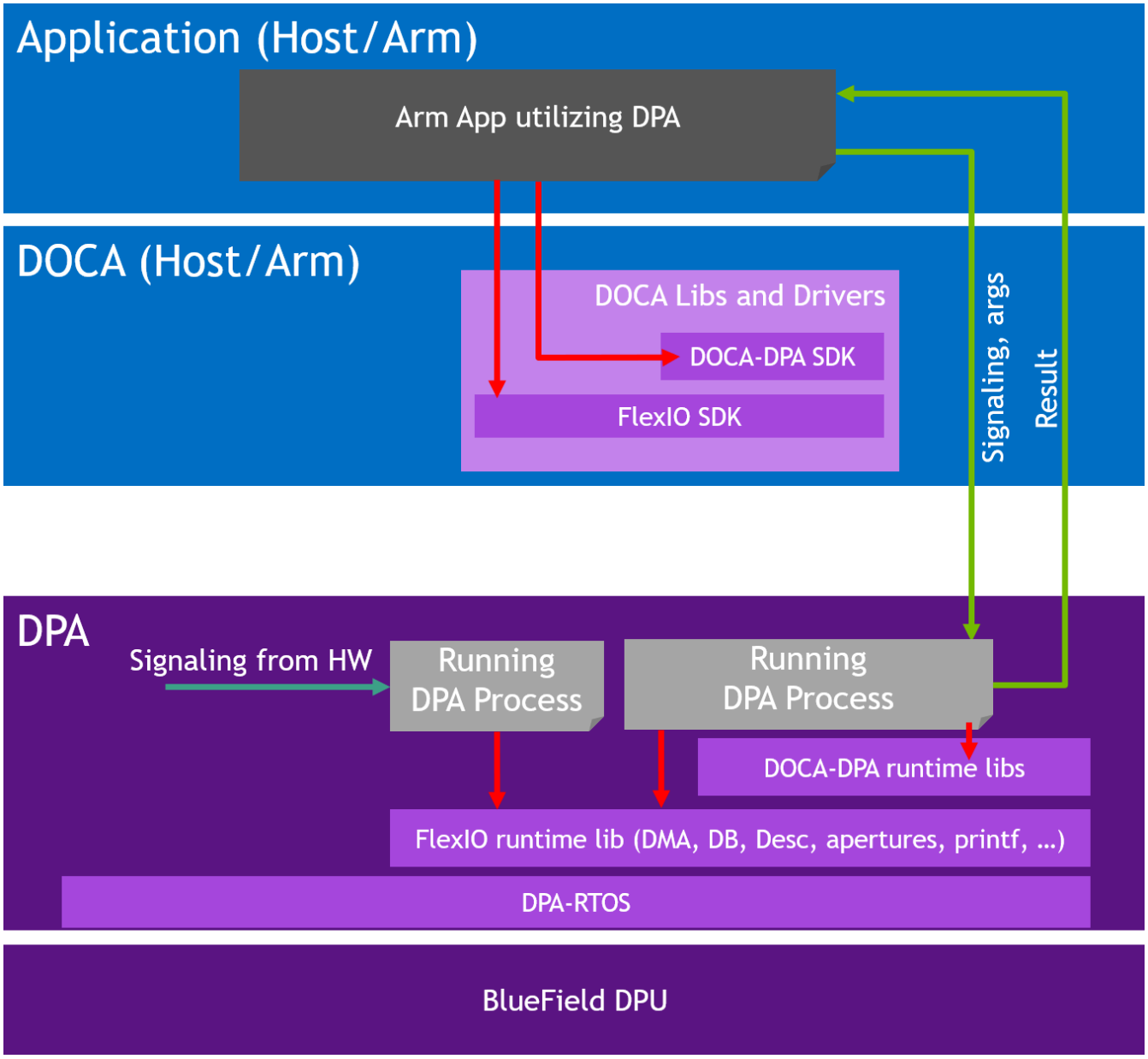
Please refer to DOCA IPsec Security Gateway Application Guide.

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.

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.

DPA RTOS

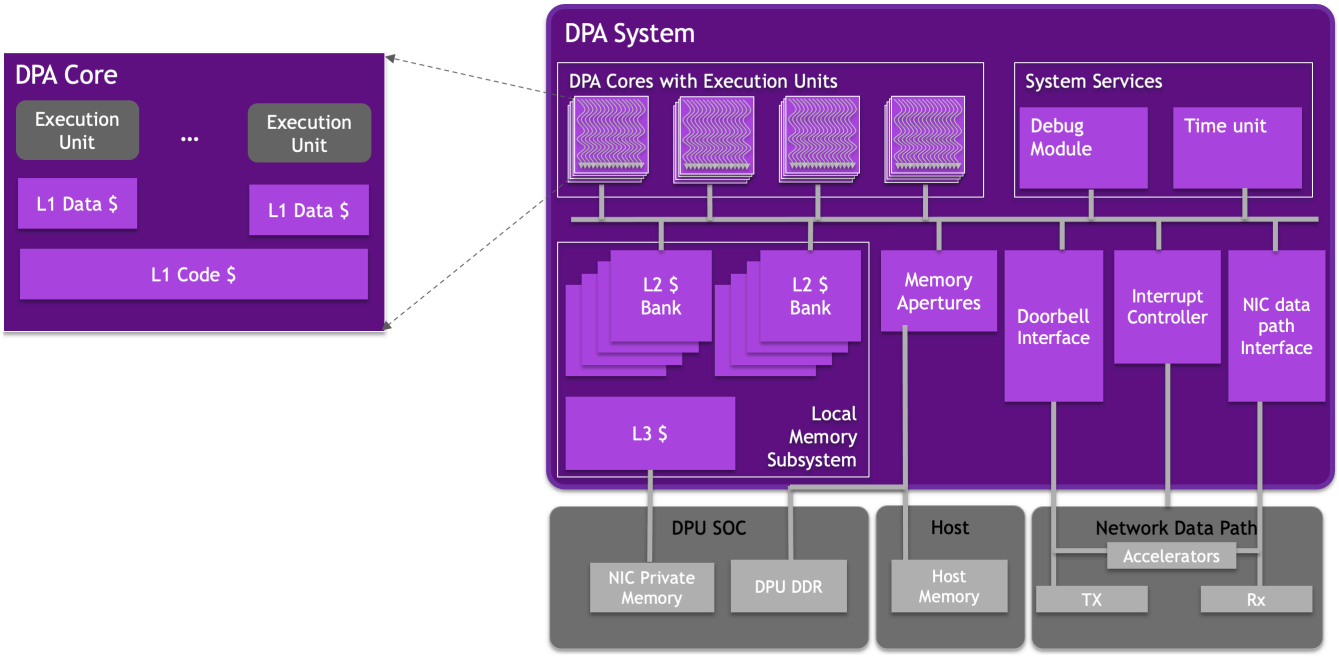
The RTOS is designed to rely on hardware-based scheduling to enable low activation latency for the execution handlers. The RTOS works in a cooperative run-to-completion scheduling model.

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.

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.

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.

DPA Development

DPA Development Overview

DOCA Libs and Drivers

The NVIDIA DOCA framework is the key for unlocking the potential of the BlueField DPU.

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

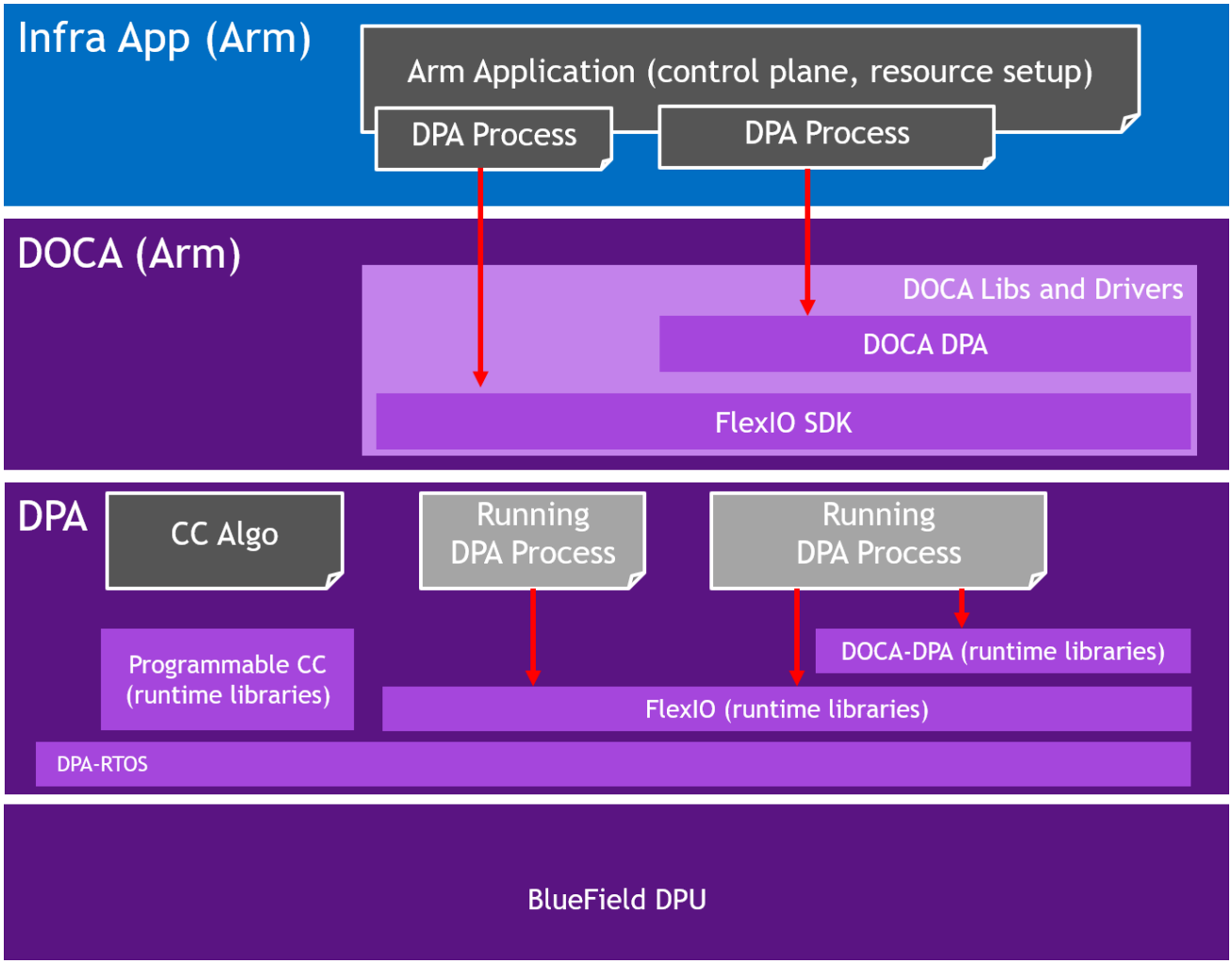
Programming Model

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 alpha 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.

Host-to-DPA Work Submission

The work submission APIs enable a host application to invoke a function on the DPA and supply it with arguments. The work is executed in an asynchronous manner only when the work's dependencies are satisfied. Using this model, the user can define an arbitrary sequence of self-triggering work on the DPA. To borrow common CUDA terminology, these functions are called "kernels". This frees up the host's CPU to focus on its tasks after submitting the list of work to the DPA.

The following is an example where the host submits three functions, `func1`, `func2`, and `func3` that execute one after the other. The functions are chained using a DPA sync event, which is an abstract data type that contains a 64-bit counter.

```
/* func1 -> func2 -> func3 */
/* wait on event, until threshold of 0 (satisfied immediately),
 * add 1 to the same event when func1 is complete */
doca_dpa_kernel_launch_update_add(dpa, event, 0, event, 1,
nthreads, func1, <args>);

/* wait on event, until threshold of 1 (i.e., wait for func1),
 * add 1 to event when func2 is complete */
doca_dpa_kernel_launch_update_add(dpa, event, 1, event, 1,
nthreads, func2, <args>);

/* wait on event, until threshold of 2 (i.e., wait for func2),
 * add 1 to event when func3 is complete */
doca_dpa_kernel_launch_update_add(dpa, event, 2, event, 1,
nthreads, func3, <args>);
```

DOCA Sync Events

For more information, refer to DOCA Sync Event in the *DOCA Core*.

The previous example demonstrates how events can be used to chain functions together for execution on the DPA. In addition to triggered scheduling, events can be directly signaled by either the CPU, GPU, DPA, or by remote nodes. This provides flexibility of coordinating work on the DPA.

The following are some examples of how events can be used:

- Signaling and waiting from CPU (host or DPU's Arm) – the CPU thread signals the event while using its counter. The event can control the execution flow on the DPA. Using the wait operation, the CPU thread can wait in either polling or blocking mode until the corresponding event is signaled.
    - CPU signals an event:

```
doca_sync_event_update_<add|set>(event, value)
```

    - CPU waits for an event:

```
doca_sync_event_wait_<gt|gt_yield>(event, value, mask)
```

- Signaling from the DPA from within a kernel – the event is written to in the user's kernel during its execution. When waiting, the DPA kernel thread waits until the event value is satisfied (e.g., greater than).
    - DPA kernel signals an event:

```
doca_dpa_dev_sync_event_update_<add|set>(event, value)
```

    - DPA kernel waits for an event:

```
doca_dpa_dev_sync_event_wait_gt(event, value, mask)
```

- Signaling from remote nodes – the event is written by the remote side. The remote node may update the count of a sync event in the target node that it has connected to via an RDMA handle.
    - Remote node signals an event:

```
doca_dpa_dev_rdma_signal_<add|set>(rdma, event, count);
```

Event Usage Example

The following example demonstrates how to construct a pipeline of functions on the DPA using events:

```
/* Host */
main()
{
      // create event for usage on DPA, i.e., subscriber=DPA,
publisher=DPA
      doca_sync_event_create(&event);
      doca_sync_event_publisher_add_location_dpa(event, ctx);
      doca_sync_event_subscriber_add_location_dpa(event, ctx);
      doca_sync_event_start(event);

      // export a handle representing the event for DPA to use
      doca_sync_event_export_to_dpa(event, ctx, &event_handle) }

/* DPA: func1 -> func2 */
__dpa_global__ func1(args)
{
      work1();
      // signal next thread by adding to event counter
      doca_dpa_dev_sync_event_update_add(event_handle, 1);
}

__dpa_global__ func2(args)
{
     // wait for event counter to be greater than 0 (i.e. 1)
     doca_dpa_dev_sync_event_wait_gt(event_handle, 0, ...);

     work2();
     // signal next thread by adding to event counter
     doca_dpa_dev_sync_event_update_add(event_handle, 1);
}
```

Memory Management

The DPA program can access several memory spaces using the provided APIs. The following presents models to access DPA process heap, host memory, GPU memory, and NIC device memory:

- DPA process heap – this is the DPA process' main memory. The memory may either be in the stack or on the heap. Heap allocations must be obtained using the `doca_dpa_mem_alloc()` API. The low-level memory model in this space is determined by the processor architecture.

- Host memory – this is the address space of the host program. Handles to this memory space can be obtained by DOCA Buff Array API. DMA access to this space is provided from the DPA using the `doca_dpa_dev_memcpy()` routine.

  For more information, refer to DOCA Core Inventories.

Communication APIs

Communication APIs are currently implemented for InfiniBand only.

The communication APIs enable the application to initiate and be a target of remote memory operations. The communication APIs are implemented over RDMA transport on InfiniBand.

All communications are initiated on an RDMA context. An RDMA context is an opaque representation of a queue pair (QP). RDMAs can be either Reliable Connected or Reliable Dynamic Connected Transport. RDMAs are created on the host-side of the application and then a handle to the RDMA can be passed to the DPA-side of the application.

The following code demonstrates a ping-pong operation between two processes, each with their code offloaded to a DPA. The program uses remote memory access semantic to transfer host memory from one node to the other. The DPA initiates the data transfer and detects its completion.

For more information related to the following code section, refer to DOCA Core Inventories and DOCA RDMA.

```
/* Host */
main()
{
   // Create a rdma instance
   doca_rdma_create(doca_dev, &rdma);
   // Set rdma datapath to dpa
   doca_ctx_set_datapath_on_dpa(rdma_as_doca_ctx, dpa_ctx);
   // Set other rdma attributes
   // Start rdma ctx
   doca_ctx_start(rdma_as_doca_ctx);
   // Export rdma to get connection info to pass to the remote
side
   doca_rdma_export(rdma, &connection_info);
   // Application does out-of-band passing of address to remote
side
   // Assume connection info is now in 'rem_rdma_info'
   // Connect my rdma to the remote rdma
   doca_rdma_connect(rdma, rem_rdma_info);
   // Get the rdma dpa handle to pass to the kernel
   doca_rdma_get_dpa_handle(rdma, &rdma_dpa_handle);

   // Allocate local buffer in host memory
   malloc(&local_buf, size);
   // Register buffer for remote access and obtain an object
representing the memory
   doca_mmap_create(&local_mmap);
   doca_mmap_set_memrange(local_mmap, local_buf, size);
   // Obtain memory export to pass to remote side
   doca_mmap_export_rdma(local_mmap, &mem_export);
   // Create buff inventory from local mmap
   doca_buf_arr_create(local_mmap, &buf_arr);
   // Set the target of memory representor to DPA
   doca_buf_arr_set_target_dpa(buf_arr, dpa_ctx);
   // Get DPA handle to pass to kernel
   doca_buf_arr_start(buf_arr);
```

```
   doca_buf_arr_get_dpa_handle(buf_arr, &buf_arr_dpa_handle);

   // Allocate an event that can be signaled by the remote side
   // to indicate that message is ready to be read
   doca_sync_event_create(&event);
   doca_sync_event_publisher_add_location_dpa(event, ctx);
   doca_sync_event_subscriber_add_location_dpa(event, ctx);
   doca_sync_event_start(event);
   // Obtain a handle to event that can be passed to the remote
side
   doca_sync_event_export_remote(event, &remote_event_handle);
   // OOB Pass the remote event handle to the other side
   // OOB Pass mmap export to remote side
 …
}
/* DPA */
func()
{
    // Write contents of local_buf to remote_buf
    doca_dpa_dev_rdma_write(rdma_handle, remote_buf, local_buf,
size);
    // add `1' atomically to remote_event_handle
    doca_dpa_dev_rdma_signal_set(rdma_handle, remote_event, 1);
    // Wait for my partner (remote node) write to complete
    doca_dpa_dev_sync_event_wait_gt(local_event, 0);
    […]
}
```

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` above. 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.

FlexIO

This chapter provides an overview and configuration instr uctions 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 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/`.

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.

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_os` – library for DPA OS-level access.
- `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.

API

Please refer to the *NVIDIA DOCA Driver APIs*.

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;
      ...
}
```

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.

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.

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.

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.

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

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/<library_name>/<sample_name>
meson /tmp/build
ninja -C /tmp/build
```

The binary `flexio_<sample_name>` will be created under `/tmp/build/host/`.

3. Sample (e.g., `flexio_rpc`) usage:

```
Usage: flexio_rpc [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
  --sdk-log-level                           Set the SDK
(numeric) log level for the program
<10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG
  -j, --json <path>                         Parse all command
flags from an input json file
```

4. For additional information per sample, use the `-h` option:

```
/tmp/build/host/flexio_<sample_name> -h
```

Samples

FlexIO RPC

This sample illustrates how to invoke a function on the DPA.

The sample logic includes:

1. Creating FlexIO process.
2. Calling the remote function `flexio_rpc_calculate_sum` on the DPA.
3. Printing return value to the standard output.

References:

- `/opt/mellanox/doca/samples/flexio/flexio_rpc/device/flexio_rpc_de`
- `/opt/mellanox/doca/samples/flexio/flexio_rpc/host/flexio_rpc_samp`
- `/opt/mellanox/doca/samples/flexio/flexio_rpc/host/meson.build`
- `/opt/mellanox/doca/samples/flexio/flexio_rpc/flexio_rpc_main.c`
- `/opt/mellanox/doca/samples/flexio/flexio_rpc/meson.build`

FlexIO Window

This sample illustrates how to use FlexIO's Window API to access host memory from the DPA device.

The sample logic includes:

1. Creating FlexIO process and FlexIO Window.
2. Registering host buffer with FlexIO Window.
3. Calling remote function on the device which overwrites the host buffer.
4. Printing the altered host buffer.

References:

- `/opt/mellanox/doca/samples/flexio/flexio_window/device/flexio_win`
- `/opt/mellanox/doca/samples/flexio/flexio_window/host/flexio_windo`
- `/opt/mellanox/doca/samples/flexio/flexio_window/host/meson.build`
- `/opt/mellanox/doca/samples/flexio/flexio_window/flexio_window_com`
- `/opt/mellanox/doca/samples/flexio/flexio_window/flexio_window_mai`
- `/opt/mellanox/doca/samples/flexio/flexio_window/meson.build`

FlexIO Multithread

This samples illustrates how to use the FlexIO command queue (`flexio_cmdq_*`) API to run a multithread on the DPA device.

The sample logic includes:

1. Creating FlexIO Process, Window, cmdq, and other FlexIO resources.
2. Copying two matrixes to be multiplied to the device (each matrix is 5*5).
3. Allocating result matrix on the host for the results. Each DPA thread writes to this matrix.
4. Generating 5*5 jobs and submitting in the command queue. Each job is responsible for one cell calculation.
5. Starting the command queue.
6. Printing the result matrix.

References:

- `/opt/mellanox/doca/samples/flexio/flexio_multithread/device/flexi`
- `/opt/mellanox/doca/samples/flexio/flexio_multithread/host/flexio_`

- `/opt/mellanox/doca/samples/flexio/flexio_multithread/host/meson.b`
- `/opt/mellanox/doca/samples/flexio/flexio_multithread/flexio_multi`
- `/opt/mellanox/doca/samples/flexio/flexio_multithread/flexio_multi`
- `/opt/mellanox/doca/samples/flexio/flexio_multithread/meson.build`

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

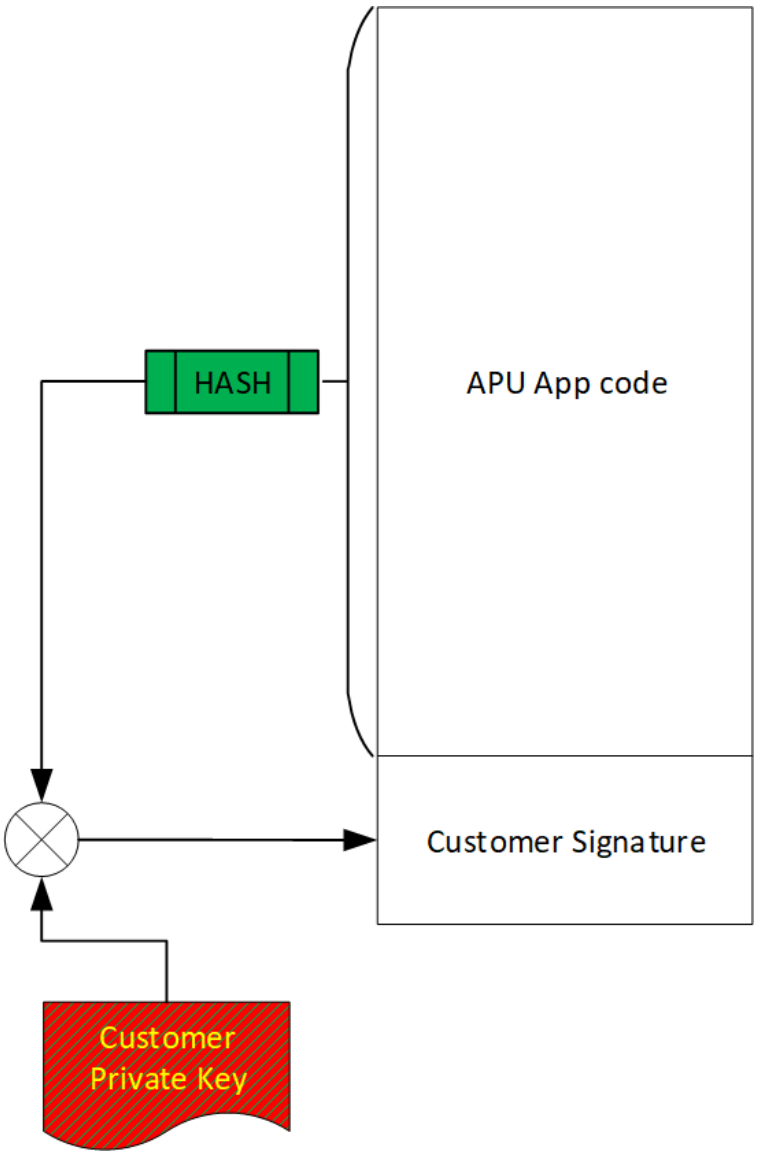
Root of Trust Principles

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.

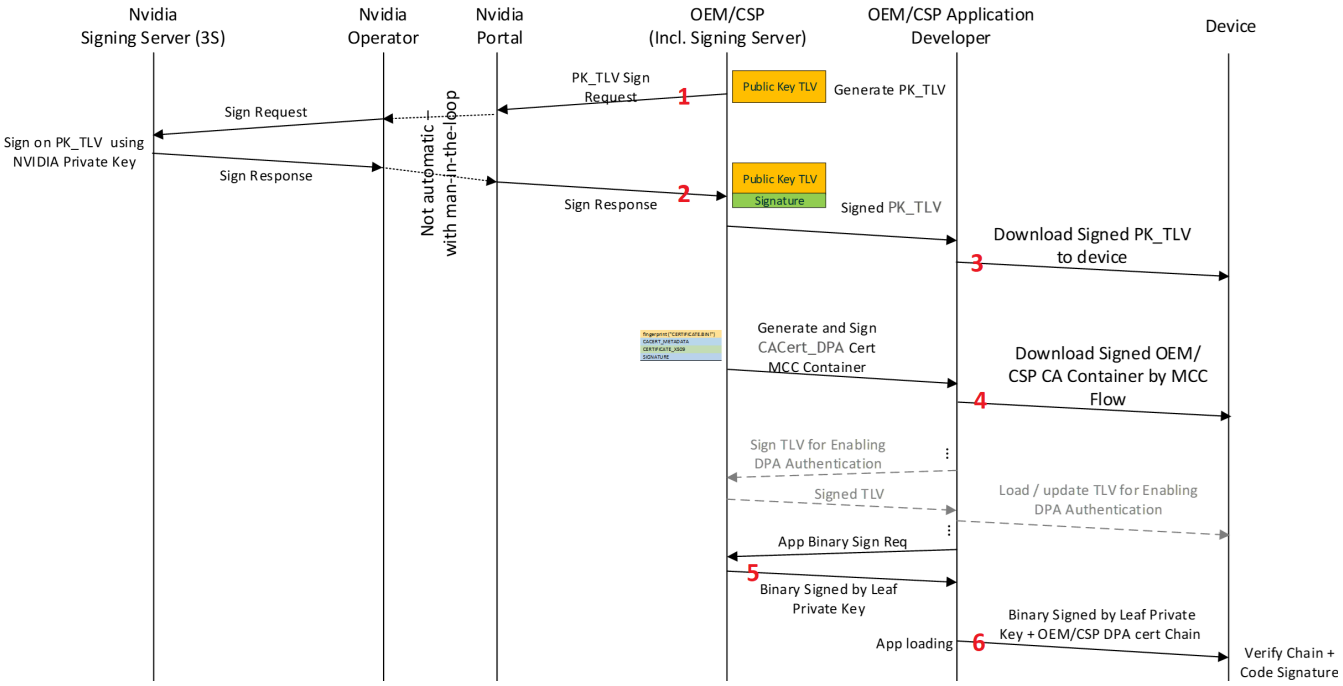
1. Customer provides NVIDIA Enterprise Support the public key for device ownership.
2. NVIDIA signs the customer's public key and sends it back to the customer.
3. Customer uploads the NVIDIA-signed public key to the device, enabling "Transfer of Ownership" to the customer (from NVIDIA).
4. Using the 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:

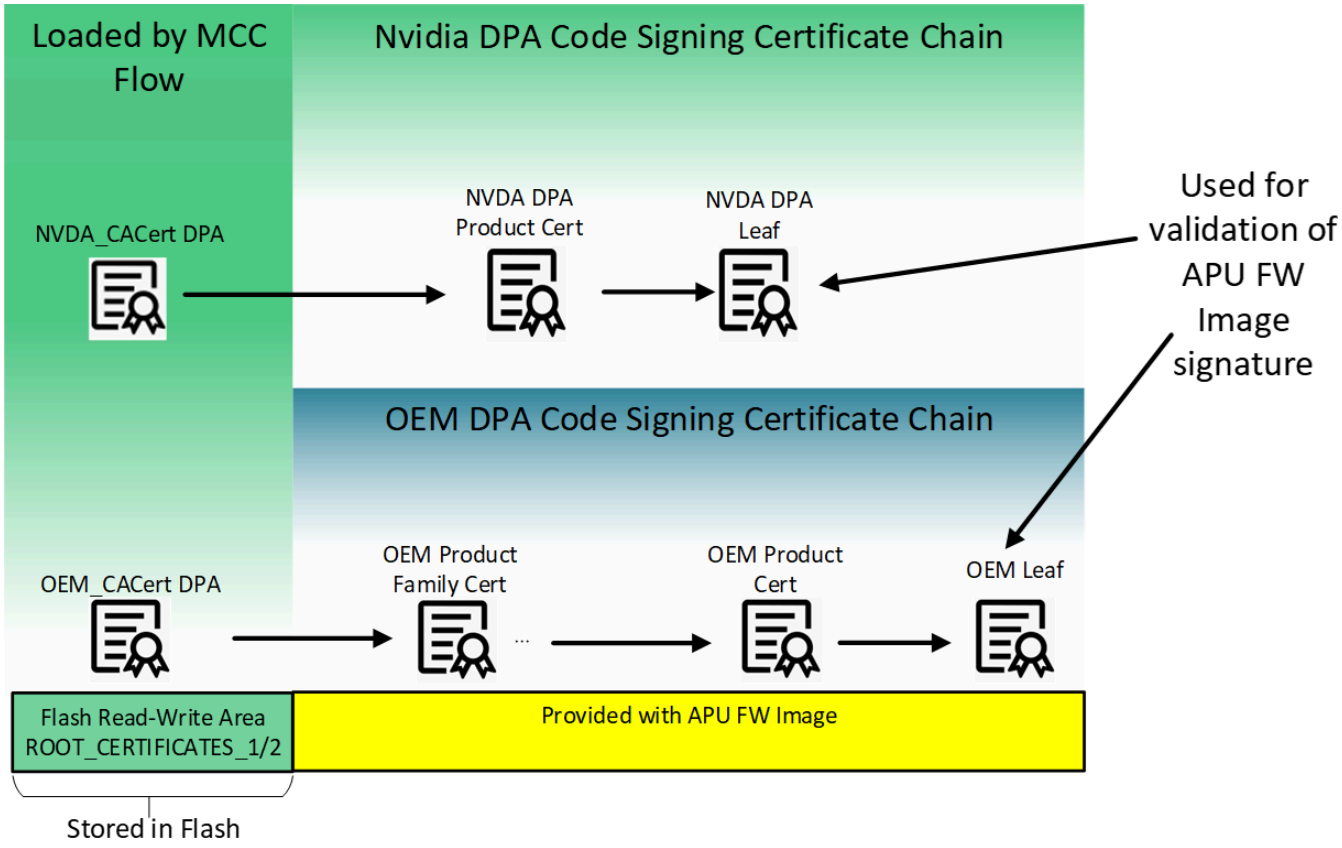
```
flint -d <mst device> -i <signed-certificate-container> burn
```

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



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 firmware 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`):

    1. 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".

    2. 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".

    3. 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 of these matters are out of the scope of this document.

> **Note**
>
> 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.
    1. Generating UUID for the key pair:

        ```
        uuidgen -t
        ```

        Example output:

        ```
        77dd4ef0-c633-11ed-9e20-001dd8b744ff
        ```

    2. 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)
```

3. 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+27A3pQ7
mtpyuHO1JY9AUeKaHUXkWRiopL9Puswx1KcGfWJSNzlEPZRevTHraYlLQC
W9NBE/qIwS2n7kiFwCCvZK6FKUUqZAuMJTpfuNtv9o4C4v0ZiX4TQqWDND
hPf3QLRiJ/ux4G6uHIFwENSwagershuKD0RI6BaZ1g9S9IxdXcD0vTdEuD
CwEs/3xnksNRLUM+TiPEZoc5MoEoKyJv4GFbGttabhDCt5sr9RqAqTNUSD
XoQBQQpqRgYd3lQ31Fhh3G9GjtoAcUQ6l0Gct3DXKFTAADV3Lyo1vjFNrO
pjDKzNmZAsxyIZI0buc24TCgj1yPyFboJtpnHmltyxfm9e+EJsdSIpRiX8
aIzNj08VswULwbKow5Gu5FFpE/uXDE3cXjLOUNnKihszFv4qkqsQjKaK4G
jfiEwsDKwS+cuWd9ihnyLrIWF23+OX0S5xjFXDJE8UthOD+3j3gGmP3kze
Qvh3ITPRsqQltaiYh+CivqaCHC0voIMOP1ilAEZ/rW85pi6LA8EsudNMG2
SznBzZI/OxMk4qKx9nGgjaP2YjmcPw2Ffc9zZcwl57ThEOhlyS6w3E9xwB
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.

    1. 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=O -days 3650
    ```

    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.

    1. 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!
    ```

    2. 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:

    1. Generate a signed container to remove all certificates.

        ```
        mlxdpa --cert_container_type remove --remove_all_certs -
        o <path-to-output> --life_cycle_priority
        <Nvidia/OEM/User> create_cert_container
        ```

        Output example:

        ```
        Certificate container signed successfully!
        ```

2. 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:

1. Generate a signed container to remove certificate based on UUID.

```
mlxdpa --cert_container_type remove --cert_uuid <uuid> -o
<path to output> --life_cycle_priority <Nvidia/OEM/User>
create_cert_container
Certificate container signed successfully!

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!
```

Output example:

```
Certificate container signed successfully!
```

2. 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.

    1. 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
…
```

2. Edit the text file to contain the following TLVs:

```
file_applicable_to                                  1
nv_file_id_vendor                                   1
nv_dpa_auth                                         1
```

3. 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>
```

4. 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='1' 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".

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> --read_only -e q
DPA_AUTHENTICATION
```

Output:

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

Device type:    ConnectX7
Name:           MCX753106ASHEA_DK_Ax
Description:    NVIDIA ConnectX-7 VPI adapter card; 200Gb/s;
dual-port QSFP; single port InfiniBand and second port VPI
(InfiniBand or Ethernet); PCIe 5.0 x16; secure boot; no
crypto; for Nvidia DGX storage - IPN for QA
Device:         e2:00.0

Configurations:
Default         Current         Next Boot
  RO     DPA_AUTHENTICATION
False(0)        False(0)        False(0)
```

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.

[Internal] This NVCONFIG file must have the `disable_override` flag set to ensure that a user cannot force device ownership transfer (DOT) from another user who already has ownership of the device.

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.

ELF File Structure

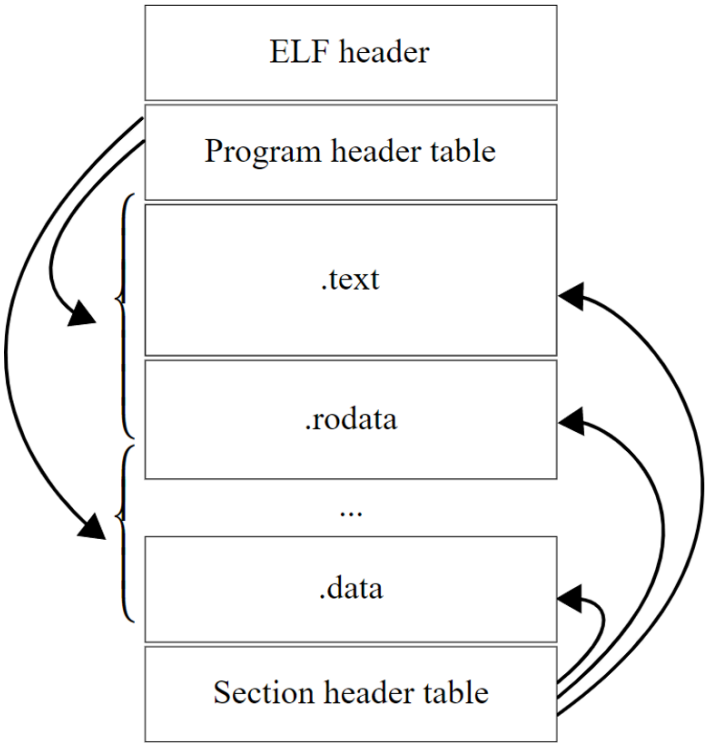
For maximal firmware code reuse, the format of the DPA image loaded from driver should be the same as for the file loaded from flash. As for files loaded from the host, ELF is the default file format. This is chosen as the format for the DPA image, both for flash and for files loaded from the host.

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
    1. Input – host ELF
    2. Output – apps list data table to include:

1. DPA app index
2. DPA app name
3. Offset in host ELF
4. Size of app
5. Name of corresponding crypto data section

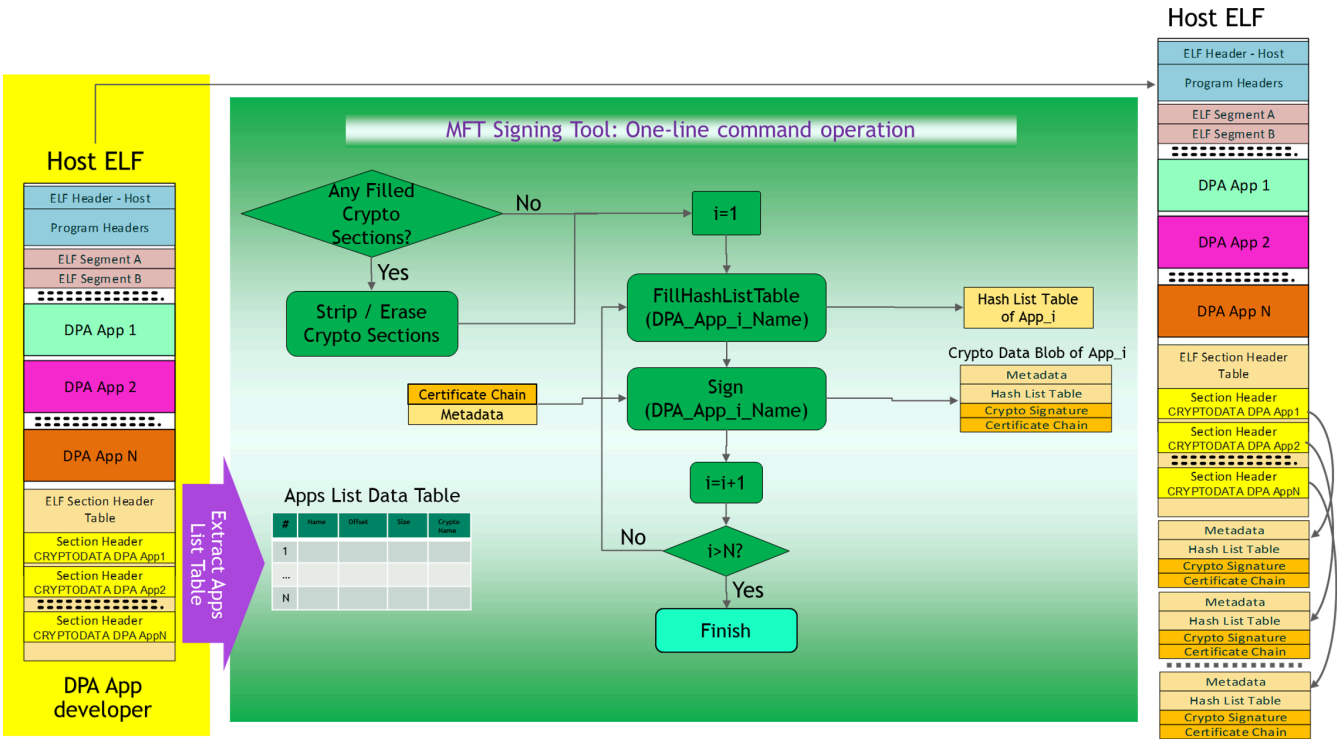
   For each DPA application (from i=1 to i=N, N- number of DPA apps in the host ELF) run steps 2 and 3.

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

Name	Offset	Range	Description
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

Name	Offset	Range	Description
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.

Known Limitations

Supported Devices

- BlueField-3 based DPUs

Supported Host OS

- Windows is not supported

Supported SDKs

- DOCA FlexIO at beta level
- DOCA DPA at alpha level
- DOCA DPA is tested only for InfiniBand devices

Toolchain

- DPA image-signing and signature-verification are not currently supported
- Debugger (GDB) is currently not supported

DOCA DPA

Introduction

> **Note**
>
> Supported at beta level.

This chapter provides an overview and configuration instr uctions 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 DPU. DOCA DPA provides a high-level programming interface to the DPA processor.

DOCA DPA enables the user to submit a function (called a kernel) that runs on the DPA processor. The user can choose to run the kernel with multiple threads .

DOCA DPA offers:

- Support for the DPA subsystem, an offload engine that executes user-written kernels on a highly multi-threaded processor.
- Full control on execution-ordering and notifications/synchronization of the DPA and host/DPU.
- Abstractions for memory services.
- Abstractions for remote communication primitives (integrated with remote event signaling).
- C API for application developers.

DPACC is used to compile and link kernels with the DOCA DPA device library 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*.

Prerequisites

DOCA DPA applications can run either on the host machine or on the target DPU. 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*.

API

Please refer to the *NVIDIA DOCA Library APIs*.

Development Flow

DOCA enables developers to program the DPA processor using both DOCA DPA library and a suite of other tools (mainly DPACC).

These are the main steps to start DPA offload programming:

1. Write DPA device code, or kernels, (`.c` files) with the `__dpa_global__` keyword before function (see examples later).
2. Use DPACC to build a DPA program (i.e., a host library which contains an embedded device executable). Input for DPACC are kernels from the previous steps and DOCA DPA device library.

3. Build host executable using a host compiler (inputs for the host compiler are DPA program are generated in the previous step and the user application source files).

4. In your program, create a DPA context which then downloads the DPA executable to the DPA processor to allow invoking (launching) kernels from the host/DPU to run on the DPA.

DPACC is provided by the DOCA SDK installation . For more information, please refer to the *NVIDIA DOCA DPACC Compiler*.

Software Architecture

Deployment View

DOCA DPA is composed of two libraries that come with the DOCA SDK installation:

- Host/DPU library and header file (used by user application)
  - `doca_dpa.h`
  - `libdoca_dpa.a` / `libdoca_dpa.so`
- Device library and header file
  - `doca_dpa_dev.h`
  - `doca_dpa_dev_sync_event.h`
  - `doca_dpa_dev_buf_array.h`
  - `doca_dpa_dev_rdma.h`
  - `libdoca_dpa_dev.a`

DPA Queries

Before invoking the DPA API, make sure that DPA is indeed supported on the relevant device.

The API that 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.

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(doca_dpa_t 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(doca_dpa_t 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.

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). An application must obtain a DPA context to begin working with the DPA API (several DPA contexts may be created by the application).
Memory	DOCA DPA provides an API to allocate/manage DPA memory, as well as handling host memory that has been exported to DPA.
Sync Event	Data structure in either CPU, DPU, GPU, or DPA-heap. An event contains a counter that can be updated and waited on.
Kernel	User function (and arguments) launched from host and executing on the DPA asynchronously. A kernel may be executed by one or more DPA threads (specified by the application as part of the launch API).

Term	Definition
RDMA	Abstraction around a network transport object (encapsulates RDMA RC QP). Allows sending data to remote RDMA for example, and collect communication completions.

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 so as to avoid race conditions.

DPA Context

Context creation:

```
doca_error_t doca_dpa_create(struct doca_dev *dev, doca_dpa_app_t app, doca_dpa_t *dpa, unsigned int flags)
```

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. This is a host-side API and it is expected to be the first programming step.

The `app` parameter of this call is the same name used when running DPACC to get the DPA program.

Memory Subsystem

A DPA program can allocate (from the host API) and access (from both the host and device API) several memory locations using the DOCA DPA API. DOCA DPA supports access from the host/DPU to DPA heap memory and also enables device access to host memory (e.g., kernel writes to host heap).

Normal memory usage flow would be to:

1. Allocate memory on the host.

2. Register the memory.

3. Export the registered memory so it can be accessed by DPA kernels.

4. Access/use the memory from the kernel (see device side APIs later).

Host-side API

- To free previously allocated DPA memory (if it exists):

```
doca_dpa_mem_free(doca_dpa_dev_uintptr_t dev_ptr)
```

- To allocate DPA heap memory:

```
doca_dpa_mem_alloc(doca_dpa_t dpa, size_t size,
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 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 set memory:

```
doca_dpa_memset(doca_dpa_t dpa, doca_dpa_dev_uintptr_t
dev_ptr, int value, size_t size)
```

- To export host memory to DPA and get a handle to use on your kernels, one must use a DOCA Core Memory Inventory Object in the following manner (refer to TBD *DOCA Core Memory Subsystem API*):

```
doca_buf_arr_set_target_dpa(struct doca_buf_arr
*buf_arr, struct doca_dpa *dpa_handler)
doca_buf_arr_get_dpa_handle(const struct doca_buf_arr
*buf_arr, struct doca_dpa_dev_buf_arr **dpa_buf_arr)
```

  Use the output parameter (`dpa handle`) by passing it as kernel parameter in `doca_dpa_kernel_launch(...)` API.

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).

Kernels get the `doca_dpa_dev_buf_t` handle in their kernel and invoke the following API:

- 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 get the address pointed to by the buffer handle:

```
uintptr_t doca_dpa_dev_buf_get_addr(doca_dpa_dev_buf_t buf)
```

- To get the length of the buffer:

```
uint64_t doca_dpa_dev_buf_get_len(doca_dpa_dev_buf_t buf)
```

- To get a pointer to external memory registered on the host:

```
doca_dpa_dev_uintptr_t
doca_dpa_dev_buf_get_external_ptr(doca_dpa_dev_buf_t buf,
uint64_t buf_offset)
```

- To copy memory between two registered memory regions:

```
doca_dpa_dev_memcpy_nb(doca_dpa_dev_buf_t dst_mem, uint64_t
dst_offset, doca_dpa_dev_buf_t src_mem, uint64_t
src_offset, size_t length)
```

- To transpose memory between two registered memory regions:

```
doca_dpa_dev_memcpy_transpose2D_nb(doca_dpa_dev_buf_t
dst_mem, uint64_t dst_offset, doca_dpa_dev_buf_t src_mem,
uint64_t
src_offset, size_t length, size_t element_size, size_t num_col
```

- Since memory operations in the device are non-blocking, developers are given the following API to drain all previous memory operations:

```
doca_dpa_dev_memcpy_synchronize()
```

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.

Communication Model

DOCA DPA communication primitives allow sending data from one node to another. The communication object between two nodes is called an RDMA DPA handle. RDMAs represent a unidirectional communication pipe between two nodes. Currently, RDMA DPA handles are built on top of RDMA RC QP implementation. RDMA DPA handles can be used by kernels only. The RDMA DPA handle allows the DPA to track the completion of all communications since they are asynchronous.

RDMA DPA handles are created when setting a DOCA RDMA context to DPA data path. For more information, please refer to *DOCA RDMA*.

Host-side API

- To obtain a DPA RDMA handle:

```
doca_ctx_set_datapath_on_dpa(doca_ctx rdma_as_doca_ctx
,doca_dpa_t dpa)
doca_rdma_get_dpa_handle(struct doca_rdma
*rdma, struct doca_dpa_dev_rdma **dpa_rdma)
```

DPA RDMAs are not thread-safe and, therefore, must not be used from different kernels/threads concurrently.

Device-side API

The flow in DOCA DPA begins with creation and configuration of an object on the host/DPU side and then exporting it to a device so the kernels and DPA can use v arious entities. This is no different for RDMA. After the host application connects two RDMAs, the local RDMA is exported to the device and can be used by the device-side API.

- To read to a local buffer from the remote side buffer:

```
doca_dpa_dev_rdma_read(doca_dpa_dev_rdma_t rdma,
doca_dpa_dev_buf_t dst_mem, uint64_t dst_offset,
doca_dpa_dev_buf_t src_mem, uint64_t
src_offset, size_t length)
```

- To copy local memory to the remote side buffer:

```
doca_dpa_dev_rdma_write(doca_dpa_dev_rdma_t rdma,
doca_dpa_dev_buf_t dst_mem, uint64_t dst_offset,
doca_dpa_dev_buf_t src_mem, uint64_t
src_offset, size_t length)
```

- To do an atomic add operation on the remote side buffer:

```
doca_dpa_dev_rdma_atomic_fetch_add(doca_dpa_dev_rdma_t rdma,
doca_dpa_dev_buf_t dst_mem, uint64_t
dst_offset, size_t value)
```

- 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)
```

As all DPA RDMA operations are non-blocking, the following API is provided to kernel 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)`).

Limitations

- DOCA DPA has only been enabled and tested with InfiniBand networking protocol.
- RDMAs are not thread-safe.

Execution Model

DOCA DPA provides an API which enables full control for launching and monitoring kernels.

Understanding the following terms and concepts is important:

- Thread – just as with modern operating systems, DOCA DPA provides a notion of sequential execution of programmer code in an "isolated" environment (i.e., variables which are thread-local and not global or shared between other threads). Moreover, DOCA DPA provides hardware threads which allow a dedicated execution pipe per thread without having the execution of one thread impact other ones (no preemption, priorities etc).

- Kernel – this is a C function written by the programmer and compiled with DPACC that the programmer can invoke from their host application. Programmers can use the `doca_dpa_kernel_launch_update_<add|set>(...)` API to run their kernels. One of the parameters for this call is the number of threads to run this kernel (minimum is 1). So, for example, a programmer can launch their kernel and ask to run it with 16 threads.

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>(doca_dpa_t
dpa, struct doca_sync_event *wait_event, uint64_t
wait_threshold, struct doca_sync_event *comp_event, uint64_t
comp_count, unsigned int nthreads, doca_dpa_func_t *func, ... /* args
*/)
```

- This function asks DOCA DPA to run `func` in DPA by `nthreads` 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 resources to run 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/DPU) 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

- ```
  unsigned int doca_dpa_dev_thread_rank()
  ```

  Retrieves 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_num_threads()
  ```

Returns the number of threads running current kernel.

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);
}
```

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_publisher_add_location_cpu(ev_a, doca_dev);
    doca_sync_event_subscriber_add_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_publisher_add_location_dpa(ev_b, dpa_ctx);
    doca_sync_event_subscriber_add_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_publisher_add_location_dpa(ev_c, dpa_ctx);
```

```
    doca_sync_event_subscriber_add_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_publisher_add_location_dpa(comp_ev, dpa_ctx);
    doca_sync_event_subscriber_add_location_cpu(comp_ev,
doca_dev);
    doca_sync_event_start(comp_ev);

    /* Export kernel events and acquire their handles */
    doca_sync_event_export_to_dpa(ev_b, dpa_ctx, &ev_b_handle);
    doca_sync_event_export_to_dpa(ev_c, dpa_ctx, &ev_c_handle);
    doca_sync_event_export_to_dpa(comp_ev, dpa_ctx,
&comp_ev_handle);

    /* Launch kernels */
    doca_dpa_kernel_launch_update_add(wait_ev = ev_a,
wait_threshold = 1, &linear_kernel, kernel_args:
                          NULL, ev_b_handle);
    doca_dpa_kernel_launch_update_add(wait_ev = NULL,
&linear_kernel, nthreads = 1, kernel_args:
                          ev_b_handle, ev_c_handle);
    doca_dpa_kernel_launch_update_add(wait_ev = NULL,
&linear_kernel, nthreads = 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);
}
```

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_export_to_dpa(&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, nthreads = 1, &diamond_kernel, kernel_args:
                            NULL, 0, ev_b_handle, ev_c_handle);
    doca_dpa_kernel_launch_update_set(wait_ev = NULL, nthreads =
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, nthreads =
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, nthreads =
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, nthreads =
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>(..., nthreads = x)`, the next call with `nthreads = 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 `nthreads` 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`.

Logging

The following device-side API (to be used in kernels) supports logging to stdout:

```
doca_dpa_dev_printf(const char *format, ...)
```

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 call:

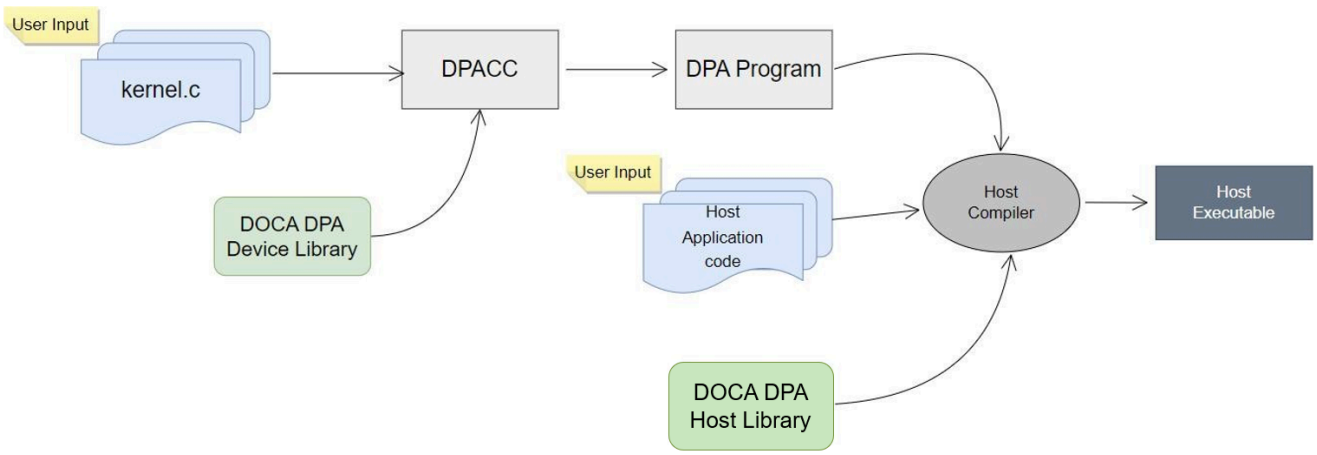
```
doca_dpa_peek_at_last_error(const doca_dpa_t dpa)
```

This data path call indicates if an error occurred on device. If so, the DPA context enters a fatal state and can no longer be used by the application. Therefore, the application must destroy the DPA context.

Hello World Example

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:
    1. Kernels from step 1.
    2. DOCA DPA device library.
3. Build host executable using a host compiler. Input for the host compiler:
    1. DPA program.
    2. User host application `.c` / `.cpp` files.



Procedure Steps

The following code snippets show a very basic DPA code that eventually prints "Hello World" to `stdout`.

1. Prepare kernel code:

```
#include "doca_dpa_dev.h"

__dpa_global__ void hello_world_kernel(void)
{
        doca_dpa_dev_printf("Hello World\n");
}
```

2. Prepare application code:

```
#include <stdio.h>
#include <unistd.h>
#include <doca_dev.h>
#include <doca_error.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 doca_dpa_app_t 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 the kernel launch API as follows:
* doca_dpa_kernel_launch_update_set(..., &hello_world_kernel, ...);
*/
doca_dpa_func_t hello_world_kernel;

int main(int argc, char **argv)
{
        /* local variables... */

        /* Open doca device */
        printf("\n----> Open DOCA device\n");
        /* Open appropriate DOCA device doca_dev... */

        /* Create doca_dpa context */
        printf("\n----> Create DOCA DPA context\n");
        doca_dpa_create(doca_dev, dpa_hello_world_app, &doca_dpa, 0);

        /* Create a CPU event that will be signaled when kernel is finished */
        printf("\n----> Create DOCA Sync event\n");
        doca_sync_event_create(&cpu_event);
```

```
        doca_sync_event_publisher_add_location_dpa(cpu_event,
doca_dpa);
        doca_sync_event_subscriber_add_location_cpu(cpu_event,
doca_dev);
        doca_sync_event_start(cpu_event);

        /* Kernel launch */
        printf("\n----> Launch hello_world_kernel\n");
        doca_dpa_kernel_launch_update_set(doca_dpa, NULL, 0,
cpu_event, 1, 1, &hello_world_kernel);

        /* Waiting for completion of kernel launch */
        printf("\n----> Waiting for hello_world_kernel to finish\n");
        doca_sync_event_wait_gt(cpu_event, 0, 0xFFFFFFFFFFFFFFFF);

        /* Tear down */
        printf("\n----> Destroy DOCA Sync event\n");
        doca_sync_event_destroy(cpu_event);

        printf("\n----> Destroy DOCA DPA context\n");
        doca_dpa_destroy(doca_dpa);

        printf("\n----> Destroy DOCA device\n");
        doca_dev_close(doca_dev);

        printf("\n----> DONE!\n");
        return 0;
}
```

3. Build DPA program:

```
/opt/mellanox/doca/tools/dpacc \
    libdoca_dpa_dev.a \
    kernel.c \
    -o dpa_program.a \
    -hostcc=gcc \
    -hostcc-options="-Wno-deprecated-declarations" \
    --devicecc-options="-D__linux__ -Wno-deprecated-declarations" \
    --app-name="dpa_hello_world_app" \
    -I/opt/mellanox/doca/include/
```

4. Build host application:

```
gcc hello_world.c -o hello_world \
    dpa_program.a libdoca_dpa.a \
    -I/opt/mellanox/doca/include/ \
    -DDOCA_ALLOW_EXPERIMENTAL_API \
    -L/opt/mellanox/doca/lib/x86_64-linux-gnu/ -ldoca_common
\
    -L/opt/mellanox/flexio/lib/ -lflexio \
    -lstdc++ -libverbs -lmlx5
```

5. Execution:

```
$ ./hello_world mlx5_0


*********************************************
**********    Hello World Example    **********
*********************************************


----> Open DOCA device

----> Create DOCA DPA context

----> Create DOCA Sync event

----> Launch hello_world_kernel

----> Waiting for hello_world_kernel to finish

----> Destroy DOCA Sync event
/  7/Hello World

----> Destroy DOCA DPA context

----> Destroy DOCA device

----> DONE!
```

DOCA DPA Samples

This section provides DPA sample implementation on top of the BlueField-3 DPU.

DPA Sample Prerequisites

The BlueField-3 DPU's link layer must be configured to InfiniBand, and it must also be set to operate in DPU mode.

Running DPA 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_dpa/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build
   ```

   The binary `doca_<sample_name>` will be created under `/tmp/build/`.

3. Sample (e.g., `dpa_kernel_launch`) usage:

```
Usage: doca_dpa_kernel_launch [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
  --sdk-log-level                           Set the SDK
(numeric) log level for the program
<10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG
  -j, --json <path>                         Parse all command
flags from an input json file

Program Flags:
  -d, --device <IB device name>             IB device name that
supports DPA (optional). If not provided then a random IB
device will be chosen
```

4. For additional information per sample, use the `-h` option:

```
/tmp/build/doca_<sample_name> -h
```

Samples

DPA Kernel Launch

This sample illustrates how to launch a DOCA DPA kernel with a completion event.

The sample logic includes:

1. Allocating DOCA DPA resources.
2. Initializing completion event for the DOCA DPA kernel.
3. Running hello_world DOCA DPA kernel that prints "Hello from kernel".
4. Waiting on completion event of the kernel.
5. Destroying the event and resources.

Reference:

- `/opt/mellanox/doca/samples/doca_dpa/dpa_kernel_launch/dpa_kernel_`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_kernel_launch/host/dpa_ke`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_kernel_launch/device/dpa_`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_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`

DPA Wait Kernel Launch

This sample illustrates how to launch a DOCA DPA kernel with wait and completion events.

The sample logic includes:

1. Allocating DOCA DPA resources.
2. Initializing wait and completion 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_wa`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_wait_kernel_launch/host/d`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_wait_kernel_launch/device`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_wait_kernel_launch/meson.`
- `/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`

DPA Binary Tree

This sample illustrates how to launch multiple DOCA DPA kernels in a binary tree pattern.

The sample logic includes:

1. Allocating DOCA DPA resources.
2. Initializing wait and completion events for the DOCA DPA kernels.
3. Running 7 DOCA DPA kernels, such that each kernel (except the first) waits on the parent kernel in a binary tree-like pattern.
4. Waiting on all 7 completion events (completion event for each kernel).
5. Destroying the events and resources.

Reference:

- `/opt/mellanox/doca/samples/doca_dpa/dpa_binary_tree/dpa_binary_tr`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_binary_tree/host/dpa_bina`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_binary_tree/device/dpa_bi`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_binary_tree/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`

DPA Diamond Tree

This sample illustrates how to launch multiple DOCA DPA kernels in a diamond tree-like pattern.

The sample logic includes:

1. Allocating DOCA DPA resources.
2. Initializing wait and completion events for the DOCA DPA kernel.
3. Running 4 kernels, such that each kernel (except the first) waits on the parent kernel in a diamond tree-like pattern.
4. Waiting on the completion event of the last kernel.
5. Destroying the events and resources.

Reference:

- `/opt/mellanox/doca/samples/doca_dpa/dpa_diamond_tree/dpa_diamond_`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_diamond_tree/host/dpa_dia`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_diamond_tree/device/dpa_d`

- `/opt/mellanox/doca/samples/doca_dpa/dpa_diamond_tree/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`

DPA RDMACopy

This sample illustrates how to perform RDMA copy using DOCA DPA kernels and DOCA DPA RDMAs. This sample launches another thread considered to be a "remote thread" to copy to (the thread is not actually remote as the main process spawns it).

> **Info**
>
> To avoid confusion, the main process is referred to as the "main thread".

The goal is to illustrate how the main thread can copy a buffer (in this example, an integer) to the remote thread's memory.

The sample logic includes:

1. The main thread allocating DOCA DPA resources.
2. The main thread creating DOCA DPA events for wait and completion of DOCA DPA kernels, and another event for communication with the remote thread.
3. Launching the remote thread.
4. The main thread preparing the DOCA DPA RDMA resources, registering the buffer to copy as a DOCA DPA host memory, and signaling (using the communication event) to the remote thread that it is ready.
5. The remote thread preparing the DOCA DPA RDMA resources, registering a buffer to copy to it as a DOCA DPA host memory, and signaling (using the communication

event) to the main thread that it is ready.

6. The main and remote threads connecting to each other's RDMAs.
7. The main thread launching a DOCA DPA kernel that copies the main thread's buffer to the remote thread's buffer.
8. The DOCA DPA kernel synchronizing the RDMA and finishing.
9. The main thread waiting on the completion event of the kernel.
10. The main thread signaling to the remote event that the copy has finished.
11. The remote thread destroying its resources and joining the main thread.
12. The main thread destroying the events and resources.

Reference:

- `/opt/mellanox/doca/samples/doca_dpa/dpa_rdma_copy/dpa_rdma_copy_m`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_rdma_copy/host/dpa_rdma_c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_rdma_copy/device/dpa_rdma`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_rdma_copy/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`

DOCA PCC

This guide provides an overview and configuration instructions for DOCA Programmable Congestion Control (PCC) API.

Introduction

The DOCA PCC library provides a high-level programming interface that allows users to implement their own customized congestion control (CC) algorithm.

The DOCA PCC library provides an API to:

- 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.

This library uses the NVIDIA® BlueField®-3 DPU hardware acceleration for CC management, while providing an API that simplifies hardware complexity, allowing users to focus on the functionality of the CC algorithm.

Prerequisites

DOCA PCC-based applications can run either on the host machine or on the NVIDIA® BlueField®-3 (or later) target DPU.

To enable PCC:

1. Run the following on the host/VM:

```
mlxconfig -d <mlx_device> -y s USER_PROGRAMMABLE_CC=1
```

2. Power cycle the host.

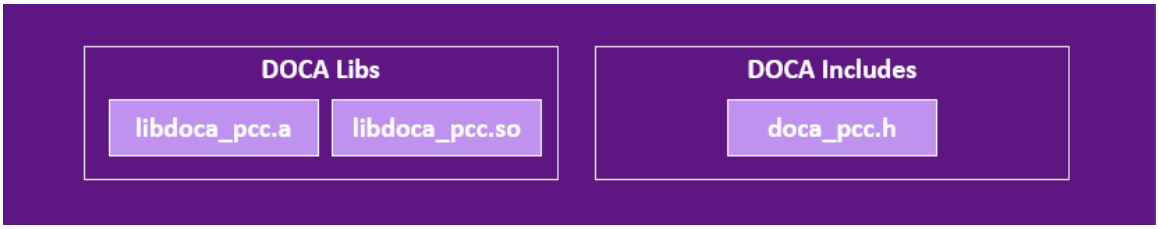
The DPACC tool is used to compile and link user algorithm and device code with the DOCA PCC device library to get applications that can be loaded from the host program .

DPACC is bundled as part of the DOCA SDK installation package. For more information on DPACC, refer to *NVIDIA DOCA DPACC Compiler*.

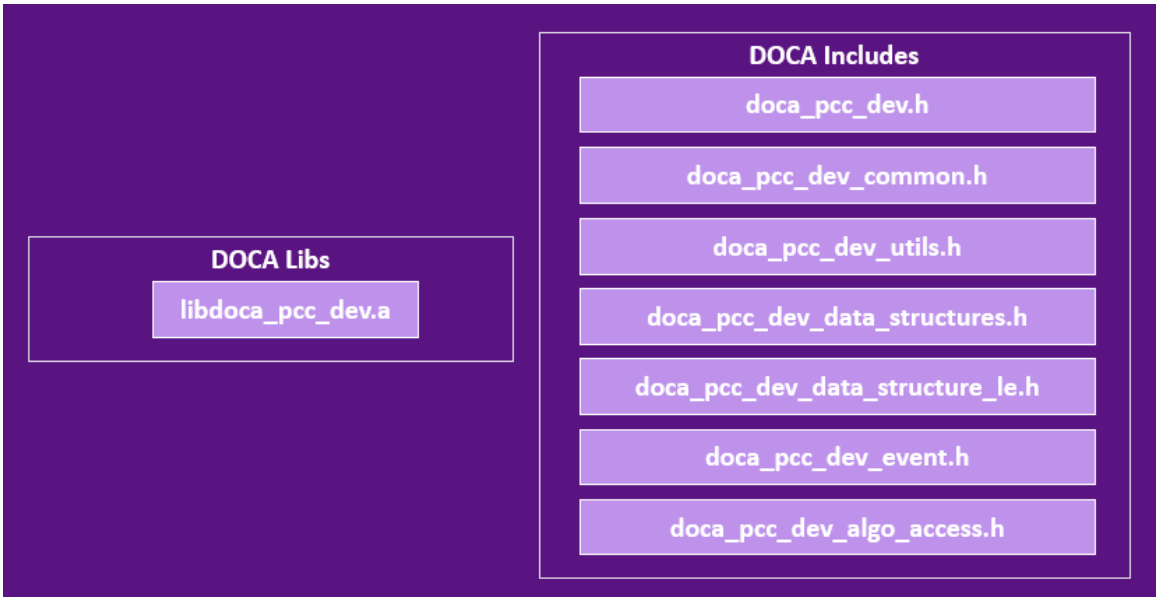
Architecture

DOCA PCC is composed of two libraries which are part of the DOCA SDK installation package :

- Host/DPU library and header files



- Device library and header files



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*.

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:

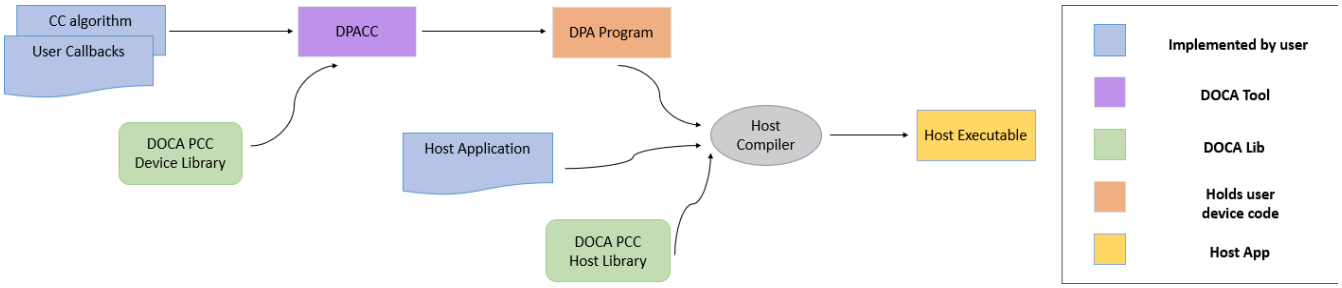
1. (Optional) Implement CC algorithms using the API provided by the device header files to be run by the library.

2. Implement the following calls defined by the library: `doca_pcc_dev_user_init()`, `doca_pcc_dev_user_set_algo_params()`, `doca_pcc_dev_user_algo()`.

3. Use DPACC to build a DPA program (i.e., a host library which contains an embedded device executable). Input for DPACC are the files containing the implementation of the previous steps.

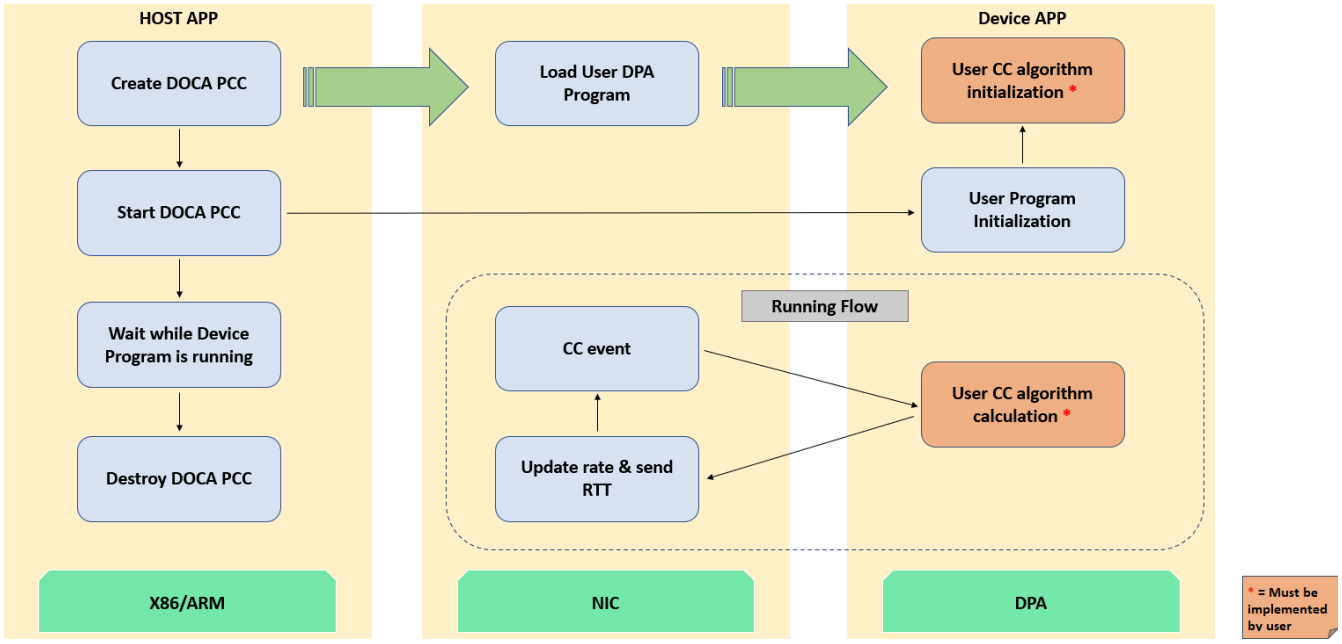
4. Build host executable using a host compiler. Inputs for the host compiler are the DPA program generated in the previous step and the user application source files.

5. In the host executable, create and start a DOCA PCC context which is set with the DPA program containing the device code.



For a more descriptive example, refer to *NVIDIA DOCA PCC Application Guide*.

System Design



Dependencies

The library requires firmware version 32.38.1000 and higher.

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.

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 PCC using `doca_devinfo_get_is_pcc_supported()` .

Setting Up and Starting DOCA PCC Context

After selecting a DOCA device, a PCC context can be created using `doca_pcc_create()`.

Afterwards, the following attributes must be set for the PCC context:

- Context app – the name of the DPA program 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.

> **Note**
>
> 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.

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, w ithout 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.

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;
```

Device API

The device library API consists of calls to setup the CC algorithm to handle CC events arriving on hardware.

Algorithm Access

The device library API provides a set of functions to initiate and 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 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 API provides 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 library API consists of specific user callbacks used by the library to initiate and run the CC algorithm. These callbacks must be implemented by the user and, to be part of the DPA program, compiled by DPACC to provide to the DOCA PCC context.

The set of callbacks to be implemented is as follows:

- `doca_pcc_dev_user_init()` – this is 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.

DOCA DMA

This guide provides instructions on building and developing applications that require copying memory using Direct Memory Access (DMA).

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.

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

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 DPU Modes of Operation.

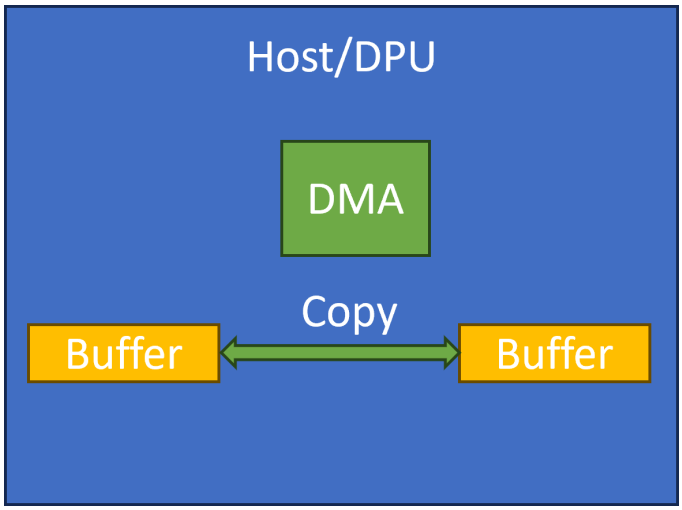
Architecture

DOCA DMA is a DOCA Context as defined by DOCA Core. See NVIDIA DOCA Core Context for more information.

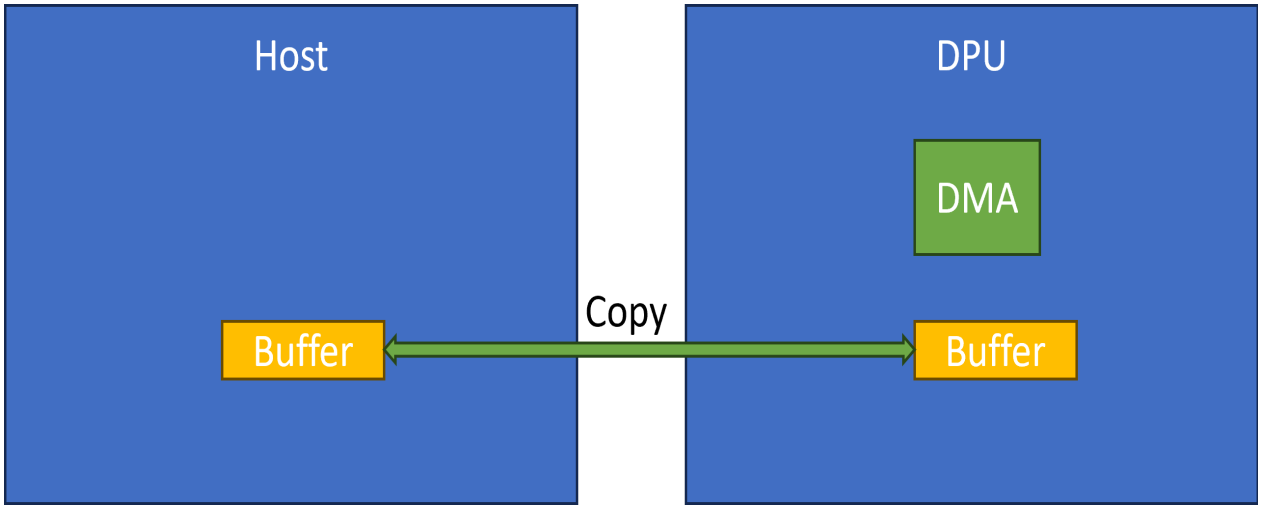
DOCA DMA leverages DOCA Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

DMA can be used to copy data as follows:

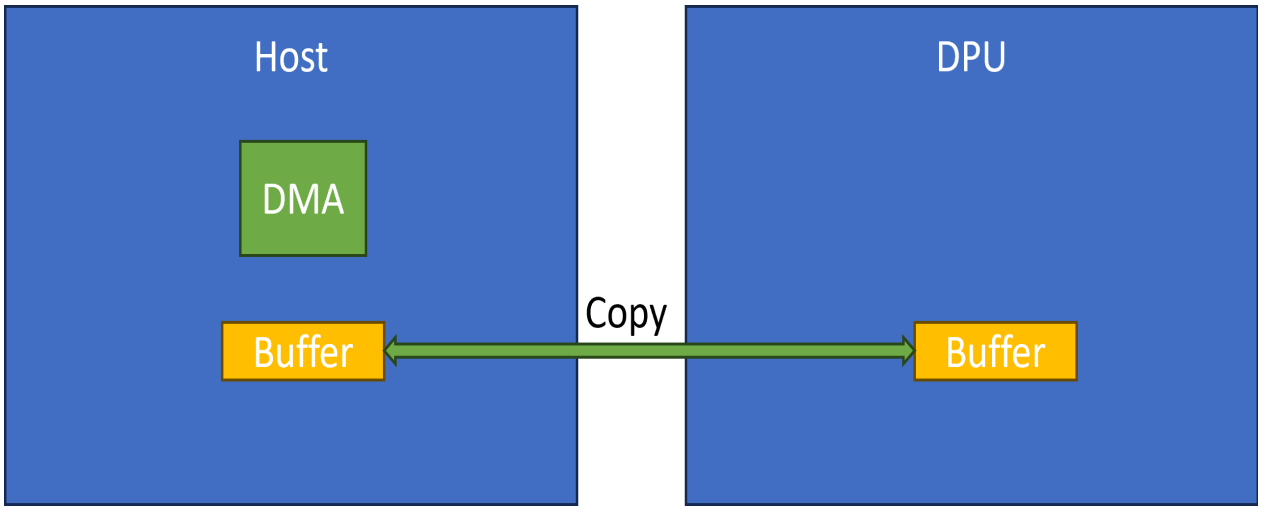
- Copying from local memory to local memory:



- Using DPU to copy memory between host and DPU:



- Using host to copy memory between host and DPU:



Objects

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.

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.

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.

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".

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
- A device with appropriate support must be provided upon creation

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

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

Execution Phase

This section describes execution on CPU using DOCA Core Progress Engine.

Tasks

DOCA DMA exposes asynchronous tasks that leverage the DPU hardware according to the DOCA Core architecture. See DOCA Core Task.

Memory Copy Task

The memory copy task allows copying memory from one location to another. Using buffers as described in Buffer Support.

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`
Number of tasks	`doca_dma_task_memcpy_set_conf`	`doca_dma_cap_get_max_num_ta`
Maximal Buffer Size	–	`doca_dma_cap_task_memcpy_ge`
Maximum buffer list size	–	`doca_dma_cap_task_memcpy_ge`

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

Output

Common output as described in DOCA Core Task.

Task Successful Completion

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 Failed Completion

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

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

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.

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.

Idle

In this state it is expected that application:

- Destorys 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

Starting

This state cannot be reached.

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

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

Alternative Datapath Options

DOCA DMA only supports datapath on the CPU. See Execution Phase.

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

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=DE
  -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
```

Samples

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 job result.
9. Destroying all DMA and DOCA core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_dma/dma_local_copy/dma_local_copy`
- `/opt/mellanox/doca/samples/doca_dma/dma_local_copy/dma_local_copy`
- `/opt/mellanox/doca/samples/doca_dma/dma_local_copy/meson.build`

DMA Copy DPU

> **Note**
>
> 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_sam`
- `/opt/mellanox/doca/samples/doca_dma/dma_copy_dpu/dma_copy_dpu_mai`
- `/opt/mellanox/doca/samples/doca_dma/dma_copy_dpu/meson.build`

DMA Copy Host

Note

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_s`
- `/opt/mellanox/doca/samples/doca_dma/dma_copy_host/dma_copy_host_m`
- `/opt/mellanox/doca/samples/doca_dma/dma_copy_host/meson.build`

DOCA Comm Channel

Info

DOCA SDK 2.5.0 introduces a new API for DOCA Comm Channel. As a result, the old API for DOCA Comm Channel will be deprecated in future releases.

The new DOCA Comm Channel 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.

The new DOCA Comm Channel API does not support event-triggered completions.

DOCA Comm Channel – New

This guide provides instructions on building and developing applications that require communication channels between the x86 host and the Arm cores on the DPU.

Introduction

DOCA Comm Channel provides a communication channel between client applications on the host and servers on the DPU.

Benefits of using Comm Channel:

- Security – Comm Channel is isolated from the network
- Network independent – the state of the Comm Channel does not depend on the state and configuration of the network
- Ease of use

Comm Channel provides two different data path APIs:

- Basic Comm Channel 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_cc_consumer/doca_cc_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

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)

Environment

DOCA Comm Channel based applications can run either on the host machine or on the NVIDIA® BlueField® DPU target.

Sending messages between the host and DPU can only be run with a DPU configured with DPU mode as described in NVIDIA BlueField DPU Modes of Operation.

For basic DOCA Comm Channel send and receive, the following configuration is required:

- `doca_cc_server` context must run on the DPU
- `doca_cc_client` context must run on the host machine

> **Note**
>
> Producer and consumer objects can run on both the host and DPU. However, there must be a valid client/server connection already established on the channel.

Architecture

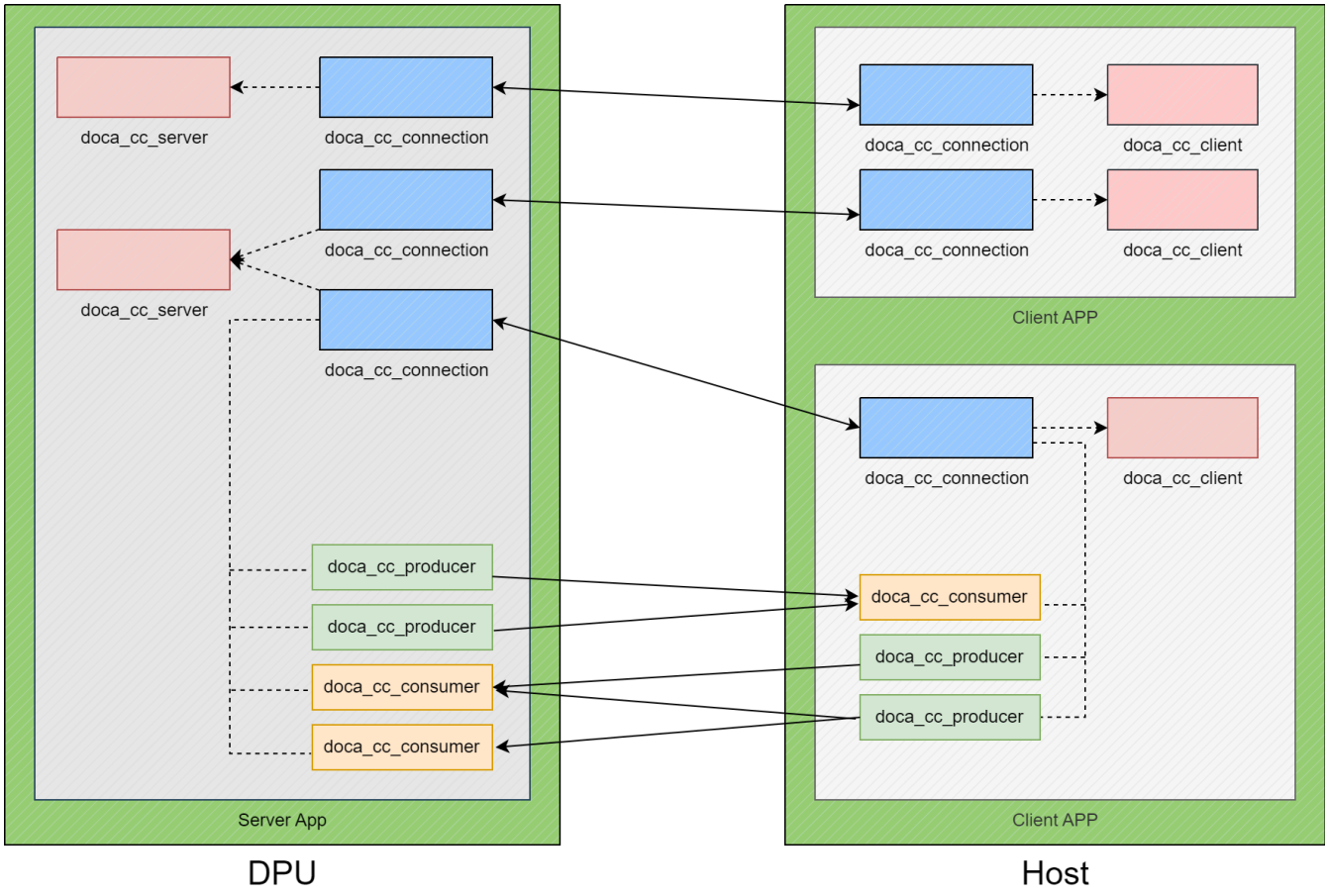
DOCA Comm Channel is comprised of four DOCA Core Contexts. All DOCA Comm Channel contexts leverage DOCA Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

A `doca_cc_server` context runs on the DPU and listens for incoming connections from the host side. Such host side connections are initiated by a `doca_cc_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_cc_connection`.

Once an established connection exists between a client and a server, the `doca_cc_producer` and `doca_cc_consumer` contexts can be used to run fast path channels.

The following diagram provides examples of the contexts use:



Objects

	Description	DPU/Host	Scope
`doca_cc_server`	Allows applications on the DPU to listen on a specific server name and accept new incoming connection from the host	DPU only	Per host PCIe function (`doca_dev` + `doca_dev_rep`)
`doca_cc_client`	Allows client applications to connect to a specific server name on the DPU	Host only	Per host PCIe function (`doca_dev`)

	Description	DPU/Host	Scope
`doca_cc_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_cc_consumer`s and `doca_cc_producer`s.	DPU and host	Per client server pair
`doca_cc_producer`	A handle for a FIFO-like send queue that provides a zero-copy API to send messages to a specific `doca_cc_consumer` on the same `doca_cc_connection`. Multiple `doca_cc_producer`s can be created per `doca_cc_connection`.	DPU and host	Per `doca_cc_connection`
`doca_cc_consumer`	A handle for a FIFO-like receive queue that provides a zero-copy API to receive messages from a `doca_cc_producer`	DPU and host	Per `doca_cc_connection`

Security Considerations

- DOCA Comm Channel 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 Comm Channel 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.

Initialization Flow

doca_cc_server Initialization Flow

1. A `doca_cc_server` is created on a specific `doca_dev` and a specific `doca_dev_rep`.
2. A `doca_cc_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_cc_server` can start receiving new connection requests.
4. For the `doca_cc_server` to process new connection requests and messages, the user must periodically call `doca_pe_progress()` .
5. When a new connection request arrives, `doca_cc_server` calls the connection request handler function and passes a `doca_cc_connection` object.

The server can now send and receive messages on the connection represented by `doca_cc_connection` .

doca_cc_client Initialization Flow

1. A `doca_cc_client` is created on a specific `doca_dev` is targeting a specific `doca_cc_server`.

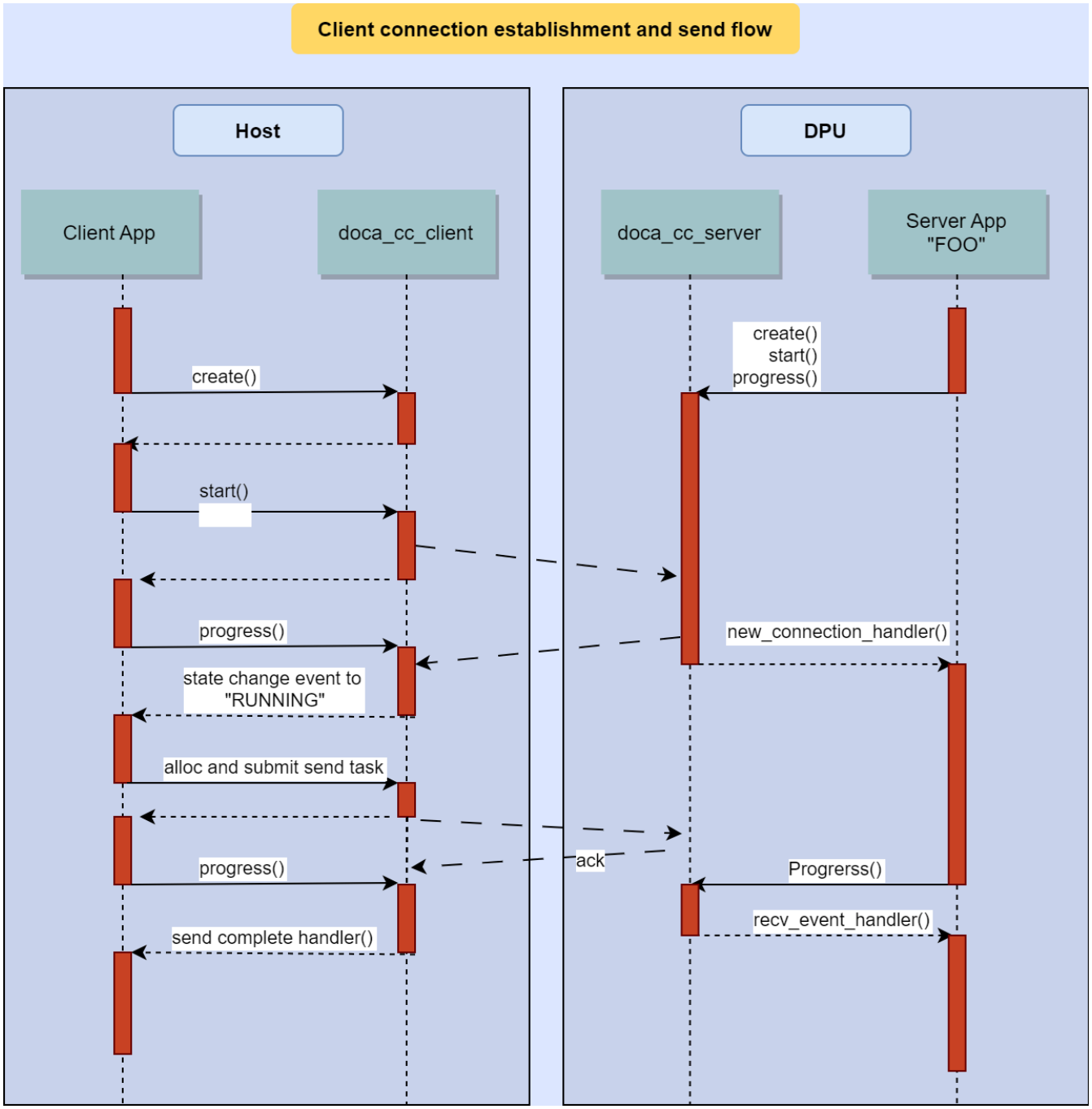
2. Once `doca_ctx_start()` is called, `doca_cc_client` asynchronously tries to connect to the server.

3. To establish the connection and receive messages, the user must periodically call `doca_pe_progress()`.

4. When the connection is established, `doca_cc_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 Comm Channel:



doca_cc_consumer Initialization Flow

1. A `doca_cc_consumer` is created on a specific `doca_cc_connection`.
2. `doca_pe_progress()` must be periodically called on the client/server PE to allow registration of the consumer.

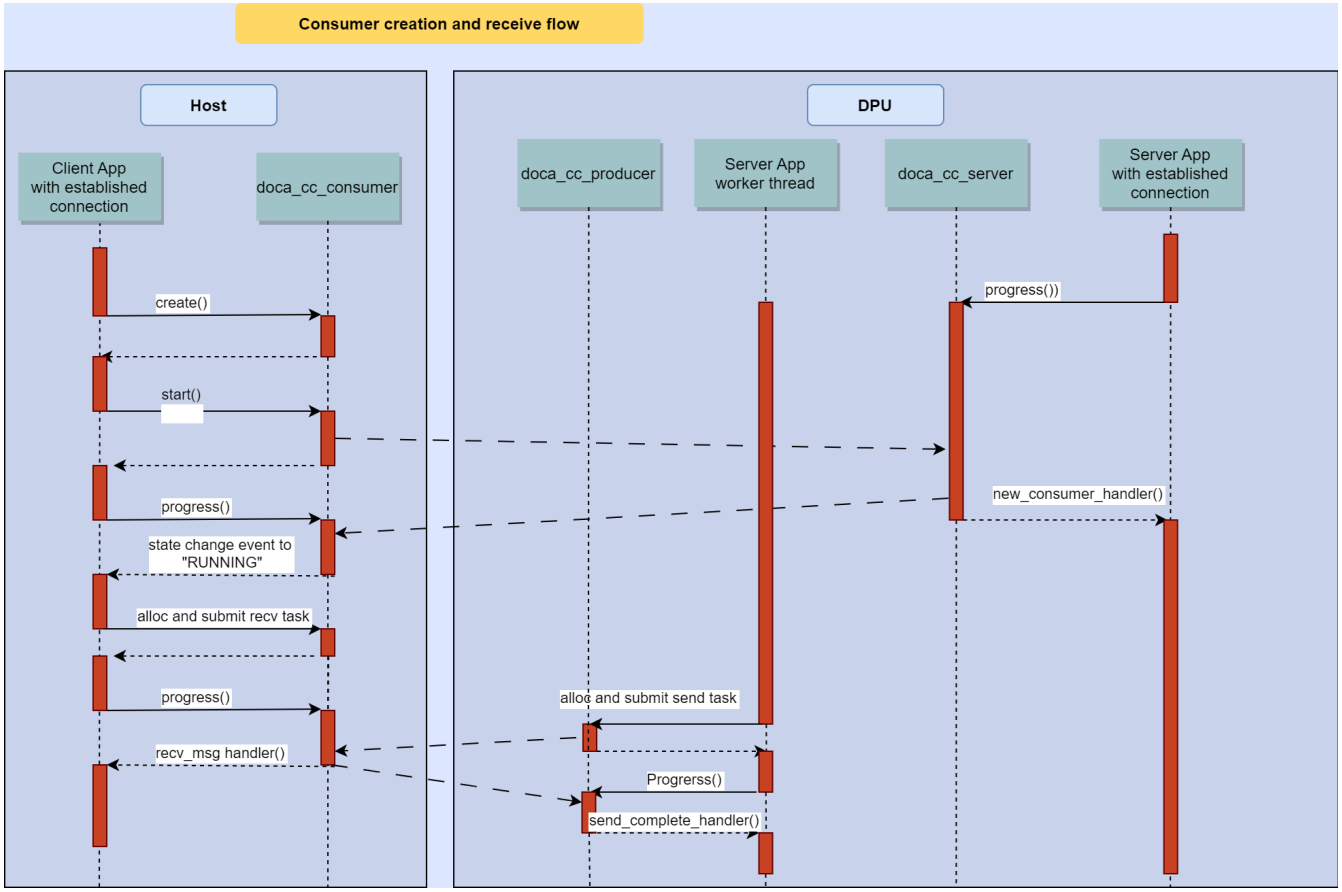
3. After the `doca_cc_consumer` moves to "RUNNING" state:

    1. `doca_cc_consumer` notifies its existence to the peer (invoking a new consumer event).

    2. The application can start posting receive tasks.

    3. A `doca_cc_producer` on the peer side can start sending messages to that consumer.

The initialization is described in the following diagram:



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.

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)

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_cc_(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_cc_(server|client)_set_recv_queue_size` – set the size of the queue to receive new messages on

Device Support

DOCA Comm Channel 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_cc_cap_server_is_supported`
  - `doca_cc_cap_client_is_supported`
- For extended fast path functionality:
  - `doca_cc_cap_is_producer_supported`
  - `doca_cc_cap_is_consumer_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

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

> **Note**
>
> Chained buffers are not supported in DOCA Comm Channel.

Execution Phase

This section describes execution on CPU using DOCA Core Progress Engine. For additional execution environments, refer to section "Alternative Datapath Options".

Tasks

DOCA Comm Channel exposes asynchronous tasks that leverage the DPU hardware according to DOCA Core architecture.

Control Channel Send Task

This task allows the sending of messages between connected client and server objects.

Configuration

Description	API to Set the Configuration	API to Query Support
Number of tasks	`doca_cc_server_send_task_set_conf` `doca_cc_client_send_task_set_conf`	`doca_cc_cap_get_max_sen`
Maximal Message Size	`doca_cc_server_set_max_msg_size` `doca_cc_client_set_max_msg_size`	`doca_cc_server_get_max_` `doca_cc_client_get_max_`

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

Output

Common output as described in DOCA Core Task.

Task Successful Completion

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 Failed Completion

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

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.

Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_cc_consumer_post_recv_task_set_conf`	`doca_cc_cap_is_c`
Number of tasks	`doca_cc_consumer_post_recv_task_set_conf`	`doca_cc_consumer`
Maximal Buffer Size	–	`doca_cc_consumer`

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 **Info** Buffers `doca_mmap` must have `DOCA_ACCESS_FLAG_PCI_READ_WRITE` flag set.

Output

Common output as described in DOCA Core Task.

Task Successful Completion

The task only completes once a producer has written to the advertised buffer, 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 Failed Completion

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

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
- Other limitations are described in DOCA Core Task

Producer Send Task

This task allows producer objects to copy buffers for use by remote consumers.

Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_cc_producer_send_task_set_conf`	`doca_cc_cap_is_produc`
Number of tasks	`doca_cc_producer_send_task_set_conf`	`doca_cc_producer_cap_`
Maximal Buffer Size	–	`doca_cc_producer_cap_`

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
Consumer ID	Identifier for the target consumer to write to	Active consumers and their IDs are advertised through consumer events

Output

Common output as described in DOCA Core Task.

Task Successful Completion

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 Failed Completion

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

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)
- All limitations described in DOCA Core Task

Events

DOCA Comm Channel 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.

Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	`doca_cc_server_event_msg_recv_register` `doca_cc_client_event_msg_recv_register`	–

Trigger Condition

The event is triggered when a remote message is received on any currently active connection associated with the client or server.

Output

Upon event detection, the registered callback is triggered, passing the following parameters:

- A pointer to the message data

> **Info**
>
> 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.

> **Note**
>
> 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.

Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	`doca_cc_server_event_connection_register`	–

Trigger Condition

The event is triggered when a new connection is either established or a current connection disconnected on a server.

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.

Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	`doca_cc_server_event_consumer_register` `doca_cc_client_event_consumer_register`	–

Trigger Condition

The event is triggered whenever a new consumer is created or a current consumer destroyed on the remote side of an established Comm Channel connection.

Output

The event hits a separate callback for either the creation or destruction of a consumer.

Callback parameters include:

- The established Comm Channel connection on which the consumer is connected (on the remote side)

- The ID of the consumer (a unique value per Comm Channel connection)

State Machine

The DOCA Comm Channel 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.

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

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

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

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

Alternative Datapath Options

DOCA Comm Channel can only be run on a CPU datapath. See Execution Phase for details.

DOCA Comm Channel Samples

This section describes DOCA Comm Channel samples based on the DOCA Comm Channel library.

The samples illustrates how to use the DOCA Comm Channel API to do the following:

- Set up a client/server between host and DPU and use it to send text messages
- Configure fast path producers and consumers, and send messages between them

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_cc/<sample_name>
meson /tmp/build
ninja -C /tmp/build
```

   The binary `doca_<sample_name>` is created under `/tmp/build/`.

3. All DOCA Comm Channel samples accept the same input arguments:

Sample	Argument	Description

`doca_cc_ctrl_path_server`
`doca_cc_ctrl_path_client`
`doca_cc_data_path_high_speed_server`
`doca_cc_data_path_high_speed_client`
	`-p`, `--pci-addr`	DOCA Comm Channel device PCIe address

	`-r`, `--rep-pci`	DOCA Comm Channel device representor PCIe address (required only on DPU)
	`-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
```

Samples

DOCA Comm Channel Control Path Client/Server

Note

`doca_cc_ctrl_path_server` must be run on the DPU side and started before `doca_cc_ctrl_path_client` is started on the host.

This sample sets up a client server connection between the host and DPU.

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_cc/cc_ctrl_path_client/cc_ctrl_pa`
- `/opt/mellanox/doca/samples/doca_cc/cc_ctrl_path_client/cc_ctrl_pa`
- `/opt/mellanox/doca/samples/doca_cc/cc_ctrl_path_server/cc_ctrl_pa`
- `/opt/mellanox/doca/samples/doca_cc/cc_ctrl_path_server/cc_ctrl_pa`
- `/opt/mellanox/doca/samples/doca_cc/cc_ctrl_path_common.c`

- `/opt/mellanox/doca/samples/doca_cc/cc_ctrl_path_common.h`

DOCA Comm Channel Data Path Client/Server

> **Note**
>
> `doca_cc_data_path_high_speed_server` should be run on the DPU side and should be started before `doca_cc_data_path_high_speed_client` is started on the host

This sample sets up a client server connection between host and DPU.

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_cc/cc_data_path_high_speed_client`

- `/opt/mellanox/doca/samples/doca_cc/cc_data_path_high_speedclient/`
- `/opt/mellanox/doca/samples/doca_cc/cc_data_path_high_speedserver/`
- `/opt/mellanox/doca/samples/doca_cc/cc_data_path_high_speedserver/`
- `/opt/mellanox/doca/samples/doca_cc/cc_data_path_high_speed_common`
- `/opt/mellanox/doca/samples/doca_cc/cc_data_path_high_speed_common`

DOCA Comm Channel – Deprecated

This guide provides instructions on how to use the DOCA Comm Channel API.

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.

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.

API

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()`.

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.

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.

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.

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 e xtracts 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.

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.

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.

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.

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 e xtracts 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()

Tip

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.

> **Note**
>
> 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.

Info

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.

Info

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.

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.

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.

Limitations

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()`

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()`.

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.

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.

Usage

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.

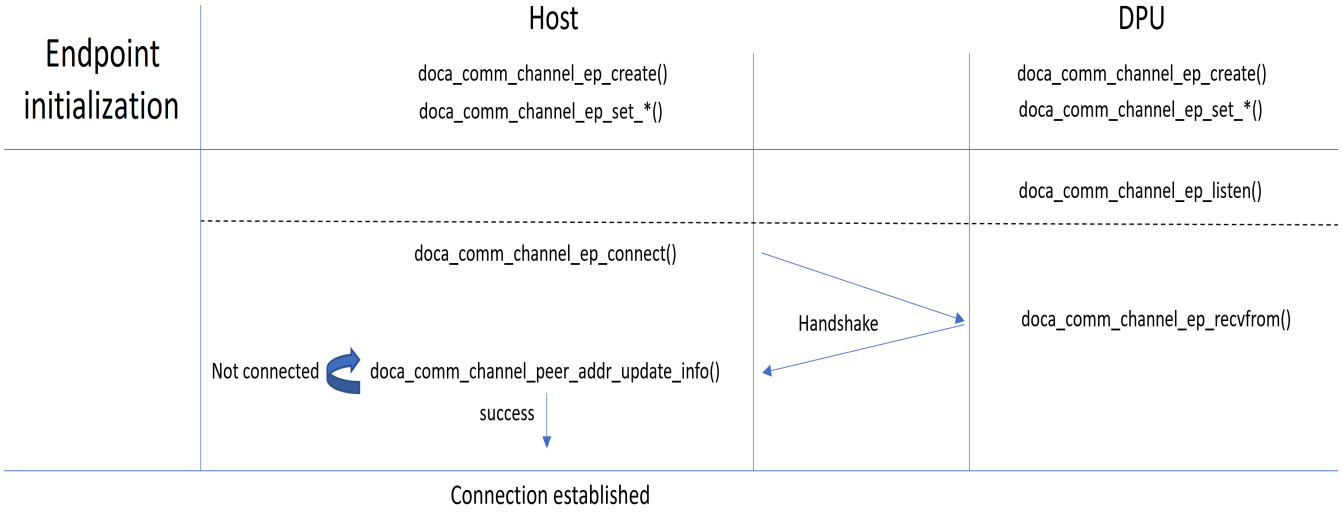
Endpoint Initialization

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".

Connection Flow

The following diagram illustrates the process of establishing a connection between the host and a service.



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()".

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()".

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".

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()`.

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`

Statistic Function	Returns
`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".

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 connect `peer_addrs`
`doca_comm_channel_ep_get_pending_connections()`	Number of pending connec for the service. If there are connections, `doca_comm_channel_ep` should be called to handle

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()".

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.

DOCA Comm Channel Samples

This section provides Comm Channel sample implementation on top of the BlueField DPU.

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
```

> **Note**
>
> 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
```

Note

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
```

Samples

Comm Channel Server

> **Note**
>
> 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/`
- `/opt/mellanox/doca/samples/doca_comm_channel/comm_channel_server/`
- `/opt/mellanox/doca/samples/doca_comm_channel/comm_channel_server/`

Comm Channel Client

> **Note**
>
> 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/`
- `/opt/mellanox/doca/samples/doca_comm_channel/comm_channel_client/`
- `/opt/mellanox/doca/samples/doca_comm_channel/comm_channel_client/`

DOCA RDMA

This guide provides an overview and configuration instructions for the DOCA RDMA API.

Introduction

> **Note**
>
> This library is currently supported at beta level only.

> **Note**
>
> DOCA RDMA supports local networks only (i.e., cross-network or routing is not supported).

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.

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

Environment

DOCA RDMA-based applications can run either on the host machine or on the NVIDIA® BlueField® DPU target.

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

Objects

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.

> **Note**
>
> The device must stay valid until the RDMA instance is destroyed.

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.

  > **Info**
  >
  > 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.

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.

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.

Configurations

The context can be configured to match the application use case.

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?[a]
	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		

Note

[a] Refers to the peer. A side that only submits tasks is never required to export an mmap.

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.

> **Info**
>
> 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.

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` fo r 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()`.

Both Bluefield-2 and Bluefield-3 devices are supported, any `doca_dev` is supported.

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

> **Info**
>
> 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".

Exporting and Connecting RDMA

To establish the communication between the peers and allow the execution of different DOCA RDMA tasks, the RDMA instances must be connected.

This step should be executed after `doca_ctx_start()` is called and when the context is in Starting state.

> **Info**
>
> Refer to State Machine section for more information.

Connecting the RDMA instances is performed by e xporting 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:



Execution Phase

This section describes execution on CPU using DOCA Core Progress Engine (PE). For additional execution environments refer to section "Alternative Datapath Options".

Tasks

DOCA RDMA exposes asynchronous tasks that leverage the DPU hardware according to the DOCA Core architecture. See DOCA Core Task.

> **Note**
>
> 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 the data to be written to the remote address.

> **Note**
>
> All buffers used in DOCA RDMA tasks must remain valid until the task result is retrieved.

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.

Configuration

Description	API to Set the Configuration	API to Query Sup
Enable the task	`doca_rdma_task_receive_set_conf`	`doca_rdma_cap`
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`

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.

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

Name	Description	Notes
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 Successful Completion

After the task completes successfully, the following happens:

- T he received data is copied to the tail segment extending the original data segment
- The data length is increased by the received data length

Task Failed Completion

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.

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 device's `max_message_size` or 2GB (whichever is lower)
- 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

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.

Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_send_set_conf`	`doca_rdma_cap_task_se`
Number of tasks	`doca_rdma_task_send_set_conf`	–
Source buffer list length	`doca_rdma_set_max_send_buf_list_len` (a)	`doca_rdma_cap_get_max`

Note

(a) This configuration affects other tasks as well.

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 purpose of the DOCA RDMA send task, the length of each buffer is considered as its data length

Output

Common output as described in DOCA Core Task.

Task Successful Completion

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 Failed Completion

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

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 or 2GB (whichever is lower)
- 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

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.

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_se`
Number of tasks	`doca_rdma_task_send_imm_set_conf`	–
Source buffer list length	`doca_rdma_set_max_send_buf_list_len` (a)	`doca_rdma_cap_get_max`

Note

[a] This configuration affects other tasks as well.

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 purpose of 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.

Output

Common output as described in DOCA Core Task.

Task Successful Completion

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 Failed Completion

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

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 or 2GB (whichever is lower)
- 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

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).

Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_read_set_conf`	`doca_rdma_cap_task_re`
Number of tasks	`doca_rdma_task_read_set_conf`	–
Destination buffer list length	`doca_rdma_set_max_send_buf_list_len` (a)	`doca_rdma_cap_get_max`

Note

(a) This configuration affects other tasks as well.

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.

Name	Description	Notes
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

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 Successful Completion

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 Failed Completion

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.

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 or 2GB (whichever is lower)
- 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

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).

Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_write_set_conf`	`doca_rdma_cap_task_wr`
Number of tasks	`doca_rdma_task_write_set_conf`	–
Source buffer list length	`doca_rdma_set_max_send_buf_list_len` (a)	`doca_rdma_cap_get_max`

Note

[a] This configuration affects other tasks as well.

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

Output

Common output as described in DOCA Core Task.

Task Successful Completion

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 Failed Completion

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.

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 or 2GB (whichever is lower)
- 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

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).

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_wr`
Number of tasks	`doca_rdma_task_write_imm_set_conf`	–
Source buffer list length	`doca_rdma_set_max_send_buf_list_len` (a)	`doca_rdma_cap_get_max`

Note

[a] This configuration affects other tasks as well.

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

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
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

Output

Common output as described in DOCA Core Task.

Task Successful Completion

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 Failed Completion

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.

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 or 2GB (whichever is lower)
- 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

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.

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`
Number of tasks	`doca_rdma_task_atomic_cmp_swp_set_conf`	–

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

Output

Common output as described in DOCA Core Task.

Task Successful Completion

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 Failed Completion

If the task fails midway:

- The context is stopped and the task should be freed by the user

Limitations

- Task buffers must remain valid until task is completed
- Other limitations are described in DOCA Core Task

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.

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_ta`
Number of tasks	`doca_rdma_task_atomic_fetch_add_set_conf`	–

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

Output

Common output as described in DOCA Core Task.

Task Successful Completion

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 Failed Completion

If the task fails midway:

- The context is stopped and the task should be freed by the user

Limitations

- Task buffers must remain valid until task is completed
- Other limitations are described in DOCA Core Task

Get Remote Sync Event Task

This task should be submitted when wishing to get the value of a remote sync event.

Configuration

Description	API to Set the Configuration	API to Qu
Enable the task	`doca_rdma_task_remote_net_sync_event_get_set_conf`	`doca_rd`
Number of tasks	`doca_rdma_task_remote_net_sync_event_get_set_conf`	–

Description	API to Set the Configuration	API to Qu
Destination buffer list length	`doca_rdma_set_max_send_buf_list_len`[(a)]	`doca_rd`

Note

[(a)] This configuration affects other tasks as well.

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

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 Successful Completion

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 Failed Completion

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.

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

Set Remote Sync Event Task

This task should be submitted when wishing to set a remote sync event to a given value.

Configuration

Description	API to Set the Configuration	A
Enable the task	`doca_rdma_task_remote_net_sync_event_notify_set_set_conf`	
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`[(a)]	

> **Note**
>
> [(a)] This configuration affects other tasks as well.

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	

Output

Common output as described in DOCA Core Task.

Task Successful Completion

After the task completes successfully, the following happens:

- The remote sync event value is set to the data in the source buffer

Task Failed Completion

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

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

Add Remote Sync Event Task

This task should be submitted when wishing to atomically increase a remote sync event by a given value.

Configuration

Description	API to Set the Configuration	Al
Enable the task	`doca_rdma_task_emote_net_sync_event_notify_add_set_conf`	`d`
Number of tasks	`doca_rdma_task_emote_net_sync_event_notify_add_set_conf`	–

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

Output

Common output as described in DOCA Core Task.

Task Successful Completion

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 Failed Completion

If the task fails midway:

- The context is stopped and the task should be freed by the user

Limitations

- Result buffer must remain valid until task is completed
- Other limitations are described in DOCA Core Task

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.

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.

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

Starting

In this state, it is expected that application:

1. Connects the RDMA instances on both peers. Refer to section "Exporting and Connecting RDMA" for more information.
2. After connecting the RDMA instance, call progress to allow transition to next state

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

Running

In this state, it is expected that application:

1. Allocates and submit tasks
2. Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocating previously configured task
- Submitting an allocated task
- Calling stop

It is possible to reach this state as follows:

Previous State	Transition Action
Starting	Call progress until context state transitions

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

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

Alternative Datapath Options

DOCA RDMA allows data path to be run on DPA.

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.

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.

DOCA RDMA Samples

These samples illustrate how to use the DOCA RDMA API to execute DOCA RDMA operations.

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
```

> **Info**
>
> 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
```

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.

Note

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:
    1. Locating and opening a device. The chosen device is one that supports the task relevant for the sample. If the sample requires no task, any device may be chosen.
    2. Creating a local MMAP and configuring it (including setting the MMAP memory range and relevant permissions)
    3. Creating a DOCA Progress Engine (PE)
    4. Creating an RDMA instance and configuring it (including setting the relevant permissions)
    5. Connecting the RDMA context to the PE
2. Sample-specific configurations:
    1. Configuring the tasks relevant to the sample, if any. Including:
        1. Setting the number of tasks for each task type.
        2. Setting callback functions for each task type, with the following logic:
            1. Successful completion callback:
                1. Verifying the data received from the remote, if any, is valid.
                2. Printing the transferred data.
                3. Freeing the task and task-specific resources (such as source/destination buffers).

4. Saving the result of the callback (If there was an error in steps a. and b., the relevant error value is saved).

5. Decreasing the number of remaining tasks and stopping the context once it reaches 0.

2. Failed completion callback:

1. Saving the task result.

2. Freeing the task and task specific resources (such as source/destination buffers).

3. Decreasing the number of remaining tasks and stopping the context once it reaches 0.

2. 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.

> **Note**
>
> During this step, the user is responsible for copying the descriptors between the two peers.

> **Note**
>
> 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.
- In others, prepare and submit a task:
    1. If needed, create an mmap from the received exported mmap descriptor, passed from the peer.
    2. Request the required buffers from the buffer inventory.
    3. Allocate and initiate the required task, together with setting the number of remaining tasks parameter as the task's user data.
    4. 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.

> **Info**
>
> After starting the context, the state change callback function is called by the PE which executes the relevant steps.

> **Info**
>
> 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 successful run in which all tasks have been successfully completed, or due to a fatal error.

7. Cleaning up the resources.

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 is 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_rea`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_read_requester/rdma_rea`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_read_requester/meson.bu`

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_rea`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_read_responder/rdma_rea`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_read_responder/meson.bu`

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 is 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_wr`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_requester/rdma_wr`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_requester/meson.b`

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_wr`

- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_responder/rdma_wr`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_responder/meson.b`

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_request`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_request`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_request`

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:

    1. Checking the result opcode, to verify that the receive task has completed after receiving a write with immediate request.

    2. 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_respond`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_respond`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_respond`

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_sa`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_receive/rdma_receive_ma`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_receive/meson.build`

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_sen`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_send_immediate/rdma_sen`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_send_immediate/meson.bu`

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:
    1. Checking the result opcode, to verify that the receive task has completed after receiving a sent message with an immediate.
    2. 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_`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_receive_immediate/rdma_`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_receive_immediate/meson`

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 includes:

1. Creating a remote sync event from the peer's exported sync event.
2. Configuring "sync event notify set" and "sync event get" tasks.
3. Notifying the sync event from the remote peer side.
4. Reading the sync event from the remote peer until signaled by the local peer.
5. Notifying for completion by setting the sync event from the remote side.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_requester/rd`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_requester/rd`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_requester/me`

RDMA Remote Sync Event Responder

The sample logic includes:

1. Creating a local sync event.
2. Exporting the local sync event to the remote memory to allow the peer to create a remote handle.
3. Waiting for the sync event to be signaled from the remote side.
4. Notifying the sync event from the local side.
5. On the local side, waiting for completion notification from the remote side.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_responder/rd`

- `/opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_responder/rd`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_responder/me`

DOCA Ethernet

This guide provides an overview and configuration instructions for the DOCA ETH API.

Introduction

> **Note**
>
> 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.

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)

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.

Architecture

DOCA ETH is comprised of two parts: DOCA ETH RXQ and DOCA ETH TXQ.

DOCA ETH RXQ

Operating Modes

DOCA ETH RXQ can operate in the three modes, each exposing a slightly different control/datapath.

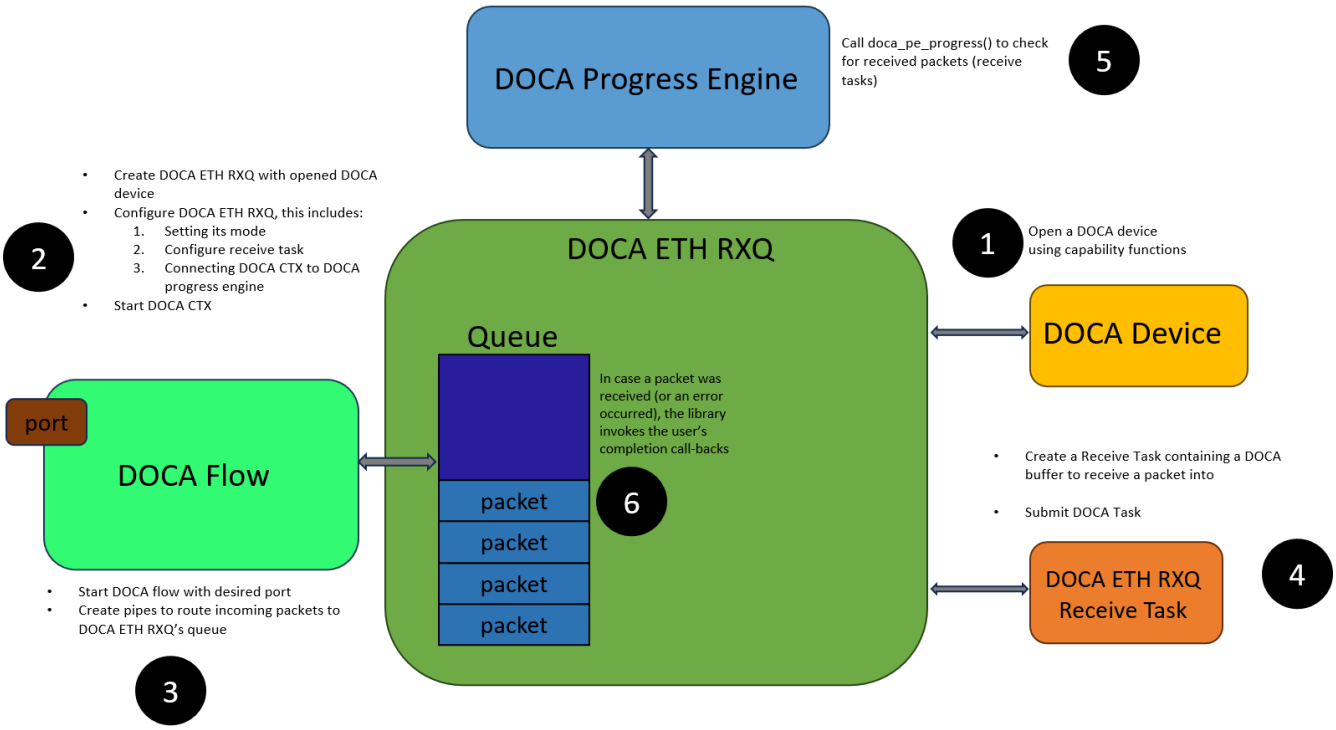
Regular Receive

> **Info**
>
> 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



Cyclic Receive

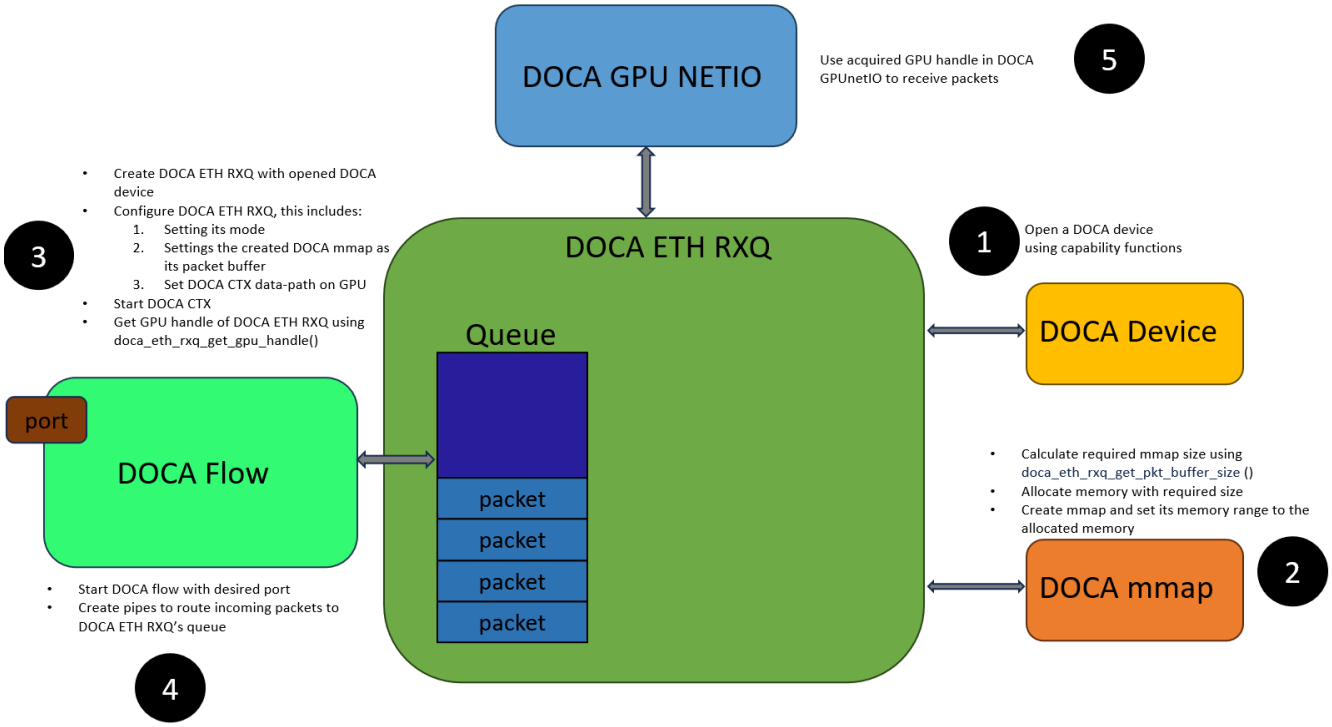
> **Info**
>
> This mode is supported only for GPU datapath.

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_buffer()`). The buffer should be large enough to avoid packet loss (see `doca_eth_rxq_get_pkt_buffer_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



Managed Memory Pool Receive

> **Info**
>
> This mode is supported only for CPU datapath.

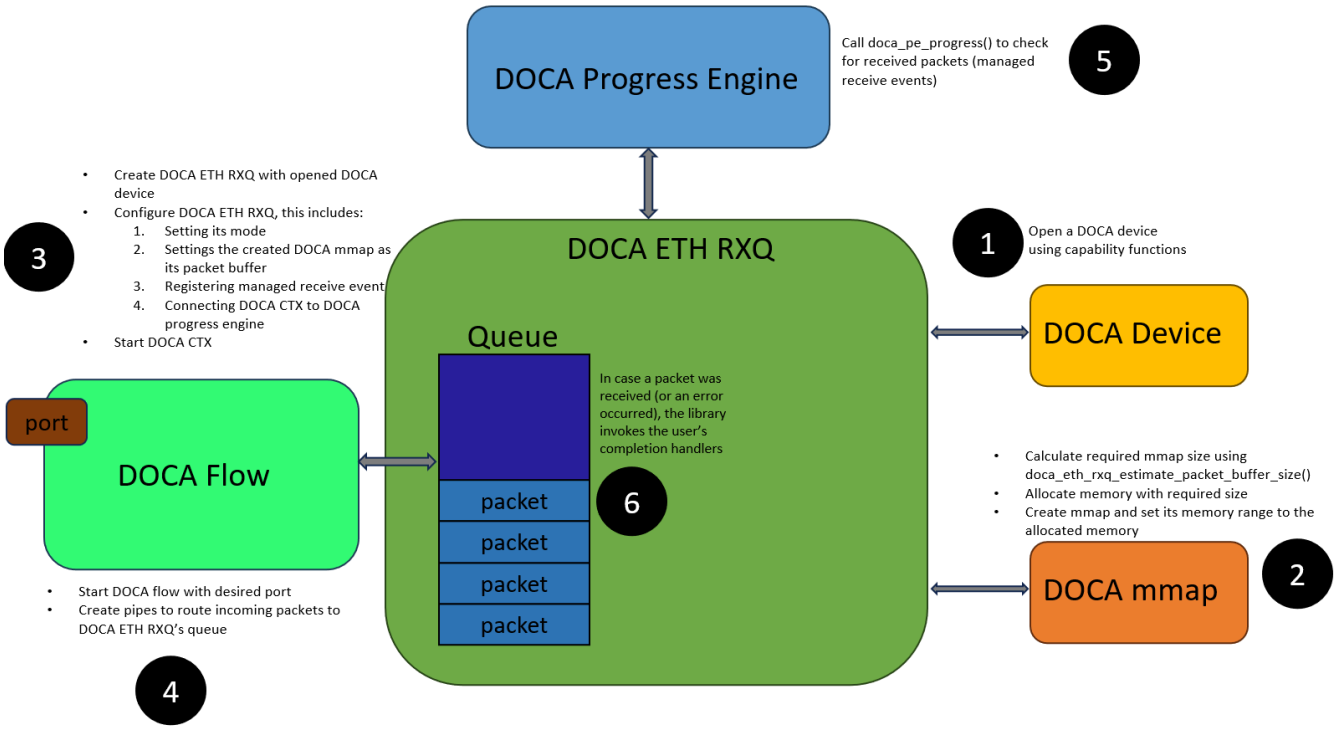
In this mode, the library uses various optimizations to manage the packet buffers. Packets acquired by the user cannot be overwritten by the library unless explicitly freed by the application. Thus, if the application does not release the packet buffers fast enough, the library would run out of memory 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_buffer()`). The buffer should be large enough to avoid packet loss (see `doca_eth_rxq_estimate_packet_buffer_size()`).

The application uses this mode if:

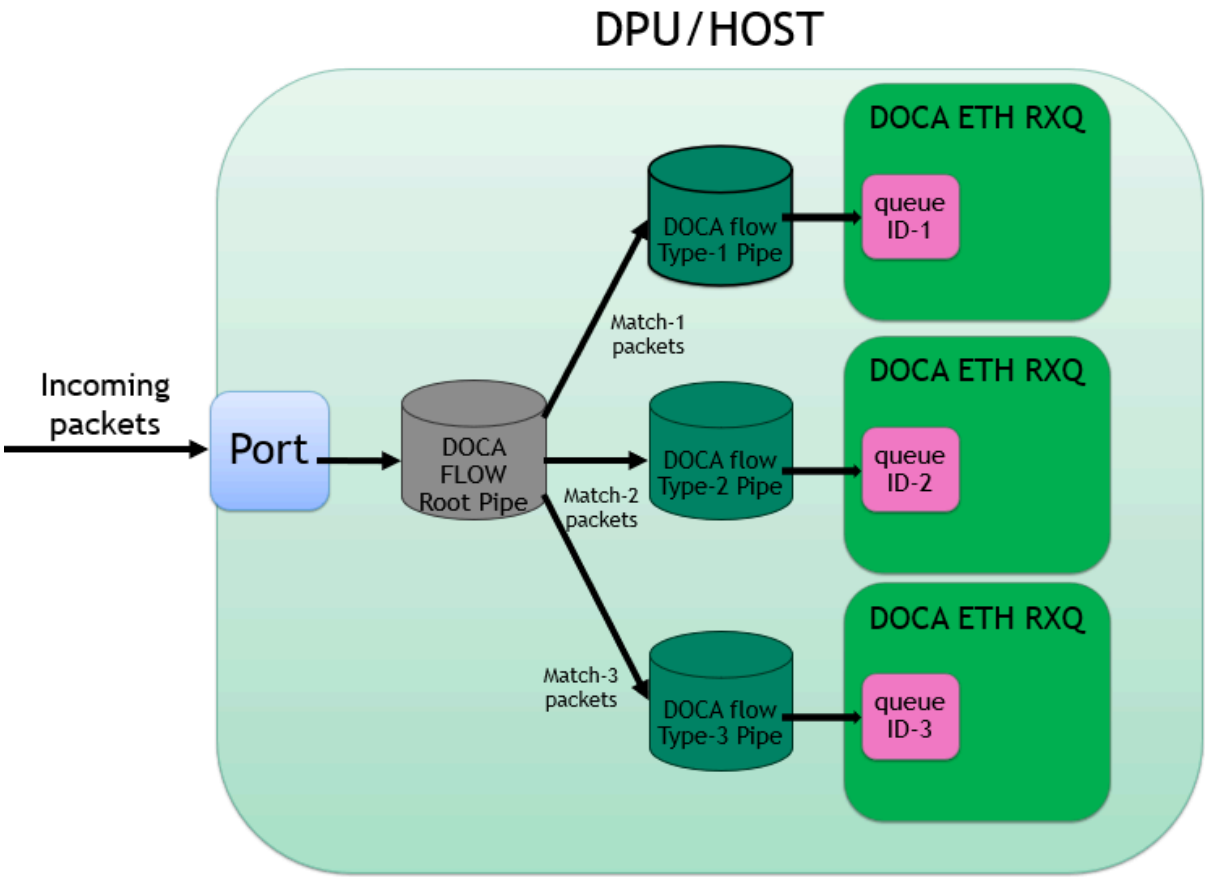
- It wants to run on CPU
- It has a deterministic packet processing time, where a packet is guaranteed to be processed before the library runs out of memory and packets start dropping
- It wants to forward the received packets to other DOCA libraries
- It wants best performance



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.

DOCA ETH TXQ

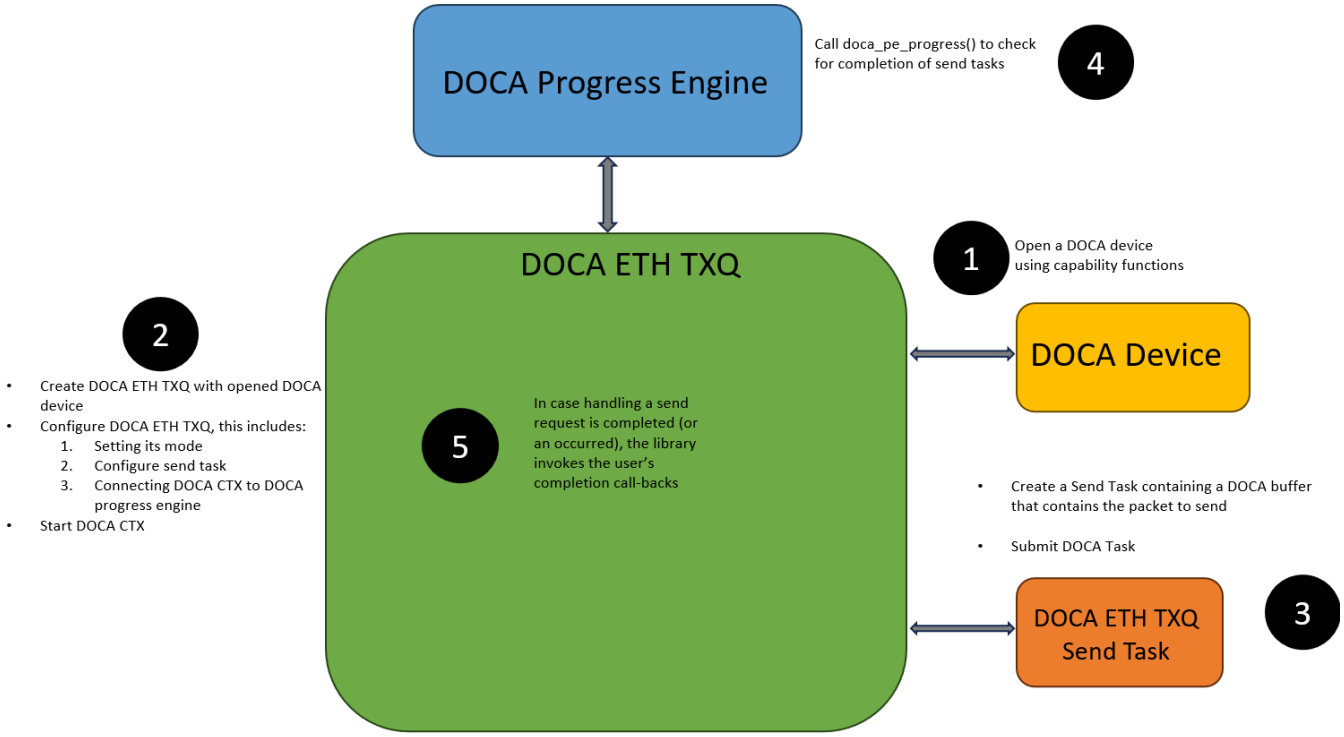
Operating Modes

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.



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()`.

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).

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.

> **Note**
>
> DOCA ETH in GPU datapath does not need to be associated with a DOCA PE (since the datapath is not on the CPU).

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.

Mandatory Configurations

These configurations are mandatory and must be set by the application before attempting to start the context.

DOCA ETH RXQ

- At least one task/event type must be configured. See configuration of Tasks and/or Events.
- 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 opon 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_buffer()`)
- In case of a GPU datapath, A DOCA GPU sub-device must be provided using `doca_ctx_set_datapath_on_gpu()`

DOCA ETH TXQ

- At least one task type must be configured. See configuration of Tasks and/or Events.
- 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
- If configuring send LSO tasks, max LSO header size and MSS must be provided at configuration of send LSO tasks (See `doca_eth_txq_task_lso_send_set_conf()`)
- In case of a GPU datapath, a DOCA GPU sub-device must be provided using `doca_ctx_set_datapath_on_gpu()`

Optional Configurations

The following configurations are optional. If they are not set then a default value will be used.

DOCA ETH RXQ

Setting the DOCA ETH RXQ mode is optional (See `doca_eth_rxq_set_type()`). The default type is Regular Receive.

DOCA ETH TXQ

Setting L3/L4 offload checksum (enabling/disabling hardware to calculate L3/L4 checksum values and write them into the specific field in the transmitted packet) is optional (see `doca_eth_txq_set_l3_chksum_offload()` / `doca_eth_txq_set_l4_chksum_offload()`). The default value is false (offloading is disabled).

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_get_type_supported()` and `doca_eth_txq_get_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)

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
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.

Execution Phase

This section describes execution on CPU (unless stated otherwise) using DOCA Core Progress Engine.

> **Note**
>
> For information regarding GPU datapath, see DOCA GPUNetIO.

Tasks

DOCA ETH exposes asynchronous tasks that leverage the DPU hardware according to the DOCA Core architecture. See DOCA Core Task.

DOCA ETH RXQ

Receive Task

This task allows receiving packets from a `doca_dev` .

Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_eth_rxq_task_recv_set_conf()`	`doca_eth_txq_get` Regular Receive mode
Number of tasks	`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_get`
Maximal packet size	`doca_eth_rxq_create()`	`doca_eth_rxq_get`

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

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 Successful Completion

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 Failed Completion

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

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.

DOCA ETH TXQ

Send Task

This task allows sending packets from a `doca_dev` .

Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_eth_txq_task_send_set_conf()`	`doca_eth_txq_get` Regular Send mode
Number of tasks	`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_get`

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

Output

Common output as described in DOCA Core Task.

Task Successful Completion

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 Failed Completion

If the task fails midway:

- The context enters stopping state
- The packet buffer `doca_buf` object is not modified

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).

Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_eth_txq_task_lso_send_set_conf()`	`doca_eth_txq_get` Regular Send mode

Description	API to Set the Configuration	API to Query Support
Maximal LSO message size	-	`doca_eth_txq_get`
Number of tasks	`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_get`

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`
MSS	Max segment size of each sent packet (after splitting)	This value is set for the all the LSO send tasks in `doca_eth_txq_task_lso_send_set_conf()`

Output

Common output as described in DOCA Core Task.

Task Successful Completion

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 Failed Completion

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

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
- All limitations described in DOCA Core Task

Events

DOCA ETH exposes asynchronous events to notify about changes that happen asynchronous, 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:

DOCA ETH RXQ

Managed Receive Event

This event allows receiving packets from a `doca_dev` (without the application managing the memory the packets are written too).

Configuration

Description	API to Set the Configuration	API to Query Supp
Register to the event	`doca_eth_rxq_event_managed_recv_register()`	`doca_eth_rxq_g` checking support f Pool Receive mode

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

> **Note**
>
> 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

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.

Configuration

Description	API to Set the Configuration	API to Query Suppor
Register to the event	`doca_eth_txq_gpu_event_error_send_packet_register()`	Always suppor

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.

Configuration

Description	API to Set the Configuration	API to Query Suppo
Register to the event	`doca_eth_txq_gpu_event_notify_send_packet_register()`	Alway suppo

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

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.

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

Starting

This state cannot be reached.

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

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

Alternative Datapath Options

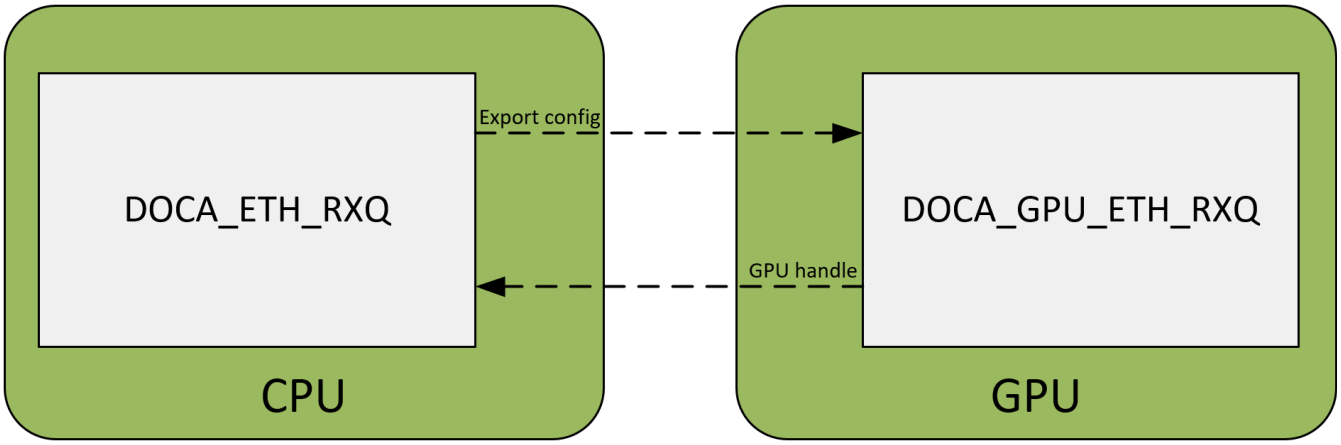
In addition to the CPU datapath (mentioned in Execution Phase), DOCA ETH supports running on GPU datapath. This allows applications to release the CPU from datapath management and allow low latency GPU processing of network traffic.

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.

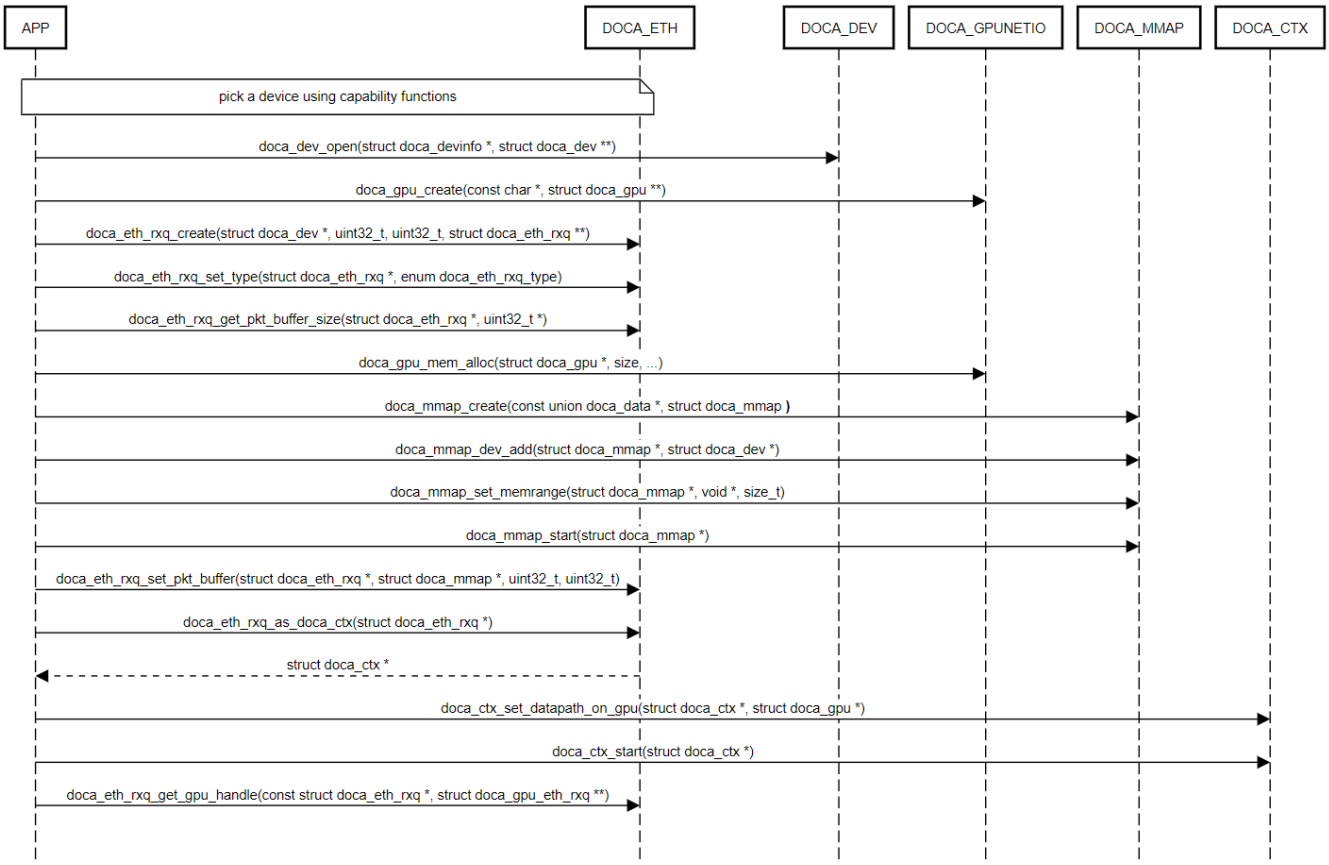
> **Note**
>
> The datapath cannot be managed concurrently for the GPU and the CPU.

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.



For more information regarding the GPU datapath see DOCA GPUNetIO.

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

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_f
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=DE

   --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
```

Samples

> **Note**
>
> The following samples are for the CPU datapath. For GPU datapath samples, see DOCA GPUNetIO.

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 element in 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 using doca_pe_progress().
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/`
- `/opt/mellanox/doca/samples/doca_eth/eth_txq_send_ethernet_frames/`
- `/opt/mellanox/doca/samples/doca_eth/eth_txq_send_ethernet_frames/`

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 element in 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 using doca_pe_progress().

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_fra`
- `/opt/mellanox/doca/samples/doca_eth/eth_txq_lso_send_ethernet_fra`
- `/opt/mellanox/doca/samples/doca_eth/eth_txq_lso_send_ethernet_fra`

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. Initialize DPDK.
2. Locating DOCA device.
3. Probing DOCA device and finding its port ID.
4. Starting DPDK ETH device.
5. Initializing the required DOCA core structures.
6. Populating DOCA memory map with one buffer to the packet's data.
7. Allocating element in DOCA buffer inventory for each buffer.
8. Initializing and Configuring DOCA ETH RXQ context.
9. Starting the DOCA ETH RXQ context.
10. Initializing DOCA Flow.

11. Starting DOCA Flow.
12. Creating a pipe connecting to DOCA ETH RXQ's RX queue.
13. Allocating DOCA ETH RXQ receive task.
14. Submitting DOCA ETH RXQ receive task into progress engine.
15. Retrieving DOCA ETH RXQ receive task from the progress engine using doca_pe_progress().
16. Handling the completed task using the provided callback.
17. Stopping DOCA Flow.
18. Destroying DOCA Flow.
19. Stopping the DOCA ETH RXQ context.
20. Destroying DOCA ETH RXQ context.
21. Destroying all DOCA core structures.
22. Destroying DPDK.

Reference:

- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_regular_receive/eth_r`
- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_regular_receive/eth_r`
- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_regular_receive/meson`

ETH RXQ Managed Receive

This sample illustrates how to receive a packet using DOCA ETH RXQ in Managed Memory Pool Receive mode .

The sample logic includes:

1. Initialize DPDK.

2. Locating DOCA device.
3. Probing DOCA device and finding its port ID.
4. Starting DPDK ETH device.
5. Initializing the required DOCA core structures.
6. Calculating the size of the required buffer to receiving the packets from DOCA ETH RXQ.
7. Populating DOCA memory map with a packets buffer.
8. Initializing and Configuring DOCA ETH RXQ context.
9. Registering DOCA ETH RXQ managed receive event.
10. Starting the DOCA ETH RXQ context.
11. Initializing DOCA Flow.
12. Starting DOCA Flow.
13. Creating a pipe connecting to DOCA ETH RXQ's RX queue.
14. Retrieving DOCA ETH RXQ managed receive events from the progress engine using doca_pe_progress().
15. Handling the completed events using the provided callback.
16. Stopping DOCA Flow.
17. Destroying DOCA Flow.
18. Stopping the DOCA ETH RXQ context.
19. Destroying DOCA ETH RXQ context.
20. Destroying all DOCA core structures.
21. Destroying DPDK.

Reference:

- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_managed_mempool_recei`

- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_managed_mempool_recei`
- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_managed_mempool_recei`

DOCA GPUNetIO

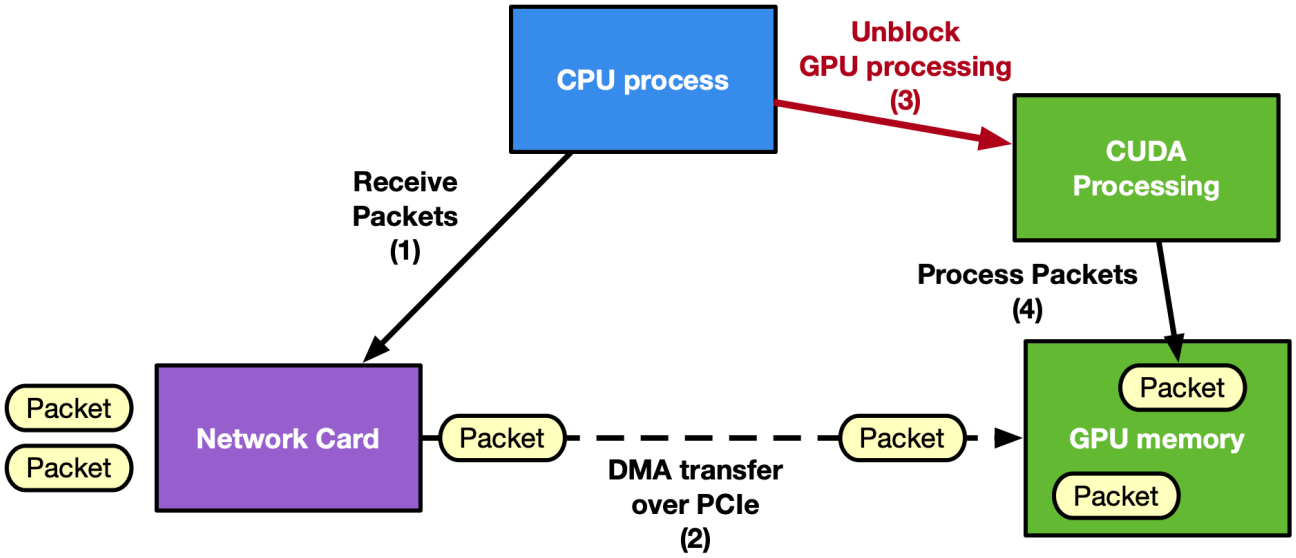
This document provides an overview and configuration instructions for DOCA GPUNetIO API.

Introduction

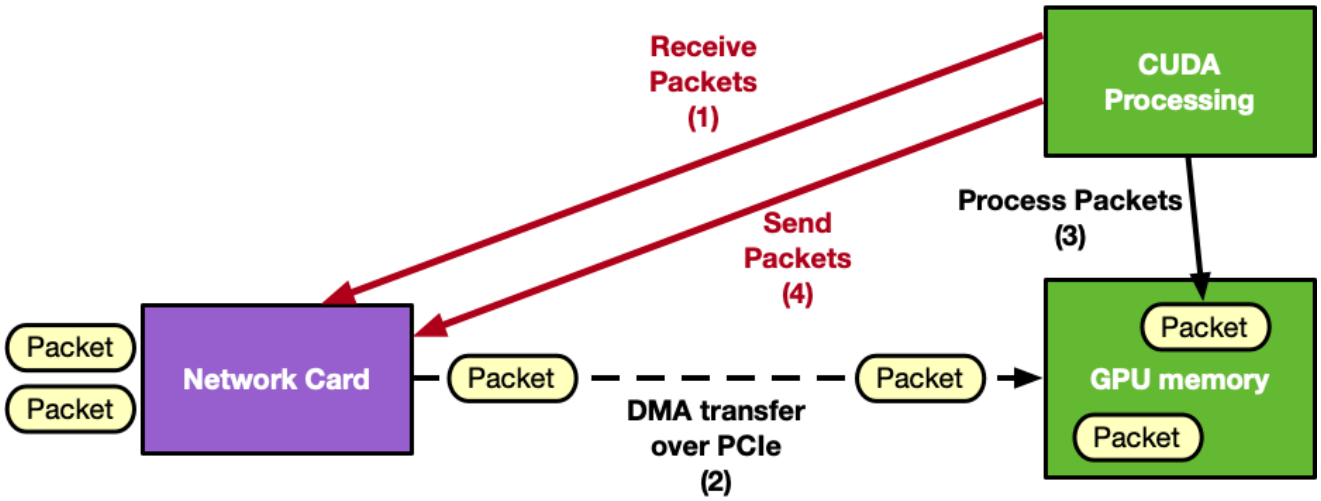
Real-time GPU processing of network packets is a technique useful for application domains involving signal processing, network security, information gathering, input reconstruction, and more. These applications involve the CPU in the critical path (CPU-centric approach) to coordinate the network card (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 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 Network (GDAKIN) communications – a CUDA kernel can invoke GPUNetIO device functions to receive or send, directly interacting with the NIC
    - CPU intervention is not needed in the application critical path
- GPUDirect RDMA – receive packets directly into a contiguous GPU memory area
- Semaphores – provide a standardized I/O communication protocol between the receiving entity and the CUDA kernel real-time packet processing

- Smart memory allocation – allocate aligned GPU memory buffers exposing them to direct CPU access
    - Combination of CUDA and DPDK gpudev library (which requires the GDRCopy library) already embedded in the DPDK released with DOCA
- Ethernet protocol management on GPU

Morpheus and Aerial 5G SDK are examples of NVIDIA applications actively using DOCA GPUNetIO.

For a deep dive into the technology and motivations, please refer to the NVIDIA Blog post Inline GPU Packet Processing with NVIDIA DOCA GPUNetIO. A second NVIDIA blog post Realizing the Power of Real-Time Network Processing with NVIDIA DOCA GPUNetIO has been published to provide more example use-cases where DOCA GPUNetIO has been useful to improve the execution.

Prerequisites

DOCA GPUNetIO requires a properly configured environment. The following subsections describe the required setup.

Info

Currently, DOCA GPUNetIO is included only in DOCA for Host packages for Ubuntu 18.04, 20.04, 22.04 and RHEL/CentOS 7.6, 8.0, 8.2 and 8.6 with CUDA 12.1 or newer.

Note

After installing the DOCA package on the system, to install all DOCA GPUNetIO components, run:

```
apt install -y doca-gpu doca-gpu-dev
```

Note

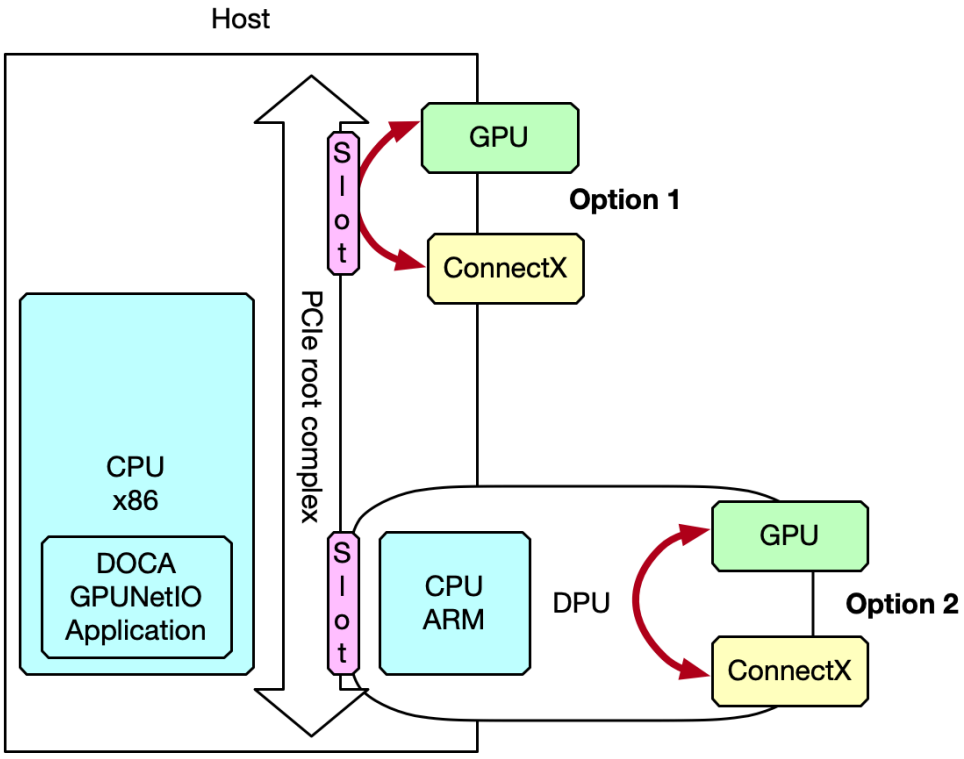
DOCA GPUNetIO has been tested on bare-metal and in docker but never in a virtualized environment. Using KVM is discouraged for now.

Hardware Topology

Internal hardware topology of the system should be GPUDirect-RDMA-friendly to maximize the internal throughput between the GPU and the NIC.

Assuming the application is running on the host's CPU cores, there must be a dedicated PCIe connection between the GPU and the NIC which can be implemented in two ways:

- Connecting an additional PCIe switch to a PCIe slot in the host system bus
- Connecting a DPU converged card exposing the GPU and NIC to the host



You may check the topology of your system using `lspci -tvvv` or `nvidia-smi topo -m`.

Option 1: Network Card in Ethernet Mode

> **Note**
>
> NVIDIA® ConnectX® firmware must be 22.36.1010 or later. It is highly recommended to only use NVIDIA® ConnectX®-6 Dx based adapter cards and later.

DOCA GPUNetIO allows a CUDA kernel to control the NIC when dealing with Ethernet protocol. For this reason, the ConnectX on your system must be set in Ethernet mode.

To do that, follow these steps:

1. Start MST and check status.

```
# 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).

```
mlxconfig -d /dev/mst/mt4125_pciconf0 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 /dev/mst/mt4125_pciconf0 --yes set
ACCURATE_TX_SCHEDULER=1 REAL_TIME_CLOCK_ENABLE=1
```

3. Perform graceful shutdown of the Arm OS.
4. Cold reboot.

```
ipmitool power cycle
```

Option 2: DPU Converged Card

> **Note**
>
> DPU firmware must be 24.35.2000 or newer.

To expose and use the GPU and the NIC on the DPU converged card from an application running on the host, configure the DPU to operate in NIC mode.

To do that, follow these steps:

1. Start MST and check status.

```
# 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 DPU converged card to host. For BlueField-2 the offset should be 4, for BlueField-3 the offset should be 8.

```
sudo mlxconfig -d /dev/mst/mt41686_pciconf0 --yes s
PCI_DOWNSTREAM_PORT_OWNER[4]=0x0
```

3. Configure the DPU to Ethernet mode and enable Accurate Send Scheduling (if required on the send side). Also set the DPU in NIC mode.

```
sudo mlxconfig -d /dev/mst/mt41686_pciconf0 --yes set
LINK_TYPE_P1=2 LINK_TYPE_P2=2
sudo mlxconfig -d /dev/mst/mt41686_pciconf0 --yes set
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 /dev/mst/mt41686_pciconf0 --yes set
ACCURATE_TX_SCHEDULER=1 REAL_TIME_CLOCK_ENABLE=1
```

4. Perform graceful shutdown of the Arm OS.

5. Cold reboot.

```
ipmitool power cycle
```

6. Verify configuration.

```
sudo mlxconfig -d /dev/mst/mt41686_pciconf0 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)
```

PCIe Configuration

On some x86 systems, the Access Control Services (ACS) must be disabled to ensure direct communication between the NIC and the GPU, whether they reside on the same DPU converged card on on different PCIe slots in the system. ACS control must be disabled via BIOS (e.g., Supermicro or HPE) and this is the preferred solution. Alternatively, it is also possible to disable it via command line but it may not be as effective as the BIOS option. Assuming the system topology Option 2 with a DPU converged card like the following:

```
$ 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:

```
setpci -s b2:00.0 ECAP_ACS+6.w=0:fc
```

To verify that the setting has been applied correctly:

```
$ 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:

```
$ sudo vim /etc/default/grub
# GRUB_CMDLINE_LINUX_DEFAULT="default_hugepagesz=1G hugepagesz=1G hugepages=4
intel_iommu=off"
$ sudo update-grub
$ sudo reboot
```

GPU Configuration

On the host, CUDA Toolkit 12.1 or newer must be installed. It is also recommended to enable persistence mode to decrease initial application latency `nvidia-smi -pm 1`.

To allow the CPU to directly access the GPU memory without the need for CUDA API, DPDK and DOCA require the GDRCopy kernel module to be installed on the system:

```
# 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-12/bin:${PATH}"
export LD_LIBRARY_PATH="/usr/local/cuda-12/lib:/usr/local/cuda-12/lib64:${LD_LIBRARY_PATH}"
export CPATH="$(echo /usr/local/cuda-12/targets/{x86_64,sbsa}-linux/include | sed 's/ /:/'):${CPATH}"
```

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)

> **Note**
>
> Installing DOCA on kernel 6.2 to enable the `dmabuf` is experimental.

An example can be found in the DOCA GPU Packet Processing application:

```
/* 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.

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`:

```
$ 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:

```
$ 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.

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 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.

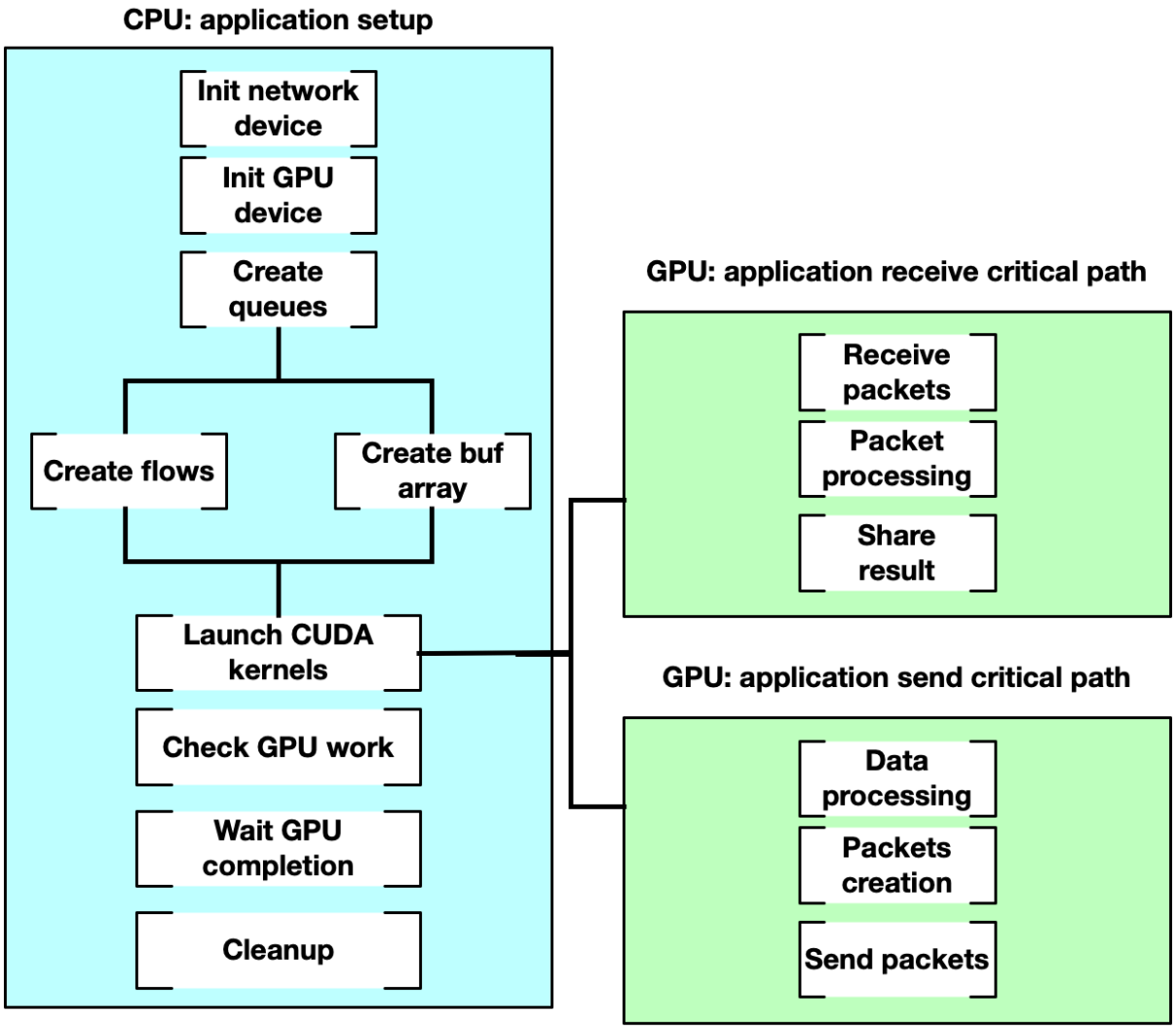
For this reason, DOCA GPUNetIO is composed of two libraries:

- `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

> **Note**
>
> The pkgconfig file for the DOCA GPUNetIO shared library is `doca-gpu.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.

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

This section lists the main functions of DOCA GPUNetIO. To better understand their usage, refer to section "Building Blocks" which includes several code examples.

Tip

To better understand structures, objects, and functions related to Ethernet send and receive, please refer to the DOCA Ethernet.

Tip

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` :

```
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` :

```
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, ...);
```

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.

CPU functions

In this section there is the list of DOCA GPUNetIO functions that can be used on the CPU only.

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,
};
```

Note

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).

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`.

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

> ⚠ **Warning**
>
> Make sure to use the right pointer on the right device! I f an application tries to access the memory using the `memptr_gpu` address from the CPU, a segmentation fault will result.

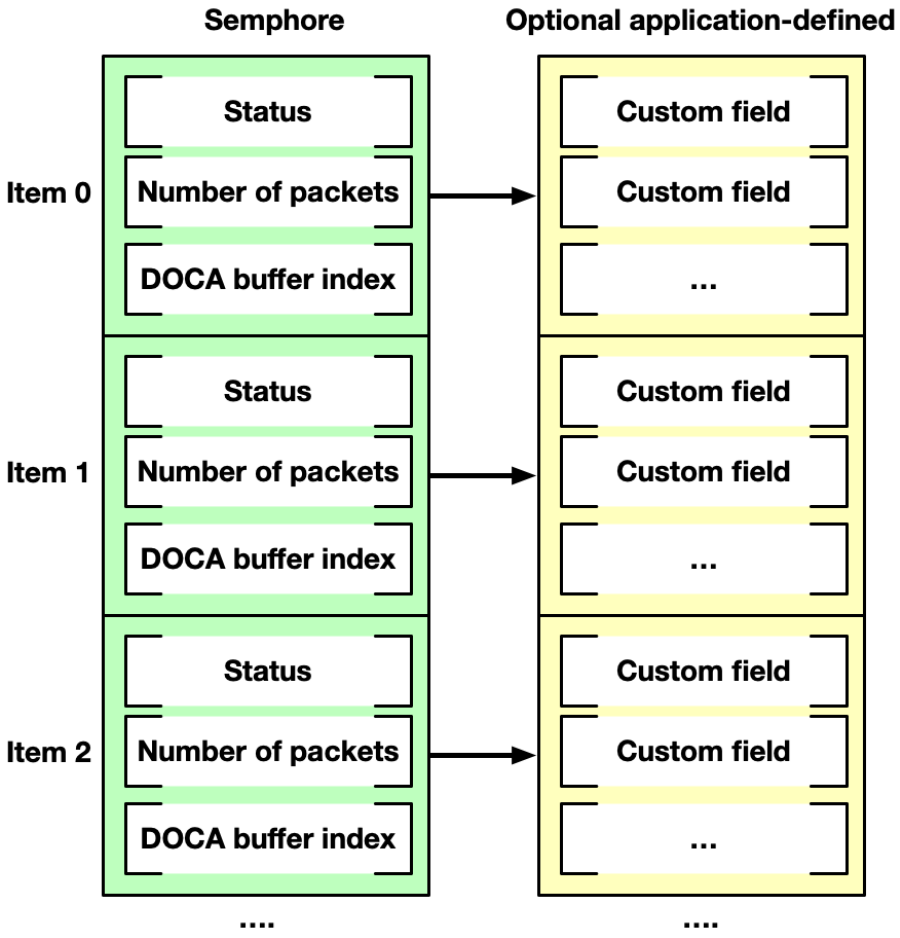
doca_gpu_semaphore_create

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:
    1. Populate the next item of the semaphore (packets' info and/or custom application-defined info).
    2. Set status flag to READY.
- Poll:
    1. Wait for the next item to have a status flag equal to `READY`.
    2. Read and process info.
    3. 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

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

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

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

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

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

GPU functions

In this section there is the list of DOCA GPUNetIO functions that can be used on the GPU only within a CUDA kernel.

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 allowed
- `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).

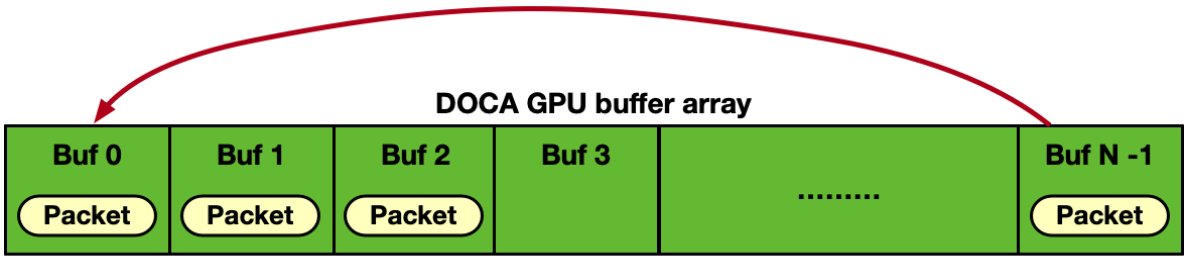
CUDA threads in the same scope (thread, warp, or block) must invoke the function on the same receive queue. The output parameters `num_rx_pkts` and `doca_gpu_buf_idx` must be visible by all threads in the scope (e.g., CUDA shared memory for warp and block).

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.

> **Note**
>
> 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 actually 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.

> **Note**
>
> 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.

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`.

> **Warning**
>
> 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,
};
```

Where:

- DOCA_GPU_SEND_FLAG_NONE – default behaviour, send is executed and no notification info is returned. If an error occurs and 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.

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`.

Strong Mode:

> **Tip**
>
> It is generally recommended to use strong mode as weak mode is more complex and is reserved for expert users.

```
__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

Weak Mode:

Note

In weak 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 wrapping at a 16-bit mask.

```
__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: 0-0xFFFF.
- `flags_bitmask` – One of the flags in the `doca_gpu_send_flags` enum

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.

Strong Mode:

Tip

It is generally recommended to use strong mode as weak mode is more complex and is reserved for expert users.

```
__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

Weak Mode:

Note

In weak 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 wrapping at a 16-bit mask.

```
__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. Value spans from 0 to `0xFFFF`.
- `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.

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_*`.

Strong Mode:

```
__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

Weak Mode:

```
__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.

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).

Building Blocks

This sections explains general concepts behind the fundamental building blocks to use when creating a DOCA GPUNetIO application.

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:

```
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.

Ethernet 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.

```
/* 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.

Ethernet 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.

```
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);
```

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.

```
#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);
```

Data Path on GPU

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. The following subsections provide examples for doing that.

Receive and Process

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:

```
#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:

```
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;

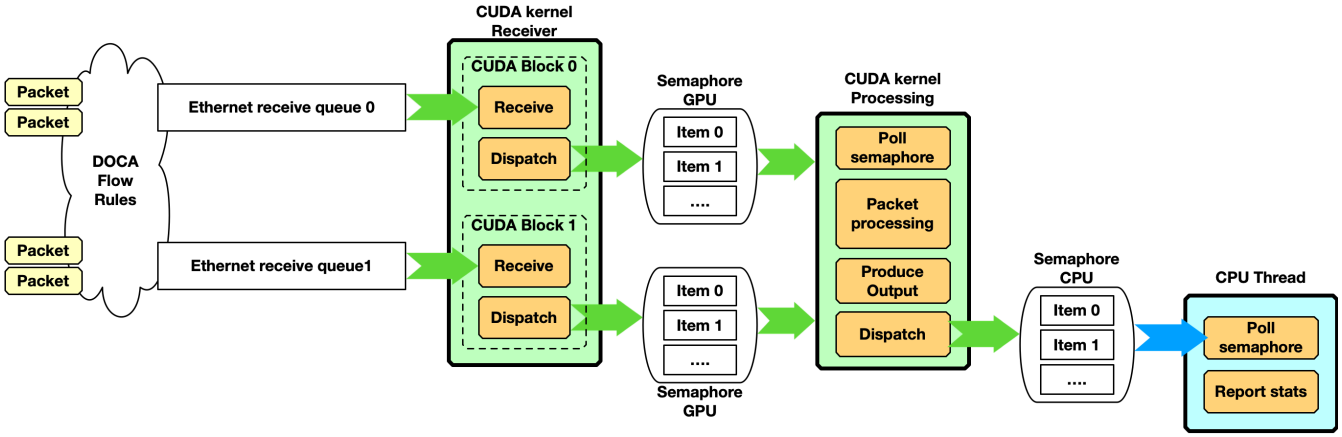
doca_gpu_dev_sem_get_custom_info_addr(sem_to_cpu_gpu,
semaphore_index_tmp, (void **)&gpu_info);
                        populate_custom_info(buf_addr, gpu_info);

doca_gpu_dev_sem_set_status(sem_to_cpu_gpu, semaphore_index_tmp,
DOCA_GPU_SEMAPHORE_STATUS_READY);
                }
                __syncthreads();

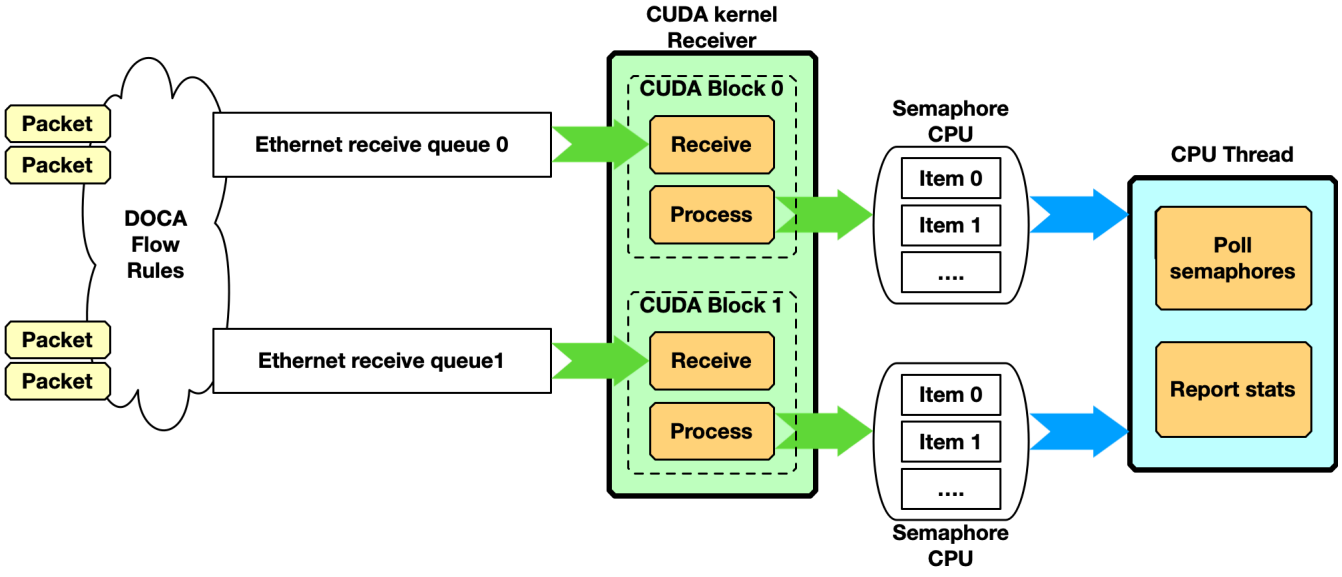
                if (threadIdx.x == 0) {
```

```
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.

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:

```
#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:

```
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;

        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 */

doca_gpu_dev_eth_txq_send_enqueue_strong(eth_txq_gpu, buf,
packet_len, );
                /* 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_strong(eth_txq_gpu,
DOCA_GPU_SEND_FLAG_NONE);
                        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 += CUDA_THREADS;
        }
```

```
}
```

GPUNetIO Samples

This section contains two samples to show how to enable simple GPUNetIO features.

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.

> **Note**
>
> This feature is supported on ConnectX-6 Dx and later .

> **Info**
>
> 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 .

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:

```
sudo apt install linuxptp
```

To start the `phc2sys` service properly, a config file must be created in `/lib/systemd/system/phc2sys.service`:

```
[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 23 /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:

```
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`:

```
$ 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 126
/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.

> ⚠ **Warning**
>
> 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.

Running the Sample

The sample is shipped with the source files that must be built:

```
# Ensure DOCA and DPDK are in the pkgconfig environment variable, change PATH according to your CUDA version
cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_send_wait_time

sudo PATH=${PATH}:/usr/local/${YOUR_CUDA_VERSION}/bin CPATH="$(echo /usr/local/${YOUR_CUDA_VERSION}/targets/{x86_64,sbsa}-linux/include | sed 's/ /:/'):${CPATH}" PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/usr/lib/pkgconfig:/opt/mellanox/dpdk/lib/x86_64-linux-gnu/pkgconfig:/opt/mellanox/dpdk/lib/x86_64-linux-gnu/pkgconfig:/opt/mellanox/doca/lib/x86_64-linux-gnu/pkgconfig meson build

sudo PATH=${PATH}:/usr/local/${YOUR_CUDA_VERSION}/bin CPATH="$(echo /usr/local/${YOUR_CUDA_VERSION}/targets/{x86_64,sbsa}-linux/include | sed 's/ /:/'):${CPATH}" 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`).

Considering a system with GPU PCIe address `ca:00.0` and NIC PCIe address `17:00.0`, to send 32 packets every 5 milliseconds:

```
$ sudo LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/local/${YOUR_CUDA_VERSION}/lib64:/opt/mellanox/dpdk/lib/x86_64-linux-gnu:/opt/mellanox/doca/lib/x86_64-linux-gnu /opt/mellanox/doca/samples/doca_gpunetio_send_wait_time -n 17:00.0 -g ca:00.0 -t 5000000
```

To verify that packets are actually sent at the right time, use a packet sniffer on the other side (e.g., `tcpdump`):

```
$ 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.

Note

`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`).

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.

> **⚠ Warning**
>
> 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:

```
cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_simple_receive

sudo PATH=${PATH}:/usr/local/${YOUR_CUDA_VERSION}/bin CPATH="$(echo
/usr/local/${YOUR_CUDA_VERSION}/targets/{x86_64,sbsa}-linux/include | sed 's/ /:/'):${CPATH}"
PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/usr/lib/pkgconfig:/opt/mellanox
linux-gnu/pkgconfig:/opt/mellanox/dpdk/lib/x86_64-linux-
gnu/pkgconfig:/opt/mellanox/doca/lib/x86_64-linux-gnu/pkgconfig
meson build

sudo PATH=${PATH}:/usr/local/${YOUR_CUDA_VERSION}/bin CPATH="$(echo
/usr/local/${YOUR_CUDA_VERSION}/targets/{x86_64,sbsa}-linux/include | sed 's/ /:/'):${CPATH}" 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 --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:

```
$ sudo
LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/local/${YOUR_CUDA_VERSION}/
linux-gnu:/opt/mellanox/doca/lib/x86_64-linux-gnu
./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
```

DOCA App Shield

This guide provides instructions on using the DOCA App Shield API.

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).

Prerequisites

1. Make sure to run App Shield from the DPU as a root user.

Dependencies

The library requires firmware version 24.32.1010 or higher.

API

For the library API reference, refer to the DOCA APSH API documentation in the NVIDIA DOCA Library APIs.

> **Note**
>
> The pkg-config (`*.pc` file) for the APSH library is included in DOCA's regular definitions (i.e., `doca`).

The following sections provide additional details about the library API.

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

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
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(struc doca_apsh_system *system, s doca_apsh_module ***modules *modules_size);`
Get processes	Returns an array with information about each process running on the system.	`doca_error_t doca_apsh_processes_get(str doca_apsh_system *system, s doca_apsh_process ***proces *processes_size);`
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 doca_apsh_process *process, doca_apsh_lib ***libs, int *l`
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(struc doca_apsh_process *process, doca_apsh_thread ***threads *threads_size);`

Getter Function Name	Functions Information	Functions Signature

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`
		`doca_apsh_process *process,`
		`doca_apsh_vad ***vads, int *v`

Get privileges	For a specified process, this function returns an array with information about each possible privilege for this process, as described here.
	
	Note
	Available on a Windows host only.
		`doca_error_t`
		`doca_apsh_privileges_get(st`
		`doca_apsh_process *process,`
		`doca_apsh_privilege ***priv`
		`*privileges_size);`

Get environment variables	For a specified process, this function returns an array with information about each environment variable within this process.
	
	Note
	Available on a Windows host only.
		`doca_error_t`
		`doca_apsh_envars_get(struct`
		`doca_apsh_process *process,`
		`doca_apsh_envar ***envars, i`
		`*envars_size);`

Getter Function Name	Functions Information	Functions Signature
Get handles	For a specified process, this function returns an array with information about each handle this process holds. **Note** Available on a Windows host only.	`doca_error_t doca_apsh_handles_get(struc doca_apsh_process *process, doca_apsh_handle ***handles *handles_size);`
Get LDR modules	For a specified process, this function returns an array with information about each loaded module within this process. **Note** Available on a Windows host only.	`doca_error_t doca_apsh_ldrmodules_get(st doca_apsh_process *process, doca_apsh_ldrmodule ***ldrm *ldrmodules_size);`

Getter Function Name	Functions Information	Functions Signature
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. **Note** Single-threaded processes are supported at beta level.	`doca_error_t doca_apsh_attestation_get(s doca_apsh_process *process, *exec_hash_map_path, struct doca_apsh_attestation ***at int *attestation_size);`
Attestation refresh	Refreshes a single attestation handler of a process with a new snapshot.	`doca_error_t doca_apsh_attst_refresh(str doca_apsh_attestation ***at int *attestation_size);`

<table>
<tr><th>Getter Function Name</th><th>Functions Information</th><th>Functions Signature</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.

Note
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="6">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></td><td><pre>doca_error_t
doca_apsh_netscan_get(struc
doca_apsh_system *system, s
doca_apsh_netscan ***connec
*connections_size);</pre></td></tr>
</table>

Getter Function Name	Functions Information	Functions Signature
	Note This feature is currently supported at beta level.	
Get process parameters	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). **Note** Available on a Windows host only. **Note** This feature is currently supported at beta level.	`doca_error_t doca_apsh_process_parameter` `doca_apsh_process *process,` `doca_apsh_process_parameter` `**process_parameters);`

Getter Function Name	Functions Information	Functions Signature

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.
	
	Note
	Available on a Windows host only.
		`doca_error_t doca_apsh_sids`
		`doca_apsh_process *process,`
		`doca_apsh_sid ***sids, int *s`

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.
	
	Note
	Available on a Windows host and Ubuntu 22.04 DPU.
		`doca_error_t doca_apsh_yara`
		`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_matcl`
		`*yara_matches_size);`
		
		Note
		To get a better understanding of the arguments, refer to documentation in `doca_apsh.h`

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);
```

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);
```

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.

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.

> **Note**
>
> 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);
```

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);
```

> **Note**
>
> 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);
```

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). **Note** Make sure that the value set is appropriate for your setup.

Parameter	Description
`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. **Note** 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.

DOCA App Shield Samples

This section provides DOCA App Shield library sample implementations on top of BlueField DPU.

Sample Prerequisites

Follow the guidelines in section "Prerequisites" then copy the generated JSON files, `symbols.json` and `mem_regions.json`, to the `/tmp/` directory.

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
```

> **Note**
>
> 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=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 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
```

Samples

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_`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_libs_get/apsh_libs_get_`
- `/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`

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_module`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_modules_get/apsh_module`
- `/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`

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_samp`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_pslist/apsh_pslist_main`
- `/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`

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_thread`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_threads_get/apsh_thread`
- `/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`

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_`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_vads_get/apsh_vads_get_`
- `/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`

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.

> **Note**
>
> 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_`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_envars_get/apsh_envars_`

- `/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`

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.

> **Note**
>
> 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_pri`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_privileges_get/apsh_pri`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_privileges_get/meson.bu`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_common.c`; `/opt/mellanox/doca/samples/doca_apsh/apsh_common.h`

DOCA Compress

This guide provides instructions on how to use the DOCA Compress API.

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.

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

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 DPU Modes of Operation.

Architecture

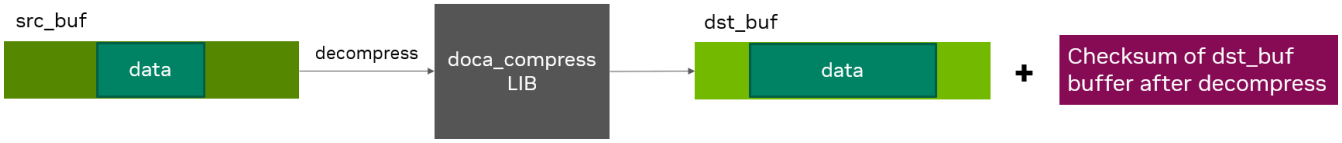
DOCA Compress is a DOCA Context as defined by DOCA Core. See NVIDA DOCA Core Context for more information.

DOCA Compress leverages DOCA Core architecture to expose asynchronous tasks that are offloaded to hardware.

Compress operation:



Decompress operation:



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

Alder and CRC

All compress/decompress tasks produce Adler and CRC.

Objects

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.

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.

Source and Destination Location

DOCA Compress can process DOCA buffers that reside on the host, the DPU, or both.

Local Host

Source and destination buffers reside on the host and the compress library runs on the host.

Local DPU

Source and destination buffers reside on the DPU and the compress library runs on the DPU.

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

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.

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.

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

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:

Supported Tasks

- `doca_compress_cap_task_compress_deflate_is_supported`
- `doca_compress_cap_task_decompress_deflate_is_supported`
- `doca_compress_cap_task_decompress_lz4_is_supported`

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_get_max_buf_size`

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
mmap From PCI Export Buffer	Yes	Yes
mmap From RDMA Export Buffer	No	No

Execution Phase

This section describes execution on CPU or DPU using DOCA Core Progress Engine.

Tasks

Compress Deflate Task

This task facilitates compressing memory using buffers as described in section "Buffer Support".

Configuration

Description	API to set the configuration	API to query sup
Enable the task	`doca_compress_task_compress_deflate_set_conf`	`doca_compres`
Number of tasks	`doca_compress_task_compress_deflate_set_conf`	`doca_compres`
Maximal buffer size	–	`doca_compres`
Maximum buffer list size	–	`doca_compres`

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

Output

Common output as described in DOCA Core Task.

Task Successful Completion

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 Failed Completion

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

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 Deflate Task

This task facilitates decompressing memory using buffers as described in section "Buffer Support".

Configuration

Description	API to Set the Configuration	API to Query
Enable the task	`doca_compress_task_decompress_deflate_set_conf`	`doca_compr`
Number of tasks	`doca_compress_task_decompress_deflate_set_conf`	`doca_compr`
Maximal buffer size	–	`doca_compr`
Maximum buffer list size	–	`doca_compr`

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

Output

Common output as described in DOCA Core Task.

Task Successful Completion

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 Failed Completion

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

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 Task

This task facilitates decompressing memory using buffers as described in section "Buffer Support".

Configuration

Description	API to Set the Configuration	API to Query Supp
Enable the task	`doca_compress_task_decompress_lz4_set_conf`	`doca_compress_`
Number of tasks	`doca_compress_task_decompress_lz4_set_conf`	`doca_compress_`
Maximal buffer size	–	`doca_compress_`
Maximum buffer list size	–	`doca_compress_`

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

Output

Common output as described in DOCA Core Task.

Task Successful Completion

After the task completes successfully:

- 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_lz4_get_adler_cs`
- CRC can be retrieved by calling `doca_compress_task_decompress_lz4_get_crc_cs`

Task Failed Completion

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

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

> **Note**
>
> When using an LZ4 operation, the source buffer must be from local memory.

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.

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.

States

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

Starting

This state cannot be reached.

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

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

Alternative Datapath Options

DOCA Compress only supports datapath on CPU, see Execution Phase.

> **Note**
>
> When using an LZ4 operation, the source buffer must be from local memory.

DOCA Compress Samples

The following samples illustrate how to use the DOCA Compress API to compress and decompress files.

DOCA Compress handles payload only. To create a compressed file, (e.g., gzip), the developer must add a gzip header. To decompress a file, (e.g. gzip) developer has to strip the header.

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
```

Info

The binary `doca_<sample_name>` is created under `/tmp/build/`.

3. Sample (e.g., `doca_compress_deflate`) usage:

```
Usage: doca_compress_deflate [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
```

4. For additional information per sample, use the `-h` option:

```
/tmp/build/doca_<sample_name> -h
```

Samples

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 DOCA Compress deflate task or DOCA Decompress deflate object.
6. Submitting a compress task.
7. Running the progress engine until the task is completed.
8. Writing the result into an output file, `out.txt`.
9. Destroying all compress and DOCA core structures.

References:

- `/opt/mellanox/doca/samples/doca_compress/compress_deflate/compres`
- `/opt/mellanox/doca/samples/doca_compress/compress_deflate/compres`
- `/opt/mellanox/doca/samples/doca_compress/compress_deflate/meson.b`
- `/opt/mellanox/doca/samples/doca_compress/decompress_deflate/decom`
- `/opt/mellanox/doca/samples/doca_compress/decompress_deflate/decom`
- `/opt/mellanox/doca/samples/doca_compress/decompress_deflate/meson`
- `/opt/mellanox/doca/samples/doca_compress/compress_common.h`
- `/opt/mellanox/doca/samples/doca_compress/compress_common.c`

- `/opt/mellanox/doca/samples/doca_compress/meson.build`

DOCA SHA

This guide provides developer focused instructions on deploying and programming the DOCA SHA library.

Introduction

The DOCA SHA library provides a flexible and unified API to leverage the SHA offload engine present in the NVIDIA® BlueField® DPU. For more information on SHA (secure hash standard algorithm), please review the FIPS 180-4 specifications.

> **Note**
>
> SHA hardware acceleration engine is only available on the BlueField-2 DPU. Thus, the DOCA SHA library is not available for BlueField-3 DPUs.

SHA is commonly used in cryptography to generate a given hash value for a supplied input buffer. Depending on the SHA algorithm used, the message length may vary: Any length less than 2^{64} bits for SHA-1, SHA-224, and SHA-256, or less than 2^{128} bits for SHA-384, SHA-512, SHA-512/224, and SHA-512/256. The resulting output from a SHA operation is called a message digest. The message digests range in length from 160 to 512 bits depending on the selected SHA algorithm. As expected from any cryptography algorithm, any change to a message will, with a very high probability, result in a different message digest and verification failure.

SHA is typically used with other cryptographic algorithms, such as digital signature algorithms and keyed-hash message authentication codes, or in the generation of random numbers.

The DOCA SHA library supports three SHA algorithms, SHA-1, SHA-256, and SHA-512, and aims to comply with the OpenSSL SHA implementation standard. It supports both one-shot and stateful SHA calculations.

- One-shot means that the input message is composed of a single segment of data and, therefore, the SHA operation is completed in a single step (i.e., one single SHA engine enqueue and dequeue operation)

- Stateful means that the input message is composed of many segments of data and, therefore, its SHA calculation needs more than one SHA enqueue and dequeue operation to finish. During any stateful operation, other SHA operations can also be executed.

Prerequisites

DOCA SHA applications can run either on the host machine or directly on the crypto-enabled DPU target. As the DOCA SHA leverages the SHA engine, users must make sure it is enabled:

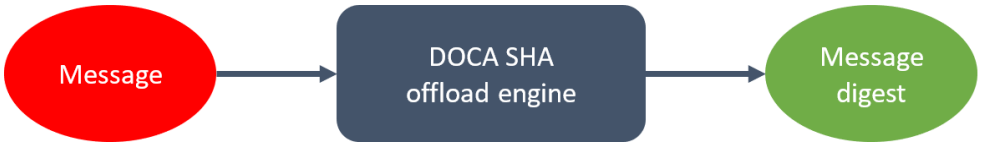
```
$ sudo mlxfwmanager
```

In the output, make sure that `Crypto Enabled` appears in the command output in the `Description` line.

Architecture

The following diagram shows how the DOCA SHA library receives a message and outputs a message digest.

From an application level, the DOCA SHA library can be seen as a black box. DOCA SHA outputs a response regardless of the nature of the input message.



- In a one-shot SHA situation, the single output is the correct message digest

- In a stateful SHA situation, multiple outputs are expected corresponding to multiple inputs but only the last output is the correct message digest

API

In the following sections, additional details about the library API are provided. For more information, please refer to the NVIDIA DOCA Library APIs reference.

doca_sha_job_type

The enum defines six job types in the DOCA SHA library.

```
enum doca_sha_job_type {
        DOCA_SHA_JOB_SHA1 = DOCA_ACTION_SHA_FIRST + 1,
        DOCA_SHA_JOB_SHA256,
        DOCA_SHA_JOB_SHA512,
        DOCA_SHA_JOB_SHA1_PARTIAL,
        DOCA_SHA_JOB_SHA256_PARTIAL,
        DOCA_SHA_JOB_SHA512_PARTIAL,
};
```

- `DOCA_SHA_JOB_SHA1`; `DOCA_SHA_JOB_SHA256`; `DOCA_SHA_JOB_SHA512` – used to specify a one-shot SHA calculation
- `DOCA_SHA_JOB_SHA1_PARTIAL`; `DOCA_SHA_JOB_SHA256_PARTIAL`; `DOCA_SHA_JOB_SHA512_PARTIAL` – used to specify a stateful SHA calculation

DOCA SHA Output Length Macro

These macros define the smallest SHA response buffer length corresponding to different job types.

```
#define DOCA_SHA1_BYTE_COUNT       20
#define DOCA_SHA256_BYTE_COUNT     32
#define DOCA_SHA512_BYTE_COUNT     64
```

- `DOCA_SHA1_BYTE_COUNT` – number of message digest bytes for `SHA1_PARTIAL` and `SHA1_PARTIAL`
- `DOCA_SHA256_BYTE_COUNT` – number of message digest bytes for `SHA256_PARTIAL` and `SHA256_PARTIAL`
- `DOCA_SHA512_BYTE_COUNT` – number of message digest bytes for `SHA512_PARTIAL` and `SHA512_PARTIAL`

doca_sha_job_flags

The enum defines flags used for `doca_sha_job` construction.

```
enum doca_sha_job_flags {
        DOCA_SHA_JOB_FLAGS_NONE = 0,
        DOCA_SHA_JOB_FLAGS_SHA_PARTIAL_FINAL
};
```

- `DOCA_SHA_JOB_FLAGS_NONE` – the default flag suitable for all SHA jobs
- `DOCA_SHA_JOB_FLAGS_SHA_PARTIAL_FINAL` – signifies that the current input is the final segment of a whole stateful job

doca_sha_job

This is the DOCA SHA job definition, suitable for one-shot SHA job types, `DOCA_ JOB_SHA1/256/512`.

```
struct doca_sha_job {
        struct doca_job base;
        struct doca_buf *req_buf;
        struct doca_buf *resp_buf;
        uint64_t flags;
};
```

- `base` – an opaque `doca_job` structure
- `req_buf` – the `doca_buf` containing the input message
- `resp_buf` – the `doca_buf` used for the output message digest
- `flags` – the `doca_sha_job_flags`

doca_sha_partial_session

An opaque structure used in a stateful SHA job.

```
struct doca_sha_partial_session;
```

doca_sha_partial_job

This is the DOCA SHA job definition, suitable for stateful SHA job types, `DOCA_JOB_SHA1/256/512_PARTIAL`.

```
struct doca_sha_partial_job {
        struct doca_sha_job sha_job;
        struct doca_sha_partial_session *session;
};
```

- `sha_job` – contain the fields for the input message, output message digest, and flags
- `session` – contain the state information for a stateful SHA calculation

doca_sha

An opaque structure for DOCA SHA API.

```
struct doca_sha;
```

doca_sha_create

Before performing any SHA operation, it is essential to create a `doca_sha` object.

```
doca_error_t doca_sha_create(struct doca_sha **ctx);
```

- `ctx [in/out]` – `doca_sha` object to be created
- Returns – `DOCA_SUCCESS` on success, error code otherwise

doca_sha_destroy

Used to destroy a doca_sha object after a SHA operation is done:

```
doca_error_t doca_sha_destroy(struct doca_sha *ctx);
```

- `ctx [in]` – `doca_sha` object to be destroyed; it is created by `doca_sha_create()`
- Returns – `DOCA_SUCCESS` on success, error code otherwise

doca_sha_job_get_supported

Check whether a device can perform `doca_sha` jobs.

```
doca_error_t doca_sha_destroy(struct doca_sha *ctx);
```

- `devinfo [in]` – a pointer to the `doca_devinfo` object
- `job_type [in]` – `d` `oca_sha` job type enum
- Returns – `D` `OCA_SUCCESS` on success, error code otherwise

doca_sha_get_max_list_buf_num_elem

Get the maximum linked_list doca_buf count for the source buffer in a doca_sha job.

```
doca_error_t doca_sha_get_max_list_buf_num_elem(const struct
doca_devinfo *devinfo, uint32_t *max_list_num_elem);
```

- `devinfo [in]` – a pointer to the doca_devinfo object

- `max_list_num_elem [out]` – maximum linked list `doca_buf` count
- Returns – `DOCA_SUCCESS` on success, error code otherwise

doca_sha_get_max_src_buffer_size

Get the maximum buffer byte count for the source buffer in a doca_sha job.

```
doca_error_t doca_sha_get_max_src_buffer_size(const struct
doca_devinfo *devinfo, uint64_t *max_buffer_size);
```

- `devinfo [in]` – a pointer to the `doca_devinfo` object
- `max_buffer_size [out]` – maximum buffer byte count
- Returns – `DOCA_SUCCESS` on success, error code otherwise

doca_sha_get_min_dst_buffer_size

Get the minimum buffer byte count for the destination buffer in a doca_sha job.

```
doca_error_t doca_sha_get_max_src_buffer_size(const struct
doca_devinfo *devinfo, uint64_t *max_buffer_size);
```

- `devinfo [in]` – A pointer to the `doca_devinfo` object
- `job_type [in]` – `doca_sha` job type enum
- `min_buffer_size [out]` – Minimum buffer byte count
- Returns – `DOCA_SUCCESS` on success, error code otherwise

doca_sha_get_hardware_supported

Check that a `doca_sha` engine is supported.

```
doca_error_t doca_sha_get_hardware_supported(const struct
doca_devinfo *devinfo);
```

- `devinfo [in]` – a pointer to the `doca_devinfo` object
- Returns – `DOCA_SUCCESS` on success, error code otherwise

doca_sha_as_ctx

Convert a doca_sha object into a doca object:

```
struct doca_ctx *doca_sha_as_ctx(struct doca_sha *ctx);
```

- `ctx [in]` – a pointer to the `doca_sha` object
- `doca_ctx [out]` – a pointer to the `doca` object
- Returns – a pointer to the `doca` object on success, `NULL` otherwise

doca_sha_partial_session_create

Before doing any stateful SHA calculation, it is necessary to create a doca_sha_partial_session object to keep the state information:

```
doca_error_t doca_sha_partial_session_create(
        struct doca_sha *ctx,
        struct doca_workq *workq,
        struct doca_sha_partial_session **session);
```

- `ctx [in]` – a pointer to the `doca_sha` object
- `workq [in]` – a pointer to the `doca_workq` object
- `session [in/out]` – a pointer to the `doca_sha_partial_session` object to be created
- Returns – `DOCA_SUCCESS` on success, error code otherwise

doca_sha_partial_session_destroy

Free stateful SHA session resource:

```
doca_error_t doca_sha_partial_session_destroy(
        struct doca_sha *ctx,
        struct doca_workq *workq,
        struct doca_sha_partial_session *session);
```

- `ctx [in]` – a pointer to the `doca_sha` object
- `workq [in]` – a pointer to the `doca_workq` object
- `session [in]` – a pointer to the `doca_sha_partial_session` object to be freed
- Returns – `DOCA_SUCCESS` on success, error code otherwise

doca_sha_partial_session_copy

Copy the stateful SHA session resource:

```
doca_error_t doca_sha_partial_session_copy(
        struct doca_sha *ctx,
        struct doca_workq *workq,
        struct doca_sha_partial_session *from,
        struct doca_sha_partial_session *to);
```

- `ctx [in]` – a pointer to the `doca_sha` object
- `workq [in]` – a pointer to the `doca_workq` object
- `from [in]` – a pointer to the source `doca_sha_partial_session` object to be copied
- `to [out]` – a pointer to the destination `doca_sha_partial_session` object
- Returns – `DOCA_SUCCESS` on success, error code otherwise

Capabilities and Limitations

Supported SHA algorithms:

- SHA1
- SHA256
- SHA512

Output message digest length:

- 20B for SHA1
- 32B for SHA256

- 64B for SHA512

Maximum single job size:

- For one-shot SHA calculation, the input message size must be $\leqslant 2^{31}$
- For stateful SHA calculation, the accumulated input message size must be $\leqslant 2^{31}$

Stateful SHA job length requirement:

- For `SHA1/256_PARTIAL`, only the last segment allows its `byte_count` != multiple-of-64
- For `SHA512_PARTIAL`, only the last segment allows its `byte_count` != multiple-of-128

Troubleshooting

Performing One-shot SHA Calculation

1. Construct a `doca_sha_job`:

```
struct doca_sha_job job = {
        .base.type = DOCA_SHA_JOB_SHA1,
        .req_buf   = user_req_buf,
        .resp_buf  = user_resp_buf,
        .flags     = DOCA_SHA_JOB_FLAGS_NONE
};
```

2. Submit the job until `DOCA_SUCCESS` is received:

```
In synchronous mode, we can use:
ret = doca_workq_submit(workq, &job.base);
if (ret != DOCA_SUCCESS)
        error_exit;
```

If `doca_workq_submit()` returns `DOCA_ERROR_INVALID_VALUE`, it means the job construction has a problem. If it returns `DOCA_ERROR_BAD_STATE`, it indicates a fatal internal error and the whole engine must be reinitialized.

In asynchronous mode, `doca_workq_submit()` may return `DOCA_ERROR_NO_MEMORY`. In that case, you must first call `doca_workq_progress_retrieve()` to receive a response so that the job resource can be freed, then retry calling `doca_workq_submit()`.

Possible `doca_workq_submit()` return codes:

- `DOCA_SUCCESS`
- `DOCA_ERROR_INVALID_VALUE`
- `DOCA_ERROR_NO_MEMORY`
- `DOCA_ERROR_BAD_STATE`

3. To retrieve a job response until `DOCA_SUCCESS` is received:

```
while ((ret = doca_workq_progress_retrieve(workq, &event,
DOCA_WORKQ_RETRIEVE_FLAGS_NONE)) == DOCA_ERROR_AGAIN);
if (ret != DOCA_SUCCESS)
        error_exit;
```

If `doca_workq_progress_retrieve()` returns `DOCA_ERROR_INVALID_VALUE` it means invalid input is received. If it returns `DOCA_ERROR_IO_FAILED`, it signifies fatal internal error and the whole engine needs reinitialized.

Possible `doca_workq_progress_retrieve()` return codes:

- `DOCA_SUCCESS`
- `DOCA_ERROR_INVALID_VALUE`
- `DOCA_ERROR_NO_MEMORY`
- `DOCA_ERROR_BAD_STATE`

Performing Stateful SHA Calculation

This section describes the steps to finish a stateful SHA1 calculation, assuming the whole job is composed of three or more segments.

1. Obtain a `doca_sha_partial_session`:

```
doca_sha_partial_session *session;
doca_sha_partial_session_create(ctx, workq, &session);
```

2. Construct a `doca_sha_partial_job` for the first segment:

```
struct doca_sha_partial_job job = {
        .sha_job.base.type = DOCA_SHA_JOB_SHA1_PARTIAL,
        .sha_job.req_buf   = user_req_buf_of_1st_segment,
        .sha_job.resp_buf  = user_resp_buf,
        .sha_job.flags     = DOCA_SHA_JOB_FLAGS_NONE,
        .session           = session,
};
```

3. Submit the job for the first segment:

```
ret = doca_workq_submit(workq, &job.base);
if (ret != DOCA_SUCCESS)
        error_exit;
```

4. Wait until first segment processing is done:

```
while ((ret = doca_workq_progress_retrieve(workq, &event,
DOCA_WORKQ_RETRIEVE_FLAGS_NONE)) == DOCA_ERROR_AGAIN);
if (ret != DOCA_SUCCESS)
        error_exit;
```

   The purpose of this call is to make sure the first segment processing is finished before continuing to send the next segment, as it is necessary to sequentially process all segments for a correct message digest generation. The `user_resp_buf` at this moment contains garbage values.

5. For the second segment, repeat the previous three steps:

```
struct doca_sha_partial_job job = {
        .sha_job.base.type = DOCA_SHA_JOB_SHA1_PARTIAL,
        .sha_job.req_buf   = user_req_buf_of_2nd_segment,
        .sha_job.resp_buf  = user_resp_buf,
        .sha_job.flags     = DOCA_SHA_JOB_FLAGS_NONE,
        .session           = session,
};

ret = doca_workq_submit(workq, &job.base);
if (ret != DOCA_SUCCESS)
        error_exit;

while ((ret = doca_workq_progress_retrieve(workq, &event,
DOCA_WORKQ_RETRIEVE_FLAGS_NONE)) == DOCA_ERROR_AGAIN);
if (ret != DOCA_SUCCESS)
        error_exit;
```

The purpose of this call is still to make sure the second segment processing is finished. The user `user_resp_buf` at this moment still contains garbage values.

6. All subsequent segments repeat the same process.

7. For the last segment, repeat the same process while setting the special flag for the last segment:

```
struct doca_sha_partial_job job = {
        .sha_job.base.type = DOCA_SHA_JOB_SHA1_PARTIAL,
        .sha_job.req_buf   =
user_req_buf_of_the_last_segment,
        .sha_job.resp_buf  = user_resp_buf,
        .sha_job.flags     =
DOCA_SHA_JOB_FLAGS_SHA_PARTIAL_LAST,
        .session           = session,
};

ret = doca_workq_submit(workq, &job.base);
if (ret != DOCA_SUCCESS)
        error_exit;

while ((ret = doca_workq_progress_retrieve(workq, &event,
DOCA_WORKQ_RETRIEVE_FLAGS_NONE)) == DOCA_ERROR_AGAIN);
if (ret != DOCA_SUCCESS)
        error_exit;
```

After the `DOCA_SUCCESS` event of the last segment is received the processing of the whole job is done now. You can get the expected SHA message digest from the `user_resp_buf` now.

8. Release the session object:

```
doca_sha_partial_session_destroy(ctx, workq, session);
```

Notes:

- Before submitting the first segment, call `doca_sha_partial_session_create()` to obtain a "session" object.
- During the whole process, make sure to use the same `doca_sha_partial_session` object used for all segments of the entire job.

- If a session object is released before the whole stateful SHA is finished, or if different objects are used for a stateful SHA, the job submission may fail due to job validity check failure. Even the job submission succeeds, a wrong SHA message digest is expected.
- The session resource is limited, it is the user's responsibility to properly call `doca_sha_partial_session_destroy()` to make sure all allocated session objects are released.
- For the last segment, the `DOCA_SHA_JOB_FLAGS_SHA_PARTIAL_FINAL` flag must be set.
- If `DOCA_SHA_JOB_FLAGS_SHA_PARTIAL_FINAL` is not properly set, the engine assumes an intermediate partial SHA calculation and returns an invalid SHA message digest. As only the user knows when the last segment arrives, it is their responsibility to properly set this flag.
- Make sure the `SHA_PARTIAL` segment length requirements are In this example, the first and second segments' byte count must be a multiple of 64. Otherwise, the job submission may fail due to job validity check failure.

Using Session Copy

This section describes the steps for `utilizing session_copy()` to reduce the stateful SHA calculation overhead.

The example assumes there are two whole jobs, `job_0` and `job_1`, where `job_0` is composed of several segments, `{header_segment, job_0's other segments}`, and `job_1` is composed of `{header_segment, job_1' other segments}`.

1. Obtain two `doca_sha_partial_session`:

```
doca_sha_partial_session *session_0;
doca_sha_partial_session_create(ctx, workq, &session_0);
doca_sha_partial_session *session_1;
doca_sha_partial_session_create(ctx, workq, &session_1);
```

2. Construct a `doca_sha_partial_job` for the `header_segment`:

```
struct doca_sha_partial_job job = {
        .sha_job.base.type = DOCA_SHA_JOB_SHA1_PARTIAL,
        .sha_job.req_buf   = user_req_buf_of_header_segment,
        .sha_job.resp_buf  = user_resp_buf,
        .sha_job.flags     = DOCA_SHA_JOB_FLAGS_NONE,
        .session           = session_0,
};
```

3. Submit the `header_segment` of `job_0`.

```
ret = doca_workq_submit(workq, &job.base);
if (ret != DOCA_SUCCESS)
        error_exit;
```

4. Wait until the processing of `header_segment` is done:

```
while ((ret = doca_workq_progress_retrieve(workq, &event,
DOCA_WORKQ_RETRIEVE_FLAGS_NONE)) == DOCA_ERROR_AGAIN);
if (ret != DOCA_SUCCESS)
        error_exit;
```

5. Perform the session copy so that `job_1` does not need to calculate its `header_segment`:

```
doca_sha_partial_session_copy(ctx, workq, session_0,
session_1);
```

6. Continue to calculate `job_0` and `job_1`'s other segments until final segment using normal `partial_sha` calculation process.

```
struct doca_sha_partial_job job = {
        .sha_job.base.type = DOCA_SHA_JOB_SHA1_PARTIAL,
        .sha_job.req_buf   =
user_req_buf_of_job_0_other_segment,
        .sha_job.resp_buf  = user_resp_buf,
        .sha_job.flags     = DOCA_SHA_JOB_FLAGS_NONE,
        .session           = session_0,
};

ret = doca_workq_submit(workq, &job.base);
if (ret != DOCA_SUCCESS)
        error_exit;

while ((ret = doca_workq_progress_retrieve(workq, &event,
DOCA_WORKQ_RETRIEVE_FLAGS_NONE)) == DOCA_ERROR_AGAIN);
if (ret != DOCA_SUCCESS)
        error_exit;

struct doca_sha_partial_job job = {
        .sha_job.base.type = DOCA_SHA_JOB_SHA1_PARTIAL,
        .sha_job.req_buf   =
user_req_buf_of_job_1_other_segment,
        .sha_job.resp_buf  = user_resp_buf,
        .sha_job.flags     = DOCA_SHA_JOB_FLAGS_NONE,
        .session           = session_1,
};

ret = doca_workq_submit(workq, &job.base);
if (ret != DOCA_SUCCESS)
        error_exit;

while ((ret = doca_workq_progress_retrieve(workq, &event,
DOCA_WORKQ_RETRIEVE_FLAGS_NONE)) == DOCA_ERROR_AGAIN);
if (ret != DOCA_SUCCESS)
```

```
        error_exit;
```

7. Release the session object:

```
doca_sha_partial_session_destroy(ctx, workq, session_0);
doca_sha_partial_session_destroy(ctx, workq, session_1);
```

Quick Start

This section provides instructions on how to test the DOCA SHA library:

1. Enable DOCA SHA test apps and build.

```
user@machine:/home/user$ cd ${YOUR-PATH}/doca
user@machine:/home/user$ meson setup build
user@machine:/home/user$ cd build
user@machine:/home/user/doca/build$ meson configure -
Dunit_test_lib_sha=true
user@machine:/home/user/doca/build$ ninja
```

2. Run the test app `test_doca_sha_lite`. This test app shows how to do the simplest one-shot SHA calculation. It sends three messages and receives three massage digests for all SHA1, SHA256, and SHA512.

```
user@machine:/home/user$ cd libs/doca_sha/src/unit_test/
user@machine:/home/user$ ./test_doca_sha_lite --pci_addr
03:00.0
```

3. Run the test app `test_doca_sha_benchmark`. This test app can be used to test throughput and more complex one-shot SHA calculation cases.

   - Case 1: Test throughput of SHA1 with a 4096-byte input message.

     ```
     user@machine:/home/user$ ./test_doca_sha_benchmark --
     pci_addr af:00.0 --data_file test_files/sha-input-
     4kbyte.txt --nb_iteration 1000000 --sha_type 0
     ```

   - Case 2: Test throughput of SHA256 with an 8192-byte input message.

     ```
     user@machine:/home/user$ ./test_doca_sha_benchmark --
     pci_addr af:00.0 --data_file test_files/sha-input-
     8kbyte.txt --nb_iteration 1000000 --sha_type 1
     ```

   - Case 3: Calculate a SHA512 message digest with a random 16-byte input message.

     ```
     user@machine:/home/user$ ./test_doca_sha_benchmark --
     pci_addr af:00.0 --nb_iteratoin 1 --use_random_data --
     data_byte_count 16 --sha_type 2
     ```

   - Case 4: Calculate a SHA1 message digest with a 1-gigabyte (2^{30}) input message.

     ```
     user@machine:/home/user$ ./test_doca_sha_benchmark --
     pci_addr af:00.0 --data_file test_files/sha-input-
     1Gbyte.txt --nb_iteration 1 --sha_type 0
     ```

4. Run the test app `test_doca_sha_partial`. This test app can be used to show how to perform a stateful SHA calculation and the `session_copy` function.

    - Case 1: Calculate a SHA1 message digest of two 129-byte messages. Each message is composed of three segments, the first and second segments are 64 bytes, the third segment is 1 byte.

      ```
      user@machine:/home/user$ ./test_doca_sha_partial --
      pci_addr af:00.0 --job_byte_count 129 --
      segment_byte_count 64 --job_count 2 --mode 0 --sha_type
      0
      ```

    - Case 2: Calculate a SHA256 message digest of a 2-gigabyte (2^{31}) message. This message is composed of 16 segments, each segment is 128 megabytes (2^{27}).

      ```
      user@machine:/home/user$ ./test_doca_sha_partial --
      pci_addr af:00.0 --job_byte_count 134217728 --
      segment_byte_count 134217728 --job_count 1 --mode 3 --
      nb_sha_partial 16 --sha_type 1
      ```

    - Case 3: Calculate four SHA512 message digests of using the `session_copy` function. All the four jobs share the same job length, 129 bytes, and the same head segment, 128 bytes.

      ```
      user@machine:/home/user$ ./test_doca_sha_partial --
      pci_addr af:00.0 --job_byte_count 129 --
      segment_byte_count 128 --job_count 4 --mode 4 --sha_type
      2
      ```

DOCA SHA Samples

This section describes SHA samples based on the DOCA SHA library. These samples illustrate how to use the DOCA SHA API to calculate secure hash algorithm on a given message.

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_sha/<sample_name>
meson /tmp/build
ninja -C /tmp/build
```

> **Note**
>
> The binary `doca_<sample_name>` will be 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=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, --data                        user data
```

For additional information per sample, use the `-h` option:

```
/tmp/build/doca_<sample_name> -h
```

Samples

SHA Create

This sample illustrates how to send A SHA job and retrieve the result.

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 the element in DOCA buffer inventory for each buffer.
5. Initializing a DOCA SHA job object.
6. Submitting the SHA job into work queue.
7. Retrieving the SHA job from the queue once it is done.
8. Printing the job result.
9. Destroying all SHA and DOCA core structures.

References:

- `/opt/mellanox/doca/samples/doca_sha/sha_create/sha_create_sample.`
- `/opt/mellanox/doca/samples/doca_sha/sha_create/sha_create_main.c`
- `/opt/mellanox/doca/samples/doca_sha/sha_create/meson.build`

SHA Partial Create

This sample illustrates how to send partial SHA jobs and retrieve the result. Each job source buffer (except the final) will be 64 bytes.

The sample logic includes:

1. Locating a DOCA device.
2. Initializing the required DOCA core structures.
3. Initializing a partial session for all the jobs.
4. Populating DOCA memory map with two relevant buffers; one for the source data and one for the result.
5. Allocating the element in DOCA buffer inventory for the result buffer.

6. Calculating total jobs; user data length divided by 64.

7. For each job:

    1. Allocating the element in DOCA buffer inventory for the relevant part in the source buffer.

    2. Initializing the DOCA SHA job object. If it is the final job, send `DOCA_SHA_JOB_FLAGS_SHA_PARTIAL_FINAL` flag.

    3. Submitting SHA job into work queue.

    4. Retrieving SHA job from the queue once it is done.

8. Printing the final job result.

9. Destroying all SHA and DOCA core structures.

References:

- `/opt/mellanox/doca/samples/doca_sha/sha_partial_create/sha_partia`
- `/opt/mellanox/doca/samples/doca_sha/sha_partial_create/sha_partia`
- `/opt/mellanox/doca/samples/doca_sha/sha_partial_create/meson.buil`

DOCA Erasure Coding

This guide provides instructions on how to use the DOCA Erasure Coding API .

Introduction

> **Note**
>
> 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.

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.

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

Environment

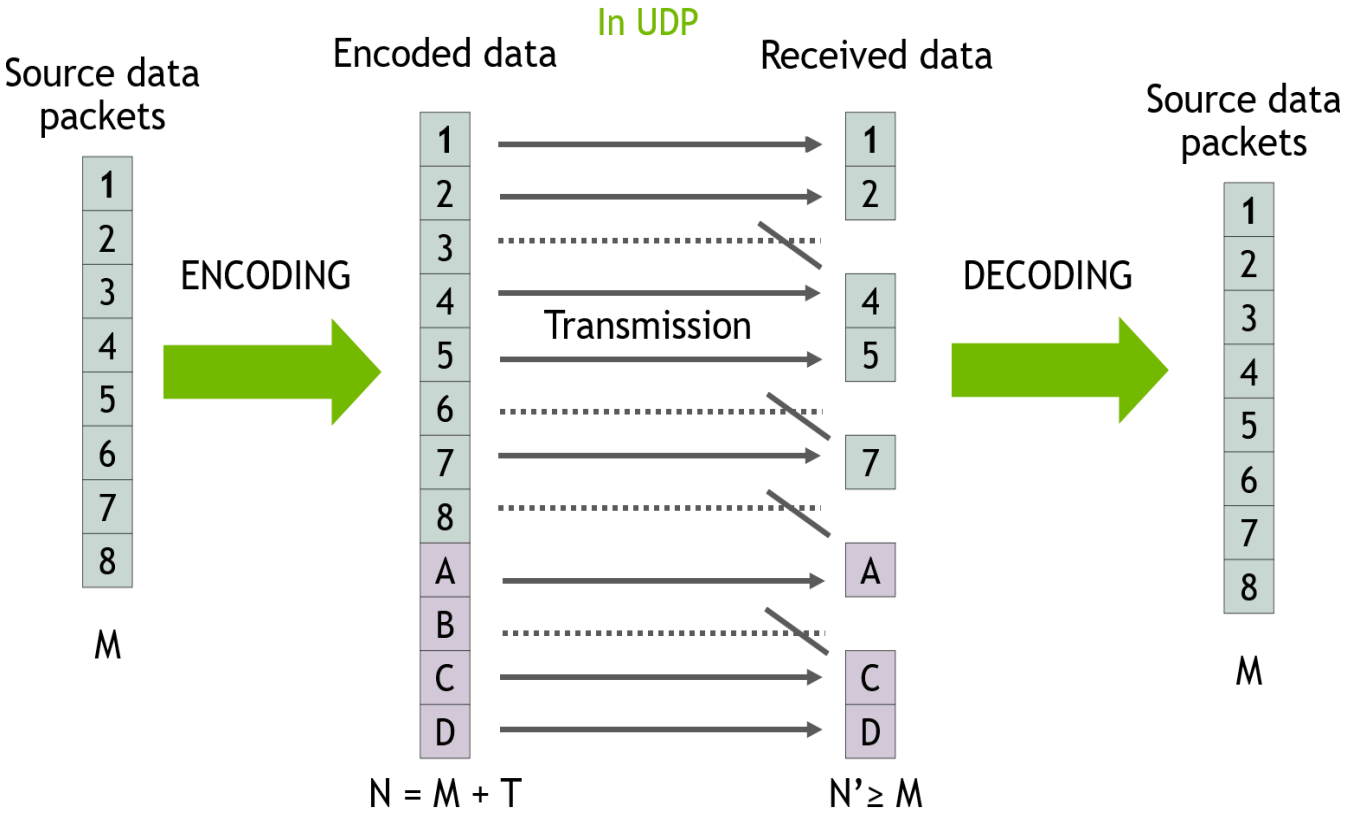
DOCA Erasure Coding-based applications can run either on the host machine or on the DPU target (NVIDIA® BlueField®-3 and above).

Erasure Coding can only be run with DPU configured in DPU mode as described in NVIDIA BlueField DPU Modes of Operation.

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.

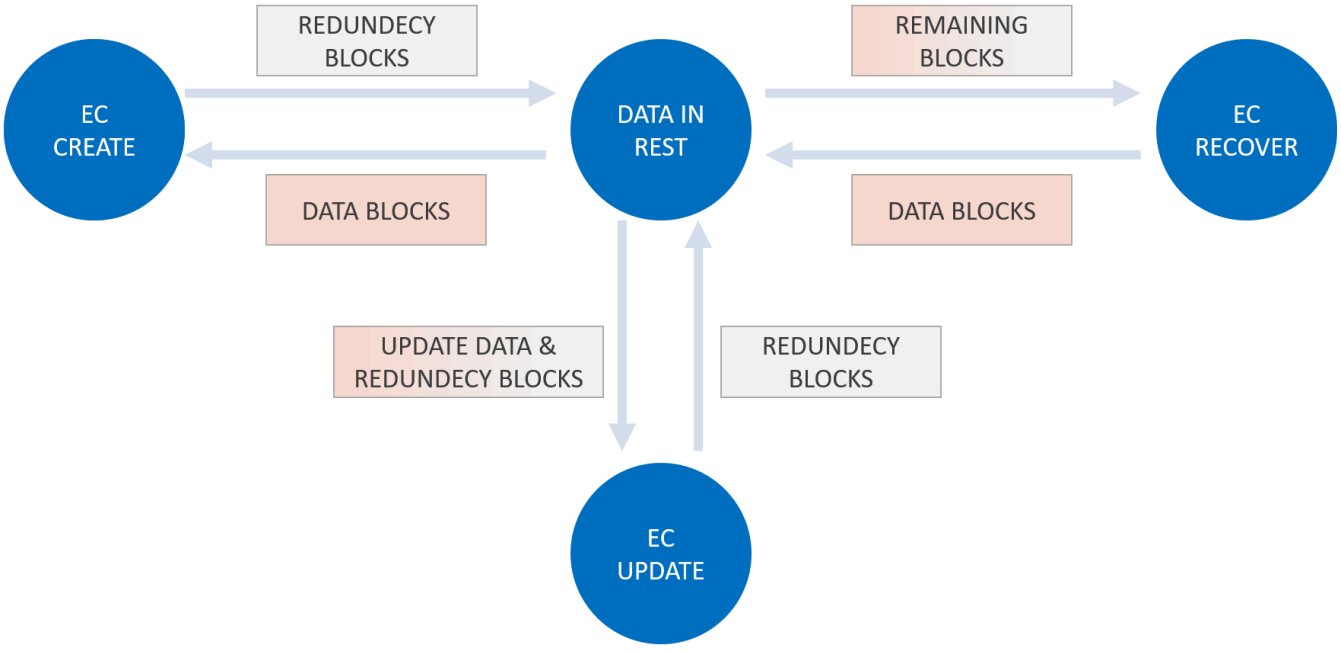
Flows

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).



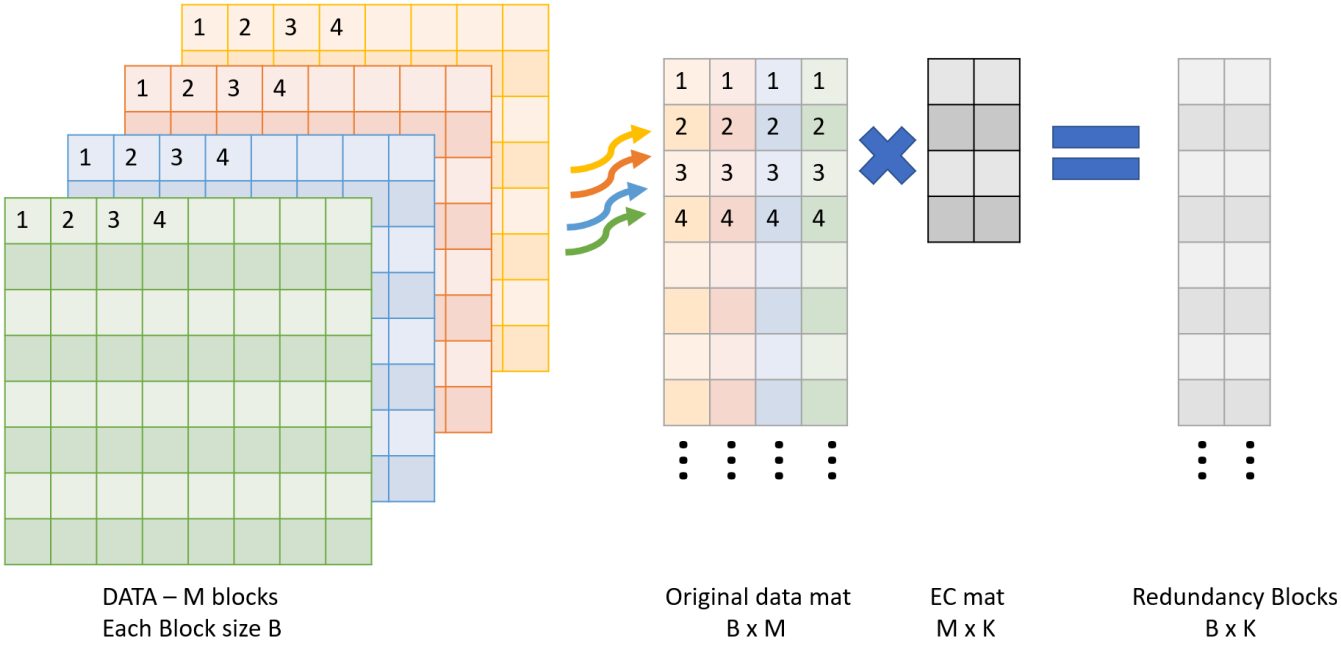
The following sections examine an M:K (where M is the original data and K is redundancy) EC.

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.

> **Note**
>
> This step can be repeated in a stream use case, as the DPU would not be the recovery or update point.



Recover Block

The user must perform the following:

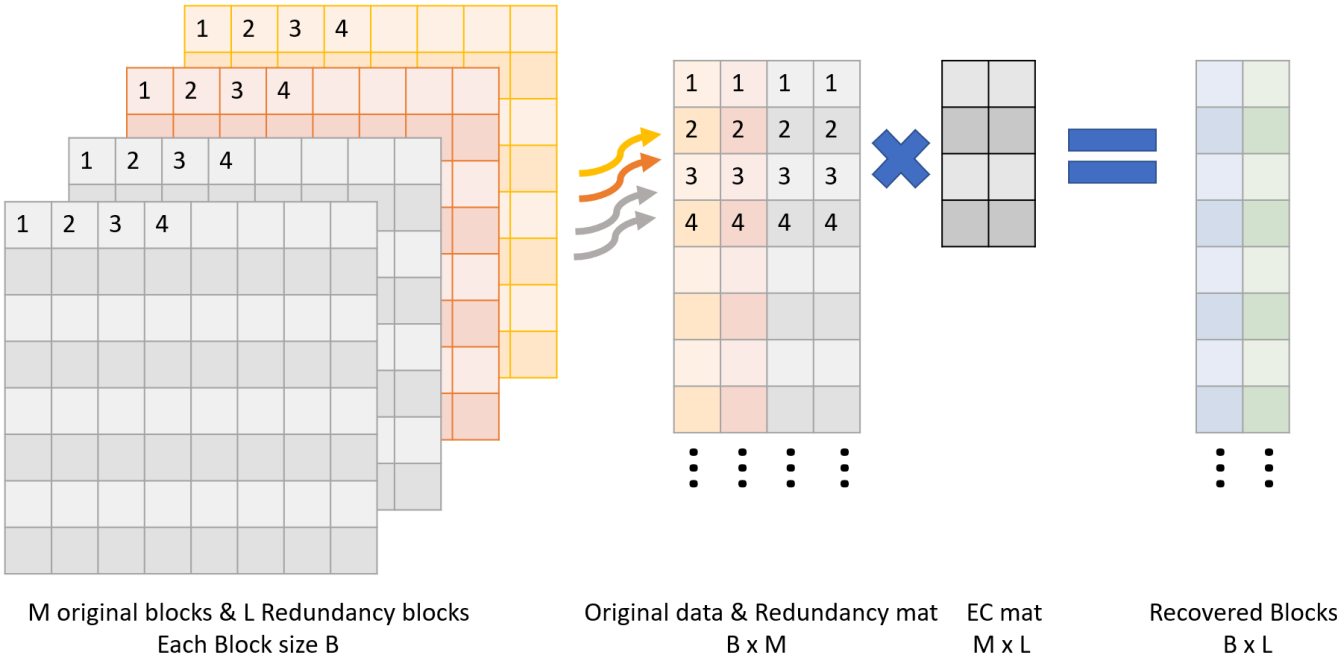
1. Input M-L original blocks via `doca_buf` (blocks that weren't impaired).
2. Input L⩽K (any) redundancy blocks via `doca_buf` (redundancy blocks originating from create \ update tasks).

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.



Objects

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. In case there are multiple cards, then it is possible to create an EC instance per card, providing each instance with a device from a different card.

For accessing memory that is not local (from Host to DPU and vice versa), the DPU side of the application will need to pick a device with appropriate representor. See DOCA Core Device Representor Discovery

The device must stay valid until the EC instance is destroyed.

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.

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.

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.

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.

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

> **Note**
>
> Current BlueField-3 limitations:
>
> - Data block count range: 1-128
> - Redundancy block count: 1-32
> - Block size: 64B-128MB

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

Execution Phase

This section describes execution on CPU or DPU using the DOCA Core Progress Engine .

Matrix Generate

All tasks require a coding matrix.

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()`	–
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()`	–

Name	Description	Notes
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.	–

Tasks

Galois Mul Task

This task executes Galois multiplication between the original blocks and the coding matrix.

Configur1ation

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_`
Maximum block size	–	`doca_ec_cap_get_max_bloc`
Maximum buffer list length	–	`doca_ec_cap_get_max_buf_`

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. T he 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.

Note

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

Output

Common output as described in DOCA Core Task .

Task Successful Completion

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 Failed Completion

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

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

Create Task

This task creates redundancy blocks for the given original data blocks using a given coding matrix.

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_s`
Maximum block size	–	`doca_ec_cap_get_max_block_si`
Maximum buffer list length	–	`doca_ec_cap_get_max_buf_list`

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()`	–

Name	Description	Notes
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 upo n 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

Note

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

Output

Common output as described in DOCA Core Task .

Task Successful Completion

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 Failed Completion

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

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

Update Task

This task executes updates the redundancy blocks for the given original data blocks, using an update coding matrix.

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_s`
Maximum block size	–	`doca_ec_cap_get_max_block_si`
Maximum buffer list length	–	`doca_ec_cap_get_max_buf_list`

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_r` should be a multiplication of t • The data length should also b 64B and with a minimum size

Name	Description	Notes
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 upo n successful completion of the task	• The data is written to the tail extending the data segment • The minimal available memory `updated_rdnc_blocks` sho number of redundancy blocks size, aligned to 64B and, in an least 64B

Note

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

Output

Common output as described in DOCA Core Task.

Task Successful Completion

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 Failed Completion

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

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

Recover Task

This task executes recovers data blocks for, using given available original data blocks and redundancy blocks and a given coding matrix.

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`
Maximum block size	–	`doca_ec_cap_get_max_block_s`

Description	API to Set the Configuration	API to Query Support
Maximum buffer list length	–	`doca_ec_cap_get_max_buf_lis`

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 upo n 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.

Note

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

Output

Common output as described in DOCA Core Task.

Task Successful Completion

After the task is completed successfully t he data is transformed to destination.

Task Failed Completion

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

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

DOCA Erasure Coding Samples

This section provides DOCA Erasure Coding sample implementation on top of the BlueField-3 DPU (and higher).

Sample Prerequisites

N/A

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
```

Note

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 seperated i.e. 0,3,4 - default: 0
```

> **Note**
>
> 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
```

Samples

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:

        1. Writing the resulting redundancy blocks to the output directory (count is specified by `<redundancy block count>`).

        2. Freeing the task.

        3. Saving the result of the task and the callback. If there was an error in step a., the relevant error value is saved.

        4. Stopping the context.

    - Failed completion callback:

        1. Saving the result of the task and the callback.

        2. Freeing the task.

        3. Stopping the context.

10. Creating EC encoding matrix by the matrix type specified to the sample.

11. Allocating and s ubmitting 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:

        1. Writing the resulting recovered blocks to the output directory.

        2. Writing the recovered file to the output path.

        3. Freeing the task.

4. Saving the result of the task and the callback. If there was an error in step a., the relevant error value is saved.

5. Stopping the context.

- Failed completion callback:

  1. Saving the result of the task and the callback.

  2. Freeing the task.

  3. 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 s ubmitting 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_codin`
- `/opt/mellanox/doca/samples/doca_erasure_coding/doca_erasure_codin`
- `/opt/mellanox/doca/samples/doca_erasure_coding/doca_erasure_codin`

DOCA AES-GCM

This guide provides instructions on building and developing applications that require data encryption and decryption using the AES-GCM algorithm.

Introduction

> **Note**
>
> 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.

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

Environment

DOCA AES-GCM-based applications can run either on the host machine or on the NVIDIA® BlueField® DPU target.

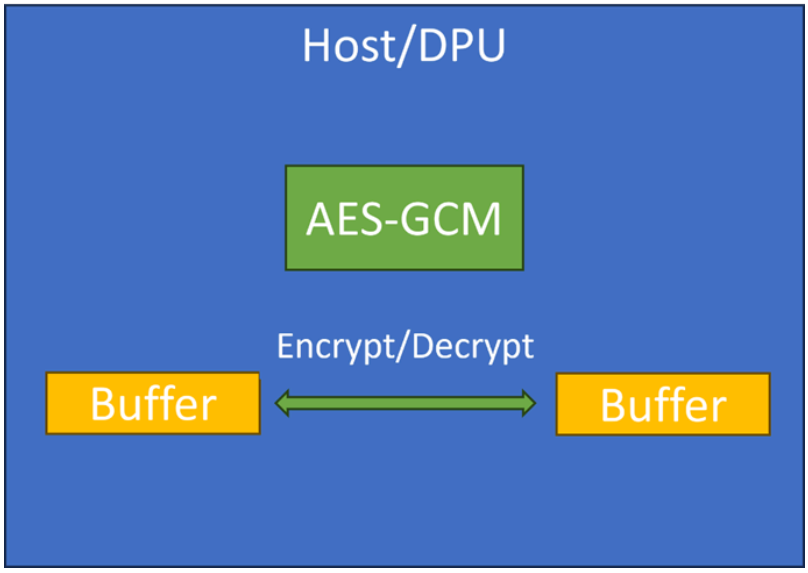
Encrypting/decrypting from the host to DPU and vice versa can only be run when the DPU is configured in DPU mode.

Architecture

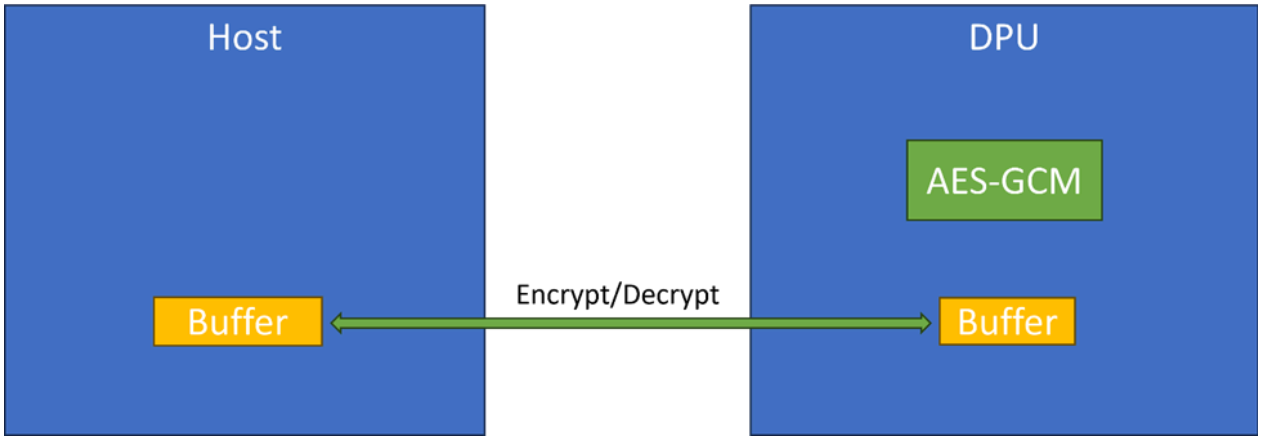
DOCA AES-GCM is a DOCA Core Context. This library leverages the DOCA Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

AES-GCM can be used to encrypt/decrypt data as illustrated in the following diagrams:

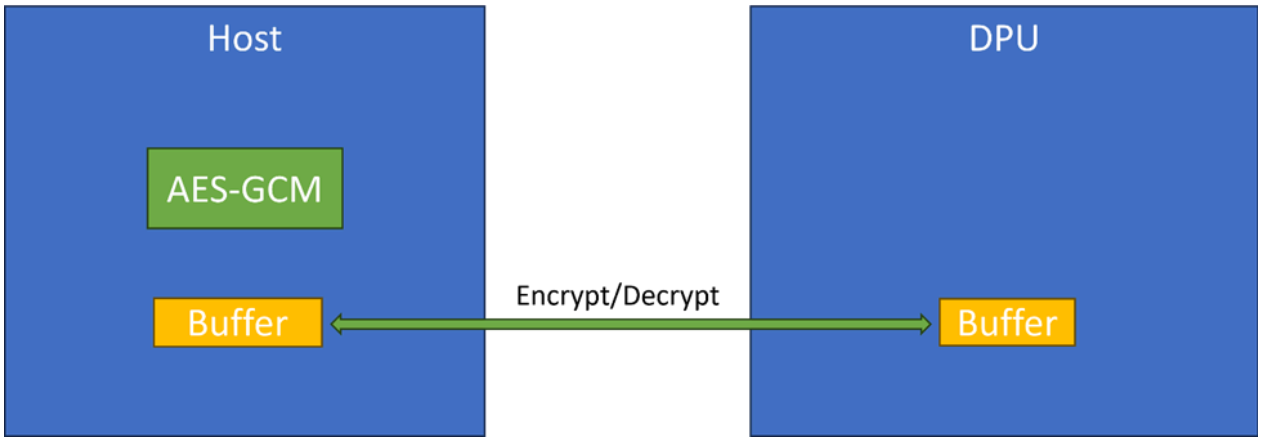
- Encrypt/decrypt from local memory to local memory:



- Using the DPU to copy memory between the host and the DPU:



- Using the host to copy memory between the host and the DPU:



Objects

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.

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.

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.

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.

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

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

Buffer Support

> **Info**
>
> AES-GCM tasks can also support encrypt/decrypt in place.

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

Execution Phase

This section describes execution on the CPU using DOCA Core Progress Engine.

Tasks

DOCA AES-GCM exposes asynchronous tasks that leverage DPU hardware according to the DOCA Core architecture.

Encrypt Task

The encrypt task allows data encryption using buffers as described in Buffer Support.

Configuration

Description	API to Set Configuration	API to Query Support
Enable the task	`doca_aes_gcm_task_encrypt_set_conf`	`doca_aes_gcm_cap_tas`
Number of tasks	`doca_aes_gcm_task_encrypt_set_conf`	`doca_aes_gcm_cap_get`
Maximal buffer size	-	`doca_aes_gcm_cap_tas`
Maximum buffer list size	-	`doca_aes_gcm_cap_tas`
Maximum initialization vector length	-	`doca_aes_gcm_cap_tas`
Enable authentication tag size	-	`doca_aes_gcm_cap_tas` `doca_aes_gcm_cap_tas`
Enable key type	-	`doca_aes_gcm_cap_tas`

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

Name	Description	Notes
destination buffer	Buffer pointing to where memory is encrypted to	The encrypted data is appended to the tail segment
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

Output

Common output as described in DOCA Core Task.

Task Successful Completion

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 Failed Completion

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

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

Decrypt Task

The decrypt task allows data decryption. Using buffers as described in Buffer Support.

Configuration

Description	API to Set Configuration	API to Query Support
Enable the task	`doca_aes_gcm_task_decrypt_set_conf`	`doca_aes_gcm_cap_tas`
Number of tasks	`doca_aes_gcm_task_decrypt_set_conf`	`doca_aes_gcm_cap_get`
Maximal buffer size	–	`doca_aes_gcm_cap_tas`
Maximum buffer list size	–	`doca_aes_gcm_cap_tas`
Maximum initialization vector length	–	`doca_aes_gcm_cap_tas`

Description	API to Set Configuration	API to Query Support
Enable authentication tag size	–	`doca_aes_gcm_cap_tas` `doca_aes_gcm_cap_tas`
Enable key type	–	`doca_aes_gcm_cap_tas`

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

Name	Description	Notes
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

Output

Common output as described in DOCA Core Task.

Task Successful Completion

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 Failed Completion

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

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

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.

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.

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

Previous State	Transition Action
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 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 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

Alternative Datapath Options

DOCA AES-GCM only supports datapath on the CPU. See Execution Phase.

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

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
```

Info

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=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
  -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 0-96-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
```

Samples

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_e`
- `/opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_encrypt/aes_gcm_e`

- `/opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_encrypt/meson.bui`

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_d`
- `/opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_decrypt/aes_gcm_d`
- `/opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_decrypt/meson.bui`

DOCA Rivermax

This guide provides instructions on building and developing applications that require media/data streaming.

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.

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

Environment

Info

DOCA Rivermax-based applications can run on the target DPU only.

Info

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
```

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.

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.

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.

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".

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)

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

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.

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)

Execution Phase

This section describes execution on CPU using DOCA Core Progress Engine.

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.

Rx Data

The Rx Data event is used by the stream to notify application that data has been received from the network.

Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	`doca_rmax_in_stream_event_rx_data_register`	–

Trigger Condition

The event is triggered anytime packet(s) arrive.

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

> **Note**
>
> 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.

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.

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.

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

Starting

This state is not expected to be reached.

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

Stopping

This state is not expected to be reached.

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 per formance
- 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

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
```

Info

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
```

> **Note**
>
> 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
```

Samples

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_`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_list_devices/rmax_list_`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_list_devices/meson.buil`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_common.h`; `/opt/mellanox/doca/samples/doca_rmax/rmax_common.c`

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_a`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_set_affinity/rmax_set_a`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_set_affinity/meson.buil`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_common.h`; `/opt/mellanox/doca/samples/doca_rmax/rmax_common.c`

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_cloc`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_set_clock/rmax_set_cloc`
- `/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`

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_crea`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_create_stream/rmax_crea`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_create_stream/meson.bui`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_common.h`; `/opt/mellanox/doca/samples/doca_rmax/rmax_common.c`

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_`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_create_stream_hds/rmax_`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_create_stream_hds/meson`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_common.h`; `/opt/mellanox/doca/samples/doca_rmax/rmax_common.c`

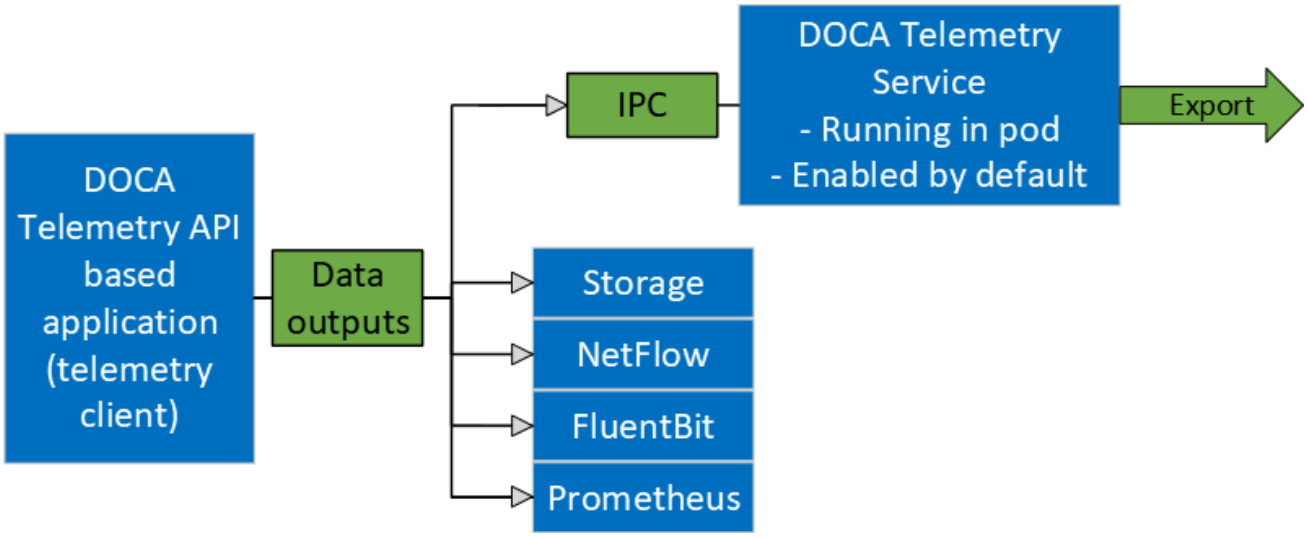
DOCA Telemetry Exporter

This guide provides an overview and configuration instructions for DOCA Telemetry API.

Introduction

DOCA Telemetry 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 API. The telemetry client side, based on the telemetry 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 clients and can collect pre-defined counters by itself. Additionally, telemetry API has built-in data outputs that can be used from telemetry 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 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.

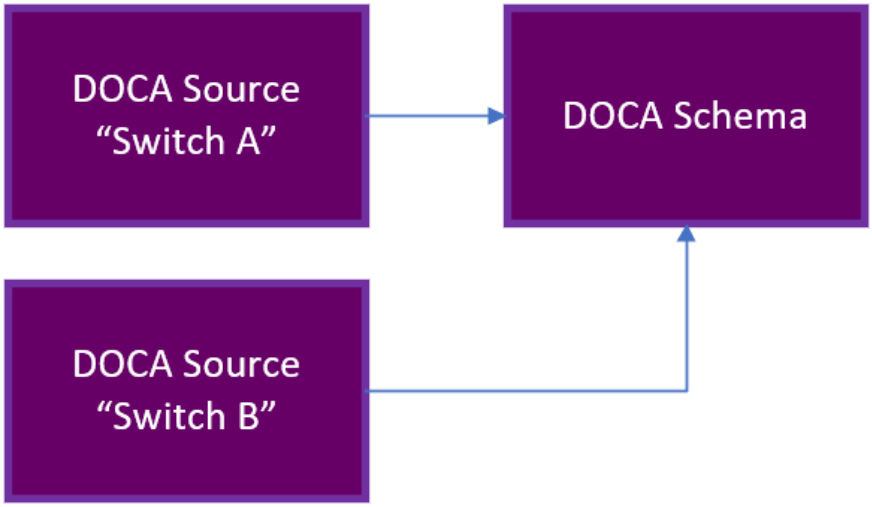
The library stores data in an internal buffer and flushes it to DTS/exporters in the following scenarios:

- Once the buffer is full. Buffer size is configurable with different attributes.
- When `doca_telemetry_source_flush(void *doca_source)` function is invoked.
- When the telemetry client terminates. If the buffer has data, it is processed before the library's context cleanup.

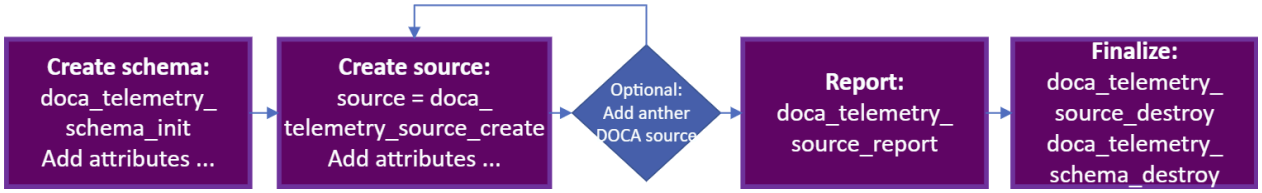
Architecture

DOCA Telemetry API is fundamentally built around four major parts:

- DOCA schema – defines a reusable structure (see `doca_telemetry_type`) of telemetry data which can be used by multiple sources



- Source – the unique identifier of the telemetry source that periodically reports telemetry data.
- Report – exports the information to the DTS
- Finalize – releases all the resources



DOCA Telemetry API Walkthrough

The NVIDIA DOCA Telemetry API's definitions can be found in the `doca_telemetry.h` file.

The following is a basic walkthrough of the needed steps for using the DOCA Telemetry API.

1. Create `doca_schema`.

1. Initialize an empty schema with default attributes:

```
struct doca_telemetry_schema *doca_schema;
doca_telemetry_schema_init("example_doca_schema_name",
&doca_schema);
```

2. Set the following attributes if needed:

   - `doca_telemetry_schema_set_buffer_attr_*(…)`
   - `doca_telemetry_schema_set_file_write_*(…)`
   - `doca_telemetry_schema_set_ipc_*(…)`

3. Add user event types:

   Event type (`struct doca_telemetry_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_type_index_t`) which can be used to refer to the event type in future API calls.

```
struct doca_telemetry_type *doca_type;
struct doca_telemetry_field *field1;

doca_telemetry_type_create(&doca_type);
doca_telemetry_field_create(&field1);

doca_telemetry_field_set_name(field1, "sport");
doca_telemetry_field_set_description(field1, "Source port")
doca_telemetry_field_set_type_name(field1,
DOCA_TELEMETRY_FIELD_TYPE_UINT16);
doca_telemetry_field_set_array_length(field1, 1);

/* The user loses ownership on field1 after a successful invocation of the function */
doca_telemetry_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_schema_add_type(doca_schema, "example_event",
doca_type, &type_index);
```

4. Apply attributes and types to start using the schema:

```
doca_telemetry_schema_start(doca_schema)
```

2. Create `doca_source`:

    1. Initialize:

```
struct doca_telemetry_source *doca_source;
doca_telemetry_source_create(doca_schema, &doca_source);
```

2. Set source ID and tag:

```
doca_telemetry_source_set_id(doca_source, "example id");
doca_telemetry_source_set_tag(doca_source, "example tag");
```

3. Apply attributes to start using the source:

```
doca_telemetry_source_start(doca_source)
```

You may optionally add more `doca_sources` if needed.

3. Collect the data per source and use:

```
doca_telemetry_source_report(source, type_index,
&my_app_test_ev1, num_events)
```

4. Finalize:

    1. For every source:

```
doca_telemetry_source_destroy(source)
```

    2. Destroy:

```
doca_telemetry_schema_destroy(doca_schema)
```

Example implementation may be found in the `telemetry_export` DOCA sample (`telemetry_export_sample.c`).

DOCA Telemetry NetFlow API Walkthrough

The DOCA telemetry API also supports NetFlow using DOCA Telemetry 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 Netflow API's definitions can be found in the `doca_telemetry_netflow.h` file.

The following are the steps to use the NetFlow API:

1. Initiate the API with an appropriate source ID:

```
doca_telemetry_netflow_init(source_id)
```

2. Set the relevant attributes:
   - `doca_telemetry_netflow_set_buffer_*(…)`
   - `doca_telemetry_netflow_set_file_write_*(…)`
   - `doca_telemetry_netflow_set_ipc_*(…)`
   - `doca_telemetry_netflow_source_set_*()`
3. Start the API to use the configured attribute:

```
doca_telemetry_netflow_start();
```

4. Form a desired NetFlow template and the corresponding NetFlow records.
5. Collect the NetFlow data.

```
doca_telemetry_netflow_send(…)
```

6. (Optional) Flush the NetFlow data to send data immediately instead of waiting for the buffer to fill:

```
doca_telemetry_netflow_flush()
```

7. Clean up the API:

```
doca_telemetry_netflow_destroy()
```

Example implementation may be found in the `telemetry_netflow_export` DOCA sample (`telemetry_netflow_export_sample.c`).

API

Refer to NVIDIA DOCA Library APIs, for more detailed information on DOCA Telemetry API.

> **Note**
>
> The pkg-config (`*.pc` file) for the DOCA Telemetry library is included in DOCA's regular definitions (i.e., `doca`).

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 the setter functions of respective attributes.

DOCA Telemetry 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_schema_set_buffer_size(doca_schema, 16 * 1024); /*
16KB - arbitrary value */
doca_telemetry_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 File Write Attributes".

DOCA Telemetry 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_schema_set_file_write_enabled(doca_schema);
doca_telemetry_schema_set_file_write_max_size(doca_schema, 1 *
1024 * 1024); /* 1 MB */
doca_telemetry_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.

DOCA Telemetry 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.

> **Note**
>
> It is important to make sure that the IPC location matches the IPC location used by DTS, otherwise IPC communication will fail.

```
doca_telemetry_schema_set_ipc_enabled(doca_schema);
doca_telemetry_schema_set_ipc_sockets_dir(doca_schema, "/path/to/sockets/");
doca_telemetry_schema_set_ipc_reconnect_time(doca_schema, 100); /* 100 milliseconds */
doca_telemetry_schema_set_ipc_reconnect_tries(doca_schema, 3);
doca_telemetry_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 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.

DOCA Telemetry 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.

> **Note**
>
> Sources attributes are mandatory and must be configured before invoking `doca_telemetry_source_start()`.

```
doca_telemetry_source_set_id(doca_source, "example_source");
doca_telemetry_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 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`.

DOCA Telemetry Netflow Collector Attributes

DOCA Telemetry 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_netflow_set_collector_addr("127.0.0.1");
doca_telemetry_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)`.

doca_telemetry_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_source_report(struct
doca_telemetry_source *doca_source,

doca_telemetry_type_index_t index,
                                          void *data,
                                          int count);
```

- `doca_source [in]` – a pointer to the `doca_telemetry_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.

doca_telemetry_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_schema_add_type(struct
doca_telemetry_schema *doca_schema,
                                            const char
*new_type_name,
                                            struct
doca_telemetry_type *type,

doca_telemetry_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.

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_schema_add_type` API call.

Note

See available `DOCA_TELEMETRY_FIELD_TYPE`s in `doca_telemetry.h`. See example of usage in `/opt/mellanox/doca/samples/doca_telemetry/telemetry_export/teler`.

Note

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_get_timestamp(doca_telemetry_timestamp_t
*timestamp);
```

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.

Inter-process Communication

IPC transport automatically transfers the data from the telemetry-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-based program must share the same `ipc_sockets` directory.

When IPC transport is enabled, the data is sent from the DOCA-telemetry-based application to the DTS process via shared memory.

To enable IPC, use the `doca_telemetry_schema_set_ipc_enabled` API function.

> **Note**
>
> IPC transport relies on system folders. For the host's usage, run the DOCA-telemetry-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_check_ipc_status(struct
doca_telemetry_source *doca_source,

doca_telemetry_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.

Using IPC with Non-container Application

When developing and testing a non-container DOCA Telemetry-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.

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 NetFlow API.

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
```

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-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
```

Note

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}
```

DOCA Telemetry Samples

This section provides DOCA Telemetry sample implementations on top of the BlueField DPU.

The telemetry samples in this document demonstrate an initial recommended configuration that covers two use cases:

- Standard DOCA Telemetry data
- DOCA Telemetry for NetFlow data

The telemetry 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.

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/<sample_name>
meson /tmp/build
ninja -C /tmp/build
```

> **Note**
>
> 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=CRITICAL, 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
```

Samples

Telemetry Export

This sample illustrates how to use the telemetry API. The sample uses a custom schema for telemetry.

The sample logic includes:

1. Configuring schema attributes.
2. Initializing schema.
3. Creating telemetry source.

4. Creating example events.
5. Reporting example events via DOCA Telemetry.
6. Destroying source and schema.

Reference:

- `/opt/mellanox/doca/samples/doca_telemetry/telemetry_export/teleme`
- `/opt/mellanox/doca/samples/doca_telemetry/telemetry_export/teleme`
- `/opt/mellanox/doca/samples/doca_telemetry/telemetry_export/meson.`

Telemetry NetFlow Export

This sample illustrates how to use the NetFlow functionality of the telemetry API.

The sample logic includes:

1. Configuring NetFlow attributes.
2. Initializing NetFlow.
3. Creating telemetry source.
4. Starting NetFlow.
5. Creating example events.
6. Reporting example events via DOCA Telemetry.
7. Destroying NetFlow.

Reference:

- `/opt/mellanox/doca/samples/doca_telemetry/telemetry_netflow_expor`
- `/opt/mellanox/doca/samples/doca_telemetry/telemetry_netflow_expor`
- `/opt/mellanox/doca/samples/doca_telemetry/telemetry_netflow_expor`

DOCA Telemetry

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.

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.

Prerequisites

In order to use DOCA telemetry Diagnostics, please ensure that the following prerequisites are met:

- fwctl, an OFED s ubsystem, must be installed. Please refer to NVIDIA MLNX_OFED Documentation v24.07-0.6.1.0 | Changes and New Features for more information.
- System must have a minimal FW version XX.43.1000 (XX - HW type; CX7 is 28, BF3 is 32).

Environment

DOCA telemetry-based applications can run either on the host machine (ConnectX-7 or Bluefield-3 and above) or on the DPU target (NVIDIA® BlueField®-3 and above).

DOCA telemetry can only be run with DPU configured in DPU mode as described in NVIDIA BlueField Modes of Operation.

Architecture

Diagnostic data is stored in hardware as a cyclic buffer of samples. Each sample represents the values of 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 each 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 obtained 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.

Sampling Period

The sampling period can be configured using `doca_telemetry_diag_set_sample_period`. In some cases, depending on the number and type of data IDs configured, the actual sample period may be higher. The actual sample period can be queried using `doca_telemetry_diag_get_sample_period` after configuring the data IDs.

Considerations for Repetitive Sampling Mode

When configuring the DOCA Telemetry Diagnostics library to repetitive sampling, it is important to ensure that the buffer is adequately sized to handle the data flow between hardware sampling and software retrieval.

To ensure smooth data processing and prevent data loss, the buffer should be large enough to accommodate at least twice the average number of samples collected during the retrieval period.

1. Determine sampling rates:

- Hardware sampling rate – the frequency at which the hardware collects data (e.g., every 100 µ sec)

- Software retrieval rate – the average time interval between successive data retrievals by the software (e.g., every 500 msec)

2. Calculate `AverageSamplesPerRetrieval` using the following equation:

   $$AverageSamplesPerRetrieval = \frac{RetrievalInterval}{SamplingInterval}$$

   For example:

   $$\frac{500msec}{100\mu sec} = 5000$$

3. To ensure reliability, configure the buffer should to hold at least twice the average number of samples per retrieval:

   $$BufferSize = 2 \times AverageSamplesPerRetrieval$$

   For example:

   $$2 \times 5000 = 10000 \text{ samples}$$

Moreover, the number of samples in the buffer should be enlarged if the retrieving process may spike occasionally. For example, if the process time between retrieval calls is up to 6 times of the average, then the number of samples should be multiplied by 6+1=7.

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.

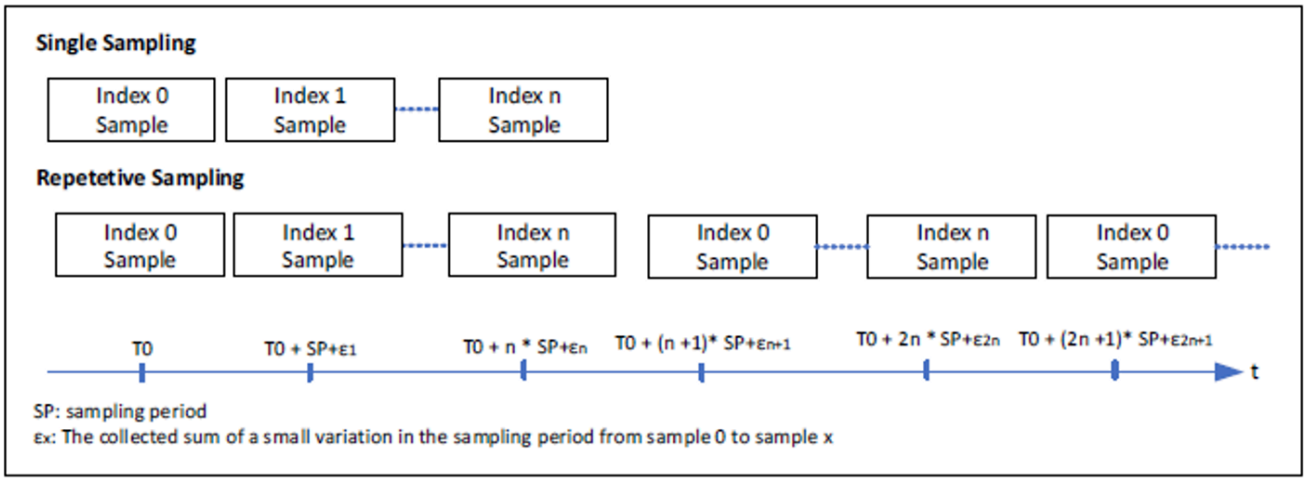
Synchronized start mode enables diagnostics counters to begin all data measurements at the same time (i.e., during the same clock cycle). This way, the sample period is guaranteed to be identical for all samples. S ynchronized start diagnostic counters can be configured to be cleared at the beginning of each sampling period,.

Note

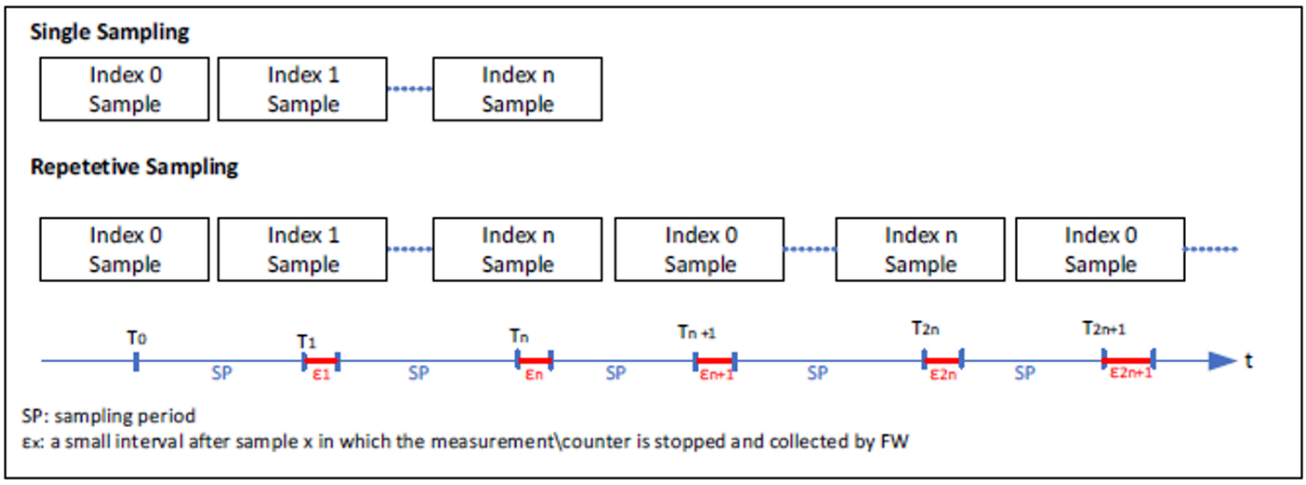
Not all data IDs can be sampled in synchronized start mode. See section "Data IDs" for additional details

The following diagrams illustrate how synchronized start affects the sampling timeline:

Without Synched Start:



With Synched Start:



Note

In synchronized start mode, counters are stopped during the collection time of each sample (illustrated in red in the diagram). If the application is required to normalize the counter to time, the actual sample period should be taken into account.

For example, if the `global_icmc_hit` (GIH) counter is sampled and the sample period is 100 µ sec, then the `global_icmc_hit` per second, should be calculated as follows: $\frac{(GIH_{Tn+1} - GIH_{Tn})}{100\mu sec}$

Output Formats

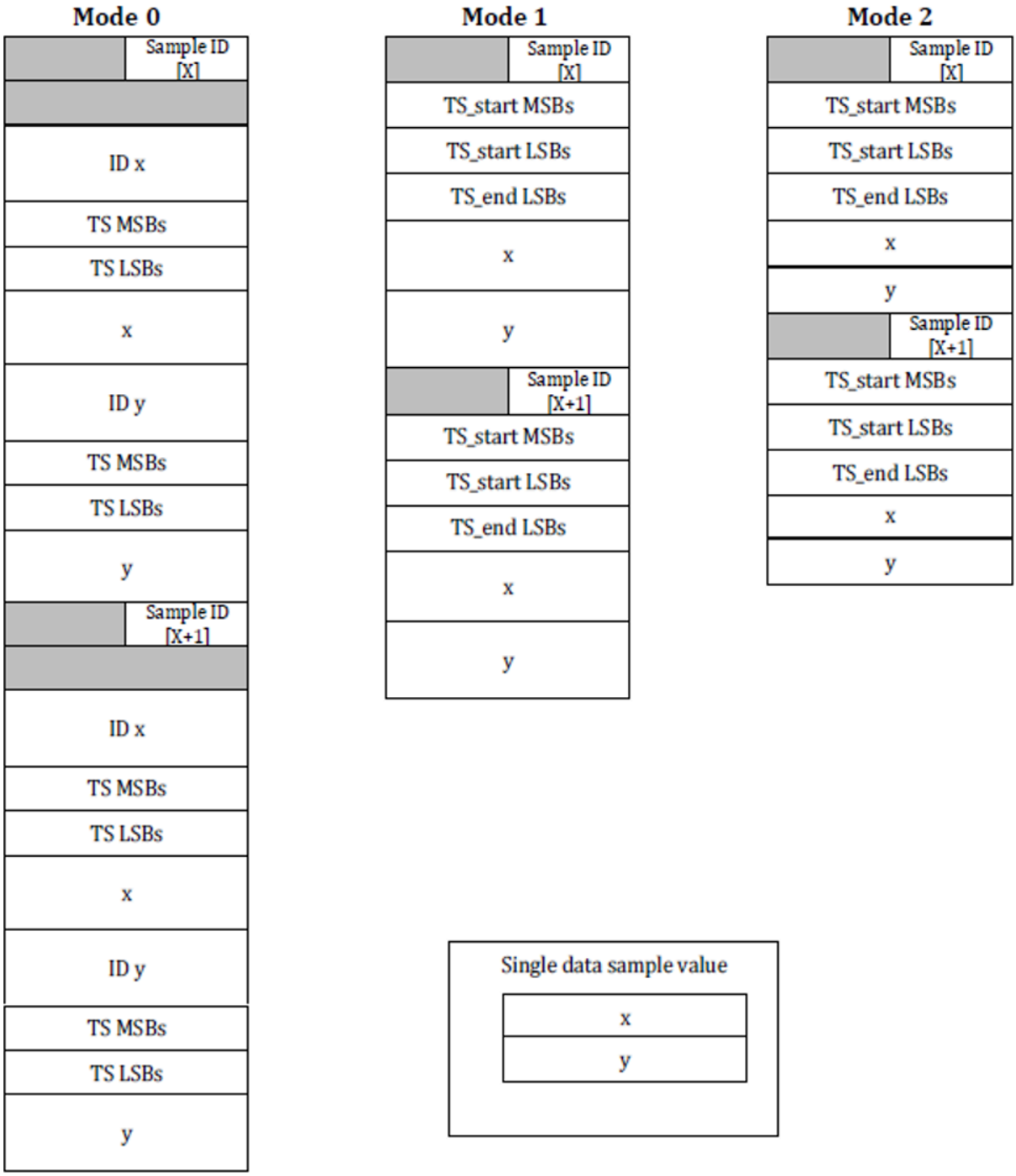
`doca_telemetry_diag` supports the following layout modes of the sampled data:

- 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)

Note

The order of the data IDs in the output is the same as the order in which the data IDs were applied, using `doca_telemetry_diag_apply_counters_list_by_id`

The sample layout of these modes is illustrated in the following diagrams:



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.

> **Note**
>
> 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.

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 should be applied by calling `doca_telemetry_diag_apply_counters_list_by_id`.

Note

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.

Note

Not all data IDs support synchronized start mode. If synchronized start mode is configured and `doca_telemetry_diag_apply_counters_list_by_id` fails with error `DOCA_ERROR_BAD_CONFIG`, this indicates that the failed data ID does not support synchronized start mode.

Configuration Phase

TBD

Execution Phase

TBD

State Machine

The following section describes the different states the `doca_telemetry_diag` context goes through, how to move states and what is allowed in each state.

Idle

The Context is Idle, and has ownership over the Diagnostics Data Registers interface.

In this state it is expected that application:

- Destroys the context (Releases the ownership).
- Applies a configuration.

Allowed operations:

- Configuring the context according to Configurations

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Configured	Call stop
Ready	Call stop
Running	Call stop

Configured

In this state it is expected that application:

- Applies the list of data IDs configuration, using the

Allowed operations:

- Checking if a data ID is supported, using `doca_telemetry_diag_check_data_id`.
- Calling stop.

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Successfully apply the configuration, calling `doca_telemetry_diag_apply_config`.

Ready

All the necessary configuration was applied and the context is ready to start sampling.

In this state it is expected that application:

- Starts the context.

Allowed operations:

- Calling stop.

It is possible to reach this state as follows:

Previous State	Transition Action
Configured	Successfully apply the list of counters, calling `doca_telemetry_diag_apply_counters_list_by_id`.

Running

In this state samples are generated and can be retrieved.

In this state it is expected that application:

- Queries the counters.

Allowed operations:

- For 'Single' sample mode - restart the context if needed.
- Calling stop.

It is possible to reach this state as follows:

Previous State	Transition Action
Ready	Successfully Start the context.

Alternative Datapath Options

DOCA Telemetry only supports datapath on the CPU.

DOCA 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=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
  -di, --data-ids                   Path to data ids JSON file
  -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, --sample-mode                sample mode (0 - single, 1 -
repetitive, 2 - on demand)
  -of, --output-format              output format
  -f, --force-ownership             Force ownership when creating
context
  -e, --example-json-path           Generate an example json file
with the default data_ids to the given path and exit immediately.
```

```
This file can be used as input later on. All other flags are
ignored
```

The sample logic includes:

1. Locating a DOCA device.
2. Initializing and configuring the `doca_telemetry_diag` instance.
3. Applying a list of data IDs to sample (either from a source JSON file or the default data IDs).
4. Starting the `doca_telemetry_diag` instance.
5. Allocating a buffer according to the sample size and amount of desired samples.
6. Querying the actual sample time, after start.
7. Retrieving samples and writing the retrieved data to a `*.csv` file (either once or periodically).
8. Stopping the data IDs sampling.
9. Releasing all resources and destroying the context.

The sample can use data IDs given by the user using a JSON file. An example of the JSON file format can be created by using the "-e" flag on the sample, to export an example JSON file containing the default data IDs to a given path.

Appendix - List of Supported Data IDs

The following table lists the data IDs currently supported by DOCA:

Name	Descriptio
`port_rx_bytes`	The numb

Name	Descriptic
`port_priority_rx_bytes`	The numb
`port_rx_packets`	The numb
`port_priority_rx_packets`	The numb
`port_rx_discard_buf_packets`	The numb physical p
`port_priority_rx_pauses_packets`	The numb and priorit
`host_rx_buffer_discards`	The numb descriptor
`host_rx_transport_pass_packets`	The numb Scatter er
`host_rx_transport_out_of_buffer_packets`	The numb associate
`host_rx_transport_out_of_buffer_hairpin_packets`	The numb associate

Name	Descriptio
`port_rx_transport_ecn_packets`	The numb which wer `11` on th
`port_rx_transport_cnp_handled_packets`	The numb Point, per
`port_tx_transport_cnp_sent_packets`	The numb
`tx_transport_done_due_to_cc_deschedule_events`	The numb limitation
`port_tx_bytes`	The numb loopback
`port_priority_tx_bytes`	The numb (excluding
`port_tx_packets`	The numb loopback
`port_priority_tx_packets`	The numb priority (e
`port_priority_tx_pauses_packets`	The numb port and p

Name	Descriptio
`pcie_link_inbound_bytes`	The numb PCIe link
`pcie_link_outbound_bytes`	The numb per PCIe li
`pcie_link_inbound_data_bytes`	The numb headers) t
`pcie_link_outbound_data_bytes`	The numb PCI (exclu
`pcie_link_write_stalled_time_no_posted_data_credits_ns`	The time p posted wr per PCIe li

Name	Descriptio
`pcie_link_write_stalled_time_no_posted_header_credits_ns`	The time p posted wr per PCIe li
`pcie_link_read_stalled_time_no_non_posted_data_credits_ns`	The time p non-poste credits pe
`pcie_link_read_stalled_time_no_non_posted_header_credits_ns`	The time p non-poste credits pe
`pcie_link_read_stalled_time_no_completion_buffers_ns`	The time p non-poste buffers pe
`pcie_link_tclass_read_stalled_time_ordering_ns`	The time p non-poste semantics

Name	Descriptio
`pcie_link_latency_total_read_ns`	The total l per PCIe li ⓘ **Inf** Div `pc` yie rea
`pcie_link_latency_total_read_packets`	The total r `pcie_li`
`pcie_link_latency_max_read_ns`	The maxir the device
`pcie_link_latency_min_read_ns`	The minim the device
`global_completion_engine_rx_cqes`	Number o
`function_completion_engine_rx_cqes`	Number o
`global_completion_engine_tx_cqes`	Number o

Name	Descriptic
`function_completion_engine_tx_cqes`	Number o
`global_icmc_request`	Number o
`global_icmc_hit`	Number o
`global_icmc_miss`	Number o

1. This counter includes loopback traffic and does not include packets discarded due to FCS, frame size, and similar errors. __ __ __ __

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

DOCA Arg Parser

This guide provides an overview and configuration instructions for DOCA Arg Parser API.

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

API

For the library API reference, refer to ARGP API documentation in NVIDIA DOCA Library APIs.

> **Note**
>
> The pkg-config (`*.pc` file) for the Arg Parser library is included in DOCA's regular definitions (i.e., `doca`).

The following sections provide additional details about the library API.

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;
```

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

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.

> **Note**
>
> 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

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.

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

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

Short Flag	Long Flag/ JSON Key	Flag Description	JSON Content	JSON Content Description
				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)

> **Note**
>
> Additional DPDK flags may be added in the "flags" JSON field.

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

Short Flag	Long Flag/JSON Key	Flag Description	JSON Content	JSON Content Description
`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
`g`	`grpc-address`	Set the IP address for the gRPC server **Note** This flag is mandatory for gRPC mode.	`"grpc-address" : "10.10.10.10"`	Set the server's IP address to 10.10.10.10
			`"grpc-address" : "10.10.10.10:1234"`	Set the server's IP address to 10.10.10.10 and the server's port to 1234

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.

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"
    }
}
```

DOCA Drivers

This section describes underlying drivers included in DOCA and includes the following pages:

- DOCA UCX
- Features Overview and Configuration

DOCA UCX

This guide provides instructions for developing applications on top of the UCX library.

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.

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>
```

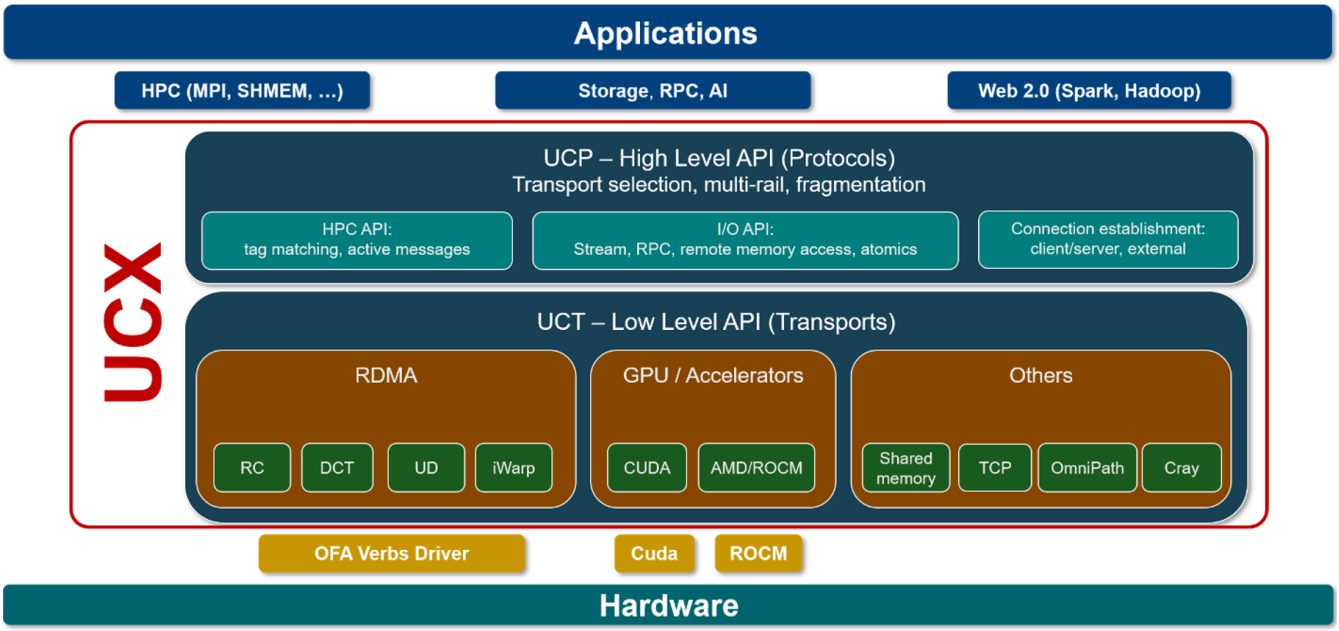
Architecture

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.



UCP Objects

This section describes the high-level communication objects that are used by most applications written on top of UCX.

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.

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.

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.

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.

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.

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).

> **Note**
>
> The pkg-config (`*.pc` file) for the UCX library is named `ucx`.

The following sections provide additional details about the library API.

ucs_status_t

An `enum` type that holds all UCX error codes.

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().

> **Note**
>
> 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.

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.

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.

> **Note**
>
> 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.

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`).

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.

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.

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.

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.

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` callback

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()`.

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.

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.

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()`.

Note

This function cannot be used from inside a callback.

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`.

> **Note**
>
> 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.

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.

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.

> **Note**
>
> 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.

UCX Best Practices

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.

> **Note**
>
> 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()` section.

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.

Features Overview and Configuration

> **Note**
>
> It is recommended to enable the "above 4G decoding" BIOS setting for features that require a large amount of PCIe resources (e.g., SR-IOV with numerous VFs, PCIe Emulated Switch, Large BAR Requests).

The chapter contains the following sections:

- InfiniBand Network
- Ethernet Network
- Storage Protocols
- Virtualization
- Resiliency

- Docker Containers
- HPC-X
- Fast Driver Unload

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

InfiniBand Interface

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.

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.

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.

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
```

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.

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.

Note

- 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.

Note

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).

Warning

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

Note

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;
```

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).

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.

> **Note**
>
> 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:

   1. Use only ports which have same MinHop

   2. First prefer the ones that go to different systemImageGuid (then the previous LID of the same LMC group)

   3. If none, prefer those which go through another NodeGuid

   4. 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.

> **Note**
>
> The user can override the node list manually.

> **Note**
>
> 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.

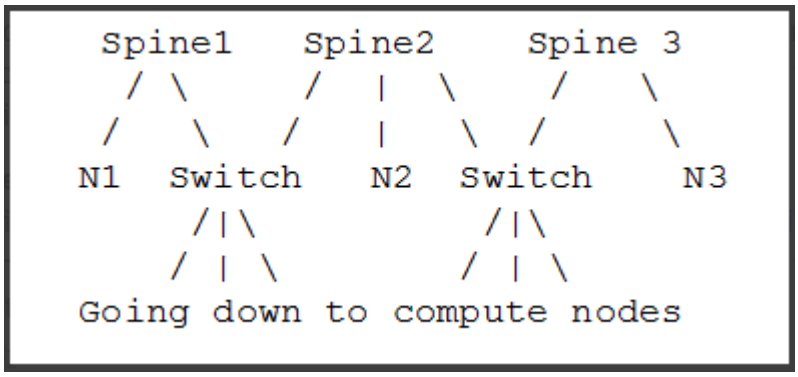
Topologies that do not comply cause a fallback to min-hop routing. Note that this can also occur on link failures which cause the topology to no longer be a "pure" fat-tree.

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.



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.

Note

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.

Note

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]:



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.

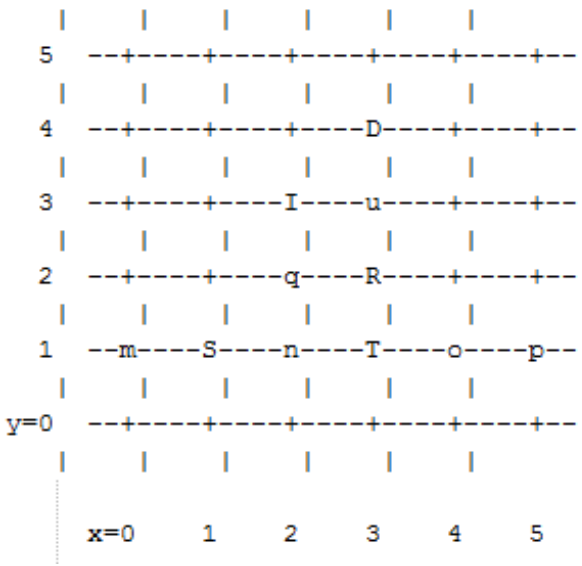
Note that it can do this without changing the path SL value; once the 1D ring m-S-n-T-o-p-m has been broken by failure, path segments using it cannot contribute to deadlock, and the x-direction dateline (between, say, x=5 and x=0) can be ignored for path segments on that ring. One result of this is that 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.

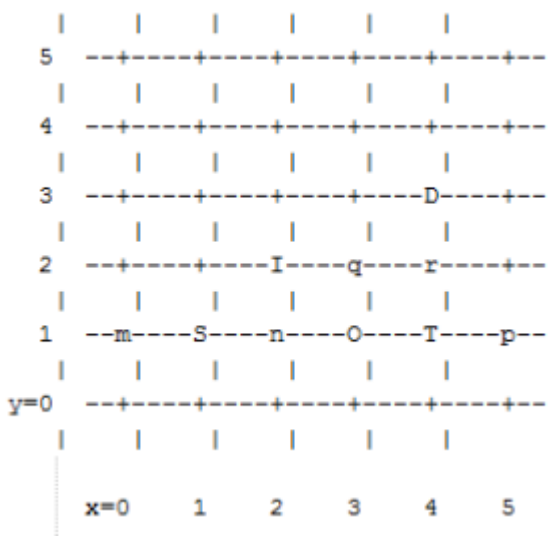
Handling a failed switch under DOR requires introducing into a path at least one turn that would be otherwise "illegal", i.e. not allowed by DOR rules. Torus-2QoS will introduce such a turn as close as possible to the failed switch in order to route around it. n the above example, suppose switch T has 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:



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:

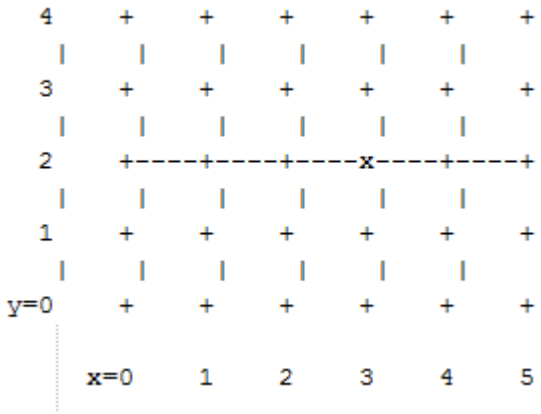


In a pristine fabric, torus-2QoS would generate the path from S to D as S-n-O-T-r-D. With failed switches O and T, torus-2QoS will generate the path S-n-I-q-r-D, with an illegal turn at switch I, and with hop I-q using a VL with bit 1 set. In contrast to the earlier examples, the second hop after the illegal turn, q-r, can be used to construct a credit loop encircling the failed switches.

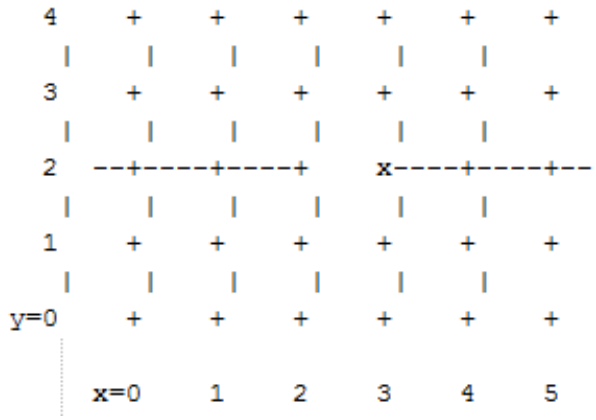
Multicast Routing

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:

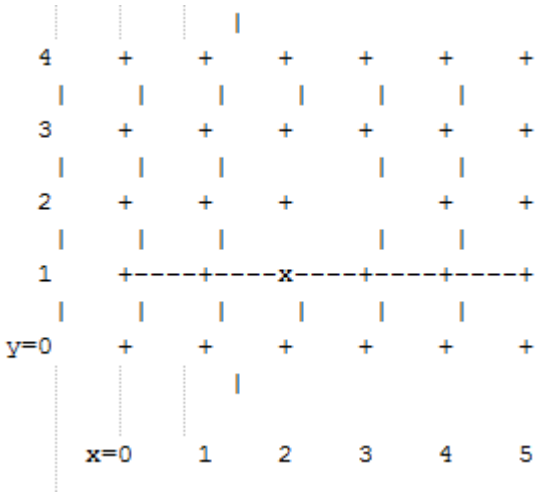


For multicast traffic routed from root to tip, every turn in the above spanning tree is a legal DOR turn. For traffic routed from tip to root, and some traffic routed through the root, turns are not legal DOR turns. However, to construct a credit loop, the union of multicast routing on this spanning 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:



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:



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

> **Note**
>
> 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`
`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`

Parameter	Description	Example
`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

> **Note**
>
> 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`

Parameter	Description	Example
`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

> **Note**
>
> 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`
`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`

Parameter	Description	Example
`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.

Note

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

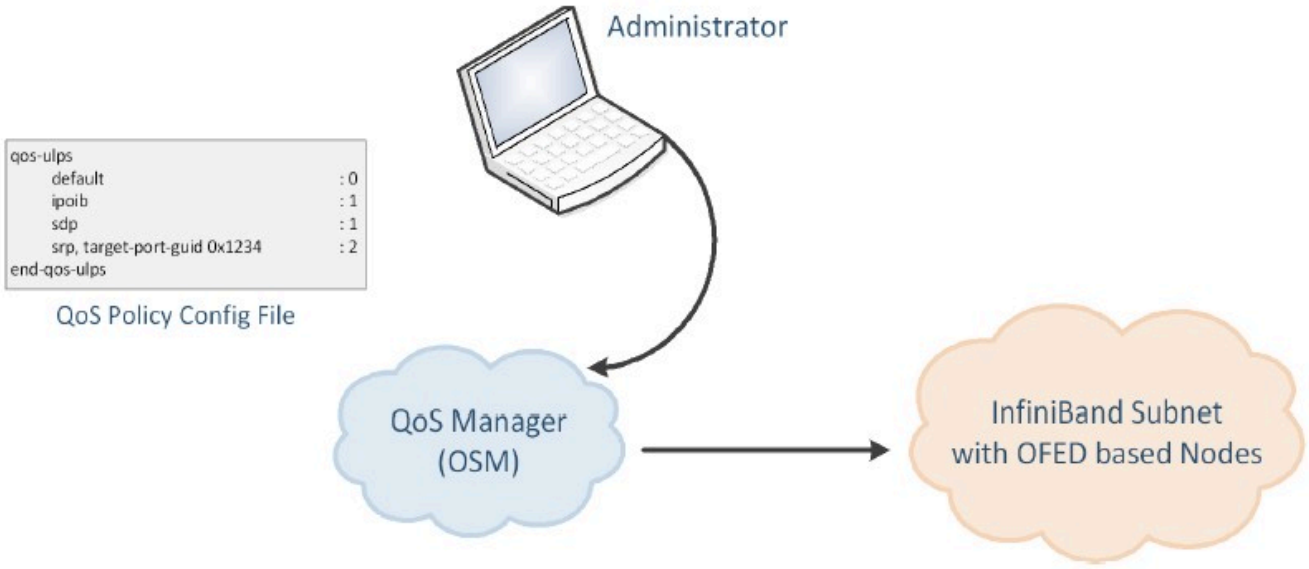
Unicast Routing Cache

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).

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



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__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.

Note

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:

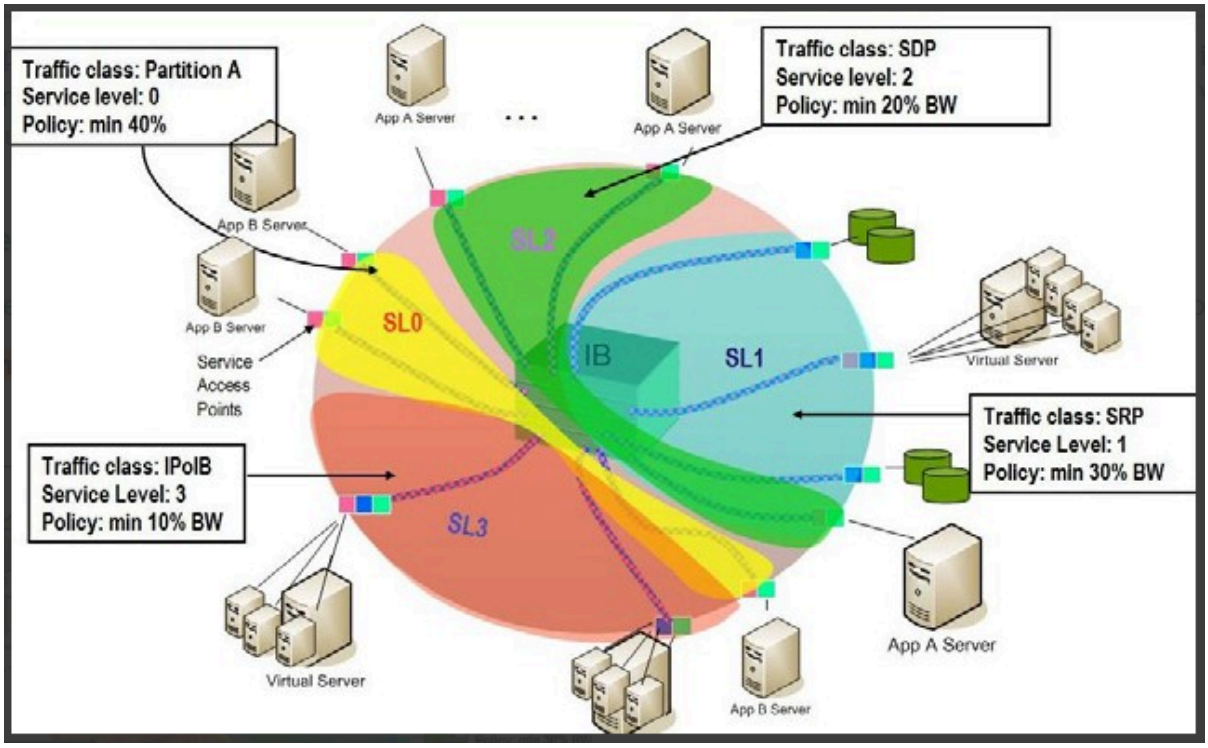
```
qos_ca_max_vls 15
qos_ca_high_limit 6
qos_ca_vlarb_high 0:4
qos_ca_vlarb_low 0:0,1:64,2:128,3:192,4:0,5:64,6:64,7:64
qos_ca_sl2vl 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,7
qos_swe_max_vls 15
qos_swe_high_limit 6
qos_swe_vlarb_high 0:4
qos_swe_vlarb_low 0:0,1:64,2:128,3:192,4:0,5:64,6:64,7:64
qos_swe_sl2vl 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)

> **Note**
>
> 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

------------------------------------------------------------------
-------
```

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
```

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

Parameter	Description	Default Value
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)

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

ACtivity ID	Activity Name	Additional Fields	Comments	Description
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_invali-dated			ucast cache invalidated
12	ucast_cache_rout-ing_done			ucast cache routing done

Offsweep Balancing

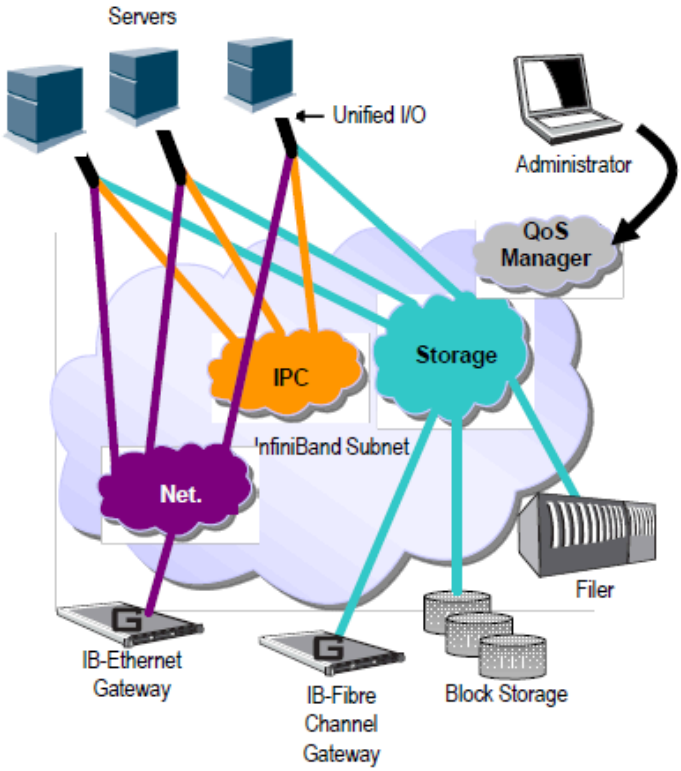
When working with minhop/dor/updn, subnet manager can re-balance routing during idle time (between sweeps).

- 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)

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.

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 fab- ric 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.

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.

> **Note**
>
> 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.

Note

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

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.

IP over InfiniBand (IPoIB)

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

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

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)

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'
```

> **Note**
>
> 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.

Note

A patch for DHCP may be required for supporting IPoIB. For further information, please see the REAME file available under the docs/dhcp/ directory.

Note

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)

> **Note**
>
> 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

> **Note**
>
> 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).

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).

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
```

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

> **Note**
>
> 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.

Note

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.

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.

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".

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.

Advanced Transport

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`.

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.

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.

> **Note**
>
> Please note that ConnectX-4 supports DCT v0 and ConnectX-5 and above support DCT v1. DCTv0 and DCT v1 are not interoperable.

MPI Tag Matching and Rendezvous Offloads

> **Note**
>
> 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 .

Optimized Memory Access

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.

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.

Note

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);
```

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.

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.

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().

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`.

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.

Relaxed Ordering (RSYNC)

> **Note**
>
> 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.

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.

Out-of-Order (OOO) Data Placement

Note

This feature is only supported on:

- ConnectX-5 adapter cards and above
- RC and XRC QPs
- DC transport

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.

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

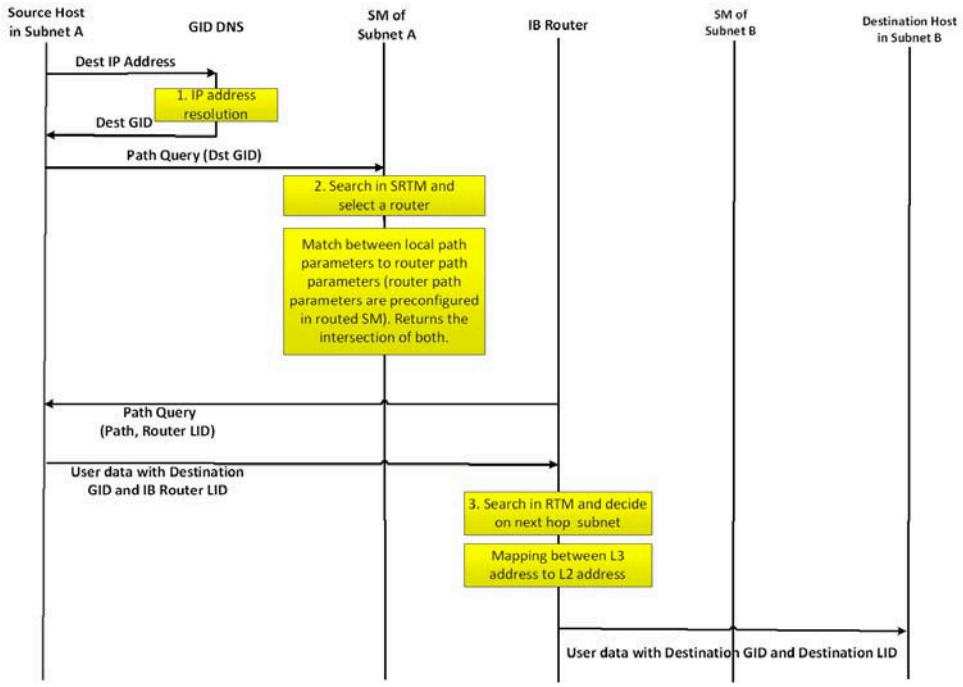
The DLID in the new LRH is built using simplified GID-to-LID mapping (where LID = 16 LSB bits of GID) thereby not requiring to send for ARP query/lookup.

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



- 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.

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

Ethernet Network

The chapter contains the following sections:

- Ethernet Interface
- Quality of Service (QoS)
- Ethtool
- Checksum Offload
- Ignore Frame Check Sequence (FCS) Errors
- RDMA over Converged Ethernet (RoCE)
- Flow Control
- Explicit Congestion Notification (ECN)
- RSS Support
- Time-Stamping
- Flow Steering
- Wake-on-LAN (WoL)

- Hardware Accelerated 802.1ad VLAN (Q-in-Q Tunneling)
- VLAN Stripping in Linux Verbs
- Offloaded Traffic Sniffer
- Dump Configuration
- Local Loopback Disable
- Kernel Transport Layer Security (kTLS) Offloads
- IPsec Crypto Offload
- IPsec Full Offload
- MACsec Full Offload

Ethernet Interface

Counters

Counters are used to provide information about how well an operating system, an application, a service, or a driver is performing. The counter data help determine system bottlenecks and fine-tune the system and application performance. The operating system, network, and devices provide counter data that an application can consume to provide users with a graphical view of how well the system is performing.

The counter index is a Queue Pair (QP) attribute given in the QP context. Multiple QPs may be associated with the same counter set. If multiple QPs share the same counter, the counter value will represent the cumulative total.

RoCE Counters

- RoCE counters are available only through sysfs located under:
  - # /sys/class/infiniband/<device>/ports/*/hw_counters/
  - # /sys/class/infiniband/<device>/hw_counters/
  - # /sys/class/infiniband/<device>/ports/*/counters/

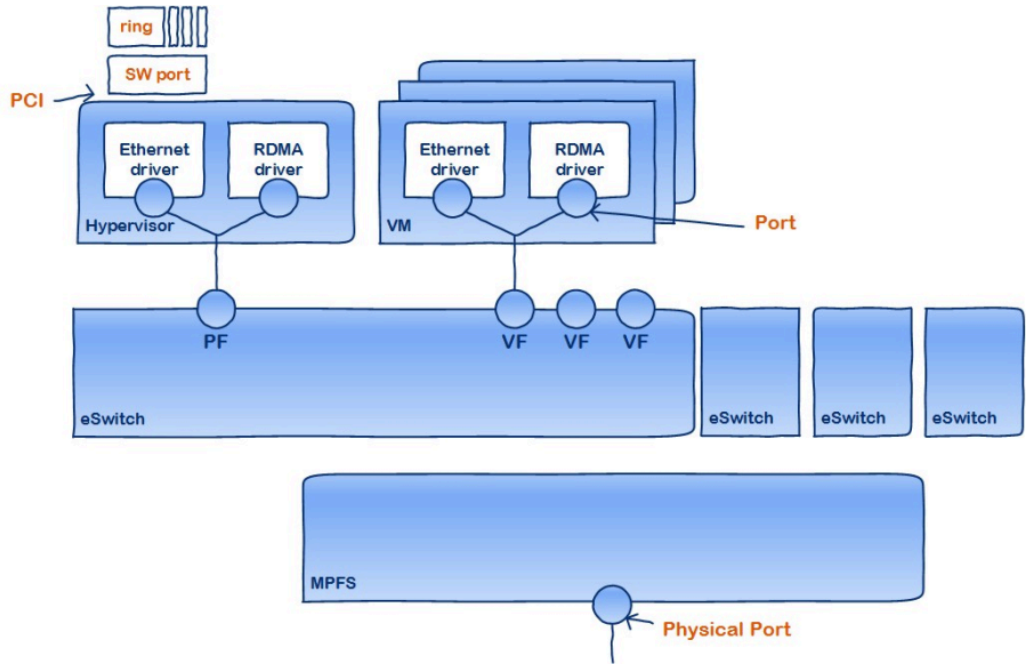
For mlx5 port and RoCE counters, refer to the Understanding mlx5 Linux Counters Community post.

SR-IOV Counters

Physical Function can also read Virtual Functions' port counters through sysfs located under # /sys/class/net/<interface_name>/device/sriov/<index>/stats/

ethtool Counters

The ethtool counters are counted in different places, according to which they are divided into groups. Each counters group may also have different counter types.



For the full list of supported ethtool counters, refer to the Understanding mlx5 ethtool Counters community post.

Persistent Naming

To avoid network interface renaming after boot or driver restart, set the desired constant interface name in the "/etc/udev/rules.d/70-persistent-net.rules" file.

- Example for Ethernet interfaces:

```
PCI device 15b3:1019 (mlx5_core)
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*",
ATTR{address}=="00:02:c9:fa:c3:50", ATTR{dev_id}=="0x0",
ATTR{type}=="1", KERNEL=="eth*", NAME="eth1"
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*",
ATTR{address}=="00:02:c9:fa:c3:51", ATTR{dev_id}=="0x0",
ATTR{type}=="1", KERNEL=="eth*", NAME="eth2"
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*",
ATTR{address}=="00:02:c9:e9:56:a1", ATTR{dev_id}=="0x0",
ATTR{type}=="1", KERNEL=="eth*", NAME="eth3"
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*",
ATTR{address}=="00:02:c9:e9:56:a2", ATTR{dev_id}=="0x0",
ATTR{type}=="1", KERNEL=="eth*", NAME="eth4"
```

- Example for IPoIB interfaces:

```
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{dev_id}=="0x0",
ATTR{type}=="32", NAME="ib0"
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{dev_id}=="0x1",
ATTR{type}=="32", NAME="ib1"
```

Interrupt Request (IRQ) Naming

Once IRQs are allocated by the driver, they are named `mlx5_comp<x>@pci:<pci_addr>`. The IRQ name is constant and is not affected by the interface state.

The mlx5_core driver allocates all IRQs during loading time to support the maximum possible number of channels. Once the driver is up, no further IRQs are freed or allocated. Changing the number of working channels does not re-allocate or free the IRQs.

Quality of Service (QoS)

Quality of Service (QoS) is a mechanism of assigning a priority to a network flow (socket, rdma_cm connection) and manage its guarantees, limitations and its priority over other flows. This is accomplished by mapping the user's priority to a hardware TC (traffic class) through a 2/3 stage process. The TC is assigned with the QoS attributes and the different flows behave accordingly.

Mapping Traffic to Traffic Classes

Mapping traffic to TCs consists of several actions which are user controllable, some controlled by the application itself and others by the system/network administrators.

The following is the general mapping traffic to Traffic Classes flow:

1. The application sets the required Type of Service (ToS).
2. The ToS is translated into a Socket Priority (sk_prio).
3. The sk_prio is mapped to a User Priority (UP) by the system administrator (some applications set sk_prio directly).
4. The UP is mapped to TC by the network/system administrator.
5. TCs hold the actual QoS parameters

QoS can be applied on the following types of traffic. However, the general QoS flow may vary among them:

- **Plain Ethernet** - Applications use regular inet sockets and the traffic passes via the kernel Ethernet driver
- **RoCE** - Applications use the RDMA API to transmit using Queue Pairs (QPs)
- **Raw Ethernet QP** - Application use VERBs API to transmit using a Raw Ethernet QP

Plain Ethernet Quality of Service Mapping

Applications use regular inet sockets and the traffic passes via the kernel Ethernet driver. The following is the Plain Ethernet QoS mapping flow:

1. The application sets the ToS of the socket using setsockopt (IP_TOS, value).

2. ToS is translated into the sk_prio using a fixed translation:

```
TOS 0 <=> sk_prio 0
TOS 8 <=> sk_prio 2
TOS 24 <=> sk_prio 4
TOS 16 <=> sk_prio 6
```

3. The Socket Priority is mapped to the UP in the following conditions:

    1. If the underlying device is a VLAN device, egress_map is used controlled by the vconfig command. This is per VLAN mapping.

    2. If the underlying device is not a VLAN device, the mapping is done in the driver.

4. The UP is mapped to the TC as configured by the mlnx_qos tool or by the lldpad daemon if DCBX is used.

Note

Socket applications can use setsockopt (SK_PRIO, value) to directly set the sk_prio of the socket. In this case, the ToS to sk_prio fixed mapping is not needed. This allows the application and the administrator to utilize more than the 4 values possible via ToS.

Note

In the case of a VLAN interface, the UP obtained according to the above mapping is also used in the VLAN tag of the traffic.

RoCE Quality of Service Mapping

Applications use RDMA-CM API to create and use QPs. The following is the RoCE QoS mapping flow:

1. The application sets the ToS of the QP using the rdma_set_option option(RDMA_OPTION_ID_TOS, value).

2. ToS is translated into the Socket Priority (sk_prio) using a fixed translation:

```
TOS 0 <=> sk_prio 0
TOS 8 <=> sk_prio 2
TOS 24 <=> sk_prio 4
TOS 16 <=> sk_prio 6
```

3. The Socket Priority is mapped to the User Priority (UP) using the tc command.

- In the case of a VLAN device where the parent real device is used for the purpose of this mapping
- If the underlying device is a VLAN device, and the parent real device was not used for the mapping, the VLAN device's egress_map is used

4. UP is mapped to the TC as configured by the mlnx_qos tool or by the lldpad daemon if DCBX is used.

> **Note**
>
> With RoCE, there can only be 4 predefined ToS values for the purpose of QoS mapping.

Map Priorities with set_egress_map

For RoCE old kernels that do not support set_egress_map, use the tc_wrap script to map between sk_prio and UP. Use tc_wrap with option -u. For example:

```
tc_wrap -i <ethX> -u <skprio2up mapping>
```

Quality of Service Properties

The different QoS properties that can be assigned to a TC are:

- Strict Priority
- Enhanced Transmission Selection (ETS)
- Rate Limit
- Trust State
- Receive Buffer
- DCBX Control Mode

Strict Priority

When setting a TC's transmission algorithm to be 'strict', then this TC has absolute (strict) priority over other TC strict priorities coming before it (as determined by the TC number: TC 7 is the highest priority, TC 0 is lowest). It also has an absolute priority over nonstrict TCs (ETS).

This property needs to be used with care, as it may easily cause starvation of other TCs.

A higher strict priority TC is always given the first chance to transmit. Only if the highest strict priority TC has nothing more to transmit, will the next highest TC be considered.

Nonstrict priority TCs will be considered last to transmit.

This property is extremely useful for low latency low bandwidth traffic that needs to get immediate service when it exists, but is not of high volume to starve other transmitters in the system.

Enhanced Transmission Selection (ETS)

Enhanced Transmission Selection standard (ETS) exploits the time periods in which the offered load of a particular Traffic Class (TC) is less than its minimum allocated bandwidth by allowing the difference to be available to other traffic classes.

After servicing the strict priority TCs, the amount of bandwidth (BW) left on the wire may be split among other TCs according to a minimal guarantee policy.

If, for instance, TC0 is set to 80% guarantee and TC1 to 20% (the TCs sum must be 100), then the BW left after servicing all strict priority TCs will be split according to this ratio.

Since this is a minimum guarantee, there is no maximum enforcement. This means, in the same example, that if TC1 did not use its share of 20%, the reminder will be used by TC0.

ETS is configured using the mlnx_qos tool (mlnx_qos) which allows you to:

- Assign a transmission algorithm to each TC (strict or ETS)
- Set minimal BW guarantee to ETS TCs

  Usage:

```
mlnx_qos -i \[options\]
```

Rate Limit

Rate limit defines a maximum bandwidth allowed for a TC. Please note that 10% deviation from the requested values is considered acceptable.

Trust State

Trust state enables prioritizing sent/received packets based on packet fields.

The default trust state is PCP. Ethernet packets are prioritized based on the value of the field (PCP/DSCP).

For further information on how to configure Trust mode, please refer to HowTo Configure Trust State on NVIDIA Adapters community post.

Note

Setting the Trust State mode shall be done before enabling SR-IOV in order to propagate the Trust State to the VFs.

Receive Buffer

By default, the receive buffer configuration is controlled automatically. Users can override the receive buffer size and receive buffer's xon and xoff thresholds using mlnx_qos tool.

For further information, please refer to HowTo Tune the Receive buffers on NVIDIA Adapters community post.

DCBX Control Mode

DCBX settings, such as "ETS" and "strict priority" can be controlled by firmware or software. When DCBX is controlled by firmware, changes of QoS settings cannot be done by the software. The DCBX control mode is configured using the mlnx_qos -d os/fw command.

For further information on how to configure the DCBX control mode, please refer to mlnx_qos community post.

Quality of Service Tools

mlnx_qos

mlnx_qos is a centralized tool used to configure QoS features of the local host. It communicates directly with the driver thus does not require setting up a DCBX daemon on the system.

The mlnx_qos tool enables the administrator of the system to:

- Inspect the current QoS mappings and configuration

  The tool will also display maps configured by TC and vconfig set_egress_map tools, in order to give a centralized view of all QoS mappings.

- Set UP to TC mapping
- Assign a transmission algorithm to each TC (strict or ETS)
- Set minimal BW guarantee to ETS TCs
- Set rate limit to TCs
- Set DCBX control mode
- Set cable length
- Set trust state

> **Note**
>
> For an unlimited ratelimit, set the ratelimit to 0.

Usage

```
mlnx_qos -i <interface> \[options\]
```

Options

--version	Show the program's version number and exit
-h, --help	Show this help message and exit
-f LIST, --pfc=LIST	Set priority flow control for each priority. LIST is a comma separated value for each priority starting from 0 to 7. Example: 0,0,0,0,1,1,1,1 enable PFC on TC4-7

-p LIST, --prio_tc=LIST	Maps UPs to TCs. LIST is 8 comma-separated TC numbers. Example: 0,0,0,0,1,1,1,1 maps UPs 0-3 to TC0, and UPs 4-7 to TC1
-s LIST, --tsa=LIST	Transmission algorithm for each TC. LIST is comma separated algorithm names for each TC. Possible algorithms: strict, ets and vendor. Example: vendor,strict,ets,ets,ets,ets,ets,ets sets TC0 to vendor, TC1 to strict, TC2-7 to ets
-t LIST, --tcbw=LIST	Set the minimally guaranteed %BW for ETS TCs. LIST is comma-separated percents for each TC. Values set to TCs that are not configured to ETS algorithm are ignored but must be present. Example: if TC0,TC2 are set to ETS, then 10,0,90,0,0,0,0,0will set TC0 to 10% and TC2 to 90%. Percents must sum to 100
-r LIST, --ratelimit=LIST	Rate limit for TCs (in Gbps). LIST is a comma-separated Gbps limit for each TC. Example: 1,8,8 will limit TC0 to 1Gbps, and TC1,TC2 to 8 Gbps each
-d DCBX, --dcbx=DCBX	Set dcbx mode to firmware controlled(fw) or OS controlled(os). Note, when in OS mode, mlnx_qos should not be used in parallel with other dcbx tools, such as lldptool
--trust=TRUST	set priority trust state to pcp or dscp
--dscp2prio=DSCP2PRIO	Set/del a (dscp,prio) mapping. Example 'set,30,2' maps dscp 30 to priority 2. 'del,30,2' resets the dscp 30 mapping back to the default setting priority 0
--cable_len=CABLE_LEN	Set cable_len for buffer's xoff and xon thresholds
-i INTF, --interface=INTF	Interface name
-a	Show all interface's TCs

Get Current Configuration

```
ofed_scripts/utils/mlnx_qos -i ens1f0
DCBX mode: OS controlled
Priority trust state: dscp
dscp2prio mapping:
        prio:0 dscp:07,06,05,04,03,02,01,00,
        prio:1 dscp:15,14,13,12,11,10,09,08,
        prio:2 dscp:23,22,21,20,19,18,17,16,
        prio:3 dscp:31,30,29,28,27,26,25,24,
        prio:4 dscp:39,38,37,36,35,34,33,32,
        prio:5 dscp:47,46,45,44,43,42,41,40,
        prio:6 dscp:55,54,53,52,51,50,49,48,
        prio:7 dscp:63,62,61,60,59,58,57,56,
Cable len: 7
PFC configuration:
        priority 0 1 2 3 4 5 6 7
        enabled 0 0 0 0 0 0 0 0
tc: 0 ratelimit: unlimited, tsa: vendor
         priority: 1
tc: 1 ratelimit: unlimited, tsa: vendor
         priority: 0
tc: 2 ratelimit: unlimited, tsa: vendor
         priority: 2
tc: 3 ratelimit: unlimited, tsa: vendor
         priority: 3
tc: 4 ratelimit: unlimited, tsa: vendor
         priority: 4
tc: 5 ratelimit: unlimited, tsa: vendor
         priority: 5
tc: 6 ratelimit: unlimited, tsa: vendor
         priority: 6
tc: 7 ratelimit: unlimited, tsa: vendor
         priority: 7
```

Set ratelimit. 3Gbps for tc0 4Gbps for tc1 and 2Gbps for tc2

```
# mlnx_qos -i <interface> -p 0,1,2 -r 3,4,2
tc: 0 ratelimit: 3 Gbps, tsa: strict
         up:  0
                  skprio: 0
                  skprio: 1
                  skprio: 2 (tos: 8)
                  skprio: 3
                  skprio: 4 (tos: 24)
                  skprio: 5
                  skprio: 6 (tos: 16)
                  skprio: 7
                  skprio: 8
                  skprio: 9
                  skprio: 10
                  skprio: 11
                  skprio: 12
                  skprio: 13
                  skprio: 14
                  skprio: 15
         up:  3
         up:  4
         up:  5
         up:  6
         up:  7
tc: 1 ratelimit: 4 Gbps, tsa: strict
         up:  1
tc: 2 ratelimit: 2 Gbps, tsa: strict
         up:  2
```

ConfigureQoS. Map UP0,7 to tc0,1,2,3 to tc1 and 4,5,6 to tc2. Set tc0,tc1 as ets and tc2 as strict. Divide ets 30% for tc0 and 70% for tc1

```
# mlnx_qos -i <interface> -s ets,ets,strict -p 0,1,1,1,2,2,2 -t 30,70
tc: 0 ratelimit: 3 Gbps, tsa: ets, bw: 30%
         up:  0
                 skprio: 0
                 skprio: 1
                 skprio: 2 (tos: 8)
                 skprio: 3
                 skprio: 4 (tos: 24)
                 skprio: 5
                 skprio: 6 (tos: 16)
                 skprio: 7
                 skprio: 8
                 skprio: 9
                 skprio: 10
                 skprio: 11
                 skprio: 12
                 skprio: 13
                 skprio: 14
                 skprio: 15
  up:  7
tc: 1 ratelimit: 4 Gbps, tsa: ets, bw: 70%
         up:  1
         up:  2
         up:  3
tc: 2 ratelimit: 2 Gbps, tsa: strict
         up:  4
         up:  5
         up:  6
```

tc and tc_wrap.py

The tc tool is used to create 8 Traffic Classes (TCs).

The tool will either use the sysfs (/sys/class/net//qos/tc_num) or the tc tool to create the TCs.

Usage

```
tc_wrap.py -i <interface> \[options\]
```

Options

--version	show program's version number and exit
-h, --help	show this help message and exit
-u SKPRIO_UP, --skprio_up=SKPRIO_UP	maps sk_prio to priority for RoCE. LIST is <=16 comma separated priority. index of element is sk_prio
-i INTF, --interface=INTF	Interface name

Example

Run:

```
tc_wrap.py -i enp139s0
```

Output:

```
Tarrfic classes are set to 8

UP  0
        skprio: 0 (vlan 5)
UP  1
        skprio: 1 (vlan 5)
UP  2
        skprio: 2 (vlan 5 tos: 8)
UP  3
        skprio: 3 (vlan 5)
UP  4
        skprio: 4 (vlan 5 tos: 24)
UP  5
        skprio: 5 (vlan 5)
UP  6
        skprio: 6 (vlan 5 tos: 16)
UP  7
        skprio: 7 (vlan 5)
```

Additional Tools

tc tool compiled with the sch_mqprio module is required to support kernel v2.6.32 or higher. This is a part of iproute2 package v2.6.32-19 or higher. Otherwise, an alternative custom sysfs interface is available.

- mlnx_qos tool (package: ofed-scripts) requires python version 2.5 < = X
- tc_wrap.py (package: ofed-scripts) requires python version 2.5 < = X

Packet Pacing

ConnectX-4 and above devices allow packet pacing (traffic shaping) per flow. This capability is achieved by mapping a flow to a dedicated send queue and setting a rate limit

on that Send queue.

Note the following:

- Up to 512 send queues are supported
- 16 different rates are supported
- The rates can vary from 1 Mbps to line rate in 1 Mbps resolution
- Multiple queues can be mapped to the same rate (each queue is paced independently)
- It is possible to configure rate limit per CPU and per flow in parallel

System Requirements

- Driver v3.3 or higher
- Linux kernel v4.1 or higher
- ConnectX-4 or ConnectX-4 Lx adapter cards with an official firmware version

Packet Pacing Configuration

> **Note**
>
> This configuration is non-persistent and does not survive driver restart.

1. Firmware Activation:

   First, make sure MFT service (mst) is started:

```
# mst start
```

Then run:

```
#echo "MLNX_RAW_TLV_FILE" > /tmp/mlxconfig_raw.txt
#echo “0x00000004 0x0000010c 0x00000000 0x00000001" >>
/tmp/mlxconfig_raw.txt
#yes | mlxconfig -d <mst_dev> -f /tmp/mlxconfig_raw.txt
set_raw > /dev/null
#reboot /mlxfwreset
```

```
#echo "MLNX_RAW_TLV_FILE" > /tmp/mlxconfig_raw.txt
#echo “0x00000004 0x0000010c 0x00000000 0x00000000" >>
/tmp/mlxconfig_raw.txt
#yes | mlxconfig -d <mst_dev >-f /tmp/mlxconfig_raw.txt
set_raw > /dev/null
#reboot /mlxfwreset
```

2. Driver Activation:

   There are two operation modes for Packet Pacing:

1. Rate limit per CPU core:

   When XPS is enabled, traffic from a CPU core will be sent using the corresponding send queue. By limiting the rate on that queue, the transmit rate on that CPU core will be limited. For example:

```
echo 300 > /sys/class/net/ens2f1/queues/tx-0/tx_maxrate
```

In this case, the rate on Core 0 (tx-0) is limited to 300Mbit/sec.

2. Rate limit per flow:

    1. The driver allows opening up to 512 additional send queues using the following command:

        ```
        ethtool -L ens2f1 other 1200
        ```

        In this case, 1200 additional queues are opened

    2. Create flow mapping.

        Users can map a certain destination IP and/or destination layer 4 Port to a specific send queue. The match precedence is as follows:

- IP + L4 Port
- IP only
- L4 Port only
- No match (the flow would be mapped to default queues)

    To create flow mapping:

    Configure the destination IP. Write the IP address in hexadecimal representation to the relevant sysfs entry. For example, to map IP address 192.168.1.1 (0xc0a80101) to send queue 310, run the following command:

    ```
    echo 0xc0a80101 > /sys/class/net/ens2f1/queues/tx-310/flow_map/dst_ip
    ```

    To map Destination L4 3333 port (either TCP or UDP) to the same queue, run:

```
echo 3333 > /sys/class/net/ens2f1/queues/tx-310/flow_map/dst_port
```

From this point on, all traffic destined to the given IP address and L4 port will be sent using send queue 310. All other traffic will be sent using the original send queue.

iii. Limit the rate of this flow using the following command:

```
echo 100 > /sys/class/net/ens2f1/queues/tx-310/tx_maxrate
```

Note

Each queue supports only a single IP+Port combination.

Ethtool

Ethtool is a standard Linux utility for controlling network drivers and hardware, particularly for wired Ethernet devices. It can be used to:

- Get identification and diagnostic information
- Get extended device statistics
- Control speed, duplex, auto-negotiation and flow control for Ethernet devices
- Control checksum offload and other hardware offload features
- Control DMA ring sizes and interrupt moderation

- Flash device firmware using a .mfa2 image

Ethtool Supported Options

Options	Description
ethtool --set-priv-flags eth<x> <priv flag> <on/off>	Enables/disables driver feature matching the given private flag.
ethtool --show-priv-flags eth<x>	Shows driver private flags and their states (ON/OFF).
ethtool -a eth<x>	Queries the pause frame settings.
ethtool -A eth<x> [rx on|off] [tx on|off]	Sets the pause frame settings.
ethtool -c eth<x>	Queries interrupt coalescing settings.
ethtool -C eth<x> [pkt-rate-low N] [pkt-rate-high N] [rx-usecs-low N] [rx-usecs-high N]	Sets the values for packet rate limits and for moderation time high and low values.
ethtool -C eth<x> [rx-usecs N] [rx-frames N]	Sets the interrupt coalescing setting. rx-frames will be enforced immediately, rx-usecs will be enforced only when adaptive moderation is disabled. **Note**: usec settings correspond to the time to wait after the *last* packet is sent/received before triggering an interrupt.
ethtool -C eth<x> adaptive-rx on|off	Enables/disables adaptive interrupt moderation. By default, the driver uses adaptive interrupt moderation for the receive path, which adjusts the moderation time to the traffic pattern.
ethtool -C eth<x> adaptive-tx on|off	**Note**: Supported by mlx5e for ConnectX-4 and above adapter cards. Enables/disables adaptive interrupt moderation. By default, the driver uses adaptive interrupt moderation for the transmit path, which adjusts the moderation parameters (time/frames) to the traffic pattern.
ethtool -g eth<x>	Queries the ring size values.
ethtool -G eth<x> [rx <N>] [tx <N>]	Modifies the ring size.

Options	Description
ethtool -i eth<x>	Checks driver and device information. For example: driver: mlx5_core version: 5.1-0.4.0 firmware-version: 4.6.4046 (MT_QEMU000000) expansion-rom-version: bus-info: 0000:07:00.0 supports-statistics: yes supports-test: yes supports-eeprom-access: no supports-register-dump: no supports-priv-flags: yes
ethtool -k eth<x>	Queries the stateless offload status.

Options	Description
ethtool -K eth<x> [rx on|off] [tx on|off] [sg on|off] [tso on|off] [lro on|off] [gro on|off] [gso on|off] [rxvlan on|off] [txvlan on|off] [ntuple on/off] [rxhash on/off] [rx-all on/off] [rx-fcs on/off]	Sets the stateless offload status. TCP Segmentation Offload (TSO), Generic Segmentation Offload (GSO): increase outbound throughput by reducing CPU overhead. It works by queuing up large buffers and letting the network interface card split them into separate packets. Large Receive Offload (LRO): increases inbound throughput of high-bandwidth network connections by reducing CPU overhead. It works by aggregating multiple incoming packets from a single stream into a larger buffer before they are passed higher up the networking stack, thus reducing the number of packets that have to be processed. LRO is available in kernel versions < 3.1 for untagged traffic. Hardware VLAN insertion Offload (txvlan): When enabled, the sent VLAN tag will be inserted into the packet by the hardware. **Note**: LRO will be done whenever possible. Otherwise GRO will be done. Generic Receive Offload (GRO) is available throughout all kernels. Hardware VLAN Striping Offload (rxvlan): When enabled received VLAN traffic will be stripped from the VLAN tag by the hardware. RX FCS (rx-fcs): Keeps FCS field in the received packets.Sets the stateless offload status. RX FCS validation (rx-all): Ignores FCS validation on the received packets.
ethtool -l eth<x>	Shows the number of channels.
ethtool -L eth<x> [rx <N>] [tx <N>]	Sets the number of channels. **Notes**: • This also resets the RSS table to its default distribution, which is uniform across the cores on the NUMA (non-uniform memory access) node that is closer to the NIC. • For ConnectX®-4 cards, use ethtool -L eth<x> combined <N> to set both RX and TX channels.
ethtool -m|--dump-module-eeprom eth<x> [raw on|off] [hex on|off] [offset N] [length N]	Queries/decodes the cable module eeprom information.
ethtool -p|--identify DEVNAME	Enables visual identification of the port by LED blinking [TIME-IN-SECONDS].

<table>
<tr><th>Options</th><th>Description</th></tr>
<tr><td>ethtool -p|--identify eth<x> <LED duration></td><td>Allows users to identify interface's physical port by turning the ports LED on for a number of seconds.
Note: The limit for the LED duration is 65535 seconds.</td></tr>
<tr><td>ethtool -S eth<x></td><td>Obtains additional device statistics.</td></tr>
<tr><td>ethtool -s eth<x> advertise <N> autoneg on</td><td>Changes the advertised link modes to requested link modes <N>
To check the link modes' hex values, run <code><man ethtool></code> and to check the supported link modes, run <code>ethtool eth<x></code>
For advertising new link modes, make sure to configure the entire bitmap as follows:

<table>
<tr><td>200GAUI-4 / 200GBASE-CR4/KR4</td><td>0x7c000000000000000</td></tr>
<tr><td>100GAUI-2 / 100GBASE-CR2 / KR2</td><td>0x3E00000000000000</td></tr>
<tr><td>CAUI-4 / 100GBASE-CR4 / KR4</td><td>0xF000000000</td></tr>
<tr><td>50GAUI-1 / LAUI-1/ 50GBASE-CR / KR</td><td>0x1F0000000000000</td></tr>
<tr><td>50GAUI-2 / LAUI-2/ 50GBASE-CR2/KR2</td><td>0x10C00000000</td></tr>
<tr><td>XLAUI-4/XLPPI-4 // 40G</td><td>0x7800000</td></tr>
<tr><td>25GAUI-1/ 25GBASE-CR / KR</td><td>0x380000000</td></tr>
<tr><td>XFI / XAUI-1 // 10G</td><td>0x7C0000181000</td></tr>
<tr><td>5GBASE-R</td><td>0x1000000000000</td></tr>
<tr><td>2.5GBASE-X / 2.5GMII</td><td>0x820000000000</td></tr>
<tr><td>1000BASE-X / SGMII</td><td>0x20000020020</td></tr>
</table>

Notes:

Both previous and new link modes configurations are supported, however, they must be run separately.
Any link mode configuration on Kernels below v5.1 and ConnectX-6 HCAs will result in the advertisement of the full capabilities.
<code><autoneg on></code> only sends a hint to the driver that the user wants to modify advertised link modes and not speed.

</td></tr>
</table>

Options	Description
ethtool -s eth<x> msglvl [N]	Changes the current driver message level.
ethtool -s eth<x> speed <SPEED> autoneg off	Changes the link speed to requested <SPEED>. To check the supported speeds, run `ethtool eth<x>`. **Note**: does not set autoneg OFF, it only hints the driver to set a specific speed.
ethtool -t eth<x>	Performs a self-diagnostics test.
ethtool -T eth<x>	Shows time stamping capabilities
ethtool -x eth<x>	Retrieves the receive flow hash indirection table.
ethtool -X eth<x> equal a b c...	Sets the receive flow hash indirection table. **Note**: The RSS table configuration is reset whenever the number of channels is modified (using ethtool -L command).
ethtool --show-fec eth<x>	Queries current Forward Error Correction (FEC) encoding in case FEC is supported. **Note**: An output of "baser" implies Firecode encoding.
ethtool --set-fec eth<x> encoding auto|off|rs|baser	Configures Forward Error Correction (FEC). **Note**: 'baser' encoding applies to the Firecode encoding, and 'auto' regards the HCA's default.
ethtool -f|--flash <devname> FILE [N]	Flash firmware image on the device using the specified .mfa2 file (FILE). By default, the command flashes all the regions on the device unless a region number (N) is specified.

Checksum Offload

The following Receive IP/L4 Checksum Offload modes are supported.

- **CHECKSUM_UNNECESSARY**: When this mode is used, the driver indicates to the Linux Networking Stack that the hardware successfully validated the IP and L4 checksum so the Linux Networking Stack does not need to deal with IP/L4 Checksum validation.

- **CHECKSUM_COMPLETE**: When this mode is used, the driver still reports to the OS the calculated by hardware checksum value. This allows accelerating checksum validation in Linux Networking Stack, since it does not have to calculate the whole checksum including payload by itself.

- **CHECKSUM_NONE**: When this mode is used, the driver indicates to the Linux Networking Stack that it must calculate and validate the IP/L4 checksum.

Ignore Frame Check Sequence (FCS) Errors

Upon receiving packets, the packets go through a checksum validation process for the FCS field. If the validation fails, the received packets are dropped.

When FCS is enabled (disabled by default), the device does not validate the FCS field even if the field is invalid.

It is not recommended to enable FCS.

For further information on how to enable/disable FCS, please refer to ethtool option rx-fcs on/off.

RDMA over Converged Ethernet (RoCE)

Remote Direct Memory Access (RDMA) is the remote memory management capability that allows server-to-server data movement directly between application memory without any CPU involvement. RDMA over Converged Ethernet (RoCE) is a mechanism to provide this efficient data transfer with very low latencies on lossless Ethernet networks. With advances in data center convergence over reliable Ethernet, ConnectX® Ethernet adapter cards family with RoCE uses the proven and efficient RDMA transport to provide the platform for deploying RDMA technology in mainstream data center application at 10GigE and 40GigE link-speed. ConnectX® Ethernet adapter cards family with its hardware offload support takes advantage of this efficient RDMA transport (InfiniBand) services over Ethernet to deliver ultra-low latency for performance-critical and transaction-intensive applications such as financial, database, storage, and content delivery networks.

When working with RDMA applications over Ethernet link layer the following points should be noted:

- The presence of a Subnet Manager (SM) is not required in the fabric. Thus, operations that require communication with the SM are managed in a different way in RoCE. This does not affect the API but only the actions such as joining the multicast group, that need to be taken when using the API

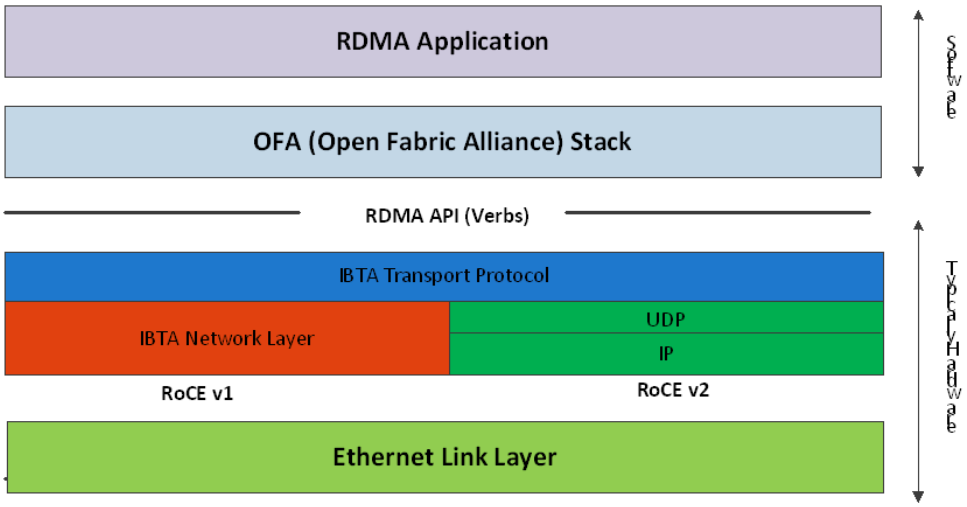
- Since LID is a layer 2 attribute of the InfiniBand protocol stack, it is not set for a port and is displayed as zero when querying the port
- With RoCE, the alternate path is not set for RC QP. Therefore, APM (another type of High Availability and part of the InfiniBand protocol) is not supported
- Since the SM is not present, querying a path is impossible. Therefore, the path record structure must be filled with relevant values before establishing a connection. Hence, it is recommended working with RDMA-CM to establish a connection as it takes care of filling the path record structure
- VLAN tagged Ethernet frames carry a 3-bit priority field. The value of this field is derived from the IB SL field by taking the 3 least significant bits of the SL field
- RoCE traffic is not shown in the associated Ethernet device's counters since it is offloaded by the hardware and does not go through Ethernet network driver. RoCE traffic is counted in the same place where InfiniBand traffic is counted; /sys/class/infiniband/<device>/ports/<port number>/counters/

RoCE Modes

RoCE encapsulates IB transport in one of the following Ethernet packets:

- **RoCEv1** - dedicated ether type (0x8915)
- **RoCEv2** - UDP and dedicated UDP port (4791)

RoCEv1 and RoCEv2 Protocol Stack



RoCEv1

RoCE v1 protocol is defined as RDMA over Ethernet header (as shown in the figure above). It uses ethertype 0x8915 and can be used with or without the VLAN tag. The regular Ethernet MTU applies on the RoCE frame.

RoCEv2

A straightforward extension of the RoCE protocol enables traffic to operate in IP layer 3 environments. This capability is obtained via a simple modification of the RoCE packet format. Instead of the GRH used in RoCE, IP routable RoCE packets carry an IP header which allows traversal of IP L3 Routers and a UDP header (RoCEv2 only) that serves as a stateless encapsulation layer for the RDMA Transport Protocol Packets over IP.

The proposed RoCEv2 packets use a well-known UDP destination port value that unequivocally distinguishes the datagram. Similar to other protocols that use UDP encapsulation, the UDP source port field is used to carry an opaque flow-identifier that allows network devices to implement packet forwarding optimizations (e.g. ECMP) while staying agnostic to the specifics of the protocol header format.

Furthermore, since this change exclusively affects the packet format on the wire, and due to the fact that with RDMA semantics packets are generated and consumed below the AP, applications can seamlessly operate over any form of RDMA service, in a completely transparent way.

Note

Both RoCEv1 and RoCEv2 are supported by default; the driver associates all GID indexes to RoCEv1 and RoCEv2, thus, a single entry for each RoCE version.

For further information, please refer to Recommended Network Configuration Examples For RoCE Deployment Community post.

GID Table Population

GID table entries are created whenever an IP address is configured on one of the Ethernet devices of the NIC's ports. Each entry in the GID table for RoCE ports has the following fields:

- GID value
- GID type
- Network device

The GID table is occupied with two GIDs, both with the same GID value but with different types. The network device in an entry is the Ethernet device with the IP address that GID is associated with. The GID format can be of 2 types; IPv4 and IPv6. IPv4 GID is an IPv4-mapped IPv6 address, while IPv6 GID is the IPv6 address itself. Layer 3 header for packets associated with IPv4 GIDs will be IPv4 (for RoCEv2) and IPv6/GRH for packets associated with IPv6 GIDs and IPv4 GIDs for RoCEv1.

GID table is exposed to userspace via sysfs

- GID values can be read from:

```
/sys/class/infiniband/{device}/ports/{port}/gids/{index}
```

- GID type can be read from:

```
/sys/class/infiniband/{device}/ports/{port}/gid_attrs/types/{ind
```

- GID net_device can be read from:

```
/sys/class/infiniband/{device}/ports/{port}/gid_attrs/ndevs/{ind
```

Setting the RoCE Mode for a Queue Pair (QP)

Setting RoCE mode for devices that support two RoCE modes is different for RC/UC QPs (connected QP types) and UD QP.

To modify an RC/UC QP (connected QP) from INIT to RTR, an Address Vector (AV) must be given. The AV, among other attributes, should specify the index of the port's GID table for the source GID of the QP. The GID type in that index will be used to set the RoCE type of the QP.

Setting RoCE Mode of RDMA_CM Applications

RDMA_CM interface requires only the active side of the peer to pass the IP address of the passive side. The RDMA_CM decides upon the source GID to be used and obtains it from the GID table. Since more than one instance of the GID value is possible, the lookup should be also according to the GID type. The type to use for the lookup is defined as a global value of the RDMA_CM module. Changing the value of the GID type for the GID table lookups is done using the cma_roce_mode script.

- ***To print the current RoCE mode for a device port:***

```
cma_roce_mode -d <dev> -p <port>
```

- ***To set the RoCE mode for a device port:***

```
cma_roce_mode -d <dev> -p <port> -m <1|2>
```

GID Table Example

The following is an example of the GID table.

DEV	PORT	INDEX	GID	IPv4	Type	Ne
mlx5_0	1	0	fe80:0000:0000:0000:ba59:9fff:fe1a:e3ea		v1	p4
mlx5_0	1	1	fe80:0000:0000:0000:ba59:9fff:fe1a:e3ea		v2	p4
mlx5_0	1	2	0000:0000:0000:0000:0000:ffff:0a0a:0a01	10.10.10.1	v1	p4
mlx5_0	1	3	0000:0000:0000:0000:0000:ffff:0a0a:0a01	10.10.10.1	v2	p4

DEV	PORT	INDEX	GID	IPv4	Type	Ne
mlx5_1	1	0	fe80:0000:0000:0000:ba59:9fff:fe1a:e3eb		v1	p4
mlx5_1	1	1	fe80:0000:0000:0000:ba59:9fff:fe1a:e3eb		v2	p4

where:

- Entries on port 1 index 0/1 are the default GIDs, one for each supported RoCE type
- Entries on port 1 index 2/3 belong to IP address 192.168.1.70 on eth1
- Entries on port 1 index 4/5 belong to IP address 193.168.1.70 on eth1.100
- Packets from a QP that is associated with these GID indexes will have a VLAN header (VID=100)
- Entries on port 1 index 6/7 are IPv6 GID. Packets from a QP that is associated with these GID indexes will have an IPv6 header

RoCE Lossless Ethernet Configuration

In order to function reliably, RoCE requires a form of flow control. While it is possible to use global flow control, this is normally undesirable, for performance reasons.

The normal and optimal way to use RoCE is to use Priority Flow Control (PFC). To use PFC, it must be enabled on all endpoints and switches in the flow path.

Configuring SwitchX® Based Switch System

To enable RoCE, the SwitchX should be configured as follows:

- Ports facing the host should be configured as access ports, and either use global pause or Port Control Protocol (PCP) for priority flow control
- Ports facing the network should be configured as trunk ports, and use Port Control Protocol (PCP) for priority flow control
- For further information on how to configure SwitchX, please refer to SwitchX User Manual

Installing and Loading the Driver

To install and load the driver:

1. Install MLNX_OFED (See Installation section for further details).

   RoCE is installed as part of mlx5 and other modules upon driver's installation.

   > **Note**
   >
   > The list of the modules that will be loaded automatically upon boot can be found in the configuration file /etc/infiniband/openib.conf.

2. Query for the device's information. Example:

   ```
   ibv_devinfo MLNX_OFED_LINUX-5.0-2.1.8.0:
   ```

3. Display the existing MLNX_OFED version.

```
ofed_info -s
hca_id: mlx5_0
        transport:                      InfiniBand (0)
        fw_ver:                         16.28.0578
        node_guid:                      ec0d:9a03:0044:3764
        sys_image_guid:                 ec0d:9a03:0044:3764
        vendor_id:                      0x02c9
        vendor_part_id:                 4121
        hw_ver:                         0x0
        board_id:                       MT_0000000009
        phys_port_cnt:                  1
                port:   1
                        state:                  PORT_ACTIVE
(4)

                        max_mtu:                4096 (5)
                        active_mtu:             1024 (3)
                        sm_lid:                 0
                        port_lid:               0
                        port_lmc:               0x00
                        link_layer:             Ethernet
```

Output Notes:

The port's state is: Ethernet is in PORT_ACTIVE state	The port state can also be obtained by running the following command: `# cat /sys/class/infiniband/mlx5_0/ports/1/state: ACTIVE`
link_layer parameter shows that port 1 is Ethernet	The link_layer can also be obtained by running the following command: `# cat /sys/class/infiniband/mlx5_0/ports/1/link_layer Ethernet`

The fw_ver parameter shows that the firmware version is 16.28.0578.	The firmware version can also be obtained by running the following command:
	`# cat /sys/class/infiniband/mlx5_0/fw_ver`
	`16.28.0578`

Associating InfiniBand Ports to Ethernet Ports

The mlx5_ib driver holds a reference to the net device for getting notifications about the state of the port, as well as using the mlx5_core driver to resolve IP addresses to MAC that are required for address vector creation. However, RoCE traffic does not go through the mlx5_core driver; it is completely offloaded by the hardware.

```
# ibdev2netdev
mlx5_0 port 1 <====> eth2
#
```

Configuring an IP Address to the netdev Interface

To configure an IP address to the netdev interface:

1. Configure an IP address to the netdev interface on both sides of the link.

```
# ifconfig eth2 20.4.3.220
# ifconfig eth2
eth2    Link encap:Ethernet HWaddr 00:02:C9:08:E8:11
        inet addr:20.4.3.220 Bcast:20.255.255.255 Mask:255.0.0.0
        UP BROADCAST MULTICAST MTU:1500 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:1000
        RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)
```

2. Make sure that ping is working.

```
ping 20.4.3.219
PING 20.4.3.219 (20.4.3.219) 56(84) bytes of data.
64 bytes from 20.4.3.219: icmp_seq=1 ttl=64 time=0.873 ms
64 bytes from 20.4.3.219: icmp_seq=2 ttl=64 time=0.198 ms
64 bytes from 20.4.3.219: icmp_seq=3 ttl=64 time=0.167 ms
20.4.3.219 ping statistics —
3 packets transmitted, 3 received, 0% packet loss, time
2000ms rtt min/avg/max/mdev = 0.167/0.412/0.873/0.326 ms
```

Adding VLANs

To add VLANs:

1. Make sure that the 8021.q module is loaded.

```
modprobe 8021q
```

2. Add VLAN.

```
# vconfig add eth2 7
Added VLAN with VID == 7 to IF -:eth2:-
#
```

3. Configure an IP address.

```
ifconfig eth2.7 7.4.3.220
```

Defining Ethernet Priority (PCP in 802.1q Headers)

1. Define Ethernet priority on the server.

```
# ibv_rc_pingpong -g 1 -i 2 -l 4
local address: LID 0x0000, QPN 0x1c004f, PSN 0x9daf6c, GID
fe80::202:c900:708:e799
remote address: LID 0x0000, QPN 0x1c004f, PSN 0xb0a49b, GID
fe80::202:c900:708:e811
8192000 bytes in 0.01 seconds = 4840.89 Mbit/sec
1000 iters in 0.01 seconds = 13.54 usec/iter
```

2. Define Ethernet priority on the client.

```
# ibv_rc_pingpong -g 1 -i 2 -l 4 sw419
local address: LID 0x0000, QPN 0x1c004f, PSN 0xb0a49b, GID
fe80::202:c900:708:e811
remote address: LID 0x0000, QPN 0x1c004f, PSN 0x9daf6c, GID
fe80::202:c900:708:e799
8192000 bytes in 0.01 seconds = 4855.96 Mbit/sec
1000 iters in 0.01 seconds = 13.50 usec/iter
```

Using rdma_cm Tests

1. Use rdma_cm test on the server.

```
# ucmatose
cmatose: starting server
initiating data transfers
completing sends
receiving data transfers
data transfers complete
cmatose: disconnecting
disconnected
test complete
return status 0
#
```

2. Use rdma_cm test on the client.

```
# ucmatose -s 20.4.3.219
cmatose: starting client
cmatose: connecting
receiving data transfers
sending replies
data transfers complete
test complete
return status 0
#
```

This server-client run is without PCP or VLAN because the IP address used does not belong to a VLAN interface. If you specify a VLAN IP address, then the traffic should go over VLAN.

Type Of Service (ToS)

The TOS field for rdma_cm sockets can be set using the rdma_set_option() API, just as it is set for regular sockets. If a TOS is not set, the default value (0) is used. Within the

rdma_cm kernel driver, the TOS field is converted into an SL field. The conversion formula is as follows:

- SL = TOS >> 5 (e.g., take the 3 most significant bits of the TOS field)

In the hardware driver, the SL field is converted into PCP by the following formula:

- PCP = SL & 7 (take the 3 least significant bits of the TOS field)

> **Note**
>
> SL affects the PCP only when the traffic goes over tagged VLAN frames.

DSCP

A new entry has been added to the RDMA-CM configfs that allows users to select default TOS for RDMA-CM QPs. This is useful for users that want to control the TOS field without changing their code. Other applications that set the TOS explicitly using the rdma_set_option API will continue to work as expected to override the configfs value.

For further information about DSCP marking, refer to HowTo Set Egress ToS/DSCP on RDMA CM QPs Community post.

RoCE LAG

RoCE LAG is a feature meant for mimicking Ethernet bonding for IB devices and is available for dual port cards only.

This feature is supported on kernel versions 4.9 and above.

RoCE LAG mode is entered when both Ethernet interfaces are configured as a bond in one of the following modes:

- active-backup (mode 1)
- balance-xor (mode 2)

- 802.3ad (LACP) (mode 4)

Any change of bonding configuration that negates one of the above rules (i.e, bonding mode is not 1, 2 or 4, or both Ethernet interfaces that belong to the same card are not the only slaves

of the bond interface), will result in exiting RoCE LAG mode and the return to normal IB device per port configuration.

Once RoCE LAG is enabled, instead of having two IB devices; mlx5_0 and mlx5_1, there will be one device named mlx5_bond_0.

For information on how to configure RoCE LAG, refer to HowTo Configure RoCE over LAG (ConnectX-4/ConnectX-5/ConnectX-6) Community post.

Disabling RoCE

By default, RoCE is enabled on all mlx5 devices. When RoCE is enabled, all traffic to UDP port 4791 is treated as RoCE traffic by the device.

In case you are only interested in Ethernet (no RDMA) and wish to enable forwarding of traffic to this port, you can disable RoCE through sysfs:

```
echo <0|1> > /sys/devices/{pci-bus-address}/roce_enable
```

Note

Once RoCE is disabled, only Ethernet traffic will be supported. Therefore, there will be no GID tables and only Raw Ethernet QPs will be supported.

The current RoCE state can be queried by sysfs:

```
cat /sys/devices/{pci-bus-address}/roce_enable
```

Enabling/Disabling RoCE on VMs via VFs

By default, when configuring VFs on the hypervisor, all VFs will be enabled with RoCE. This means they require more OS memory (from the VM). In case you are only interested in Ethernet (no RDMA) on the VM, and you wish to save the VM memory, you can disable RoCE on the VF from the hypervisor. In addition, by disabling RoCE, a VM can have the capability of utilizing the RoCE UDP port (4791) for standard UDP traffic.

For details on how to enable/disable RoCE on a VF, refer to HowTo Enable/Disable RoCE on VMs via VFs Community post.

Force DSCP

This feature enables setting a global traffic_class value for all RC QPs, or setting a specific traffic class based on several matching criteria.

Usage

- To set a single global traffic class to be applied to all QPs, write the desired global traffic_class value to /sys/class/infiniband/<dev>/tc/<port>/traffic_class.

  Note the following:

  - Negative values indicate that the feature is disabled. traffic_class value can be set using ibv_modify_qp()
  - Valid values range between 0 - 255

> **Note**
>
> The ToS field is 8 bits, while the DSCP field is 6 bits. To set a DSCP value of X, you need to multiply this value by 4 (SHIFT 2). For example, to set DSCP value of 24, set the ToS bit to 96 (24x4=96).

- To set multiple traffic class values based on source and/or destination IPs, write the desired rule to /sys/class/infiniband/<dev>/tc/<port>/traffic_class. For example:

```
echo "tclass=16,src_ip=1.1.1.2,dst_ip=1.1.1.0/24" >
/sys/class/infiniband/mlx5_0/tc/1/traffic_class
```

Note: Adding "tclass" prefix to tclass value is optional.

In the example above, traffic class 16 will be set to any QP with source IP 1.1.1.2 and destination IP 1.1.1.0/24.

Note that when setting a specific traffic class, the following rule precedence will apply:

- If a global traffic class value is set, it will be applied to all QPs
- If no global traffic class value is set, and there is a rule with matching source and destination IPs applicable to at least one QP, it will be applied
- Rules only with matching source and/or destination IPs have no defined precedence over other rules with matching source and/or destination IPs

Notes:

- A mask can be provided when using destination IPv4 addresses
- The rule precedence is not affected by the order in which rules are inserted
- Overlapping rules are entirely up to the administrator.
- "tclass=-1" will remove the rule from the database

Force Time to Live (TTL)

This feature enables setting a global TTL value for all RC QPs.

Write the desired TTL value to /sys/class/infiniband/<dev>/tc/<port>/ttl. Valid values range between 0 - 255

Flow Control

Priority Flow Control (PFC)

Priority Flow Control (PFC) IEEE 802.1Qbb applies pause functionality to specific classes of traffic on the Ethernet link. For example, PFC can provide lossless service for the RoCE traffic and best-effort service for the standard Ethernet traffic. PFC can provide different levels of service to specific classes of Ethernet traffic (using IEEE 802.1p traffic classes).

Configuring PFC on ConnectX-4 and above

1. Enable PFC on the desired priority:

```
mlnx_qos -i <ethX> --pfc <0/1>,<0/1>,<0/1>,<0/1>,<0/1>,<0/1>,
<0/1>,<0/1>
```

Example (Priority=4):

```
mlnx_qos -i eth1 --pfc 0,0,0,0,1,0,0,0
```

2. Create a VLAN interface:

```
vconfig add <ethX> <VLAN_ID>
```

Example (VLAN_ID=5):

```
vconfig add eth1 5
```

3. Set egress mapping:

    1. For Ethernet traffic:

```
vconfig set_egress_map <vlan_einterface> <skprio> <up>
```

Example (skprio=3, up=5):

```
vconfig set_egress_map eth1.5 3 5
```

4. Create 8 Traffic Classes (TCs):

```
tc_wrap.py -i <interface>
```

5. Enable PFC on the switch.

   For information on how to enable PFC on your respective switch, please refer to Switch FC/PFC Configuration sections in the RDMA/RoCE Solutions Community page .

PFC Configuration Using LLDP DCBX

PFC Configuration on Hosts

PFC Auto-Configuration Using LLDP Tool in the OS

1. Start lldpad daemon on host.

```
lldpad -d
```

   OR

```
service lldpad start
```

2. Send lldpad packets to the switch.

```
lldptool set-lldp -i <ethX>  adminStatus=rxtx
lldptool -T -i <ethX>   -V  sysName enableTx=yess
lldptool -T -i <ethX>   -V  portDesc enableTx=yess
lldptool -T -i <ethX>   -V  sysDesc enableTx=yess
lldptool -T -i <ethX>   -V sysCap enableTx=yess
lldptool -T -i <ethX>   -V mngAddr enableTx=yess
lldptool -T -i <ethX>   -V PFC enableTx=yes;
lldptool -T -I <ethX>   -V CEE-DCBX enableTx=yess
```

3. Set the PFC parameters.

- For the CEE protocol, use dcbtool:

```
dcbtool sc <ethX> pfc pfcup:<xxxxxxxx>
```

Example:

```
dcbtool sc eth6 pfc pfcup:01110001
```

where:

[pfcup:xxxxxxxx]	Enables/disables priority flow control. From left to right (priorities 0-7) - x can be equal to either 0 or 1. 1 indicates that the priority is configured to transmit priority pause.

- For IEEE protocol, use lldptool:

```
lldptool -T -i <ethX> -V PFC enabled=x,x,x,x,x,x,x,x
```

Example:

```
lldptool -T -i eth2 -V PFC enabled=1,2,4
```

where:

enabled	Displays or sets the priorities with PFC enabled. The set attribute takes a comma-separated list of priorities to enable, or the string none to disable all priorities.

PFC Auto-Configuration Using LLDP in the Firmware (for mlx5 driver)

There are two ways to configure PFC and ETS on the server:

1. **Local Configuration** - Configuring each server manually.
2. **Remote Configuration** - Configuring PFC and ETS on the switch, after which the switch will pass the configuration to the server using LLDP DCBX TLVs.

   There are two ways to implement the remote configuration using mlx5 driver:

   1. Configuring the adapter firmware to enable DCBX.
   2. Configuring the host to enable DCBX.

For further information on how to auto-configure PFC using LLDP in the firmware, refer to the HowTo Auto-Config PFC and ETS on ConnectX-4 via LLDP DCBX Community post.

PFC Configuration on Switches

1. In order to enable DCBX, LLDP should first be enabled:

```
switch (config) # lldp
show lldp interfaces ethernet remote
```

2. Add DCBX to the list of supported TLVs per required interface.

   For IEEE DCBX:

   ```
   switch (config) # interface 1/1
   switch (config interface ethernet 1/1) # lldp tlv-select dcbx
   ```

   For CEE DCBX:

   ```
   switch (config) # interface 1/1
   switch (config interface ethernet 1/1) # lldp tlv-select dcbx-cee
   ```

3. [**Optional**] Application Priority can be configured on the switch, with the required ethertype and priority. For example, IP packet, priority 1:

   ```
   switch (config) # dcb application-priority 0x8100 1
   ```

4. Make sure PFC is enabled on the host (for enabling PFC on the host, refer to PFC Configuration on Hosts section above). Once it is enabled, it will be passed in the LLDP TLVs.

5. Enable PFC with the desired priority on the Ethernet port.

   ```
   dcb priority-flow-control enable force
   dcb priority-flow-control priority <priority> enable
   interface ethernet <port> dcb priority-flow-control mode on
   force
   ```

   Example - Enabling PFC with priority 3 on port 1/1:

```
dcb priority-flow-control enable force
dcb priority-flow-control priority 3 enable
interface ethernet 1/1 dcb priority-flow-control mode on force
```

Priority Counters

Several ingress and egress counters per priority are supported. Run ethtool -S to get the full list of port counters.

ConnectX-4 Counters

- Rx and Tx Counters:
  - Packets
  - Bytes
  - Octets
  - Frames
  - Pause
  - Pause frames
  - Pause Duration
  - Pause Transition

Example

```
# ethtool -S eth35 | grep prio4
     prio4_rx_octets: 62147780800
     prio4_rx_frames: 14885696
     prio4_tx_octets: 0
     prio4_tx_frames: 0
     prio4_rx_pause: 0
     prio4_rx_pause_duration: 0
     prio4_tx_pause: 26832
     prio4_tx_pause_duration: 14508
     prio4_rx_pause_transition: 0
```

Note

The Pause counters in ConnectX-4 are visible via ethtool only for priorities on which PFC is enabled.

PFC Storm Prevention

PFC storm prevention enables toggling between default and auto modes.

The stall prevention timeout is configured to 8 seconds by default. Auto mode sets the stall prevention timeout to be 100 msec.

The feature can be controlled using sysfs in the following directory: /sys/class/net/eth*/settings/ pfc_stall_prevention

- To query the PFC stall prevention mode:

```
cat /sys/class/net/eth*/settings/pfc_stall_prevention
```

Example

```
$ cat /sys/class/net/ens6/settings/pfc_stall_prevention
default
```

- To configure the PFC stall prevention mode:

```
echo <option>
/sys/class/net/<interface>/settings/pfc_stall_prevention
```

The following two counters were added to the ethtool -S:

- tx_pause_storm_warning_events - when the device is stalled for a period longer than a pre-configured watermark, the counter increases, allowing the debug utility an insight into current device status.
- tx_pause_storm_error_events - when the device is stalled for a period longer than a pre-configured timeout, the pause transmission is disabled, and the counter increase.

Dropless Receive Queue (RQ)

Dropless RQ feature enables the driver to notify the FW when SW receive queues are overloaded. This scenario takes place when the handling of SW receive queue is slower than the handling of the HW receive queues.

When this feature is enabled, a packet that is received while the receive queue is full will not be immediately dropped. The FW will accumulate these packets assuming posting of new WQEs will resume shortly. If received WQEs are not posted after a certain period of time, out_of_buffer counter will increase, indicating that the packet has been dropped.

This feature is disabled by default. In order to activate it, ensure that Flow Control feature is also enabled.

```
ethtool --set-priv-flags ens6 dropless_rq on
```

To get the feature state, run:

```
ethtool --show-priv-flags DEVNAME
```

Output example:

```
Private flags for DEVNAME:
rx_cqe_moder   : on
rx_cqe_compress: off
sniffer        : off
dropless_rq    : off
hw_lro         : off
```

To disable the feature, run:

```
ethtool --set-priv-flags ens6 dropless_rq off
```

Explicit Congestion Notification (ECN)

ECN is an extension to the IP protocol. It allows reliable communication by notifying all ends of communication when congestion occurs. This is done without dropping packets.

Please note that this feature requires all nodes in the path (nodes, routers etc) between the communicating nodes to support ECN to ensure reliable communication. ECN is marked as 2 bits in the traffic control IP header. This ECN implementation refers to RoCE v2.

Enabling ECN

To enable ECN on the hosts:

1. Enable ECN in sysfs.

```
/sys/class/net/<interface>/ecn/<protocol>/enable/<X>
```

2. Query the attribute.

```
cat /sys/class/net/<interface>/ecn/<protocol>/params/<requested
attribute>
```

3. Modify the attribute.

```
echo <value>
/sys/class/net/<interface>/ecn/<protocol>/params/<requested
attribute>
```

ECN supports the following algorithms:

- r_roce_ecn_rp - Reaction point
- r_roce_ecn_np - Notification point

Each algorithm has a set of relevant parameters and statistics, which are defined per device, per port, per priority.

```
cat /sys/class/net/<interface>/ecn/<protocol>/enable/X
```

```
cat /sys/class/net/<interface>/ecn/<protocol>/requested attributes
```

```
echo 1 > /sys/class/net/<interface>/ecn/<protocol>/enable/X
```

```
echo <value> > /sys/class/net/<interface>/ecn/<protocol>/requested
attributes
```

where:

- X: priority {0..7}
- protocol: roce_rp / roce_np
- requested attributes: Next Slide for each protocol.

RSS Support

RSS Hash Function

The device has the ability to use XOR as the RSS distribution function, instead of the default Toplitz function.

The XOR function can be better distributed among driver's receive queues in a small number of streams, where it distributes each TCP/UDP stream to a different queue. provides the following option to change the working RSS hash function from Toplitz to XOR, and vice-versa:

Through sysfs, located at: /sys/class/net/eth*/settings/hfunc.

➢ ***To query the operational and supported hash functions:***

```
cat /sys/class/net/eth*/settings/hfunc
```

Example:

```
cat /sys/class/net/eth2/settings/hfunc
Operational hfunc: toeplitz
Supported hfuncs: xor toeplitz
```

To change the operational hash function:

➢

```
echo xor > /sys/class/net/eth*/settings/hfunc
```

RSS Verbs Support

Receive Side Scaling (RSS) technology allows spreading incoming traffic between different receive descriptor queues. Assigning each queue to different CPU cores allows better load balancing of the incoming traffic and improves performance.

This technology was extended to user space by the verbs layer and can be used for RAW ETH QP.

RSS Flow Steering

Steering rules classify incoming packets and deliver a specific traffic type (e.g. TCP/UDP, IP only) or a specific flow to "RX Hash" QP. "RX Hash" QP is responsible for spreading the traffic it handles between the Receive Work Queues using RX hash and Indirection Table. The Receive Work Queue can point to different CQs that can be associated with different CPU cores.

Verbs

The below verbs should be used to achieve this task in both control and data path. Details per verb should be referenced from its man page.

- ibv_create_wq, ibv_modify_wq, ibv_destory_wq

- ibv_create_rwq_ind_table, ibv_destroy_rwq_ind_table
- ibv_create_qp_ex with specific RX configuration to create the "RX hash" QP

Time-Stamping

Time-Stamping Service

Time-stamping is the process of keeping track of the creation of a packet. A time-stamping service supports assertions of proof that a datum existed before a particular time. Incoming packets are time-stamped before they are distributed on the PCI depending on the congestion in the PCI buffers. Outgoing packets are time-stamped very close to placing them on the wire.

Enabling Time-Stamping

Time-stamping is off by default and should be enabled before use.

To enable time-stamping for a socket:

Call setsockopt() with SO_TIMESTAMPING and with the following flags:

SOF_TIMESTAMPING_TX_HARDWARE:	try to obtain send time-stamp in hardware
SOF_TIMESTAMPING_TX_SOFTWARE:	if SOF_TIMESTAMPING_TX_HARDWARE is off or fails, then do it in software
SOF_TIMESTAMPING_RX_HARDWARE:	return the original, unmodified time-stamp as generated by the hardware
SOF_TIMESTAMPING_RX_SOFTWARE:	if SOF_TIMESTAMPING_RX_HARDWARE is off or fails, then do it in software
SOF_TIMESTAMPING_RAW_HARDWARE:	return original raw hardware time-stamp
SOF_TIMESTAMPING_SYS_HARDWARE:	return hardware time-stamp transformed into the system time base
SOF_TIMESTAMPING_SOFTWARE:	return system time-stamp generated in software
SOF_TIMESTAMPING_TX/RX	determine how time-stamps are generated

SOF_TIMESTAMPING_RAW/SYS	determine how they are reported

To enable time-stamping for a net device:

Admin privileged user can enable/disable time stamping through calling ioctl (sock, SIOCSH-WTSTAMP, &ifreq) with the following values:

- Send side time sampling, enabled by ifreq.hwtstamp_config.tx_type when:

```
/* possible values for hwtstamp_config->tx_type */
enum hwtstamp_tx_types {
        /*
     * No outgoing packet will need hardware time stamping;
     * should a packet arrive which asks for it, no hardware
     * time stamping will be done.
    */
        HWTSTAMP_TX_OFF,

        /*
     * Enables hardware time stamping for outgoing packets;
     * the sender of the packet decides which are to be
     * time stamped by setting %SOF_TIMESTAMPING_TX_SOFTWARE
     * before sending the packet.
     */
        HWTSTAMP_TX_ON,
/*
     * Enables time stamping for outgoing packets just as
     * HWTSTAMP_TX_ON does, but also enables time stamp insertion
     * directly into Sync packets. In this case, transmitted Sync
     * packets will not received a time stamp via the socket error
     * queue.
     */
        HWTSTAMP_TX_ONESTEP_SYNC,
};
Note: for send side time stamping currently only
HWTSTAMP_TX_OFF and
HWTSTAMP_TX_ON are supported.
```

- Receive side time sampling, enabled by ifreq.hwtstamp_config.rx_filter when:

```
/* possible values for hwtstamp_config->rx_filter */
enum hwtstamp_rx_filters {
        /* time stamp no incoming packet at all */
        HWTSTAMP_FILTER_NONE,

        /* time stamp any incoming packet */
        HWTSTAMP_FILTER_ALL,
      /* return value: time stamp all packets requested plus some others */
        HWTSTAMP_FILTER_SOME,

        /* PTP v1, UDP, any kind of event packet */
        HWTSTAMP_FILTER_PTP_V1_L4_EVENT,
        /* PTP v1, UDP, Sync packet */
        HWTSTAMP_FILTER_PTP_V1_L4_SYNC,
        /* PTP v1, UDP, Delay_req packet */
        HWTSTAMP_FILTER_PTP_V1_L4_DELAY_REQ,
        /* PTP v2, UDP, any kind of event packet */
        HWTSTAMP_FILTER_PTP_V2_L4_EVENT,
        /* PTP v2, UDP, Sync packet */
        HWTSTAMP_FILTER_PTP_V2_L4_SYNC,
        /* PTP v2, UDP, Delay_req packet */
        HWTSTAMP_FILTER_PTP_V2_L4_DELAY_REQ,
        /* 802.AS1, Ethernet, any kind of event packet */
        HWTSTAMP_FILTER_PTP_V2_L2_EVENT,
        /* 802.AS1, Ethernet, Sync packet */
        HWTSTAMP_FILTER_PTP_V2_L2_SYNC,
        /* 802.AS1, Ethernet, Delay_req packet */
        HWTSTAMP_FILTER_PTP_V2_L2_DELAY_REQ,

        /* PTP v2/802.AS1, any layer, any kind of event packet */
        HWTSTAMP_FILTER_PTP_V2_EVENT,
        /* PTP v2/802.AS1, any layer, Sync packet */
        HWTSTAMP_FILTER_PTP_V2_SYNC,
        /* PTP v2/802.AS1, any layer, Delay_req packet */
        HWTSTAMP_FILTER_PTP_V2_DELAY_REQ,
```

```
};
Note: for receive side time stamping currently only
HWTSTAMP_FILTER_NONE and
HWTSTAMP_FILTER_ALL are supported.
```

Getting Time-Stamping

Once time stamping is enabled time stamp is placed in the socket Ancillary data. recvmsg() can be used to get this control message for regular incoming packets. For send time stamps the outgoing packet is looped back to the socket's error queue with the send time-stamp(s) attached. It can

be received with recvmsg (flags=MSG_ERRQUEUE). The call returns the original outgoing packet data including all headers prepended down to and including the link layer, the scm_time-stamping control message and a sock_extended_err control message with ee_errno==ENOMSG and ee_origin==SO_EE_ORIGIN_TIMESTAMPING. A socket with such a pending bounced packet is ready for reading as far as select() is concerned. If the outgoing packet has to be fragmented, then only the first fragment is time stamped and returned to the sending socket.

Note

When time-stamping is enabled, VLAN stripping is disabled. For more info please refer to Documentation/networking/timestamping.txt in kernel.org.

Note

On ConnectX-4 and above adapter cards, when time-stamping is enabled, RX CQE compression is disabled (features are mutually exclusive).

Time Stamping Capabilities via ethtool

To display Time Stamping capabilities via ethtool:

Show Time Stamping capabilities:

```
ethtool -T eth<x>
```

Example:

```
ethtool -T eth0
Time stamping parameters for p2p1:
Capabilities:

hardware-transmit
(SOF_TIMESTAMPING_TX_HARDWARE)

software-transmit
(SOF_TIMESTAMPING_TX_SOFTWARE)

hardware-receive
(SOF_TIMESTAMPING_RX_HARDWARE)

software-receive
(SOF_TIMESTAMPING_RX_SOFTWARE)

software-system-clock           (SOF_TIMESTAMPING_SOFTWARE)

hardware-raw-clock
(SOF_TIMESTAMPING_RAW_HARDWARE)
PTP Hardware Clock: 1
Hardware Transmit Timestamp Modes:
off
(HWTSTAMP_TX_OFF)
on
(HWTSTAMP_TX_ON)

Hardware Receive Filter Modes:
none
(HWTSTAMP_FILTER_NONE)
all
(HWTSTAMP_FILTER_ALL)
```

For more details on PTP Hardware Clock, please refer to:
https://www.kernel.org/doc/Documentation/ptp/ptp.txt

Steering PTP Traffic to Single RX Ring

As a result of Receive Side Steering (RSS) PTP traffic coming to UDP ports 319 and 320, it may reach the user space application in an out of order manner. In order to prevent this, PTP traffic needs to be steered to single RX ring using ethtool.

Example:

```
# ethtool -u ens7
8 RX rings available
Total 0 rules
# ethtool -U ens7 flow-type udp4 dst-port 319 action 0 loc 1
# ethtool -U ens7 flow-type udp4 dst-port 320 action 0 loc 0
# ethtool -u ens7
8 RX rings available
Total 2 rules
Filter: 0
Rule Type: UDP over IPv4
Src IP addr: 0.0.0.0 mask: 255.255.255.255
Dest IP addr: 0.0.0.0 mask: 255.255.255.255
TOS: 0x0 mask: 0xff
Src port: 0 mask: 0xffff
Dest port: 320 mask: 0x0
Action: Direct to queue 0
Filter: 1
Rule Type: UDP over IPv4
Src IP addr: 0.0.0.0 mask: 255.255.255.255
Dest IP addr: 0.0.0.0 mask: 255.255.255.255
TOS: 0x0 mask: 0xff
Src port: 0 mask: 0xffff
Dest port: 319 mask: 0x0
Action: Direct to queue 0
```

Tx Port Time-Stamping

Note

This feature is supported on ConnectX-6 Dx and above adapter cards only.

Transmitted packet time-stamping accuracy can be improved when using a timestamp generated at the port level instead of a timestamp generated upon CQE creation. Tx port time-stamping better reflects the actual time of a packet's transmission.

Normal Send queues (SQs) are open with CQE time-stamp support. When this feature is enabled, the driver is expected to open extra Tx port time-stamped SQ per traffic class (TC).

The stream must meet the following conditions in order to be transmitted through a Tx port time-stamped SQ.

1. SKBTX_HW_TSTAMP flag was set at tx_flag (SO_TIMESTAMPING was set via setsockopt() or similarly)
2. Packet type is:
    1. Non-IP, with EtherType of PTP over IEEE 802.3 (0x88f7)

       or

    2. UDP over IPv4/IPv6

This feature is disabled by default in order to avoid extra SQ memory allocations. The feature can be enabled or disabled using the following command.

```
ethtool --set-priv-flags <ifs-name> tx_port_ts on / off
```

PTP Cyc2time Hardware Translation Offload

Note

This feature is supported on ConnectX-6 Dx and above adapter cards only.

Device timestamp can be in one of two modes: real time or free running internal time.

In free running internal time mode, the device clock is not editable in any way. Driver and/or user space must adjust it to the real-time nanosecond values.

In real time mode, the hardware clock device can be adjusted and can provide timestamps which are already translated into real-time nanoseconds.

Both modes are global per device. Once a mode is set, all clock-related features (such as PPS, CQE TS, PCIe bar, etc) will work with the chosen clock mode only.

Free running internal time is the default mode configured in the hardware. The driver will modify the hardware real time clock based on PTP daemon clock adjustments.

Only physical functions are allowed to modify the hardware real-time clock, so PTP daemon adjustments from VFs will be treated as NOP. In case more than one physical function tries to modify the hardware real-time clock, the device will select one of the functions as its designated clock provider. All other input will also be treated as a NOP. The designated clock provide can be replaced by the device if no new adjustments have been received from the current provider after some period.

Timestamp Format

CQE hardware timestamp format for ConnectX-6 Dx and ConnectX-6 Lx NICs is 64 bit, as follows:

```
{32bit sec, 32 bit nsec}
```

Configuration

To enable the feature:

1. Set REAL_TIME_CLOCK_ENABLE in NV_CONFIG via mlxconfig

2. Restart the driver.

Limitations

- Administrator must restart the driver and perform a FW reset for the configuration to take effect. Otherwise, mismatch between HW and driver timestamp mode might occur.

- Once real time mode is activated on a given device (see configuration section), version 5.3 or newer must run on all device functions. Any older driver running on a device function at this configuration will fail to open any traffic queues (RDMA or ETH), hence becoming dysfunctional.

- In real time mode, all device functions must be PTP-synchronized by a single clock domain—do not use multiple GMs for different functions on the same device.

- Regarding hardware clock ownership, the hardware is configured only from a single elected function; other function settings are ignored by the device. There is no indication as to which function is the hardware-clock's owner. After an internal timeout without modifying the hardware clock, a function losses the hardware-clock's ownership and is open to be grasped by any of the functions.

- All PFs/VFs within the same device must sync to the same 1588 master clock. If multiple masters are used, the device will use a single elected function. This might lead to wrong clock representation by device, wrong 1588 TLVs and hiccups on replacement of elected function.

- This feature is supported on ConnectX-6 Dx and above adapter cards only.

RoCE Time-Stamping

RoCE Time-Stamping allows you to stamp packets when they are sent to the wire/received from the wire. The time-stamp is given in raw hardware cycles but could be easily converted into hardware referenced nanoseconds based time. Additionally, it enables you to query the hardware for the hardware time, thus stamp other application's event and compare time.

Query Capabilities

Time-stamping is available if and only the hardware reports it is capable of reporting it. To verify whether RoCE Time-Stamping is available, run ibv_query_device_ex.

For further information, please see ibv_query_device_ex manual page.

Creating a Time-Stamping Completion Queue

To get time stamps, a suitable extended Completion Queue (CQ) must be created via a special call to ibv_create_cq_ex verb.

For further information, please see ibv_create_cq_ex manual page.

> **Note**
>
> Time Stamping in not available when CQE zipping is used.

Querying the Hardware Time

Querying the hardware for time is done via the ibv_query_rt_values_ex verb. For example:

For further information, please see ibv_query_rt_values_ex manual page.

One Pulse Per Second (1PPS)

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.

Flow Steering

Flow steering is a new model which steers network flows based on flow specifications to specific QPs. Those flows can be either unicast or multicast network flows. In order to maintain flexibility, domains and priorities are used. Flow steering uses a methodology of flow attribute, which is a combination of L2-L4 flow specifications, a destination QP and a priority. Flow steering rules may be inserted either by using ethtool or by using InfiniBand verbs. The verbs abstraction uses different terminology from the flow attribute (ibv_flow_attr), defined by a combination of specifications (struct ibv_flow_spec_*).

Flow Steering Support

All flow steering features are enabled in the supported adapter cards.

Flow Domains and Priorities

Flow steering defines the concept of domain and priority. Each domain represents a user agent that can attach a flow. The domains are prioritized. A higher priority domain will always supersede a lower priority domain when their flow specifications overlap. Setting a lower priority value will result in a higher priority.

In addition to the domain, there is a priority within each of the domains. Each domain can have at most 2^12 priorities in accordance with its needs.

The following are the domains at a descending order of priority:

- **User Verbs** allows a user application QP to be attached to a specified flow when using `ibv_create_flow` and `ibv_destroy_flow` verbs
- `ibv_create_flow`

  ```
  struct ibv_flow *ibv_create_flow(struct ibv_qp *qp, struct
  ibv_flow_attr
  *flow)
  ```

  Input parameters:

- `struct ibv_qp` - the attached QP.
- `struct ibv_flow_attr` - attaches the QP to the flow specified. The flow contains mandatory control parameters and optional L2, L3 and L4 headers. The optional headers are detected by setting the size and `num_of_specs` fields:

  `struct ibv_flow_attr` can be followed by the optional flow headers structs:

  ```
  struct ibv_flow_spec_eth
  struct ibv_flow_spec_ipv4
  struct ibv_flow_spec_tcp_udp
  struct ibv_flow_spec_ipv6
  ```

  For further information, please refer to the `ibv_create_flow` man page.

- `ibv_destroy_flow`

  ```
  int ibv_destroy_flow(struct ibv_flow *flow_id)
  ```

  Input parameters:

  `ibv_destroy_flow` requires struct `ibv_low` which is the return value of `ibv_create_flow` in case of success.

  Output parameters:

  Returns 0 on success, or the value of errno on failure.

  For further information, please refer to the `ibv_destroy_flow` man page.

Ethtool

Ethtool domain is used to attach an RX ring, specifically its QP to a specified flow. Please refer to the most recent ethtool man page for all the ways to specify a flow.

Examples:

- ethtool –U eth5 flow-type ether dst 00:11:22:33:44:55 loc 5 action 2

  All packets that contain the above destination MAC address are to be steered into rx-ring 2 (its underlying QP), with priority 5 (within the ethtool domain)

- ethtool –U eth5 flow-type tcp4 src-ip 1.2.3.4 dst-port 8888 loc 5 action 2

  All packets that contain the above destination IP address and source port are to be steered into rx- ring 2. When destination MAC is not given, the user's destination MAC is filled automatically.

- ethtool -U eth5 flow-type ether dst 00:11:22:33:44:55 vlan 45 m 0xf000 loc 5 action 2

  All packets that contain the above destination MAC address and specific VLAN are steered into ring 2. Please pay attention to the VLAN's mask 0xf000. It is required in order to add such a rule.

- ethtool –u eth5

  Shows all of ethtool's steering rule

When configuring two rules with the same priority, the second rule will overwrite the first one, so this ethtool interface is effectively a table. Inserting Flow Steering rules in the kernel requires support from both the ethtool in the user space and in kernel (v2.6.28).

Accelerated Receive Flow Steering (aRFS)

Receive Flow Steering (RFS) and Accelerated Receive Flow Steering (aRFS) are kernel features currently available in most distributions. For RFS, packets are forwarded based on the location of the application consuming the packet. aRFS boosts the speed of RFS by adding support for the hardware. By usingaRFS(unlike RFS), the packets are directed to a CPU that is local to the thread running the application.

aRFSis an in-kernel-logic responsible for load balancing between CPUs by attaching flows to CPUs that are used by flow's owner applications. This domain allows the aRFS mechanism to use the flow steering infrastructure to support the aRFS logic by implementing the ndo_rx_flow_steer, which, in turn, calls the underlying flow steering mechanism with the aRFS domain.

To configure RFS:

Configure the RFS flow table entries (globally and per core).

Note: The functionality remains disabled until explicitly configured (by default it is 0).

- The number of entries in the global flow table is set as follows:

> **Note**
>
> /proc/sys/net/core/rps_sock_flow_entries

- The number of entries in the per-queue flow table are set as follows:

> **Note**
>
> /sys/class/net/<dev>/queues/rx-<n>/rps_flow_cnt

Example:

```
# echo 32768 > /proc/sys/net/core/rps_sock_flow_entries
# NUM_CHANNELS=`ethtool -l ens6 | grep "Combined:" | tail -1 | awk
'{print $2}'`
# for f in `seq 0 $((NUM_CHANNELS-1))`; do echo 32768 >
/sys/class/net/ens6/queues/rx-$f/rps_flow_cnt; done
```

To Configure aRFS:

The aRFS feature requires explicit configuration in order to enable it. Enabling the aRFS requires enabling the 'ntuple' flag via the ethtool.

For example, to enable ntuple for eth0, run:

```
ethtool -K eth0 ntuple on
```

aRFS requires the kernel to be compiled with the `CONFIG_RFS_ACCEL` option. This option is available in kernels 2.6.39 and above. Furthermore, aRFS requires Device Managed Flow Steering support.

> **Note**
>
> RFS cannot function if LRO is enabled. LRO can be disabled via ethtool.

Flow Steering Dump Tool

The mlx_fs_dump is a python tool that prints the steering rules in a readable manner. Python v2.7 or above, as well as pip, anytree and termcolor libraries are required to be installed on the host.

Running example:

```
./ofed_scripts/utils/mlx_fs_dump -d /dev/mst/mt4115_pciconf0
FT: 9 (level: 0x18, type: NIC_RX)
+-- FG: 0x15 (MISC)
    |-- FTE: 0x0 (FWD) to (TIR:0x7e) out.ethtype:IPv4
out.ip_prot:UDP out.udp_dport:0x140
    +-- FTE: 0x1 (FWD) to (TIR:0x7e) out.ethtype:IPv4
out.ip_prot:UDP out.udp_dport:0x13f
...
```

For further information on the mlx_fs_dump tool, please refer to mlx_fs_dump Community post.

Wake-on-LAN (WoL)

Wake-on-LAN (WoL) is a technology that allows a network professional to remotely power on a computer or to wake it up from sleep mode.

- To enable WoL:

```
ethtool -s <interface> wol g
```

- To get WoL:

```
ethtool <interface> | grep Wake-on Wake-on: g
```

Where:

"`g`" is the magic packet activity.

Hardware Accelerated 802.1ad VLAN (Q-in-Q Tunneling)

Q-in-Q tunneling allows the user to create a Layer 2 Ethernet connection between two servers. The user can segregate a different VLAN traffic on a link or bundle different VLANs into a single VLAN. Q-in-Q tunneling adds a service VLAN tag before the user's 802.1Q VLAN tags.

For Q-in-Q support in virtualized environments (SR-IOV), please refer to " Q-in-Q Encapsulation per VF in Linux (VST) " .

To enable device support for accelerated 802.1ad VLAN:

1. Turn on the new ethtool private flag "phv-bit" (disabled by default).

```
$ ethtool --set-priv-flags eth1 phv-bit on
```

Enabling this flag sets the phv_en port capability.

2. Change the interface device features by turning on the ethtool device feature "tx-vlan- stag-hw-insert" (disabled by default).

```
$ ethtool -K eth1 tx-vlan-stag-hw-insert on
```

Once the private flag and the ethtool device feature are set, the device will be ready for 802.1ad VLAN acceleration.

Note

The "phv-bit" private flag setting is available for the Physical Function (PF) only.

The Virtual Function (VF) can use the VLAN acceleration by setting the "tx-vlan-stag-hw-insert" parameter only if the private flag "phv-bit" is enabled by the PF. If the PF enables/disables the "phv-bit" flag after the VF driver is up, the configuration will take place only after the VF driver is restarted.

VLAN Stripping in Linux Verbs

Note

This capability is now accessible from userspace using the verbs.

VLAN stripping adds access to the device's ability to offload the Customer VLAN (cVLAN) header stripping from an incoming packet, thus achieving acceleration of VLAN handing in receive flow.

It is configured per WQ option. You can either enable it upon creation or modify it later using the appropriate verbs (`ibv_create_wq` / `ibv_modify_wq`).

Offloaded Traffic Sniffer

> **Note**
>
> To be able to activate this feature, make sure libpcap library v1.9 or above is installed on your setup.
>
> To download libpcap, please visit https://www.tcpdump.org/.

Offloaded Traffic Sniffer allows bypass kernel traffic (such as RoCE, VMA, and DPDK) to be captured by existing packet analyzer, such as tcpdump.

To capture the interface's bypass kernel traffic, run tcpdump on the RDMA device.

For examples on how to dump RDMA traffic using the Inbox tcpdump tool for ConnectX-4 adapter cards and above, click here .

> **Note**
>
> Note that enabling Offloaded Traffic Sniffer can cause bypass kernel traffic speed degradation.

Note

In case you do not wish to install libpcap on your setup, you can use docker to run the tcpdump. For further information, please see https://hub.docker.com/r/mellanox/tcpdump-rdma.

Dump Configuration

This feature helps dumping driver and firmware configuration using ethtool. It creates a backup of the configuration files into a specified dump file.

Dump Parameters (Bitmap Flag)

The following bitmap parameters are used to set the type of dump. If a value is not set, the default value used is "0".

Bitmap Parameters

Value	Description
1	MST dump
2	Ring dump (Software context information for SQs, EQs, RQs, CQs)
3	MST dump + Ring dump (1+2)
4	Clear this parameter

Configuration

In order to configure this feature, follow the steps below:

1. Set the dump bitmap parameter by running -W (uppercase) with the desired bitmap parameter value (see Bitmap Parameters table above). In the following example, the bitmap parameter value is 3.

```
ethtool -W ens1f0 3
```

2. Dump the file by running -w (lowercase) with the desired configuration file name.

```
ethtool -w ens1f0 data /tmp/dump.bin
```

3. [**Optional**] To get the bitmap parameter value, version and size of the dump, run the command above without the file name.

```
ethtool -w ens1f0
flag: 3, version: 1, length: 4312
```

4. To open the dump file, run:

```
mlnx_dump_parser -f /tmp/dump.bin -m mst_dump_demo.txt -r
ring_dump_demo.txt
Version: 1 Flag: 3 Number of blocks: 123 Length 327584
MCION module number: 0 status: | present |
DRIVER VERSION: 1-23 (03 Mar 2015)
DEVICE NAME 0000:81:00.0:ens1f0
Parsing Complete!
```

where:

-f	For the file to be parsed (the file that was just created)
-m	For the mst dump file
-r	For the ring dump file

For further information, refer to HowTo Dump Driver Configuration (via ethtool) Community post.

Output:

```
# mlnx_dump_parser -f /tmp/dump.bin -m mst_dump_demo.txt -r
ring_dump_demo.txt
Version: 1 Flag: 3 Number of blocks: 123 Length 327584
MCION module number: 0 status: | present |
DRIVER VERSION: 1-23 (03 Mar 2015)
DEVICE NAME 0000:81:00.0:ens1f0
Parsing Complete!
```

5. Open the files.

    1. The MST dump file will look as follows. In order to analyze it, contact NVIDIA Support.

        ```
        cat mst_dump_demo.txt
        0x00000000 0x01002000
        0x00000004 0x00000000
        0x00000008 0x00000000
        0x0000000c 0x00000000
        0x00000010 0x00000000
        0x00000014 0x00000000
        0x00000018 0x00000000
        ...
        ```

    2. The Ring dump file can help developers debug ring-related issues, and it looks as follows:

```
# cat ring_dump_demo.txt
SQ TYPE: 3, WQN: 102, PI: 0, CI: 0, STRIDE: 6, SIZE:
1024...
SQ TYPE: 3, WQN: 102, PI: 0, CI: 0, STRIDE: 6, SIZE: 1024,
WQE_NUM: 65536, GROUP_IP: 0
CQ TYPE: 5, WQN: 20, PI: 0, CI: 0, STRIDE: 6, SIZE: 1024,
WQE_NUM: 1024, GROUP_IP: 0
RQ TYPE: 4, WQN: 103, PI: 15, CI: 0, STRIDE: 5, SIZE: 16,
WQE_NUM: 512, GROUP_IP: 0
CQ TYPE: 5, WQN: 21, PI: 0, CI: 0, STRIDE: 6, SIZE: 16384,
WQE_NUM: 16384, GROUP_IP: 0
EQ TYPE: 6, CI: 1, SIZE: 0, IRQN: 109, EQN: 19, NENT: 2048,
MASK: 0, INDEX: 0, GROUP_ID: 0
SQ TYPE: 3, WQN: 106, PI: 0, CI: 0, STRIDE: 6, SIZE: 1024,
WQE_NUM: 65536, GROUP_IP: 1
CQ TYPE: 5, WQN: 23, PI: 0, CI: 0, STRIDE: 6, SIZE: 1024,
WQE_NUM: 1024, GROUP_IP: 1
RQ TYPE: 4, WQN: 107, PI: 15, CI: 0, STRIDE: 5, SIZE: 16,
WQE_NUM: 512, GROUP_IP: 1
CQ TYPE: 5, WQN: 24, PI: 0, CI: 0, STRIDE: 6, SIZE: 16384,
WQE_NUM: 16384, GROUP_IP: 1
EQ TYPE: 6, CI: 1, SIZE: 0, IRQN: 110, EQN: 20, NENT: 2048,
MASK: 0, INDEX: 1, GROUP_ID: 1
SQ TYPE: 3, WQN: 110, PI: 0, CI: 0, STRIDE: 6, SIZE: 1024,
WQE_NUM: 65536, GROUP_IP: 2
CQ TYPE: 5, WQN: 26, PI: 0, CI: 0, STRIDE: 6, SIZE: 1024,
WQE_NUM: 1024, GROUP_IP: 2
RQ TYPE: 4, WQN: 111, PI: 15, CI: 0, STRIDE: 5, SIZE: 16,
WQE_NUM: 512, GROUP_IP: 2
CQ TYPE: 5, WQN: 27, PI: 0, CI: 0, STRIDE: 6, SIZE: 16384,
WQE_NUM: 16384, GROUP_IP: 2
...
```

Local Loopback Disable

Local Loopback Disable feature allows users to force the disablement of local loopback on the virtual port (vport). This disables both unicast and multicast loopback in the hardware.

To enable Local Loopback Disable, run the following command:

```
echo 1 > /sys/class/net/<ifname>/settings/force_local_lb_disable"
```

To disable Local Loopback Disable, run the following command:

```
echo 0 > /sys/class/net/<ifname>/settings/force_local_lb_disable"
```

Note

When turned off, the driver configures the loopback mode according to its own logic.

Kernel Transport Layer Security (kTLS) Offloads

Note

This feature is supported on NVIDIA® ConnectX®-6 Dx and NVIDIA® BlueField®-2 crypto devices onwards.

Transport Layer Security (TLS) is a widely-deployed protocol used for securing TCP connections on the Internet. TLS is also a required feature for HTTP/2, the latest web standard. Kernel implementation of TLS (kTLS) provides new opportunities for offloading the protocol into the hardware.

TLS data-path offload allows the NIC to accelerate encryption, decryption and authentication of AES-GCM. TLS offload handles data as it goes through the device without storing any data, but only updating context. If the packet cannot be encrypted/decrypted by the device, then a software fallback handles the packet.

Establishing a kTLS Connection

To avoid unnecessary complexity in the kernel, the TLS handshake is kept in the user space. A full TLS connection using the socket is done using the following scheme:

1. Call `connect()` or `accept()` on a standard TCP file descriptor.
2. Use a user space TLS library to complete a handshake.
3. Create a new KTLS socket file descriptor.
4. Extract the TLS Initialization Vectors (IVs), session keys, and sequence IDs from the TLS library. Use the `setsockopt` function on the kTLS file descriptor (FD) to pass them to the kernel.
5. Use standard `read()`, `write()`, `sendfile()` and `splice()` system calls on the KTLS FD.

Drivers can offer Tx and Rx packet encryption/decryption offload from the kernel into the NIC hardware. Upon receipt of a non-data TLS message (a control message), the kTLS socket returns an error, and the message is left on the original TCP socket instead. The kTLS socket is automatically unattached. Transfer of control back to the original encrypted FD is done by calling `getsockopt` to receive the current sequence numbers, and inserting them into the TLS library.

Kernel Support

For support in the kernel, make sure the following flags are set as follows.

- `CONFIG_TLS=y`
- `CONFIG_TLS_DEVICE=y | m`

> **Note**
>
> For kTLS **Tx** device offloads with OFED drivers, kernel TLS module (kernel/net/tls) must be aligned to kernel v5.3 and above.
>
> For kTLS **Rx** device offloads with OFED drivers, kernel TLS module (kernel/net/tls) must be aligned to kernel v5.9 and above.

Configuring kTLS Offloads

To enable kTLS Tx offload, run:

```
ethtool -K <ifs> tls-hw-tx-offload on
```

To enable kTLS Rx offload, run:

```
ethtool -K <ifs> tls-hw-rx-offload on
```

For further information on TLS offloads, please visit the following kernel documentation:

- Kernel TLS Offload
- Kernel TLS

OpenSSL with kTLS Offload

OpenSSL version 3.0.0 or above is required to support kTLS TX/RX offloads.

Supported OpenSSL version is available to download from distro packages, or can be downloaded and compiled from the OpenSSL github.

Info

For a configuration example, please refer to the DOCA TLS Offload Guide.

IPsec Crypto Offload

Note

This feature is supported on crypto-enabled products of NVIDIA® BlueField®-2 DPUs, and NVIDIA® ConnectX®-6 Dx and ConnectX-7 adapters (but not of ConnectX-6).

Newer/future crypto-enabled DPU and adapter product generations should also support the feature, unless explicitly stated in their documentation.

Note

For BlueField-2 and ConnectX-6 Dx devices only: If your target application will utilize bandwidth of 100Gb/s or higher, where a substantial part of the bandwidth will be allocated for IPsec traffic, please refer to the *NVIDIA BlueField-2 DPUs Product Release Notes* or *NVIDIA ConnectX-6 Dx Adapters Product Release Notes* to learn about a potential bandwidth limitation. To access the relevant product release notes, please contact your NVIDIA sales representative.

IPsec crypto offload feature, also known as IPsec inline offload or IPsec aware offload feature enables the user to offload IPsec crypto encryption and decryption operations to the hardware.

Note

The hardware implementation only supports AES-GCM encryption scheme.

Enabling IPsec Crypto Offload

To enable the feature, support in both kernel and adapter firmware is required.

- To add IPsec crypto offload support in the kernel, set the following flags accordingly:

```
CONFIG_XFRM_OFFLOAD=y
CONFIG_INET_ESP_OFFLOAD=m
CONFIG_INET6_ESP_OFFLOAD=m
```

> **Note**
>
> These flags are enabled by default in RedHat 8 and Ubuntu 18.04.0.

- To check whether IPsec crypto offload is supported in firmware, look for the following string in the dmesg:

```
mlx5e: IPSec ESP acceleration enabled
```

Configuring Security Associations for IPsec Offloads

To program the inline offload security associations (SA), add the option `offload dev <netdev interface> dir out/in` in the `ip xfrm state` command for transmitting and receiving SA.

- Transmit inline offload SA xfrm command example:

```
sudo ip xfrm state add src 192.168.1.64/24 dst 192.168.1.65/24 proto
esp spi 0x46dc6204 reqid 0x46dc6204 mode transport aead
'rfc4106(gcm(aes))' 0x60bd6c3eafba371a46411830fd56c53af93883261ed1fb26767820ff493f43ba35b
offload dev p4p1 dir out sel src 192.168.1.64 dst 192.168.1.65
```

- Receive inline offload SA xfrm command example:

```
sudo ip xfrm state add src 192.168.1.65/24 dst 192.168.1.64/24 proto
esp spi 0xaea0846c reqid 0xaea0846c mode transport aead
'rfc4106(gcm(aes))' 0x81d5c3167c912c1dd50dab0cb4b6d815b6ace8844304db362215a258cd19deda
offload dev p4p1 dir in sel src 192.168.1.65 dst 192.168.1.64
```

Example of setting xfrm policies:

- First server:

```
+ sudo ip xfrm state add src 192.168.1.64/24 dst 192.168.1.65/24
proto esp spi 0x28f39549 reqid 0x28f39549 mode transport aead
'rfc4106(gcm(aes))' 0x492e8ffe718a95a00c1893ea61afc64997f4732848ccfe6ea07db483175cb18de9
offload dev enp4s0 dir out sel src 192.168.1.64 dst 192.168.1.65
+ sudo ip xfrm state add src 192.168.1.65/24 dst 192.168.1.64/24
proto esp spi 0x622a73b4 reqid 0x622a73b4 mode transport aead
'rfc4106(gcm(aes))' 0x093bfee2212802d626716815f862da31bcc7d9c44cfe3ab8049e7604b2feb1254
offload dev enp4s0 dir in sel src 192.168.1.65 dst 192.168.1.64
+ sudo ip xfrm policy add src 192.168.1.64 dst 192.168.1.65 dir out
tmpl src 192.168.1.64/24 dst 192.168.1.65/24 proto esp reqid 0x28f39549
mode transport
+ sudo ip xfrm policy add src 192.168.1.65 dst 192.168.1.64 dir in
tmpl src 192.168.1.65/24 dst 192.168.1.64/24 proto esp reqid 0x622a73b4
mode transport
+ sudo ip xfrm policy add src 192.168.1.65 dst 192.168.1.64 dir fwd
tmpl src 192.168.1.65/24 dst 192.168.1.64/24 proto esp reqid 0x622a73b4
mode transport
```

- Second server:

```
+ ssh -A -t root@l-csi-0921d /bin/bash
+ set -e
+ '[' 0 == 1 ']'
+ sudo ip xfrm state add src 192.168.1.64/24 dst 192.168.1.65/24
proto esp spi 0x28f39549 reqid 0x28f39549 mode transport aead
'rfc4106(gcm(aes))' 0x492e8ffe718a95a00c1893ea61afc64997f4732848ccfe6ea07db483175cb18de9
offload dev enp4s0 dir in sel src 192.168.1.64 dst 192.168.1.65
+ sudo ip xfrm state add src 192.168.1.65/24 dst 192.168.1.64/24
proto esp spi 0x622a73b4 reqid 0x622a73b4 mode transport aead
'rfc4106(gcm(aes))' 0x093bfee2212802d626716815f862da31bcc7d9c44cfe3ab8049e7604b2feb1254
offload dev enp4s0 dir out sel src 192.168.1.65 dst 192.168.1.64
+ sudo ip xfrm policy add src 192.168.1.65 dst 192.168.1.64 dir out
tmpl src 192.168.1.65/24 dst 192.168.1.64/24 proto esp reqid 0x622a73b4
mode transport
+ sudo ip xfrm policy add src 192.168.1.64 dst 192.168.1.65 dir in
tmpl src 192.168.1.64/24 dst 192.168.1.65/24 proto esp reqid 0x28f39549
mode transport
+ sudo ip xfrm policy add src 192.168.1.64 dst 192.168.1.65 dir fwd
tmpl src 192.168.1.64/24 dst 192.168.1.65/24 proto esp reqid 0x28f39549
mode transport
+ echo 'IPSec tunnel configured successfully'
```

IPsec Full Offload

Note

When using NVIDIA® BlueField®-2 DPUs and NVIDIA® ConnectX®-6 Dx adapters only: If your target application utilizes 100Gb/s or a higher bandwidth, where a substantial part of the bandwidth is allocated for IPsec traffic, please refer to the relevant DPU or adapter card Product Release Notes to learn about a potential bandwidth limitation. To access the Release Notes, visit NVIDIA Networking's documentation website, or contact your NVIDIA sales representative.

Note

This feature is supported on crypto-enabled products of BlueField-2 DPUs, as well as on ConnectX-6 Dx, ConnectX-6 Lx and ConnectX-7 adapter cards. Note that it is not supported on ConnectX-6 cards.

Newer/future crypto-enabled DPU and adapter product generations should also support this feature, unless explicitly stated otherwise in their documentation.

Note

ConnectX-6 Dx adapters only support Full Offload: Encrypted Overlay where a hypervisor controls IPsec offload (e.g., see OVS IPsec) in a Linux OS with NVIDIA drivers.

Note

This feature requires Linux kernel v6.6, or higher.

> **Note**
>
> IPsec tunnel mode is supported in alpha-level when controlled by VM (VF capability) only.

This feature is designed to enable IPsec full offload in switchdev mode. The `ip-xfrm` command is used to configure IPsec states and policies, and it is similar to legacy mode configuration. However, there are several limitations to the use of full offload in this mode:

1. Only IPsec Transport Mode and Tunnel Mode are supported.
2. The first IPsec TX state/policy is not allowed to be offloaded if any offloaded TC rule exists, and the same applies for the first RX state/policy. More specifically, IPsec RX/TX tables must be created before offloading any TC rule. For this reason, it is a common practice to configure IPsec rules before adding any TC rule.

Following is an example for IPsec configuration with a VXLan tunnel:

- Enable switchdev mode:

```
echo 1 > /sys/class/net/$PF0 /device/sriov_numvfs
echo 0000:08:00.2 > /sys/bus/pci/drivers/mlx5_core/unbind
devlink dev param set pci/0000:08:00.0 name flow_steering_mode
value dmfs cmode runtime
devlink dev eswitch set pci/0000:08:00.0 mode switchdev
echo 0000:08:00.2 > /sys/bus/pci/drivers/mlx5_core/bind
```

- Configure PF/VF/REP netdevices, and place a VF in a namespace:

```
ifconfig $PF $LOCAL_TUN/16 up
ip l set dev $PF mtu 2000

ifconfig $REP up
ip netns add ns0
ip link set dev $VF netns ns0
ip netns exec ns0 ifconfig $VF $IP/16 up
```

- Configure IPsec states and policies:

```
ip xfrm state add src $LOCAL_TUN/16 dst $REMOTE_IP/16 proto
esp spi 0xb29ed314 reqid 0xb29ed314 mode transport aead
'rfc4106(gcm(aes))' 0x20f01f80a26f633d85617465686c32552c92c42f 128 offload
packet dev $PF dir out sel src $LOCAL_TUN/16 dst
$REMOTE_IP/16 flag esn replay-window 64
ip xfrm state add src $REMOTE_IP/16 dst $LOCAL_TUN/16 proto
esp spi 0xc35aa26e reqid 0xc35aa26e mode transport aead
'rfc4106(gcm(aes))' 0x6cb228189b4c6e82e66e46920a2cde39187de4ba 128 offload
packet dev $PF dir in sel src $REMOTE_IP/16 dst $LOCAL_TUN/16
flag esn replay-window 64

ip xfrm policy add src $LOCAL_TUN dst $REMOTE_IP offload
packet dev $PF dir out tmpl src $LOCAL_TUN/16 dst
$REMOTE_IP/16 proto esp reqid 0xb29ed314 mode transport
priority 12
ip xfrm policy add src $REMOTE_IP dst $LOCAL_TUN offload
packet dev $PF dir in tmpl src $REMOTE_IP/16 dst
$LOCAL_TUN/16 proto esp reqid 0xc35aa26e mode transport priority
12
```

- Configure Openvswitch:

```
ovs-vsctl add-br br-ovs
ovs-vsctl add-port br-ovs $REP
ovs-vsctl add-port br-ovs vxlan1 -- set interface vxlan1
type=vxlan options:local_ip=$LOCAL_TUN
options:remote_ip=$REMOTE_IP options:key=$VXLAN_ID
options:dst_port=4789
```

IPsec Full Offload for RDMA Traffic

This IPsec Full Offload for RDMA Traffic option provides a significant performance improvement compared to the software IPsec counterpart, and enables the use of IPsec over RoCE packets, which are outside the network stack and cannot be used without full hardware offload. As a result, users can leverage the benefits of the IPsec protocol with RoCE V2, even when using SR-IOV VFs.

The configuration steps for this feature should be identical to the steps mentioned above, but if this feature is supported, the traffic that will be sent can also be RoCEV2 IPsec traffic.

To configure this feature:

1. Enable IPsec over VF. For more information, please see IPsec Functionality.

2. Configure IPsec policies and states on the relevant VF net device. This should be identical to the software configuration of IPsec rules, which can be done using one of the following implementation options:

Command	Offload Request Parameter
iproute2 ip xfrm	offload packet
libreswan	nic-offload=packet
strongswan [1]	

   1. For an example of using strongSwan configuration, refer to the DOCA East-West Overlay Encryption Application.

3. Configure an SR-IOV VF normally, and add its OVS/TC rules.

Note

For this feature to work, DMFS steering mode must be enabled.

- The following is a full minimalistic configuration example using iproute, whereas PF0 is the netdevice PF, F0_REP is the VF representor, and NIC is the VF netdevice to configure IPsec over:

```
1.      echo 1 > /sys/class/net/$PF0 /device/sriov_numvfs
2.      echo 0000:08:00.2 >
/sys/bus/pci/drivers/mlx5_core/unbind
3.      devlink dev eswitch set pci/0000:08:00.0 mode switchdev
4.      devlink dev param set pci/0000:08:00.0 name
flow_steering_mode value dmfs cmode runtime
5.      devlink port function set pci/0000:08:00.0/1
ipsec_packet enable
6.      echo 0000:08:00.2 > /sys/bus/pci/drivers/mlx5_core/bind
7.      tc qdisc add dev $PF0 ingress
tc qdisc add dev $VF0_REP ingress
tc filter add dev $PF0 parent ffff: protocol 802.1q chain 0
flower vlan_id 10 vlan_ethtype 802.1q cvlan_id 5 action vlan
pop action vlan pop  action mirred egress redirect dev
$VF0_REP

tc filter add dev $VF0_REP parent ffff: protocol all chain 0
flower action vlan push protocol 802.1q id 5 action vlan push
protocol 802.1q id 10 action mirred egress redirect dev $PF0

8.      ifconfig $PF0 $PF_IP/24 up
ifconfig $NIC $LOC_IP/$SUB_NET up
ip link set dev $VF_REP up
9.      ip  xfrm state flush
ip  xfrm policy flush
```

- Configure IPsec states and policies:

```
#states
ip  -4 xfrm state add src $LOC_IP/$SUB_NET dst
$REMOTE_IP/$SUB_NET proto esp spi 1000 reqid 10000 aead
'rfc4106(gcm(aes))' 0x010203047aeaca3f87d060a12f4a4487d5a5c335 128 mode
transport sel src $LOC_IP dst $REMOTE_IP offload packet dev
$NIC dir out
ip  -4 xfrm state add src $REMOTE_IP/$SUB_NET dst
$LOC_IP/$SUB_NET proto esp spi 1001 reqid 10001 aead
'rfc4106(gcm(aes))' 0x010203047aeaca3f87d060a12f4a4487d5a5c335 128 mode
transport sel src $REMOTE_IP dst $LOC_IP offload packet dev
$NIC dir in
#policies
ip  -4 xfrm policy add src $LOC_IP dst $REMOTE_IP offload
packet dev $NIC dir out tmpl src $LOC_IP/$SUB_NET dst
$REMOTE_IP/$SUB_NET proto esp reqid 10000 mode transport
ip  -4 xfrm policy add src $REMOTE_IP dst $LOC_IP offload
packet dev $NIC dir in tmpl src $REMOTE_IP/$SUB_NET dst
$LOC_IP/$SUB_NET proto esp reqid 10001 mode transport
ip  -4 xfrm policy add src $REMOTE_IP dst $LOC_IP dir fwd
tmpl src $REMOTE_IP/$SUB_NET dst $LOC_IP/$SUB_NET proto esp
reqid 10001 mode transport
```

Note that the configuration above is for one side only, yet IPsec must be configured for both sides in order for them to communicate properly. The configuration for the other side should be almost identical, but Step 9 would be configured in an asymmetrical way, meaning the first policy would look the following, and all other states/policies would be adjusted accordingly:

```
ip -4 xfrm state add src $LOC_IP/$SUB_NET dst $REMOTE_IP/$SUB_NET
proto esp spi 1001 reqid 10001 aead
'rfc4106(gcm(aes))' 0x010203047aeaca3f87d060a12f4a4487d5a5c335 128 mode transport
sel src $LOC_IP dst $REMOTE_IP offload packet dev $NIC dir out
```

Once this step is completed, you can send any RoCE traffic of your choice between the two machines with configured IPsec. For example,
`ibv_rc_pingpong -g 3 -d VF_device :` on one side, and
`ibv_rc_pingpong -g 3 -d VF_device $IP_OF_OTHER_SIDE :` on the other side.

Finally, you can verify that the traffic was encrypted using IPsec by using the ipsec counters:

```
ethtool -S VF_NETDEV | grep ipsec
```

MACsec Full Offload

Note

MACsec full offload is supported at alpha level only.

MACsec Full offload feature, also known as MACsec inline Full offload, enables the user to offload MACsec crypto encryption and decryption, MACsec headers encapsulation and decapsulation, and Anti replay operations to the hardware.

Note

Hardware implementation supports GCM-AES & GCM-AES-XPN encryption schemes and is supported with ConnectX-7 onwards.

Note

MACsec introduced in MOFED v5.9 requires a minimal Kernel version of 6.1.

To enable the feature, support in both kernel and adapter firmware is required.

For support in the kernel, make sure the following flags are set as follows:

- `CONFIG_MACSEC=y`
- `CONFIG_MLX5_EN_MACSEC=y`

For support in firmware, use version xx.34.0364 and up.

Configurations

IProute2 Configuration

Configuring Physical Interface

- Client side:

```
ip address flush <physical_device>
ip address add <client_physical_device_ip> dev <physical
interface>
ip link set dev <physical_device> up
```

- Server side:

```
ip address flush <physical_device>
ip address add <server_physical_device_ip> dev <physical
interface>
ip link set dev <physical_device> up
```

Add MACsec Device

- Client side:

```
ip link add link <physical_device> <macsec_device> type
macsec sci <client_sci> client on
```

- Server side:

```
ip link add link <physical_device> <macsec_device> type
macsec sci <server_sci> client on
```

Offload MACsec Device

- Client side:

```
ip macsec offload <macsec_device> mac
```

- Server side:

```
ip macsec offload <macsec_device> mac
```

Add MACsec rules:

- Client side:

```
ip macsec add <macsec_device> tx sa <sa_num>pn
<inital_packet_number>on key <client_key_id> <client_key>
ip macsec add <macsec_device> rx sci <server_sci> on
ip macsec add <macsec_device> rx sci <server_sci>sa <sa_num>
pn <inital_packet_number> on key <server_key_id> <server_key>
```

- Server side:

```
ip macsec add <macsec_device> tx sa <sa_num>pn
<inital_packet_number>on key <server_key_id> <server_key>
ip macsec add <macsec_device> rx sci <client_sci> on
ip macsec add <macsec_device> rx sci <client_sci>sa <sa_num>
pn <inital_packet_number> on key <client_key_id> <client_key>
```

Configure MACsec device IPs:

- Client side:

```
ip address flush <macsec_device>
ip address add <client_macsec_device_ip> dev <macsec_device>
ip link set dev <macsec_device> up
```

- Server side:

```
ip address flush <macsec_device>
ip address add <server_macsec_device_ip> dev <macsec_device>
ip link set dev <macsec_device> up
```

Configuration Example

Client side:

```
ip address flush enp8s0f0
ip address add 1.1.1.1/24 dev enp8s0f0
ip link set dev enp8s0f0 up
ip link add link enp8s0f0 macsec0 type macsec sci 1 encrypt on
ip macsec offload macsec0 mac
ip macsec add macsec0 tx sa 0 pn 1 on key 00
dffafc8d7b9a43d5b9a3dfbbf6a30c16
ip macsec add macsec0 rx sci 2 on
ip macsec add macsec0 rx sci 2 sa 0 pn 1 on key 00
ead3664f508eb06c40ac7104cdae4ce5
ip address flush macsec0
ip address add 2.2.2.1/24 dev macsec0
ip link set dev macsec0 up
```

Server side:

```
ip link del macsec0
ip address flush enp8s0f0
ip address add 1.1.1.2/24 dev enp8s0f0
ip link set dev enp8s0f0 up
ip link add link enp8s0f0 macsec0 type macsec sci 2 encrypt on
ip macsec offload macsec0 mac
ip macsec add macsec0 tx sa 0 pn 1 on key 00
ead3664f508eb06c40ac7104cdae4ce5
ip macsec add macsec0 rx sci 1 on
ip macsec add macsec0 rx sci 1 sa 0 pn 1 on key 00
dffafc8d7b9a43d5b9a3dfbbf6a30c16
ip address flush macsec0
ip address add 2.2.2.2/24 dev macsec0
ip link set dev macsec0 up
```

Note

Use `ip macsec show` command to check configuration.

To verify traffic is offloaded, check MACsec counters by running `ethtool -S <physical_device> | grep macsec`.

Info

Refer to the Linux Manual page for more information.

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/).

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.

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

> **Note**
>
> 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.

> **Note**
>
> 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.
    1. 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:
fe800000000000002c90200402bd5
          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
```

2. 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.

    1. 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=fe8000000000000002c90200402bd5,pkey=ffff,service_id
```

2. 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=fe8000000000000002c90200402bd5,pkey=ffff,service_id
> /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.

  > **Note**
  >
  > 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=fe8000000000000002c90200402bed,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/.

Note

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.

Note

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/

> **Note**
>
> 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

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:

   1. Unmount all SRP partitions that were mounted.

   2. Stop service srpd (Kill the SRP daemon instances).

   3. Make sure there are no multipath instances running. If there are multiple instances, wait for them to end or kill them.

   4. Run: `multipath -F`

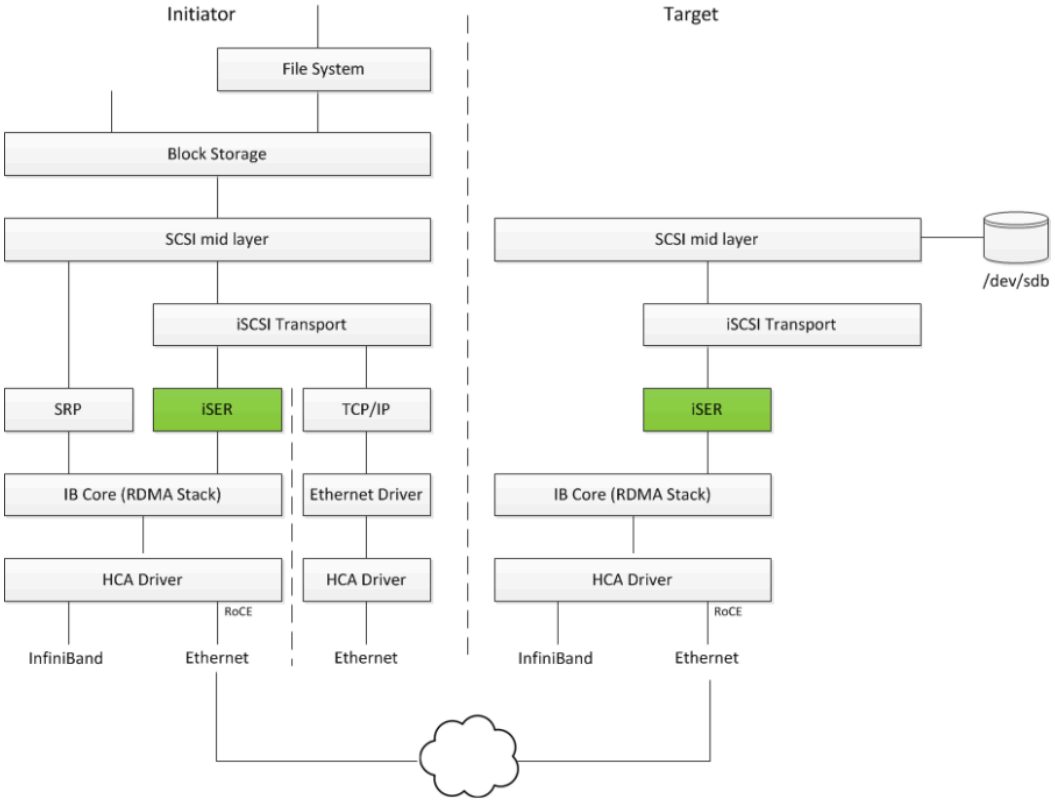
3. After Automatic Activation of High Availability

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.

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.

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.

iSER Targets

> **Note**
>
> 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.

> **Note**
>
> 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.

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:

> **Note**
>
> 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
```

```
$ 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.

NVME-oF - NVM Express over Fabrics

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.

Note

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.

NVME-oF Target Offload

> **Note**
>
> 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.

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

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.

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

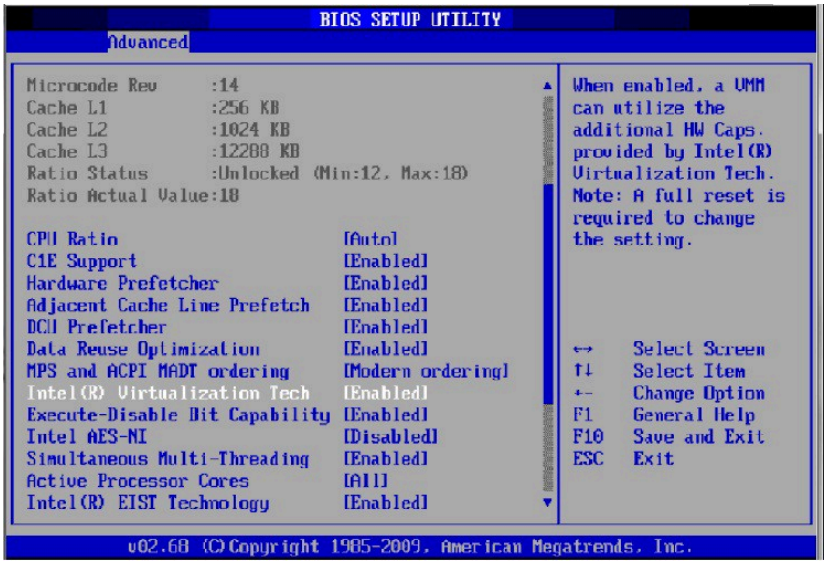
Setting Up SR-IOV

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.



2. Enable "Intel Virtualization Technology".



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").

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.

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
```

Note

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)
```

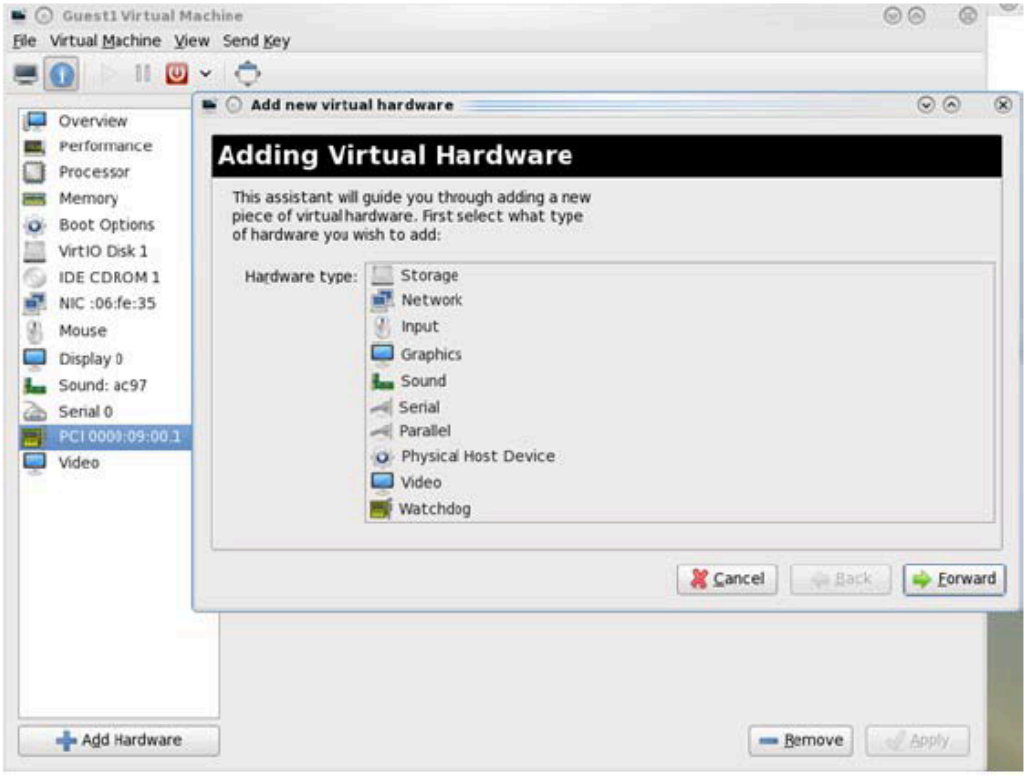
Additional SR-IOV Configurations

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.



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

> **Note**
>
> 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

> **Note**
>
> 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.
    1. To enable MAC anti-spoofing, run:

```
ip link set ens785f1 vf 0 spoofchk on
```

2. 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.

1. To enable MAC anti-spoofing, run:

```
echo "ON" > /sys/class/net/ens785f1/vf/0/spoofchk
```

2. To disable MAC anti-spoofing, run:

```
echo "OFF" > /sys/class/net/ens785f1/vf/0/spoofchk
```

Note

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//device/sriov//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:

    1. echo 10000 > /sys/class/net/<ifname>/device/sriov/5/max_tx_rate – setting individual rate limitation of VF 5 to 10G (Optional)

    2. echo 7 > /sys/class/net/<ifname>/device/sriov/5/group – moving VF 5 to group 7

    3. echo 5000 > /sys/class/net/<ifname>/device/sriov/groups/7/max_tx_rate – setting group 7 with rate limitation of 5G

    4. 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

    5. echo 3 > /sys/class/net/<ifname>/device/sriov/5/group – moving VF 5 to group 3

    6. 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.
    1. To enable trust for a specific VF, run:

        ```
        ip link set ens785f1 vf 0 trust on
        ```

    2. 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.

1. To enable trust for a specific VF, run:

```
echo "ON" > /sys/class/net/ens785f1/device/sriov/0/trust
```

2. 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 DriversCommunity 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
```

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.

SR-IOV Live Migration

> **Note**
>
> 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 **Note** 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
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.

> **Note**
>
> 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.

    1. See the name of the VFs, run:

```
devlink port show
```

    2. Unbind the VFs from mlx5_core, run:

```
echo '<VF_BDF>' > /sys/bus/pci/drivers/mlx5_core/unbind
```

    3. 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.

    1. Edit the VMs XML file, run:

    ```
    virsh edit <VM_NAME>
    ```

    2. 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>
    ```

    Note

    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.

    1. Edit the destination VM XML file, run:

```
virsh edit <VM_NAME>
```

2. 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>
```

> **Note**
>
> 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.

1.

1. Detach the VFs from libvirt management, run:

```
virsh nodedev-detach pci_<VF_BDF>
```

2. 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
```

3. Set driver override, run:

```
echo 'mlx5_vfio_pci' >
/sys/bus/pci/devices/<VF_BDF>/driver_override
```

4. 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.

    1. See the name of the VFs, run:

```
devlink port show
```

2. Unbind the VFs from mlx5_core, run:

```
echo '<VF_BDF>' > /sys/bus/pci/drivers/mlx5_core/unbind
```

3. 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:

    1. Set driver override, run:

```
echo 'mlx5_vfio_pci' >
/sys/bus/pci/devices/<VF_BDF>/driver_override
```

    2. 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"
```

> **Note**
>
> 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
```

> **Note**
>
> 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

Note

In ConnectX-7 adapter cards, migration cannot run in parallel on more than 4 VFs. It is the administrator's responsibility to control that.

Note

Live migration requires same firmware version on both the source and the target hosts.

Enabling Paravirtualization

➤ To enable Paravirtualization:

Note

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
…
```

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

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
```

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)

Q-in-Q Encapsulation per VF in Linux (VST)

Note

This feature is supported on ConnectX-5 and ConnectX-6 adapter cards only.

Note

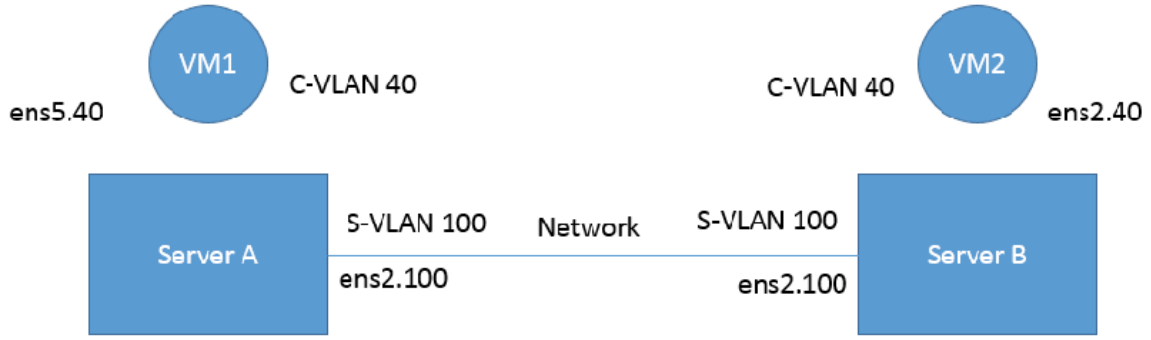
ConnectX-4 and ConnectX-4 Lx adapter cards support 802.1Q double-tagging (C-tag stack- ing on C-tag), refer to "802.1Q Double-Tagging" section.

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.

Setup

The setup assumes there are two servers equipped with ConnectX-5/ConnectX-6 adapter cards.



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.

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:

1. By using sysfs:

```
echo '100:0:802.1ad' > /sys/class/net/ens1f0/device/sriov/0/vlan
```

2. 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.

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.

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:

    1. By using sysfs:

    ```
    echo '100:0:802.1q' > /sys/class/net/ens1f0/device/sriov/0/vlan
    ```

    2. 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.

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.

Resiliency

The chapter contains the following sections:

- Reset Flow

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
```

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.

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
```

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.

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
```

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.

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.

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.

> **Note**
>
> 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
```

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

Orchestration and Clustering Tool	Version	Networking Mode	Link Layer	Virtualization Mode
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

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.

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.

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.

HPC-X

For information on HPC-X®, please refer to HPC-X User Manual at developer.nvidia.com/networking/hpc-x.

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.

DOCA Applications

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

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.

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

Installation

DOCA applications are installed under `/opt/mellanox/doca/applications` with each application having its own dedicated folder. Each application installs two directories:

- `bin` – contains pre-built binaries alongside runtime configuration files
- `src` – contains the source code and compilation files

The application's binary is located under `/opt/mellanox/doca/applications/<application_name>/bin/doca_<applicat`.

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>/src/meson.buil` – 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
```

Info

The generated applications are located under the `/tmp/build/` directory, using the following path `/tmp/build/<application_name>/src/doca_<application_name>`.

Note

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.

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
```

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
		Flow	DPA	DMA	FlexIO SDK	PCC	App Shield	IPsec	SHA	Compres
Network	GPU Packet Processing									
	NAT	✔								
	Simple Forward VNF	✔								
	Switch	✔								

Application Category	Application	Library Category								
Security	App Shield Agent						✔			
	East-west Overlay Encryption									
	Firewall	✔								
	IPsec Security Gateway							✔		
	Secure Channel									
	YARA Inspection						✔			
Data Path Acceleration	DPA All-to-all		✔							
	DPA L2 Reflector				✔					
	PCC					✔				
Storage	DMA Copy			✔						
	File Compression									✔
	File Integrity								✔	
HPC	Allreduce									

Applications

Allreduce

This application is a collective operation that allows data from many processing units to be collected and merged into a global result before being delivered to all processing units using an operator. The application is implemented using the UCX communication framework, which leverages the DPU's low-latency and high-bandwidth utilization of its network engine.

App Shield Agent

This 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.

DMA Copy

This 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.

DPA All-to-all

This application is a collective operation that allows data to be copied between multiple processes. This application is implemented using DOCA DPA, which leverages the d ata 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.

DPA L2 Reflector

This 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.

East-West Overlay Encryption

This 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.

File Compression

This 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.

File Integrity

This 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.

Firewall

This application applies network security based on DOCA Flow gRPC and is used for remote programming of the DPU hardware. It leverages DPU capabilities such as building generic execution pipes in hardware, monitoring incoming and outgoing network traffic, and more.

GPU Packet Processing

This 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.

IPsec Security Gateway

This 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.

NAT

This application, network address translation, switches packets with local IP addresses to global ones and vise versa. It is based on the DOCA Flow library which leverages DPU hardware capabilities such as building generic execution pipes in the hardware, executing specific actions on the traffic, and more.

Programmable Congestion Control

This 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.

Secure Channel

This application is used to establish a secure, network-independent communication channel between the host and the DPU based on the DOCA Comm Channel library.

Simple Forward VNF

This 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.

Switch

This application is used to establish internal switching between representor ports on the DPU. It is based on the the DOCA Flow library which leverages DPU capabilities such as building generic execution pipes in the hardware, and more.

YARA Inspection

This 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.

NVIDIA DOCA Allreduce Application Guide

This guide provides a DOCA Allreduce collective operation implementation on top of NVIDIA® BlueField® DPU using UCX.

Introduction

Allreduce is a collective operation which allows collecting data from different processing units to combine them into a global result by a chosen operator. In turn, the result is distributed back to all processing units.

Allreduce operates in stages. Firstly, each participant scatters its vector. Secondly, each participant gathers the vectors of the other participants. Lastly, each participant performs their chosen operation between all the gathered vectors. Using a sequence of different allreduce operations with different participants, very complex computations can be spread among many computation units.

Allreduce is widely used by parallel applications in high-performance computing (HPC) related to scientific simulations and data analysis, including machine learning calculation and the training phase of neural networks in deep learning.

Due to the massive growth of deep learning models and the complexity of scientific simulation tasks that utilize a network, effective implementation of allreduce is essential for minimizing communication time.

This document describes how to implement allreduce using the UCX communication framework, which leverages NVIDIA® BlueField® DPU by providing low-latency and high-bandwidth utilization of its network engine.

This document describes the following types of allreduce:

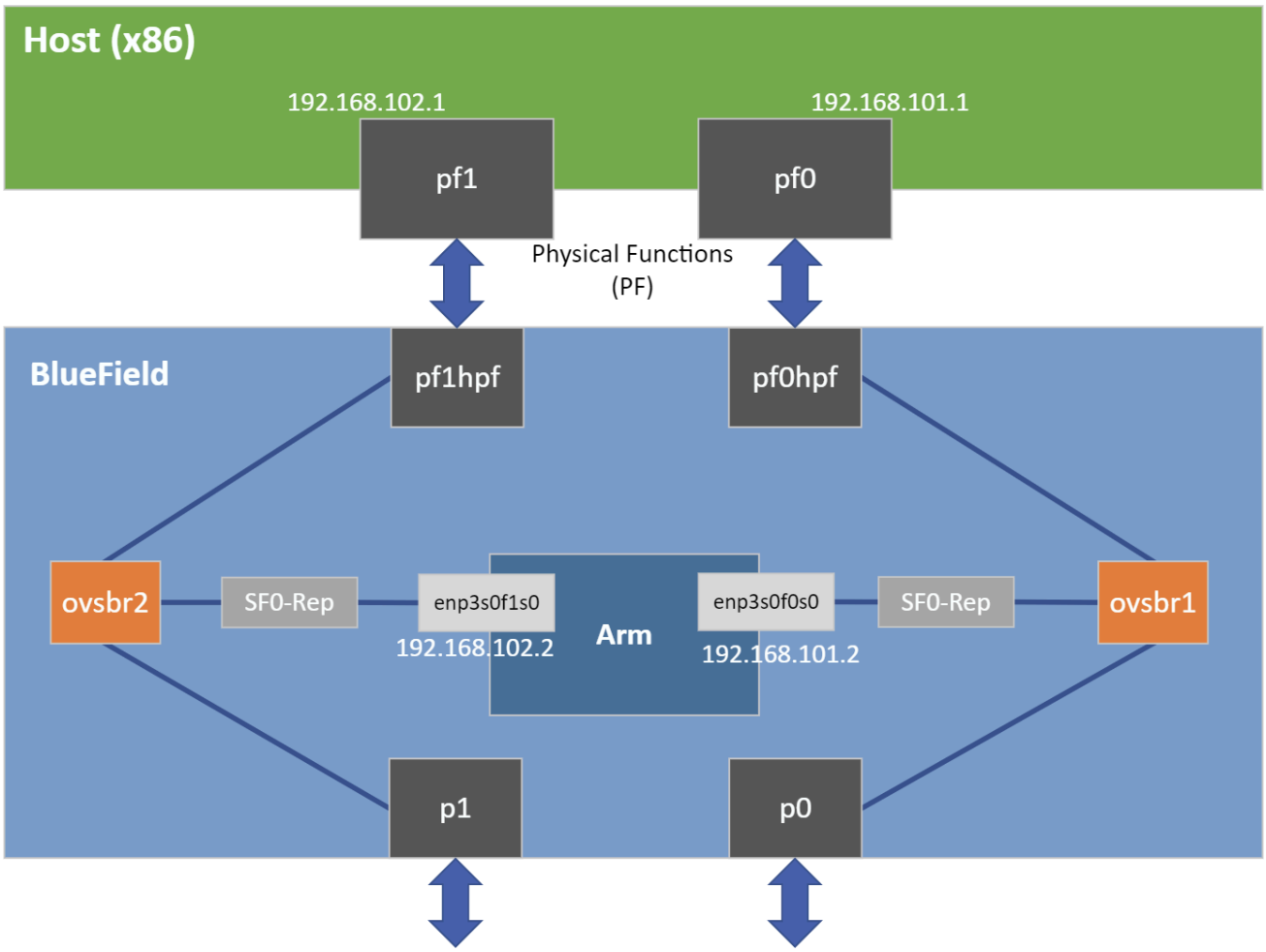
- Offloaded client – processes running on the host which only submit allreduce operation requests to a daemon running on the DPU. The daemon runs on the DPU and performs the allreduce algorithm on behalf of its on-host-clients (offloaded-client).

- Non-offloaded client – processes running on the host which execute the allreduce algorithm by themselves

System Design

The application is designed to measure three metrics:

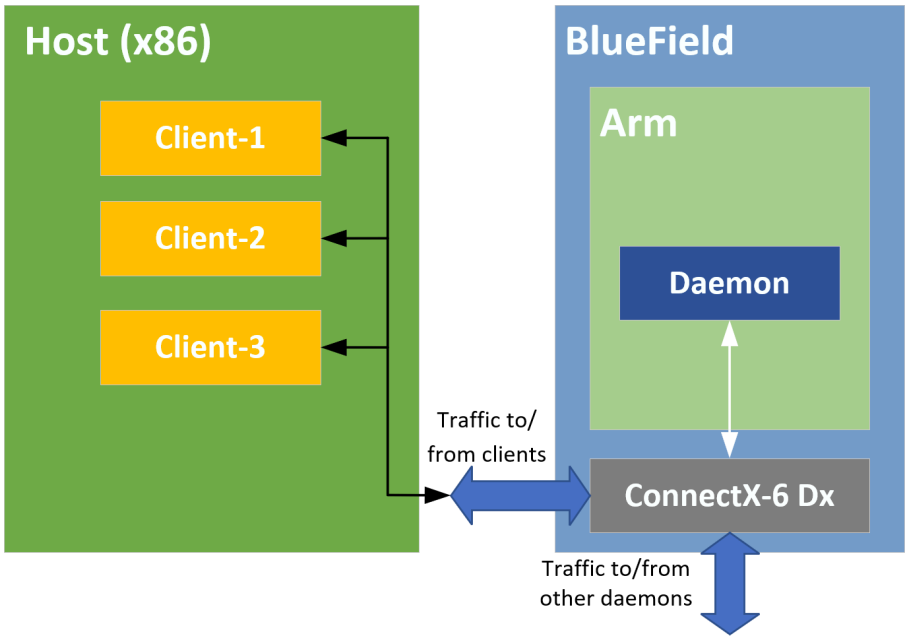
- Communication time taken by offloaded and non-offloaded allreduce operations
- Computation time taken by matrix multiplications which are done by clients until the allreduce operation is completed
- The overlap of the two previous metrics. The percentage of the total runtime during which both the allreduce and the matrix multiplications were done in parallel.



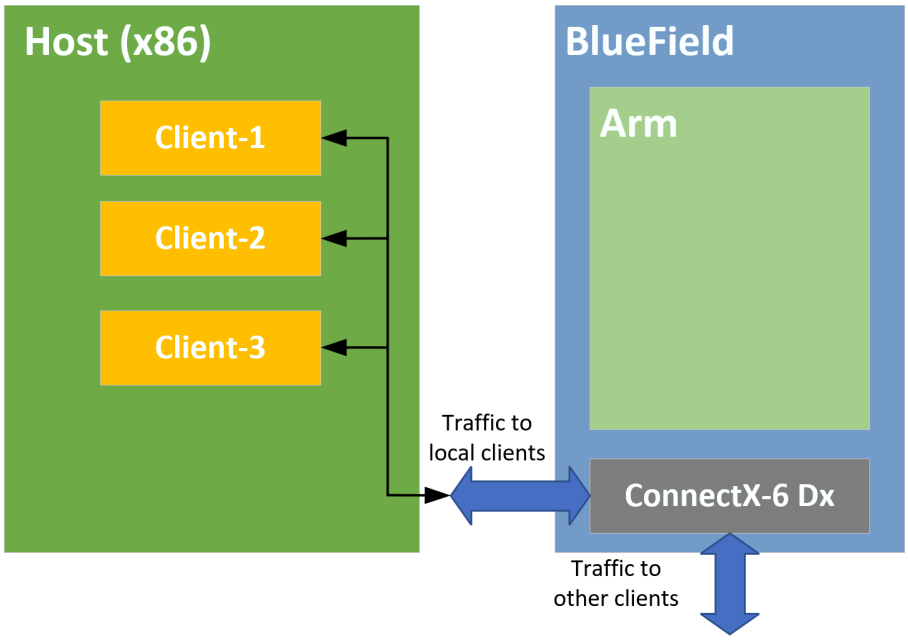
The allreduce implementation is divided into two different types of processes: clients and daemons. Clients are responsible for allocating vectors filled with data and initiating allreduce operations by sending a request with a vector to their daemon. Daemons are responsible for gathering vectors from all connected clients and daemons, applying a

chosen operator on all received buffers, and then scattering the reduced result vector back to the clients.

- Offloaded mode



- Non-offloaded mode

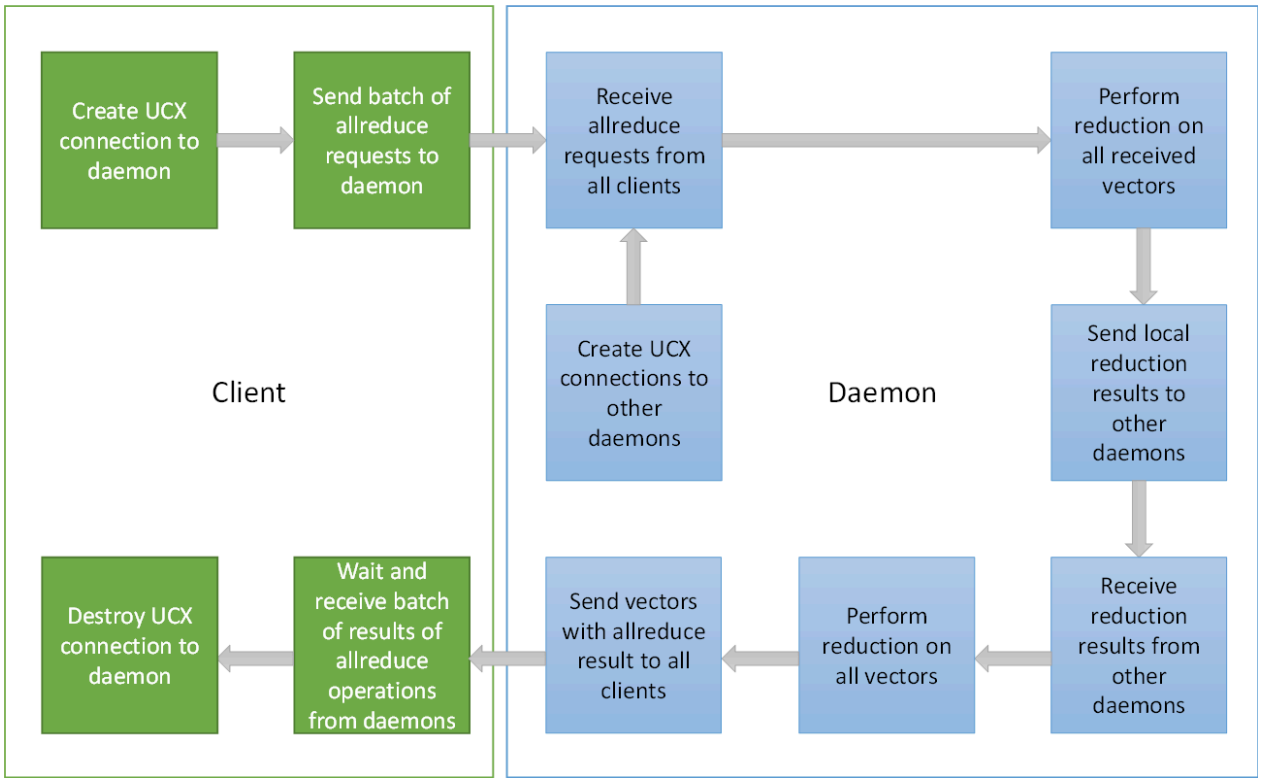


Application Architecture

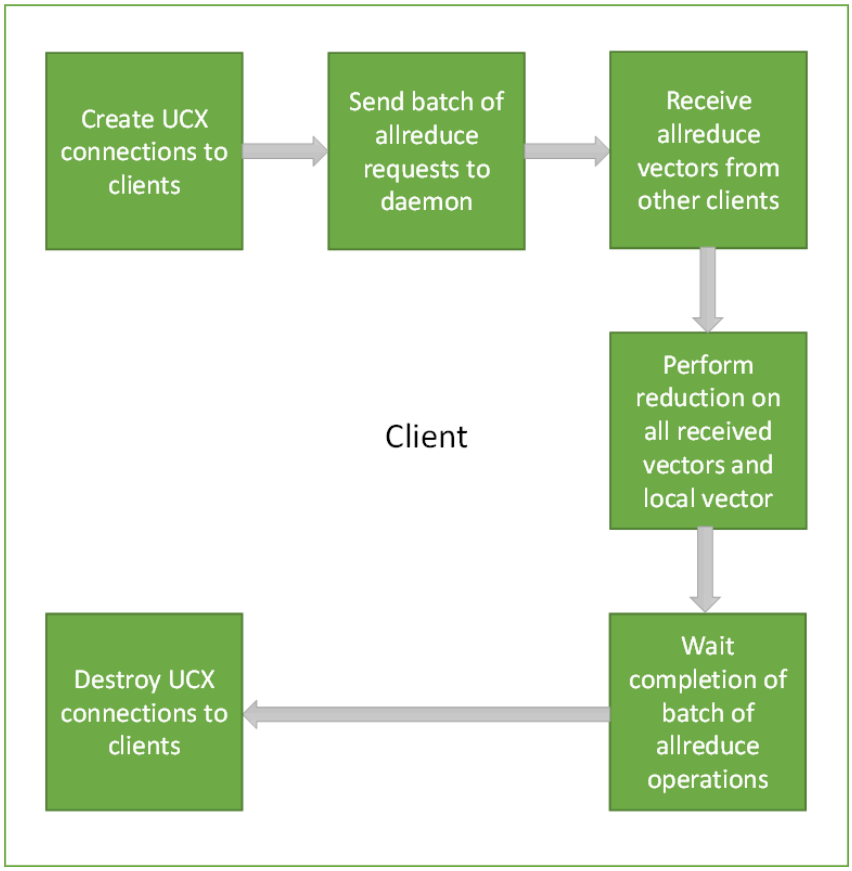
DOCA's allreduce implementation uses Unified Communication X (UCX) to support data exchange between endpoints. It utilizes UCX's sockaddr-based connection establishment

and the UCX Active Messages (AM) API for communications.

- Offloaded mode



- Non-offloaded mode



1. Connections between processes are established by UCX using IP addresses and ports of peers.

2. Allreduce vectors are sent from clients to daemons in offloaded mode, or from clients to clients in non-offloaded mode.

3. Reduce operations on vectors are done using received vectors from other daemons in offloaded mode, or other clients in non-offloaded mode.

4. Vectors with allreduce results are received by clients from daemons in offloaded mode, or are already stored in clients after completing all exchanges in non-offloaded mode.

5. After completing all allreduce operations, connections between clients are destroyed.

DOCA Libraries

This application leverages the UCX framework DOCA driver.

Running the Application

Installation

Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

Application Execution

The allreduce application is provided in both source and binary forms. The binary is located under `/opt/mellanox/doca/applications/allreduce/bin/doca_allreduce`.

1. Application usage instructions:

```
Usage: doca_allreduce [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:
  -r, --role                        Run DOCA UCX allreduce
process as: "client" or "daemon"
  -m, --mode <allreduce_mode>       Set allreduce mode:
"offloaded", "non-offloaded" (valid for client only)
  -p, --port <port>                 Set default destination
port of daemons/clients, used for IPs without a port (see '-a'
flag)
  -t, --listen-port <listen_port>   Set listening port of
daemon or client
  -c, --num-clients <num_clients>   Set the number of clients
which participate in allreduce operations (valid for daemon
only)
  -s, --size <size>                 Set size of vector to do
allreduce for
  -d, --datatype <datatype>         Set datatype ("byte", "int",
"float", "double") of vector elements to do allreduce for
  -o, --operation <operation>       Set operation ("sum",
"prod") to do between allreduce vectors
```

```
  -b, --batch-size <batch_size>     Set the number of
allreduce operations submitted simultaneously (used for
handshakes by daemons)
  -i, --num-batches <num_batches>   Set the number of batches
of allreduce operations (used for handshakes by daemons)
  -a, --address <ip_address>        Set comma-separated list
of destination IPv4/IPv6 addresses and ports optionally
(<ip_addr>:[<port>]) of daemons or clients
```

Info

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
/opt/mellanox/doca/applications/allreduce/bin/doca_allrec
-h
```

Info

For additional information, refer to section "Command Line Flags".

2. Configuration steps.

    1. All daemons should be deployed before clients. Only after connecting to their peers are daemons able to handle clients.

    2. UCX probes the system for any available net/IB devices and, by default, tries to create a multi-device connection. This means that if some network devices are available but provide an unreachable path from the daemon to the peer/client, UCX may still use that path. A common case is that a daemon tries to connect

to a different BlueField using `tmfifo_net0` which is connected to the host only. To fix this issue, follow these steps:

1. Use the UCX env variable `UCX_NET_DEVICES` to set usable devices. For example:

```
export UCX_NET_DEVICES=enp3s0f0s0,enp3s0f1s0
/opt/mellanox/doca/applications/allreduce/bin/doca_all
-r daemon -t 34001 -c 1 -s 100 -o sum -d float -b 16 -i
16
```

   Or:

```
env UCX_NET_DEVICES=enp3s0f0s0,enp3s0f1s0
/opt/mellanox/doca/applications/allreduce/bin/doca_all
-r daemon -t 34001 -c 1 -s 100 -o sum -d float -b 16 -i
16
```

2. Get the mlx device name and port of a SF to limit the UCX network interfaces and allow IB. For example:

```
BlueField> 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

BlueField>
UCX_NET_DEVICES=enp3s0f0s0,enp3s0f1s0,mlx5_2:1,mlx5_3:
/opt/mellanox/doca/applications/allreduce/bin/doca_all
-r daemon -t 34001 -c 1 -s 100 -o sum -d float -b 16 -i
16
```

3. CLI example for running the deamon on BlueField:

```
/opt/mellanox/doca/applications/allreduce/bin/doca_allreduce
-r daemon -t 34001 -c 2 -a 10.21.211.3:35001,10.21.211.4:36001 -s 65535
-o sum -d float -i 16 -b 128
```

Notes:

- The flag `-a` is necessary for communicating with other daemons. In case of an offloaded client, the address must be that of the daemon which performs the allreduce operations for them. In case of a daemon or non-offloaded clients, the flag could be a single or multiple addresses of other daemons/non-offloaded clients which exchange their local allreduce results.
- The flag `-c` must be specified for daemon processes only. It indicates how many clients submit their allreduce operations to the daemon.

- The flags `-s`, `-i`, `-b`, and `-d` must be the same for all clients and daemons participating in the allreduce operation.

> **Note**
>
> The daemon listens to incoming connection requests on all available IPs, but the actual communication after the initial "UCX handshake" does not necessarily use the same device used for the connection establishment.

4. CLI example for running the client on the host:

```
/opt/mellanox/doca/applications/allreduce/bin/doca_allreduce
-r client -m non-offloaded -t 34001 -a
10.21.211.3:35001,10.21.211.4:36001 -s 65535 -i 16 -b 128 -o sum -d float
/opt/mellanox/doca/applications/allreduce/bin/doca_allreduce
-r client -m offloaded -p 34001 -a 192.168.100.2 -s 65535 -i 16 -b
128 -o sum -d float
```

5. The application also supports a JSON-based deployment mode, in which all command-line arguments are provided through a JSON file:

```
doca_allreduce --json [json_file]
```

For example:

```
cd /opt/mellanox/doca/applications/allreduce/bin
./doca_allreduce --json ./allreduce_client_params.json
```

> **Note**
>
> Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the desired PCIe and network addresses required for the deployment.

Command Line Flags

Flag Type	Short Flag	Long Flag/JSON Key	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`	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

Flag Type	Short Flag	Long Flag/JSON Key	Description
Program flags	`r`	`role`	Run DOCA UCX allreduce process as either `client` or `daemon`
	`m`	`mode`	Set allreduce mode. Available types options: • `offloaded` • `non-offloaded` (valid for client only)
	`p`	`port`	Set default destination port of daemons/clients. Used for IPs without a port (see `-a` flag).
	`c`	`num-clients`	Set the number of clients which participate in allreduce operations Note: Valid for daemon only.
	`s`	`size`	Set size of vector to perform allreduce for
	`d`	`datatype`	Set datatype of vector elements to do allreduce for • `byte` • `int` • `float` • `double`
	`o`	`operation`	Set operation to perform between allreduce vectors
	`b`	`batch-size`	Set the number of allreduce operations submitted simultaneously. Used for handshakes by daemons.
	`i`	`num-batches`	Set the number of batches of allreduce operations. Used for handshakes by daemons.
	`t`	`listen-port`	Set listening port of daemon or client
	`a`	`address`	Set comma-separated list of destination IPv4/IPv6 address and ports optionally of daemons or clients. Format: `<ip_addr>:[<port>]`.

Info

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

Recompiling the Application

In addition to providing the application in binary form, the installation also includes all of the application sources and compilation instructions so as to allow modifying the sources and recompiling the application. For more information about the applications, as well as development and compilation tips, please refer to the DOCA Applications page.

The sources of the application can be found under the `/opt/mellanox/doca/applications/allreduce/src` directory.

Recompiling All Applications

The applications are all defined under a single meson project, so the default compilation recompiles all the DOCA applications.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

Info

`doca_allreduce` is created under `/tmp/build/allreduce/src/`.

Recompiling Allreduce Application Only

To build the allreduce application only:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -
Denable_allreduce=true
ninja -C /tmp/build
```

Info

`doca_allreduce` is created under `/tmp/build/allreduce/src/`.

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_allreduce` to `true`

2. Run the following compilation commands:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> **Info**
>
> `doca_allreduce` is created under `/tmp/build/allreduce/src/`.

Troubleshooting

Please refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application .

Running Application on NVIDIA Converged Accelerator

This section details the steps necessary to run DOCA Allreduce on NVIDIA converged accelerator.

Allreduce running on the converged accelerator has the same logic as described in previous sections except for the reducing of vectors. The reduce of incoming vectors is performed on the GPU side in batches that include the vectors from all peers or all clients. When the GPUDirect module is active, incoming vectors and outgoing vectors are received/sent directly to/from the GPU.

To make use of the GPU's capabilities, make sure to perform the following:

1. Refer to the NVIDIA DOCA Installation Guide for Linux for instructions on installing NVIDIA driver for CUDA and a CUDA-repo on your setup.

2. Create the sub-functions and configure the OVS according to NVIDIA BlueField DPU Scalable Function User Guide.

Compiling and Running Application

Since there is no pre-compiled allreduce application binary provided that uses GPU support, you must compile it and run it. All the sources needed for building, compiling, and running the application with GPU support are found under `/opt/mellanox/doca/applications/allreduce/src`.

To build and run the application:

1. Setup CUDA paths:

```
export CPATH=/usr/local/cuda/targets/sbsa-linux/include:$CPATH
export LD_LIBRARY_PATH=/usr/local/nvidia/lib:/usr/local/nvidia/lib64:
export PATH=/usr/local/nvidia/bin:/usr/local/cuda/bin:$PATH
```

2. Reinstall UCX with CUDA support. Follow the UCX installation procedure with an additional flag, `--with-cuda=/usr/local/cuda/`, passed to `configure-release`:

```
dpu# ./contrib/configure-release --with-cuda=/usr/local/cuda/
```

3. To build the application with GPU support:

    1. Set the `enable_gpu_support` flag to `true` in `/opt/mellanox/doca/applications/meson_option.txt`.

    2. Compile the application sources. Run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

`doca_allreduce_gpu` is created under `/tmp/build/allreduce/src/` alongside the regular `doca_allreduce` binary that is compiled without the GPU support.

4. To run the application with GPU support, follow the same steps as described in section "Running the Application".

Application Code Flow

1. Parse application argument.
    1. Initialize arg parser resources and register DOCA general parameters.

    ```
    doca_argp_init();
    ```

    2. Register UCX allreduce application parameters.

    ```
    register_allreduce_params();
    ```

    3. Parse all registered parameters.

    ```
    doca_argp_start();
    ```

2. UCX initialization.
    1. Initialize hash table of connections.

```
allreduce_ucx_init();
```

2. Create UCP context.

```
ucp_init();
```

3. Create UCP worker.

```
ucp_worker_create();
```

4. Set AM handler for receiving connection check packets.

```
ucp_worker_set_am_recv_handler();
```

3. Initialization of the allreduce connectivity.

```
communication_init();
```

1. Initialize hash table of allreduce super requests.
2. Set "receive callback" for handshake messages.
3. If daemon or non-offloaded client:
    1. Set AM handler for receiving allreduce requests from clients.

```
allreduce_ucx_am_set_recv_handler();
```

2. Initialize UCX listening function. This creates a UCP listener.

```
allreduce_ucx_listen();
```

4. Initialize all connections.

```
connections_init();
```

1. Go over all destination addresses and connect to each peer.
2. Repeat until a successful send occurs (to check connectivity).

```
ucp_am_send_nbx();
allreduce_ucx_request_wait();
```

3. Insert the connection to the hash table of connections.

```
allreduce_outgoing_handshake();
```

5. Scatter handshake message to peers/daemon to make sure they all have the same `-s`, `-i`, `-b`, and `-d` flags.

4. Daemon: Start UCX progress.

```
daemon_run();
```

1. Set AM handler to receive allreduce requests from clients.

```
allreduce_ucx_am_set_recv_handler();
```

2. Perform UCP worker progress.

```
while (running)
    allreduce_ucx_progress();
```

3. Callbacks are invoked by incoming/outgoing messages by calling `allreduce_ucx_progress`.

5. Client:

```
client_run();
```

1. Allocate buffers to store allreduce initial data and results.

```
allreduce_vectors_init();
```

2. Set an AM handler for receiving allreduce results.

```
allreduce_ucx_am_set_recv_handler();
```

3. Perform allreduce barrier. Check that all daemons and clients are active.

```
allreduce_barrier();
```

1. Submit a batch of allreduce operations with 0 byte.

2. Wait for completions.

4. Reset metrics and vectors.

```
allreduce_metrics_init();
```

    1. Submit some batches and calculate estimated network time.

    2. Allocate matrices to multiply.

    3. Estimate how many matrix multiplications could have been performed instead of networking (same time window).

    4. Calculate the actual computation time of these matrix multiplications.

5. Reset vectors.

6. Submit a batch of allreduce operations to daemon/peer (depends on mode).

7. Perform matrix multiplications during a time period which is approximately equal to doing a single batch of allreduce operations and calculate the actual time cost.

8. Wait for the allreduce operation to complete and calculate time cost.

9. Update metrics.

```
Do num-batches (flag) times:
    allreduce_vectors_reset();
    allreduce_batch_submit();
    cpu_exploit();
    allreduce_batch_wait();
    allreduce_metrics_calculate();
```

10. Print summary of allreduce benchmarking.

```
allreduce_metrics_print();
```

6. Arg parser destroy.

```
doca_argp_destroy();
```

7. Communication destroy.
    1. Clean up connections.

```
allreduce_ucx_disconnect();
```

        1. Remove the connection from the hash table of the connections.
        2. Close inner UCP endpoint.

```
ucp_ep_close_nbx();
```

        3. Wait for the completion of the UCP endpoint closure.
        4. Destroy connection.
        5. Free connections array.
    2. Destroy the hash table of the allreduce super requests.
8. Destroy UCX context.
    1. Destroy the hash table of the connections.

```
g_hash_table_destroy();
```

2. If the UCP listener was created, destroy it.

```
ucp_listener_destroy();
```

3. Destroy UCP worker.

```
ucp_worker_destroy();
```

4. Destroy UCP context.

```
ucp_cleanup();
```

References

- `/opt/mellanox/doca/applications/allreduce/src`
- `/opt/mellanox/doca/applications/allreduce/bin/allreduce_client_pa`
- `/opt/mellanox/doca/applications/allreduce/bin/allreduce_daemon_pa`

NVIDIA DOCA App Shield Agent Application Guide

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

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).

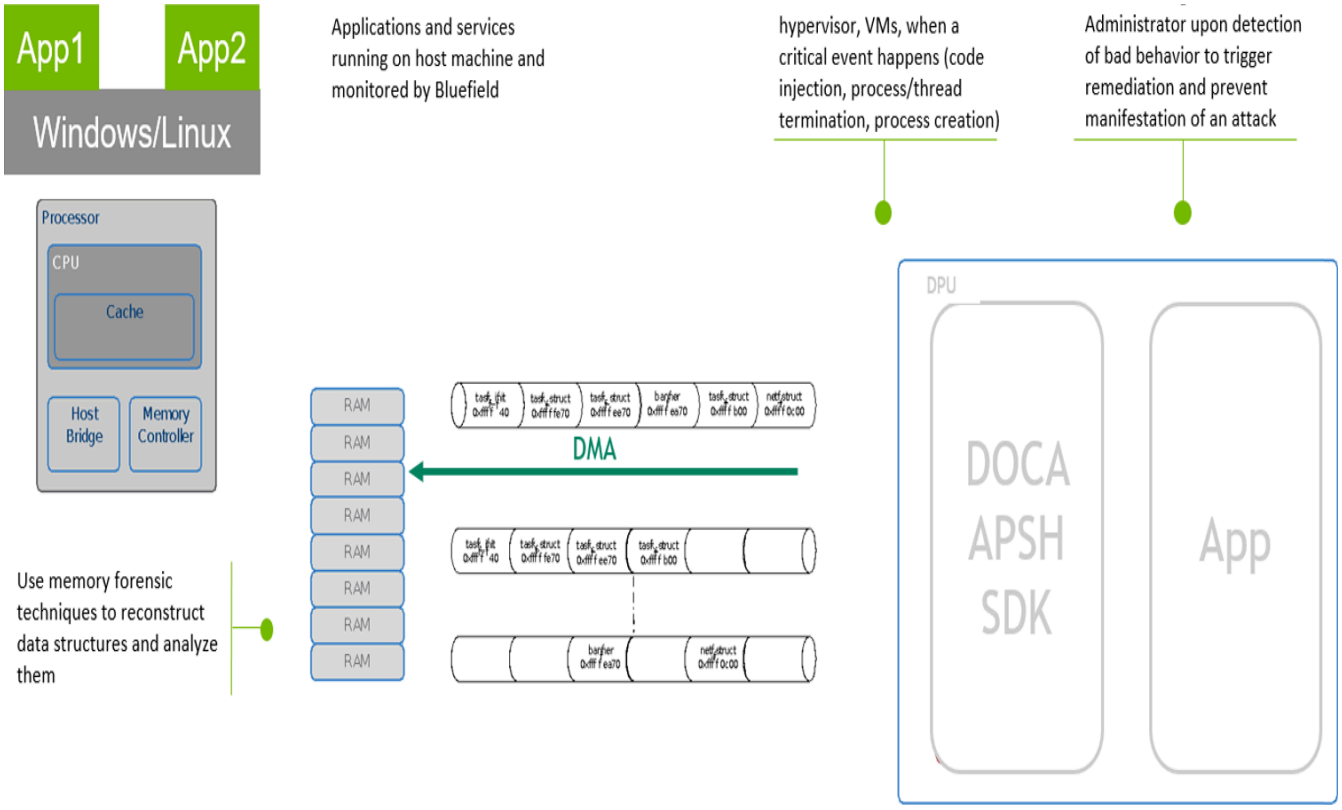
This guide describes how to build secure process monitoring using the DOCA App Shield library, which leverages the DPU's advantages such as hardware-based DMA, integrity, and more.

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.

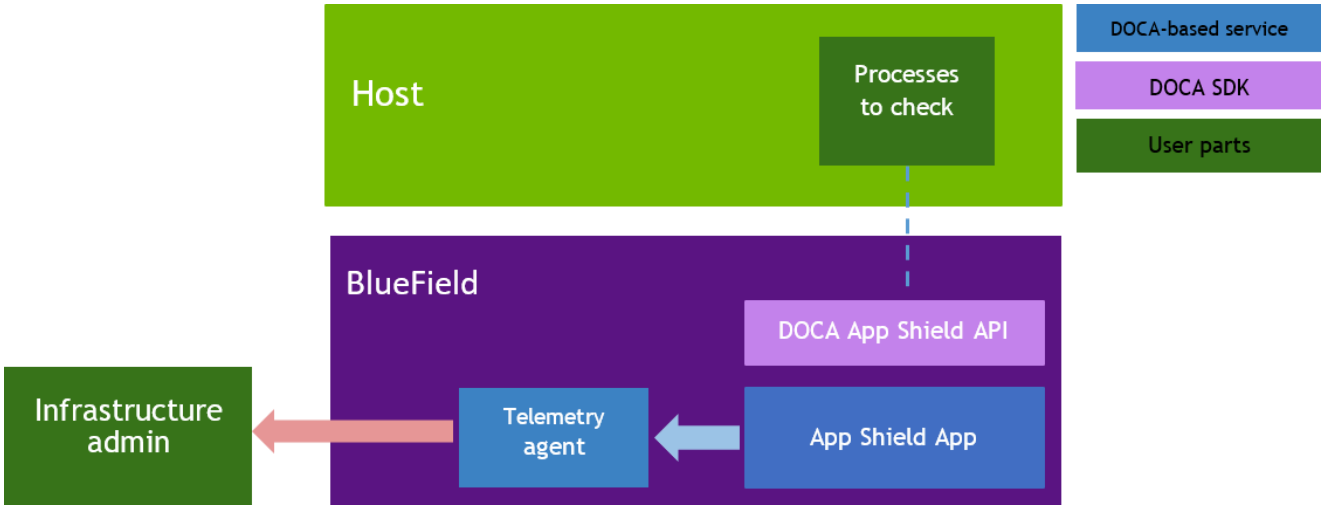
Generating the needed files can be done by running DOCA App Shield's `doca_apsh_config.py` tool on the host. See DOCA App Shield for more info.



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.

   > **Note**
   >
   > 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:
   1. Scans and hashes process memory pages (that are currently in use).
   2. Compares the hash to the original hash.
   3. 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:
   1. 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.
   2. DOCA log (without scan count).
   3. DTS IPC interface (even if no DTS is active).
6. The App Shield agent exits on first attestation failure.

DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA App Shield
- 2024-10-09_07-10-18_DOCA Telemetry

Refer to their respective programming guide for more information.

Running the Application

Installation

Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

Prerequisites

1. Configure the BlueField's firmware.

    1. 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
        ```

    2. Perform graceful shutdown and a cold boot from the host. Run:

        ```
        host> ipmitool power cycle
        ```

> **Note**
>
> These configurations can be checked 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).

> **Note**
>
> 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]
> ```

1. 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
   ```

2. 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"
   ```

3. Run:

> **Note**
>
> 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. Allocate 42 huge pages.

    1. 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
```

    2. 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.

> **Note**
>
> If the kernel and process `.exe` have not changed, there no need to redo this step.

Application Execution

1. The App Shield Agent application is provided in both source and binary forms. The binary is located under `/opt/mellanox/doca/applications/app_shield_agent/bin/doca_app_shi`.

    1. 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=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, --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
```

Info

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
/opt/mellanox/doca/applications/app_shield_agent/bin/
-h
```

Info

For additional information, please refer to section "Command Line Flags".

2. CLI example for running the application on the BlueField:

```
/opt/mellanox/doca/applications/app_shield_agent/bin/doca_
-p 13577 -e hash.zip -m mem_regions.json -o symbols.json -
f MT2125X03335MLNXS0D0F0VF1 -d mlx5_0 -t 3 -s linux
```

Note

All used identifiers (`-f`, `-p` and `-d` flags) should match the identifier of the desired devices and processes.

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

Flag Type	Short Flag	Long Flag	Description
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

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`.
			Note
			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

Flag Type	Short Flag	Long Flag	Description
			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
	`s`	`osty`	OS type (`windows` or `linux`) of the system where the process is running
	`t`	`time`	Number of seconds to sleep between scans

Info

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

Recompiling the Application

In addition to providing the application in binary form, the installation also includes all of the application sources and compilation instructions so as to allow modifying the sources and recompiling 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 `/opt/mellanox/doca/applications/app_shield_agent/src` directory.

Recompiling All Applications

The applications are all defined under a single meson project, so the default compilation recompiles all the DOCA applications.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> **Info**
>
> `doca_app_shield_agent` is created under `/tmp/build/app_shield_agent/src/`.

Recompiling App Shield Agent Application Only

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
```

Info

`doca_app_shield_agent` is created under `/tmp/build/app_shield_agent/src/`.

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
```

Info

`doca_app_shield_agent` is created under `/tmp/build/app_shield_agent/src/`.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application .

Application Code Flow

1. Parse application argument.
    1. Initialize arg parser resources and register DOCA general parameters.

    ```
    doca_argp_init();
    ```

    2. Register application parameters.

    ```
    register_apsh_params();
    ```

    3. Parse the arguments.

    ```
    doca_argp_start();
    ```

2. Initialize DOCA App Shield lib context.
    1. Create lib context.

    ```
    doca_apsh_create();
    ```

    2. Set DMA device for lib.

```
doca_devinfo_list_create();
doca_dev_open();
doca_devinfo_list_destroy();
doca_apsh_dma_dev_set();
```

3. Start the context

```
doca_apsh_start();
apsh_system_init();
```

3. Initialize DOCA App Shield lib system context handler.

    1. 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();
    ```

    2. 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();
```

    1. Initialize a new telemetry schema.
    2. Register attestation type event.
    3. Set up output to file (in addition to default IPC).
    4. Start the telemetry schema.
    5. 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_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();
```

References

- `/opt/mellanox/doca/applications/app_shield_agent/src`

NVIDIA DOCA DMA Copy Application Guide

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

Introduction

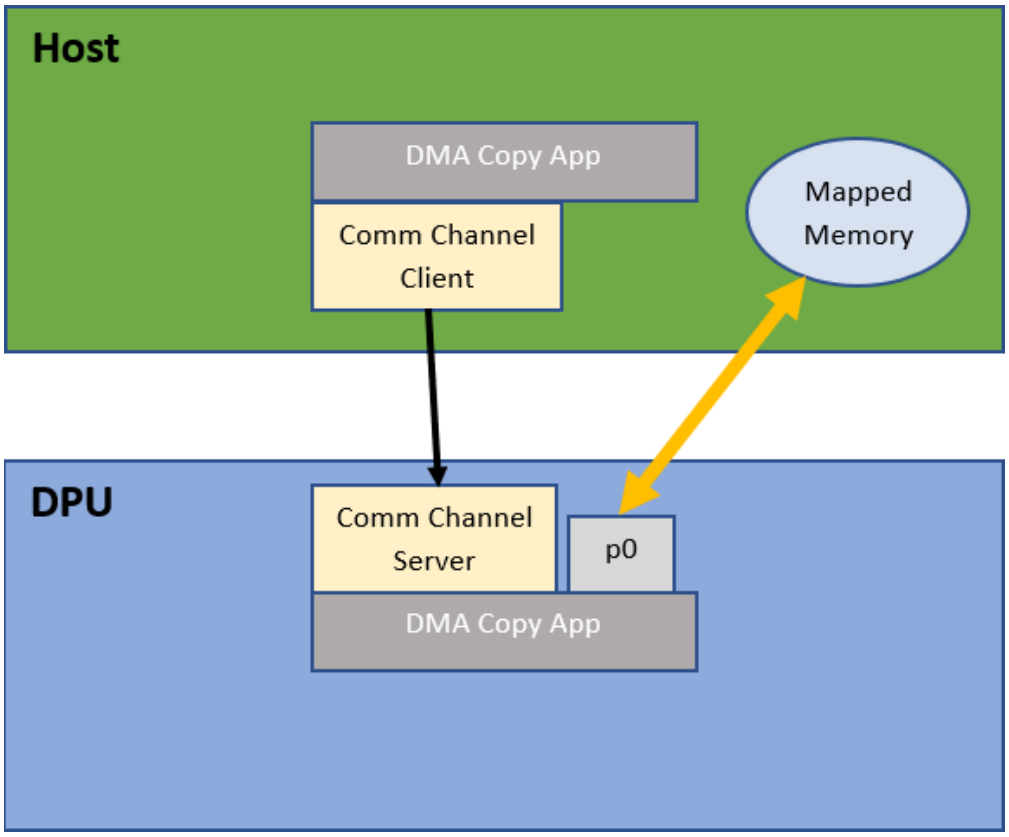
DOCA DMA (direct memory access) Copy application transfers files (data path), up to the maximum supported size by the hardware, between the DPU and the x86 host using the

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.

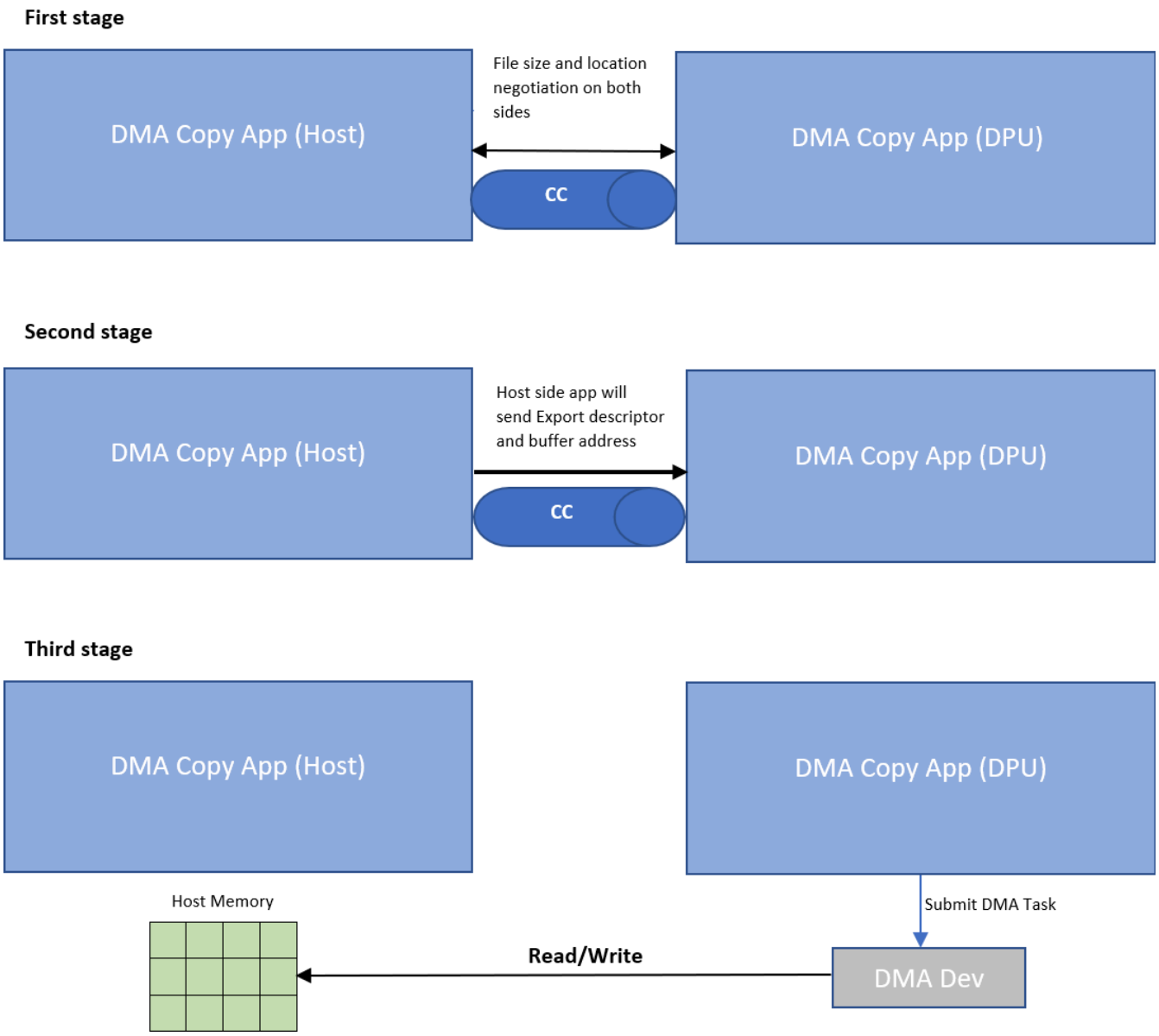
System Design

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 Comm Channel server between the two sides on which all the necessary DOCA DMA library configuration files (control path) are transferred.



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 Comm Channel 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.

DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA DMA
- DOCA Comm Channel

Refer to their respective programming guide for more information.

Running the Application

Installation

Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

Application Execution

The DMA copy application is provided in both source and binary forms. The binary is located under `/opt/mellanox/doca/applications/dma_copy/bin/doca_dma_copy`.

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=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:
  -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)
```

Info

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
/opt/mellanox/doca/applications/dma_copy/bin/doca_dma_cop
-h
```

Info

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the BlueField:

```
/opt/mellanox/doca/applications/dma_copy/bin/doca_dma_copy -p 03:00.0 -r 3b:00.0 -f received.txt
```

Note

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:

```
/opt/mellanox/doca/applications/dma_copy/bin/doca_dma_copy -p 3b:00.0 -f send.txt
```

Note

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_dma_copy --json [json_file]
```

For example:

```
cd /opt/mellanox/doca/applications/dma_copy/bin
./doca_dma_copy --json ./dma_copy_params.json
```

> **Note**
>
> Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the PCIe addresses necessary for the deployment.

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 **Note** This is a mandatory flag.	"file" : "/tmp/sample.txt"
	`p`	`pci-addr`	DOCA Comm Channel device PCIe address. **Note** This is a mandatory flag.	"pci-addr" : "b1:00.0"
	`r`	`rep-pci`	DOCA Comm Channel device representor PCIe address. **Note** This is a mandatory flag only on the DPU.	"rep-pci" : "b1:02.0"

Info

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

Recompiling the Application

In addition to providing the application in binary form, the installation also includes all of the application sources and compilation instructions so as to allow modifying the sources and recompiling 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 `/opt/mellanox/doca/applications/dma_copy/src` directory.

Recompiling All Applications

The applications are all defined under a single meson project, so the default compilation recompiles all the DOCA applications.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

Note

`doca_dma_copy` is created under `/tmp/build/dma_copy/src/`.

Recompiling DMA Copy Application Only

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
```

Info

`doca_dma_copy` is created under `/tmp/build/dma_copy/src/`.

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
```

Info

`doca_dma_copy` is created under `/tmp/build/dma_copy/src/`.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application .

Application Code Flow

1. Parse application argument.
    1. Initialize arg parser resources and register DOCA general parameters.

    ```
    doca_argp_init();
    ```

    2. Register DMA Copy application parameters.

    ```
    register_dma_copy_params();
    ```

3. Parse the arguments.

```
doca_argp_start();
```

1. Initialize Comm Channel endpoint.

```
init_cc();
```

1. Create Comm Channel endpoint.
2. Parse user PCIe address for Comm Channel device.
3. Open Comm Channel DOCA device.
4. Parse user PCIe address for Comm Channel device representor (on DPU side).
5. Open Comm Channel DOCA device representor (on DPU side).
6. Set Comm Channel endpoint properties.

2. Open the DOCA hardware device from which the copy would be made.

```
open_dma_device();
```

1. Parse the PCIe address provided by the user.
2. Create a list of all available DOCA devices.
3. Find the appropriate DOCA device according to specific properties.
4. 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.

    1. Host side application:

```
host_start_dma_copy();
```

        1. Start negotiation with the DPU side application for the location and size of the file.
        2. Allocate memory for the DMA buffer.
        3. Export the memory map and send the output (export descriptor) to the DPU side application.
        4. Send the host local buffer memory address and length on the Comm Channel to the DPU side application.
        5. Wait for the DPU to notify that DMA Copy ended.
        6. Close all memory objects.
        7. Clean resources.

    2. DPU side application:

```
dpu_start_dma_copy();
```

1. Start negotiation with the host side application for file location and size.
2. Allocate memory for the DMA buffer.
3. Receive the export descriptor on the Comm Channel.
4. Create the DOCA memory map for the remote buffer on the host.
5. Receive the host buffer information on the Comm Channel.
6. Create two DOCA buffers, one for the remote (host) buffer and one for the local buffer.
7. Submit the DMA copy task.
8. Send a host message to notify that DMA copy ended.
9. Clean resources.

6. Destroy Comm Channel.

```
destroy_cc();
```

7. Destroy DOCA core objects.

```
destroy_core_objects();
```

8. Arg parser destroy.

```
doca_argp_destroy();
```

References

- `/opt/mellanox/doca/applications/dma_copy/src`

- `/opt/mellanox/doca/applications/dma_copy/bin/dma_copy_params.json`

NVIDIA DOCA DPA All-to-all Application Guide

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

Introduction

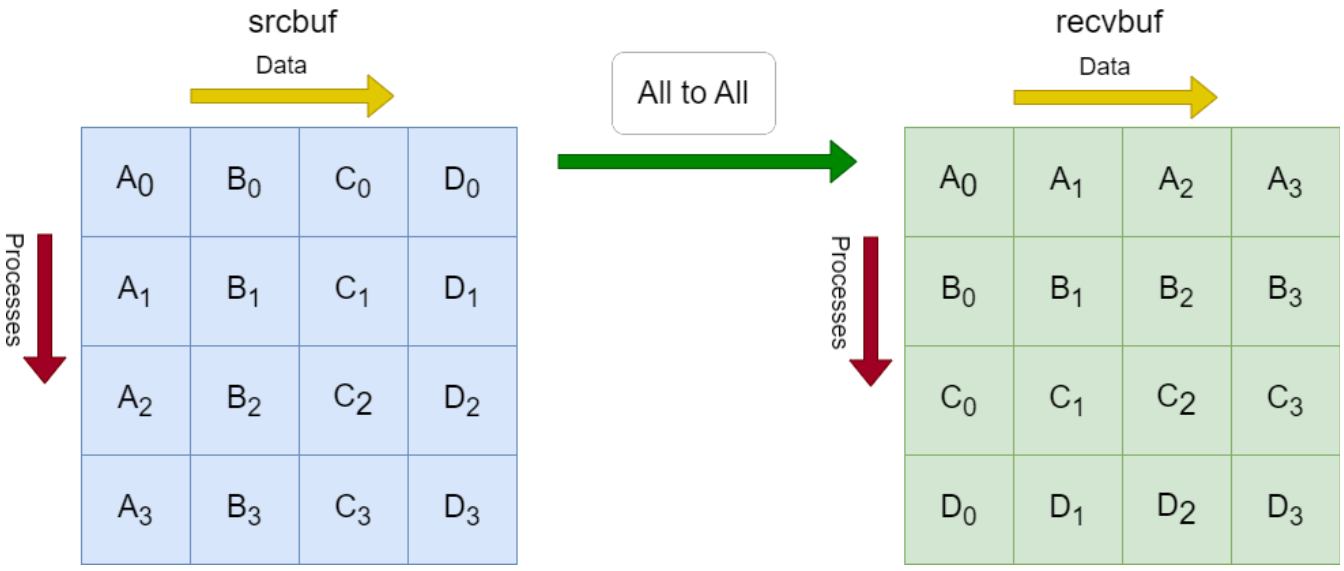
This reference application shows how the message passing interface (MPI) all-to-all collective can be accelerated on the Data Path Accelerator (DPA). In an MPI collective, all processes in the same job call the collective routine.

Given a communicator of *n* ranks, the application performs a collective operation in which all the processes send and receive the same amount of data from all the processes (hence all-to-all).

This document describes how to run the all-to-all example using DOCA DPA API .

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.

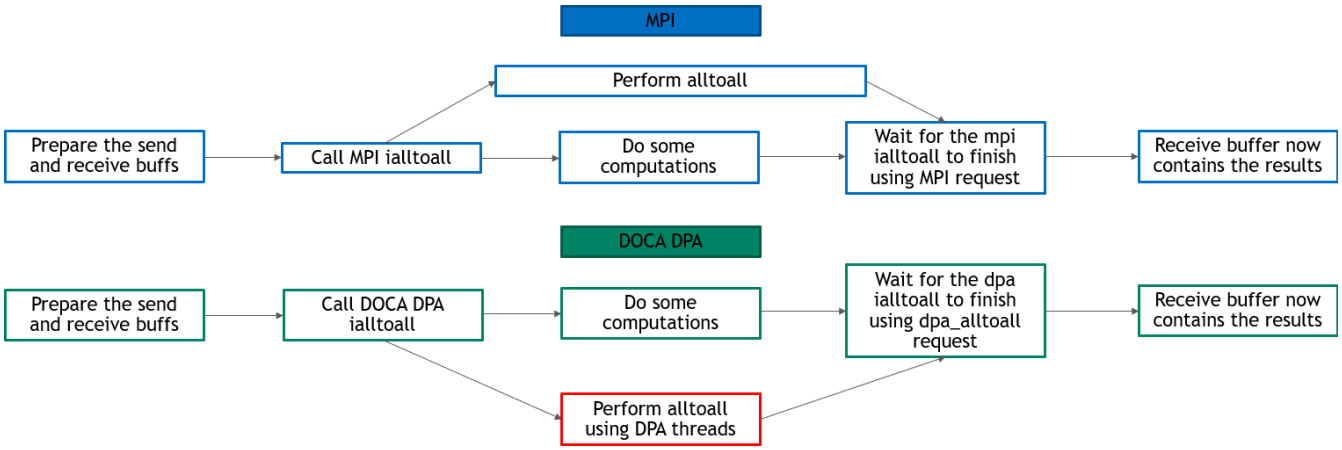


Each process in the diagram divides its local sendbuf into *n* blocks (4 in this example), each containing sendcount elements (4 in this example). Process *i* sends the *k*-th block of its local sendbuf to process *k* which places the data in the *i*-th block of its local recvbuf.

Implementing the 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.

Application Architecture

The following diagram describes the differences between the host based all-to-all and DPA all-to-all.



- In DPA all-to-all, DPA threads perform the all-to-all and the CPU is free to do other computations
- In host-based all-to-all, the CPU must still perform the all-to-all at some point and is not completely free for other computations

DOCA Libraries

This application leverages the following DOCA library:

- DOCA DPA

Refer to its programming guide for more information.

Dependencies

- NVIDIA BlueField-3 DPU is required

- The application can be run on the DPU or on the host
- Open MPI version 4.1.5rc2 or greater (included in DOCA's installation)

Running the Application

Installation

Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

Prerequisites

MPI is used for the compilation and running of this application. Make sure that MPI is installed on your setup (`openmpi` is provided as part of the installation of `doca-tools`).

Note

The installation also 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 this:

```
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}
```

Application Execution

The DPA all-to-all application is provided in both source and binary forms. The binary is located under `/opt/mellanox/doca/applications/dpa_all_to_all/bin/doca_dpa_all_to_al`.

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=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:
  -m, --msgsize <Message size>          The message size -
the size of the sendbuf and recvbuf (in bytes). Must be in
multiplies 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.
```

Info

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
/opt/mellanox/doca/applications/dpa_all_to_all/bin/doca_d
-h
```

Info

For additional information, please refer to section "Command Line Flags".

2. CLI example for running the application on the host:

Note

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
/opt/mellanox/doca/applications/dpa_all_to_all/bin/doca_dp
```

- The following runs the DPA all-to-all application with 8 processes, with 128 bytes as the message size, and with `mlx5_0` and `mlx5_1` as the InfiniBand devices:

```
mpirun-np 8
/opt/mellanox/doca/applications/dpa_all_to_all/bin/doca_dp
-m 128 -d "mlx5_0,mlx5_1"
```

> **Note**
>
> 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:

```
cd /opt/mellanox/doca/applications/dpa_all_to_all/bin
./doca_dpa_all_to_all --json ./dpa_all_to_all_params.json
```

> **Note**
>
> Before execution, ensure that the used JSON file contains the correct configuration parameters, especially the InfiniBand device identifiers.

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` **Note** 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"`

Info

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications .

Recompiling the Application

In addition to providing the application in binary form, the installation also includes all of the application sources and compilation instructions so as to allow modifying the sources and recompiling the application. For more information about the applications, as well as development and compilation tips, refer to the DOCA Applications main guide.

The sources of the application can be found under the `/opt/mellanox/doca/applications/dpa_all_to_all/src` directory.

Recompiling All Applications

The applications are all defined under a single meson project, meaning that the default compilation will recompile all the DOCA applications.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

Info

`doca_dpa_all_to_all` is created under `/tmp/build/dpa_all_to_all/src/`.

Recompiling DPA All-to-all Application Only

To directly build only the 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
```

Info

`doca_dpa_all_to_all` is created under `/tmp/build/dpa_all_to_all/src/`.

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_dpa_all_to_all` to `true`
2. Run the following compilation commands :

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

Info

`doca_dpa_all_to_all` is created under `/tmp/build/dpa_all_to_all/src/`.

Troubleshooting

Please refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application .

Application Code Flow

1. Initialize MPI.

```
MPI_Init(&argc, &argv);
```

2. Parse application arguments.
    1. Initialize arg parser resources and register DOCA general parameters.

```
doca_argp_init();
```

    2. Register the application's parameters.

```
register_all_to_all_params();
```

    3. Parse the arguments.

```
doca_argp_start();
```

1. 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.

2. The `devices_param` parameter is the names of the InfiniBand devices to use (must support DPA). It can include up to two devices names.

4. 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:

    1. Check the number of processes (maximum is 16).

    2. Check the `msgsize`. It must be in multiples of integer size and at least the number of processes times integer size.

    3. Allocate the sendbuf and recvbuf according to `msgsize`.

4. Prepare the resources required to perform the all-to-all method using DOCA DPA:

    1. Initialize DOCA DPA context:

        1. Open DOCA DPA device (DOCA device that supports DPA).

        ```
        open_dpa_device();
        ```

        2. Create DOCA DPA context using the opened device.

        ```
        doca_dpa_create();
        ```

    2. Create the required events for the all-to-all: One completion event for the kernel launch (wait location CPU and update location DPA) and kernel events

(wait location remote and update location DPA) as the number of processes.

```
create_dpa_a2a_events() {
        doca_dpa_event_create(doca_dpa,
DOCA_DPA_EVENT_ACCESS_DPA, DOCA_DPA_EVENT_ACCESS_CPU,
DOCA_DPA_EVENT_WAIT_DEFAULT, &comp_event, 0);
        for (i = 0; i < resources->num_ranks; i++)
                doca_dpa_event_create(doca_dpa,
DOCA_DPA_EVENT_ACCESS_REMOTE, DOCA_DPA_EVENT_ACCESS_DPA,
DOCA_DPA_EVENT_WAIT_DEFAULT, &(kernel_events[i]), 0);
}
```

3. Create DOCA DPA worker (for the endpoints).

```
doca_dpa_worker_create();
```

4. Prepare DOCA DPA endpoints:

    1. Create DOCA DPA endpoints as the number of processes/ranks.

    ```
    for (i = 0; i < resources->num_ranks; i++)
            doca_dpa_ep_create();
    ```

    2. Connect the local process' endpoints to the other processes' endpoints.

    ```
    connect_dpa_a2a_endpoints();
    ```

    3. Export the endpoints to DOCA DPA device endpoints (so they can be used by the DPA) and copy them to DPA heap memory.

```
for (int i = 0; i < resources->num_ranks; i++) {
        result = doca_dpa_ep_dev_export();
        doca_dpa_mem_alloc();
        doca_dpa_h2d_memcpy();
}
```

5. Prepare the memory required to perform the all-to-all method using DOCA DPA. This includes creating memory handlers for the sendbuf and recvbuf, getting the other processes' recvbufs handlers, and copying these memory handlers and their remote keys and the events' handlers to the DPA heap memory.

```
prepare_dpa_a2a_memory();
```

5. Launch the `alltoall_kernel` using DOCA DPA kernel launch with all the required parameters:

    1. Every MPI rank launches a kernel of up to `MAX_NUM_THREADS`. This example defines `MAX_NUM_THREADS` as 16.

    2. Launch `alltoall_kernel` using `kernel_launch`.

```
doca_dpa_kernel_launch();
```

    3. Using the `doca_dpa_dev_put_signal_nb()` function, every process should copy the relevant sendbuf to the correct recvbuf (according to the process' rank) for every process (including the current process itself).

```
for (i = thread_rank; i < num_ranks; i += num_threads)
        doca_dpa_dev_put_signal_nb();
```

4. Wait until the `alltoall_kernel` has finished.

```
doca_dpa_event_wait_until();
```

> **Note**
>
> Add an MPI barrier after waiting for the event to make sure that all of the processes have finished executing the `alltoall_kernel`.
>
> ```
> MPI_Barrier();
> ```

After the `alltoall_kernel` is finished, the recvbuf of all the processes now contain the expected output of the all-to-all method.

6. Destroy the `a2a_resources`:

    1. Free all the DOCA DPA memories.

    ```
    doca_dpa_mem_free();
    ```

    2. Unregister all the DOCA DPA host memories.

    ```
    doca_dpa_mem_unregister();
    ```

    3. Destroy all the DOCA DPA endpoints.

```
doca_dpa_ep_destroy();
```

4. Destroy the DOCA DPA worker.

```
doca_dpa_worker_destroy();
```

5. Destroy all the DOCA DPA events.

```
doca_dpa_event_destroy();
```

6. Destroy the DOCA DPA context.

```
doca_dpa_destroy();
```

7. Close the DOCA device.

```
doca_dev_close();
```

References

- `/opt/mellanox/doca/applications/dpa_all_to_all/src`
- `/opt/mellanox/doca/applications/dpa_all_to_all/bin/dpa_all_to_all`

NVIDIA DOCA DPA L2 Reflector Application Guide

This document provides a DPA L2 reflector implementation on top of the NVIDIA® BlueField®-3 DPU.

Introduction

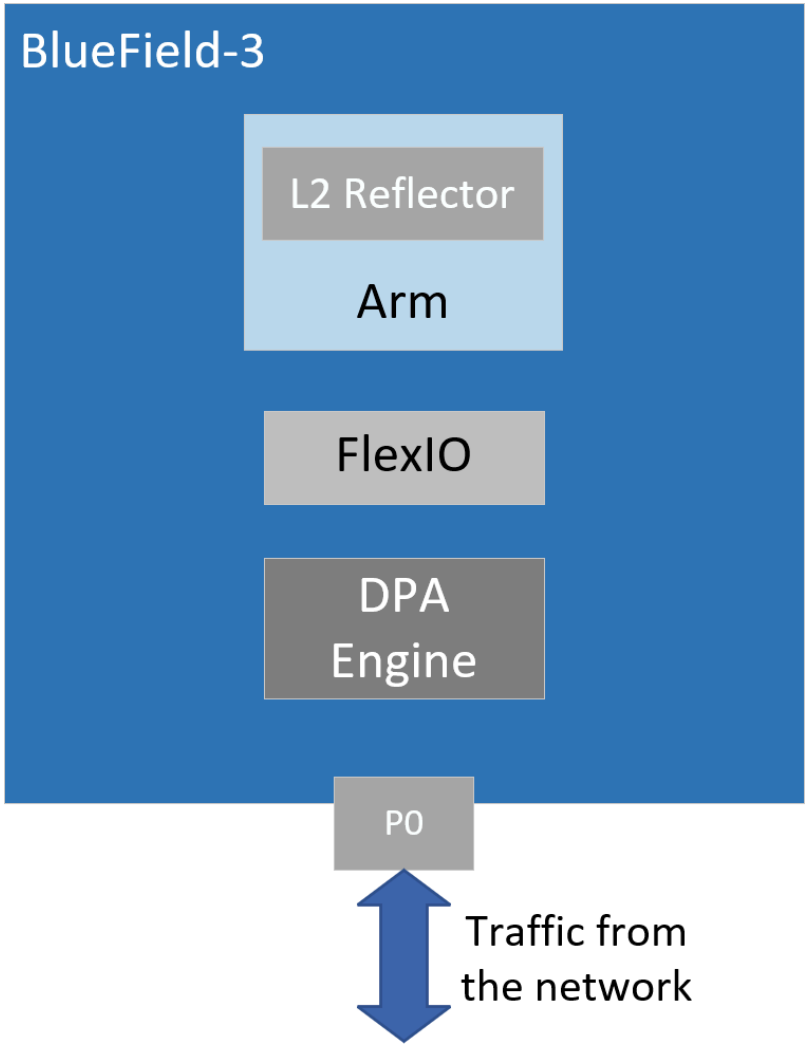
The BlueField-3 DPU supports high-speed Data Path Accelerator (DPA) . Data path accelerator allows for accelerated packet processing and manipulation.

DOCA l ayer-2 reflector uses the DPA engine to intercept network traffic and swap the source and destination MAC addresses of each packet.

System Design

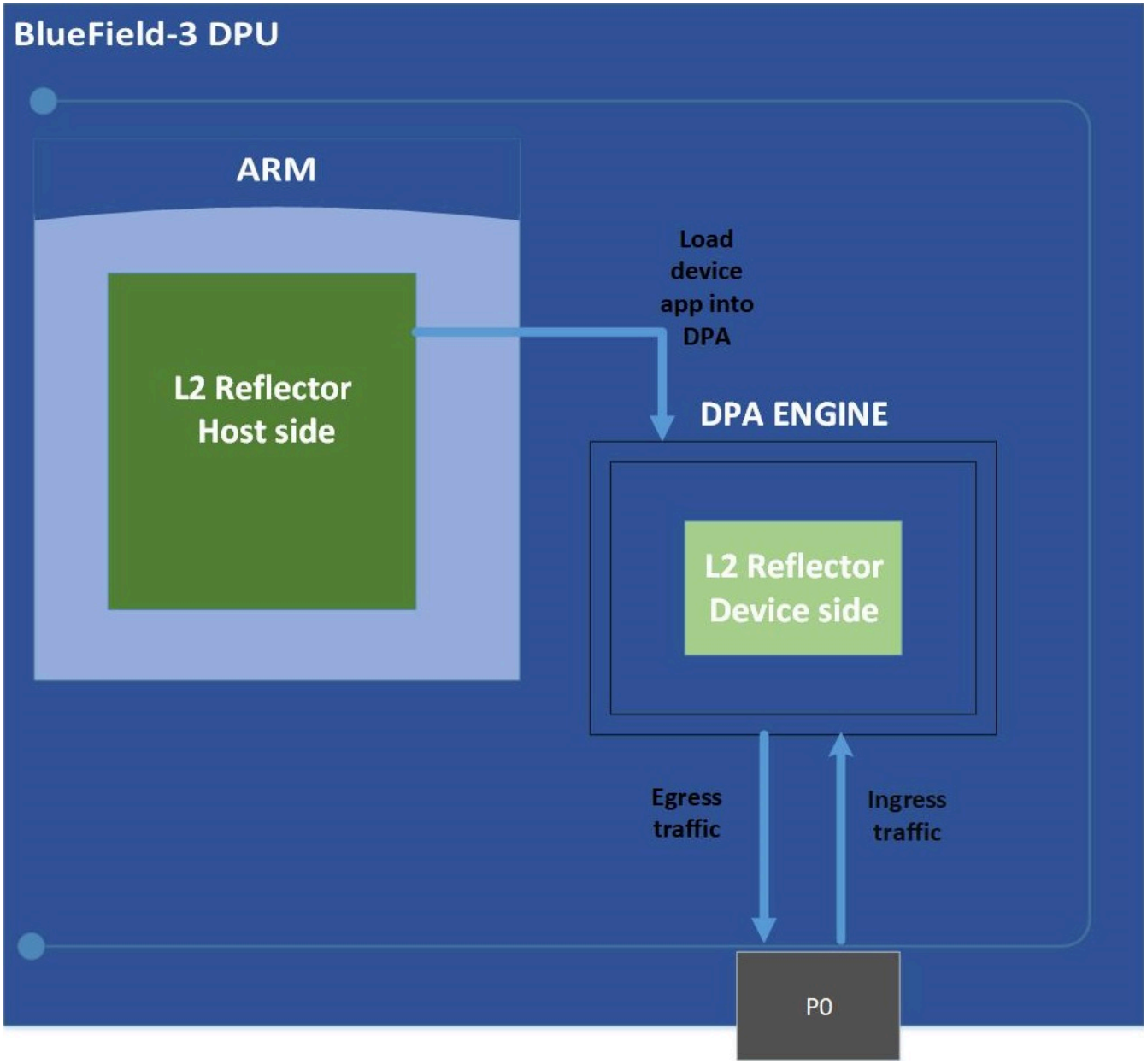
The application accepts traffic from a specific port given as an argument and leverages DPA capabilities for accelerated processing.

The following figure provides a high-level view of the components of the application:



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".

DOCA Libraries and Drivers

This application leverages the following DOCA driver:

- FlexIO SDK

Refer to its programming guide for more information.

Dependencies

NVIDIA® BlueField®-3 DPU and above is required.

Running the Application

Installation

Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

Application Execution

The L2 reflector application is provided in both source and binary forms. The binary is located under `/opt/mellanox/doca/applications/l2_reflector/bin/l2_reflector`.

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=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 <device name>        Device name
```

Info

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
/opt/mellanox/doca/applications/l2_reflector/bin/l2_refle
-h
```

Info

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on BlueField or host:

```
/opt/mellanox/doca/applications/l2_reflector/bin/l2_reflector -d mlx5_0
```

Note

The used device name (`-d` flag) must match the identifier of the desired IB device.

Info

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:

```
cd /opt/mellanox/doca/applications/l2_reflector/bin
./l2_reflector --json ./l2_reflector_params.json
```

> **Note**
>
> Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the desired PCIe addresses required for the deployment.

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	`d`	`device`	Device name	`"device" : mlx5_0`

Info

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

Troubleshooting

Note

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_reflect`.

Please refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

Recompiling the Application

In addition to providing the application in binary form, the installation also includes all of the application sources and compilation instructions so as to allow modifying the sources and recompiling 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 `/opt/mellanox/doca/applications/l2_reflector/src` directory.

Recompiling All Applications

The applications are all defined under a single meson project, so the default compilation recompiles all the DOCA applications.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

Info

`l2_reflector` is created under `/tmp/build/l2_reflector/src/host`.

Recompiling 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
```

Info

`l2_reflector` is created under `/tmp/build/l2_reflector/src/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
```

Info

`l2_reflector` is created under `/tmp/build/l2_reflector/src/host`.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the compilation of the DOCA applications.

Application Code Flow

This section lists the application's configuration flow which includes different FlexIO functions and wrappers.

1. Parse application argument.
    1. Initialize arg parser resources and register DOCA general parameters.

        ```
        doca_argp_init();
        ```

    2. Register the application's parameters.

        ```
        register_l2_reflector_params();
        ```

    3. 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. R un 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();
```

References

- `/opt/mellanox/doca/applications/l2_reflector/src`
- `/opt/mellanox/doca/applications/l2_reflector/bin/l2_reflector_par`

NVIDIA DOCA East-West Overlay Encryption Application

This guide describes IPsec-based strongSwan solution on top of NVIDIA® BlueField® DPU.

> **Note**
>
> 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.

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.

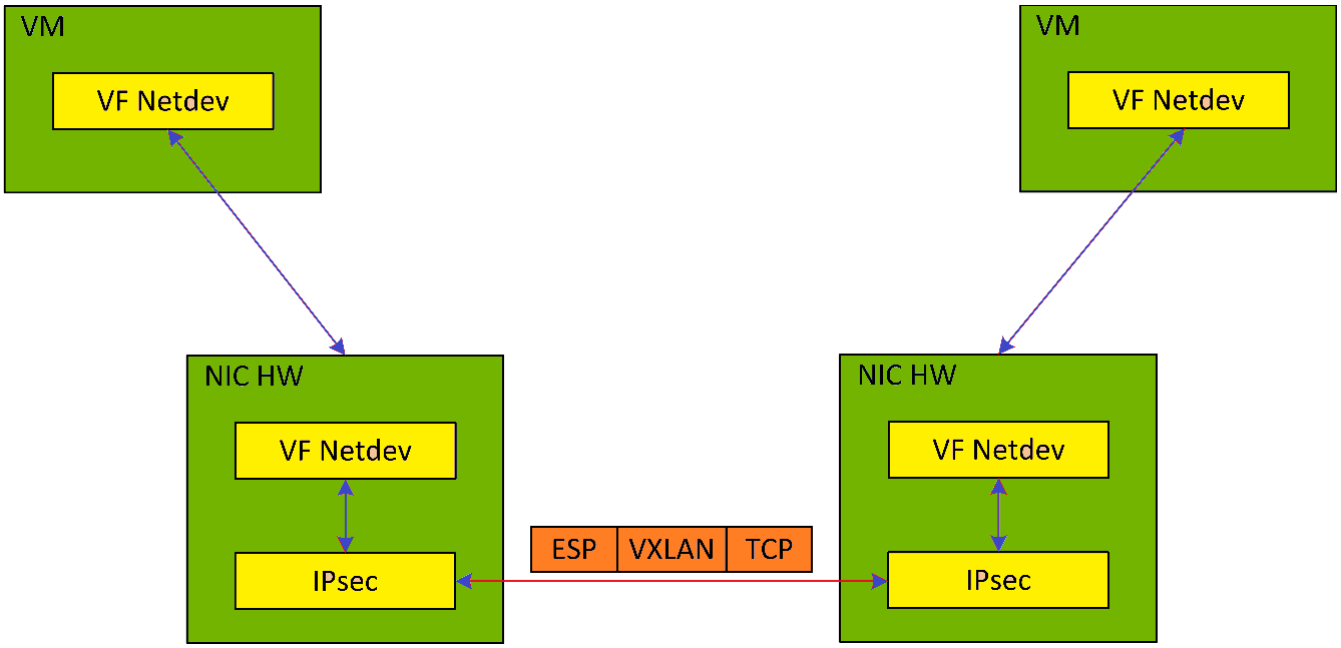
System Design

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.



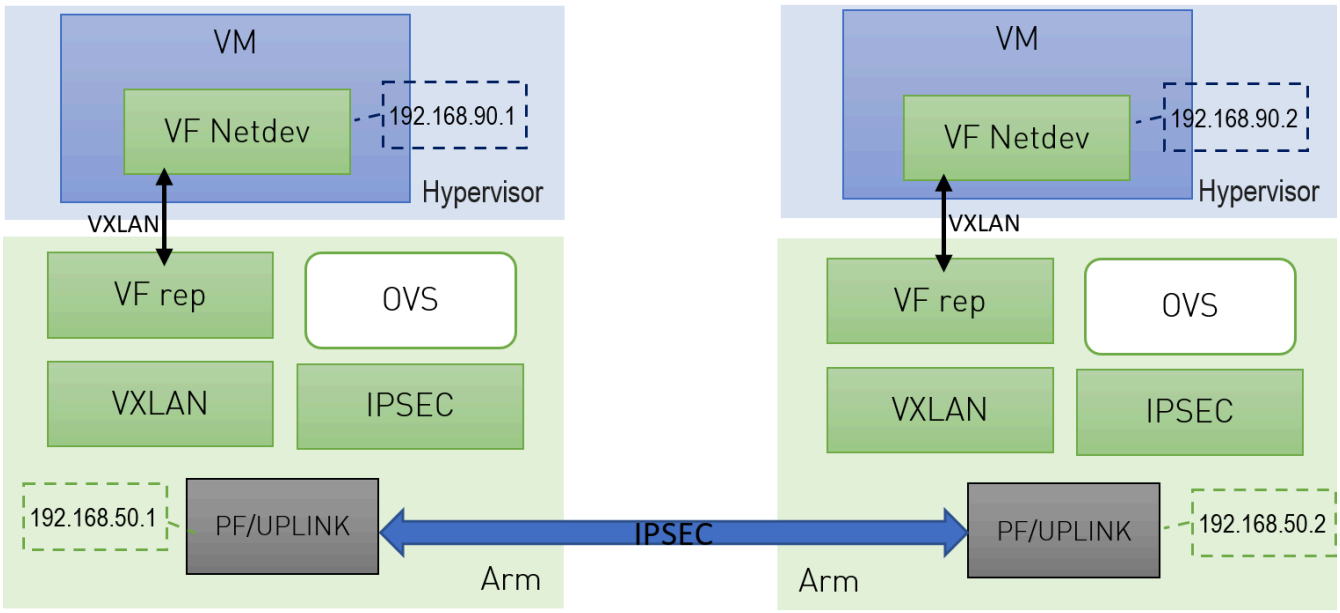
Note

IPsec packet offload is only supported on Ubuntu Bluefield kernel 5.15

Note

OVS offload and IPsec IPv6 do not work together.

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.

DOCA Libraries

N/A

Configuration Flow

The following section provides information on manually configuring IPsec packet offload in general and on using OVS IPsec with strongSwan specifically.

Note

There is a script, `east_west_overlay_encryption.sh`, which is elaborated on in section Running Application 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.

Note

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".

Enabling IPsec Packet Offload

This section explicitly enables IPsec packet offload on the Arm cores before setting up offload-aware IPsec tunnels.

> **Note**
>
> 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` .

   > **Note**
   >
   > 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
   ```

Note

If `mlx-regex` is running:

1. Disable `mlx-regex` prior to running restarting the IB driver:

```
systemctl stop mlx-regex
```

2. Restart IB driver according to the command above.

3. Re-enable `mlx-regex` after the restart has finished:

```
systemctl restart mlx-regex
```

Note

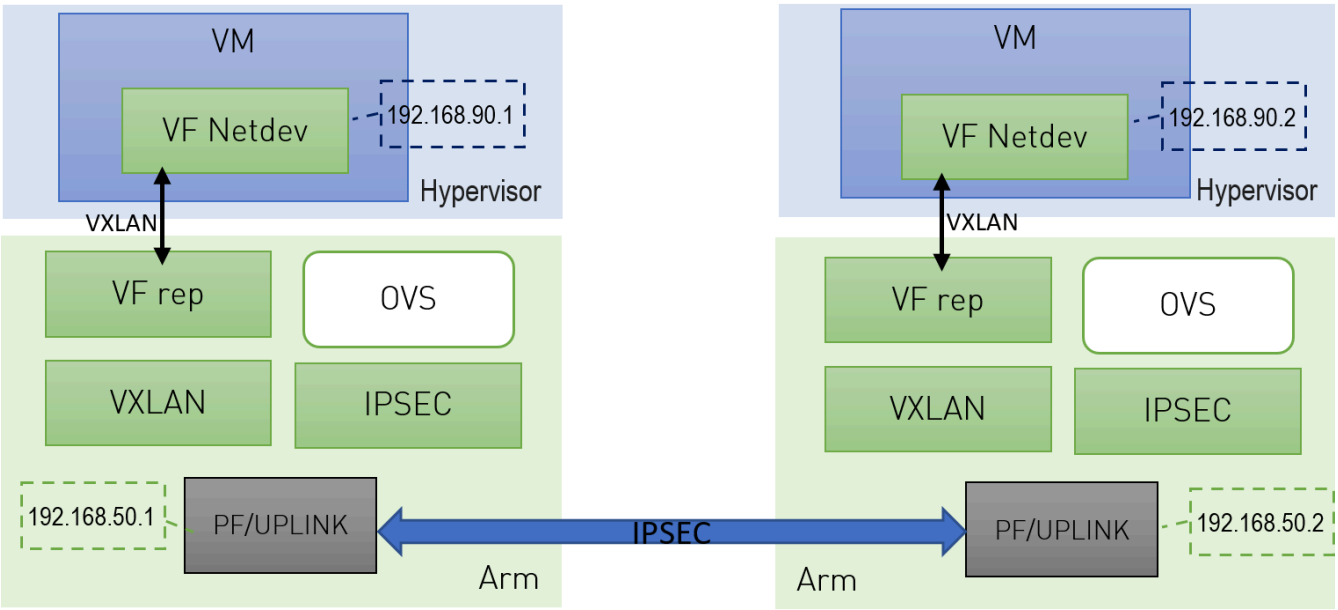
To revert IPsec full offload mode, redo the procedure from step 1, only difference is to set `IPSEC_FULL_OFFLOAD="no"` in `/etc/mellanox/mlnx-bf.conf`.

Configuring OVS IPsec

Note

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
```

Note

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):
    1. On `host_1`:

        ```
        # ifconfig $HOST_PF $host_ip1/24 up
        ```

    2. On `host_2`:

```
# ifconfig $HOST_PF $host_ip2/24 up
```

> **Note**
>
> 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:

    1. On `Arm_1`:

    ```
    # ifconfig $PF $ip1/24 up
    ```

    2. 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
```

> **Note**
>
> Configuring `other_config:hw-offload=true` sets IPsec Packet offload. Setting it to `false` sets software IPsec.

Note

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).

Note

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.

Authentication Methods

The following subsections detail the possible authentication methods for setting up the IPsec tunnel on the OVS bridge.

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:

    1. 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
    ```

    2. 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
    ```

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`:

    1. 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`.

    2. 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:

    1. Copy the certificate of `Arm_1` to `Arm_2`, and the certificate of `Arm_2` to `Arm_1`.

    2. 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.

    3. 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.

1. 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
```

2. 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:

1. 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
```

2. 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
```

Note

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`.

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`.

    1. 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.

2. 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:

    1. Copy the certificate of `Arm_1` to `Arm_2` and the certificate of `Arm_2` to `Arm_1`.

    2. 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.

    3. 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.

4. 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.

    1. On `Arm_1`:

```
# ovs-vsctl set Open_vSwitch . \

other_config:certificate=/etc/strongswan/swanctl/x509/host
\

other_config:private_key=/etc/strongswan/swanctl/private/h
privkey.pem \

other_config:ca_cert=/etc/strongswan/swanctl/x509ca/cacert
```

    2. On `Arm_2`:

```
# ovs-vsctl set Open_vSwitch . \

other_config:certificate=/etc/strongswan/swanctl/x509/host
\

other_config:private_key=/etc/strongswan/swanctl/private/h
privkey.pem \

other_config:ca_cert=/etc/strongswan/swanctl/x509ca/cacert
```

4. Set up the tunnel:

    1. 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
    ```

    2. 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
    ```

Note

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/`.

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.

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.

> **Note**
>
> 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.
    1. 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
```

2. 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
```

> **Note**
>
> 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.

> **Note**
>
> Each machine should have exactly one `swanctl.conf` file in `/etc/swanctl/conf.d/`.

6. Load the `swanctl.conf` files and initialize strongSwan. Run:

    1. On the `Arm_2`, run:

    ```
    systemctl restart strongswan.service
    swanctl --load-all
    ```

    2. On the `Arm_1`, run:

    ```
    systemctl restart strongswan.service
    swanctl --load-all
    swanctl -i --child bf
    ```

Now the IPsec connection should be established.

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.

Note

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

References

-
  - `/opt/mellanox/doca/applications/east_west_overlay_encryption/`
  - `/opt/mellanox/doca/applications/east_west_overlay_encryption/`

NVIDIA DOCA Eth L2 Forwarding Application Guide

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

Introduction

Th e Ethernet L2 Forwarding applicatio n 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.

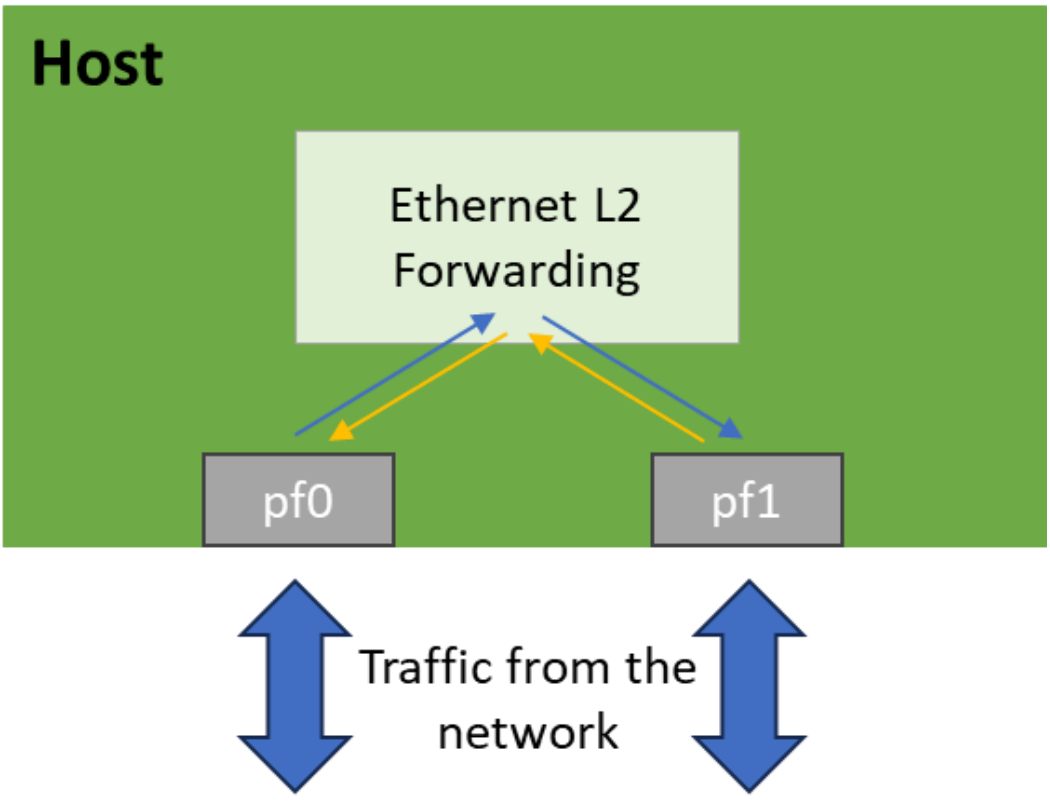
The application can run both on the **Host** and the **BlueField**, and has two main modes:

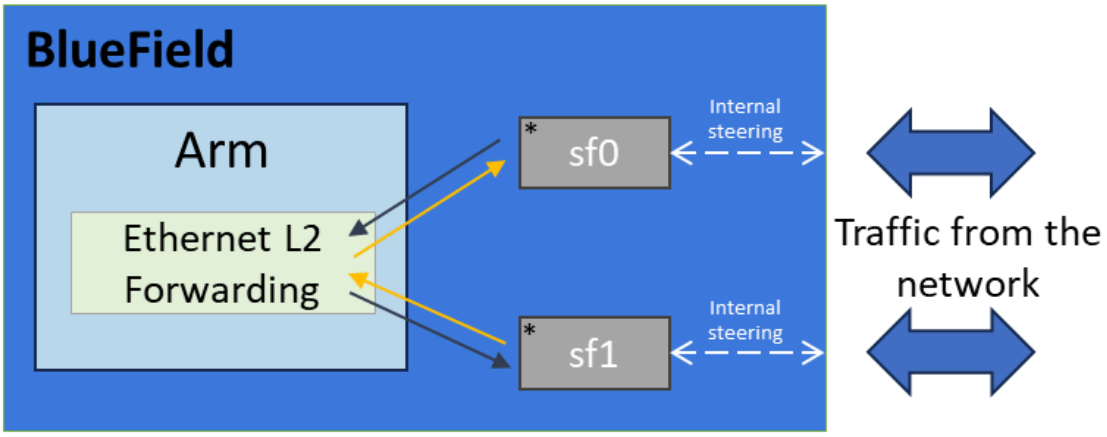
- Two-sided forwarding – device 1 → device 2 **and** device 2 → device 1
- One-sided forwarding – device 1 → device 2 **or** device 2 → device 1

The one-sided mode offers better performance, enlarging the packets forwarding rate.

System Design

The Ethernet L2 Forwarding application runs on the host or the BlueField.

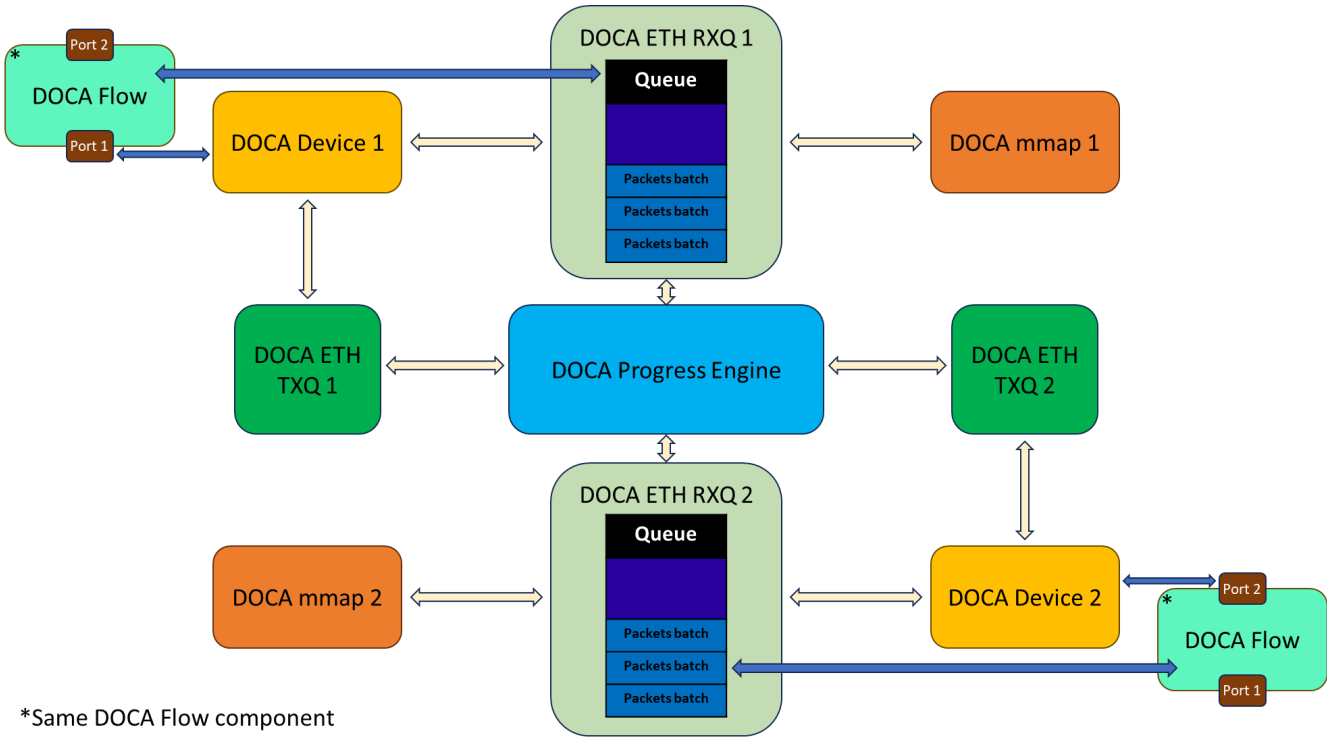




*DOCA Ethernet library only handles SFs

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).

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.

Compiling the Application

Installation

Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

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/`.

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
```

> **Note**
>
> `doca_eth_l2_fwd` will be created under `/tmp/build/eth_l2_fwd/`.

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
```

> **Note**
>
> `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:

    1. 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`

    2. 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
```

> **Note**
>
> `doca_eth_l2_fwd` will be created under `/tmp/build/eth_l2_fwd/`.

Troubleshooting

Please refer to the NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the compilation of the DOCA applications.

Running the Application

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=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, --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.

> **Note**
>
> 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
```

> **Note**
>
> 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
```

Note

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.

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

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
Program flags	`d`	`devs-names`	**Two** IB devices names, separated by a comma, without spaces. **Note** 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
	`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

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
	`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.

Troubleshooting

Please refer to the NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation or execution of the DOCA applications.

Application Code Flow

1. Parse application argument.
    1. Initialize Arg parser resources and register DOCA general parameters.

        ```
        doca_argp_init();
        ```

    2. Register Ethernet L2 Forwarding application parameters.

```
register_eth_l2_forwarding_params();
```

3. Parse the arguments.

```
doca_argp_start();
```

1. Parse DOCA flags.
2. Parse application parameters.

2. Execute Ethernet L2 Forwarding application main logic.

```
eth_l2_fwd_execute();
```

1. Open the two chosen DOCA devices.
2. Initialize necessary DOCA Core objects.
3. Initialize ETH RXQ/TXQ contexts for the devices.
4. Forward packets.

3. Clean up application resources.

```
eth_l2_fwd_cleanup();
```

1. Stop all contexts and drain tasks.
2. Free all application resources.

4. Arg parser destroy.

```
doca_argp_destroy()
```

References

- `/opt/mellanox/doca/applications/eth_l2_fwd/`
- `/opt/mellanox/doca/applications/eth_l2_fwd/eth_l2_fwd_params.json`

NVIDIA DOCA File Compression Application Guide

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

Introduction

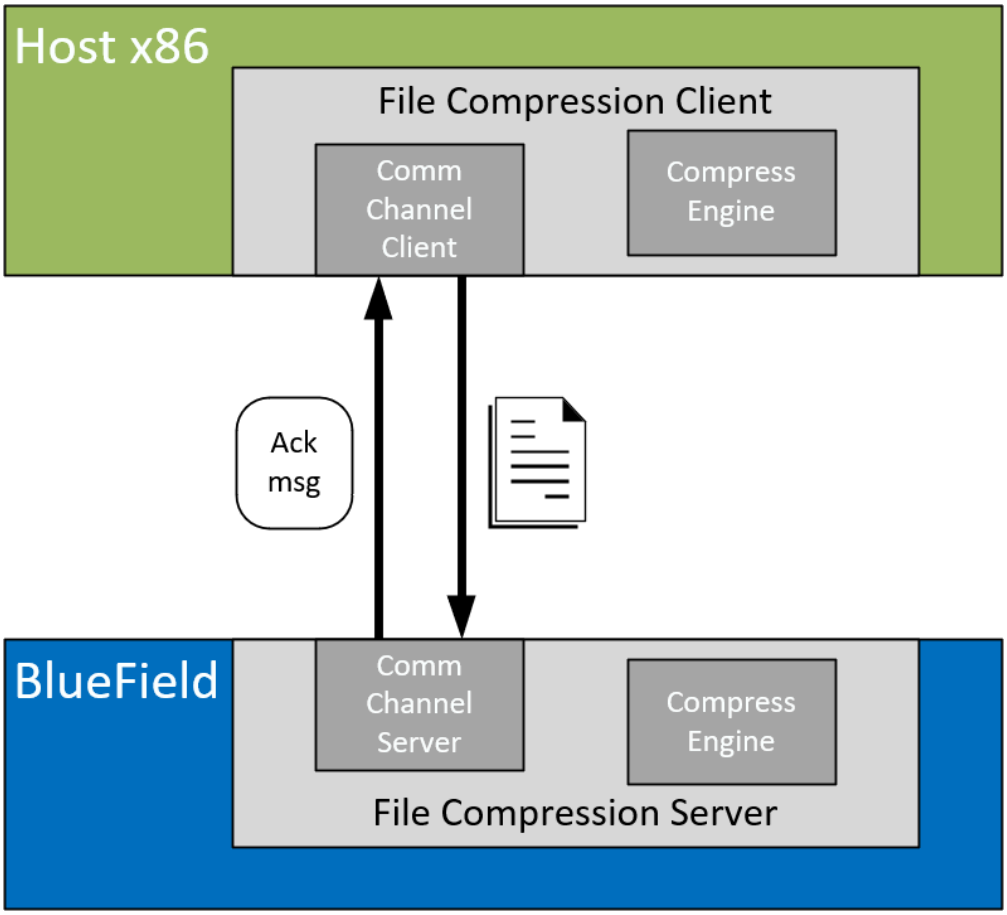
The file compression application exhibits how to use the DOCA Compress API to compress and decompress data using hardware acceleration as well as sending and receiving it using the DOCA Comm Channel API.

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

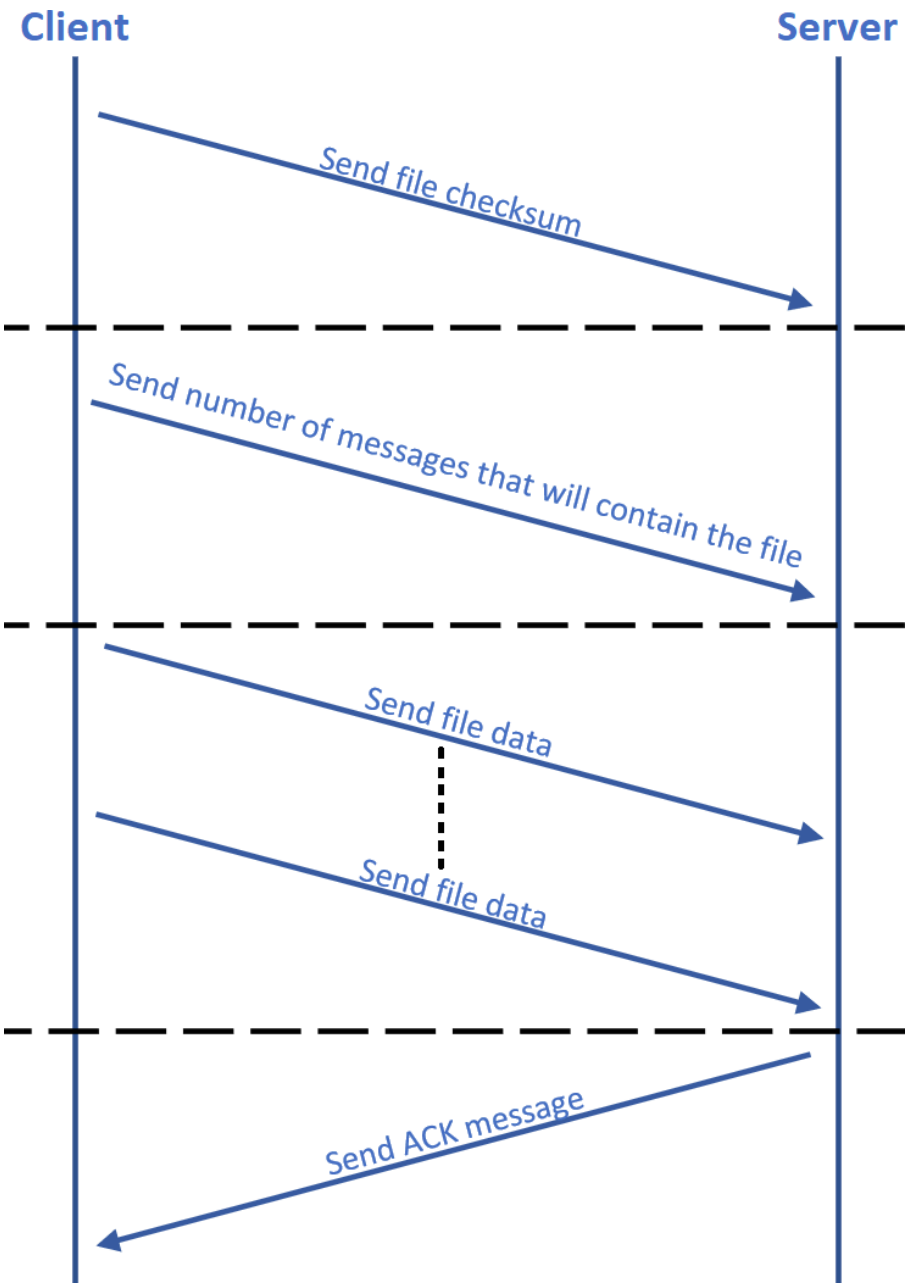
System Design

The file compression application client runs on the host and the server runs on the DPU.



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.

DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Compress
- DOCA Comm Channel

Refer to their respective programming guide for more information.

Running the Application

Installation

Refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

Application Execution

The file compression application is provided in both source and binary forms. The binary is located under `/opt/mellanox/doca/applications/file_compression/bin/doca_file_compre`.

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=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
  -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
```

Info

This usage printout can be printed to the command line using th e `-h` (or `--help`) options:

```
/opt/mellanox/doca/applications/file_compression/bin/doca
-h
```

Info

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on BlueField:

```
/opt/mellanox/doca/applications/file_compression/bin/doca_file
-p 03:00.0 -r 3b:00.0 -f received.txt
```

Note

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:

```
/opt/mellanox/doca/applications/file_compression/bin/doca_file
-p 3b:00.0 -f send.txt
```

> **Note**
>
> 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:

```
cd /opt/mellanox/doca/applications/file_compression/bin
./doca_file_compression --json ./file_compression_params.json
```

> **Note**
>
> Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the PCIe addresses necessary for the deployment.

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	`f`	`file`	For client – path to the file to be sent For server – path to write the file into **Note** This is a mandatory flag.	"file" : "/tmp/data.txt"
	`p`	`pci-addr`	Comm Channel DOCA device PCIe address **Note** This is a mandatory flag.	"pci-addr" : 03 : 00.1
	`r`	`rep-pci`	Comm Channel DOCA device representor PCIe address **Note** This flag is mandatory only on the DPU.	"rep-pci" : b1 : 00.1

Info

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

Recompiling the Application

In addition to providing the application in binary form, the installation also includes all of the application sources and compilation instructions so as to allow modifying the sources and recompiling 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 `/opt/mellanox/doca/applications/file_compression/src` directory.

Recompiling All Applications

The applications are all defined under a single meson project, so the default compilation recompiles all the DOCA applications.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

Info

`doca_file_compression` is created under `/tmp/build/file_compression/src/`.

Recompiling File Compression Application Only

To directly build only the f ile compression application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -
Denable_file_compression=true
ninja -C /tmp/build
```

Info

`doca_file_compression` is created under `/tmp/build/file_compression/src/`.

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
```

Info

`doca_file_compression` is created under `/tmp/build/file_compression/src/`.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application.

Application Code Flow

1. Parse application argument.
    1. Initialize arg parser resources and register DOCA general parameters.

    ```
    doca_argp_init();
    ```

    2. Register file compression application parameters.

```
register_file_compression_params();
```

3. Parse the arguments.

```
doca_argp_start();
```

1. Parse app parameters.

2. Set endpoint attributes.

```
set_endpoint_properties();
```

1. Set maximum message size of 4080 bytes.
2. Set maximum number of messages allowed.

3. Create comm channel endpoint.

```
doca_comm_channel_ep_create();
```

1. 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();
```

1. Free all application resources.

6. Arg parser destroy.

```
doca_argp_destroy()
```

References

- `/opt/mellanox/doca/applications/file_compression/src`
- `/opt/mellanox/doca/applications/file_compression/bin/file_compres`

NVIDIA DOCA File Integrity Application Guide

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

Introduction

The file integrity application exhibits how to use the DOCA Comm Channel and DOCA SHA libraries to send and receive a file securely.

The application's logic includes both a client and a server:

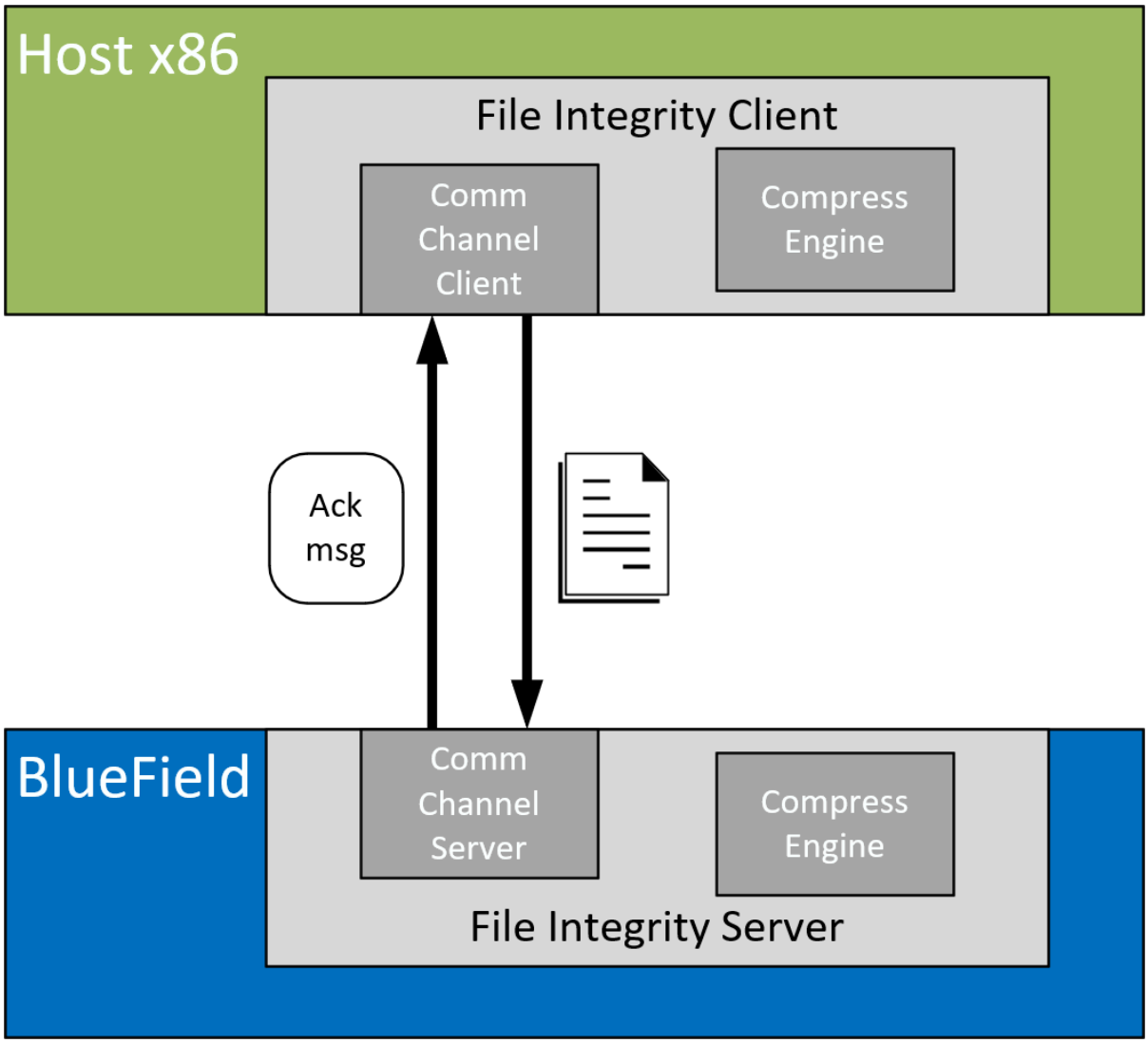
- Client side – the application opens a file, calculates the SHA (secure hash algorithm) digest on it, and sends the digest of the source file alongside the file itself to the server
- Server side – the application calculates the SHA on the received file and compares the received digest to the calculated one to check if the file has been compromised

Note

SHA hardware acceleration is only available on the BlueField-2 DPU.
This application is not supported on BlueField-3.

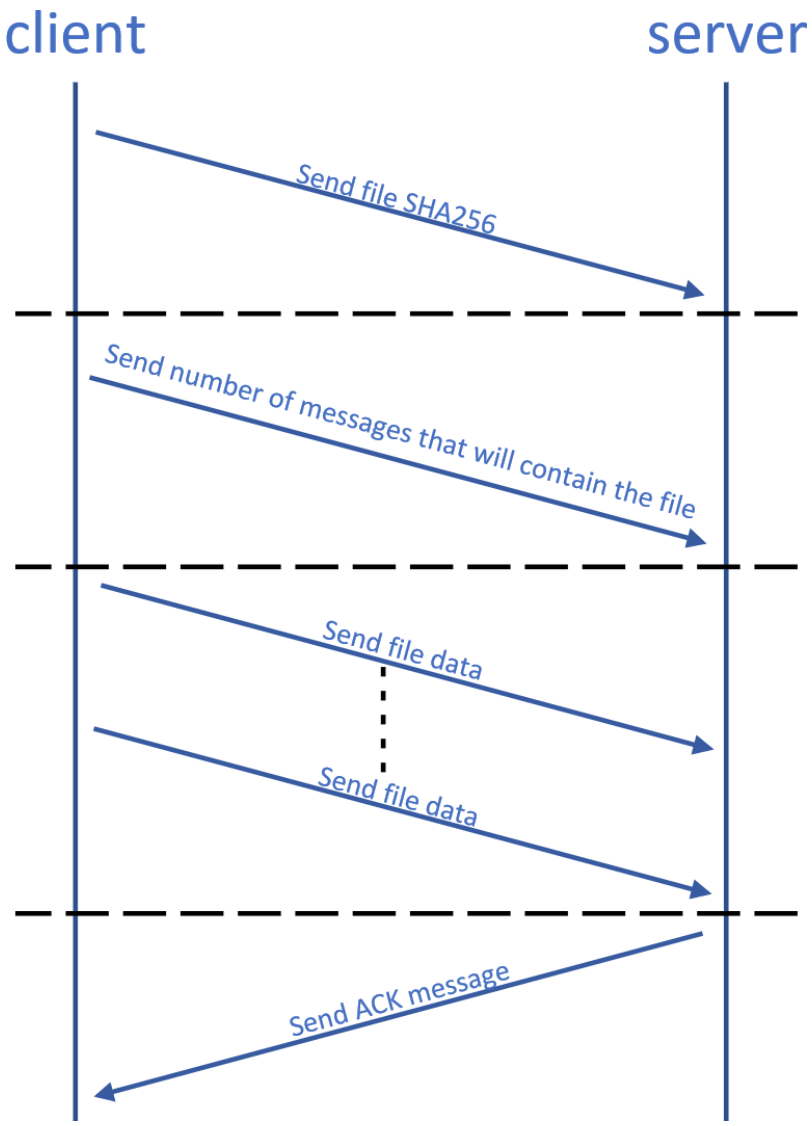
System Design

The file integrity application runs in client mode (host) and server mode (DPU).



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.

DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA SHA
- DOCA Comm Channel

Refer to their respective programming guide for more information.

Running the Application

Installation

Refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

Application Execution

The file integrity application is provided in both source and binary forms, and the binary is located under `/opt/mellanox/doca/applications/file_integrity/bin/doca_file_integrit`.

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=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
  -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
```

Info

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
/opt/mellanox/doca/applications/file_integrity/bin/doca_f
-h
```

Info

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on BlueField:

```
/opt/mellanox/doca/applications/file_integrity/bin/doca_file_i
-p 03:00.0 -r 3b:00.0 -f received.txt
```

Note

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:

```
/opt/mellanox/doca/applications/file_integrity/bin/doca_file_i
-p 3b:00.0 -f send.txt
```

> **Note**
>
> 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:

```
cd /opt/mellanox/doca/applications/file_integrity/bin
./doca_file_integrity --json ./file_integrity_params.json
```

> **Note**
>
> Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the PCIe addresses necessary for the deployment .

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	`f`	`file`	For client – path to the file to be sent For server – path to write the file into **Note** This is a mandatory flag.	"file" : "/tmp/data.txt"
	`p`	`pci-addr`	Comm Channel DOCA device PCIe address **Note** This is a mandatory flag.	"pci-addr" : 03 : 00.1
	`r`	`rep-pci`	Comm Channel DOCA device representor PCIe address **Note** This flag is mandatory only on the DPU.	"rep-pci" : b1 : 00.1

Info

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

Troubleshooting

Please refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications .

Recompiling the Application

In addition to providing the application in binary form, the installation also includes all of the application sources and compilation instructions so as to allow modifying the sources and recompiling 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 `/opt/mellanox/doca/applications/file_integrity/src` directory.

Recompiling All Applications

The applications are all defined under a single meson project, so the default compilation recompiles all the DOCA applications.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

Info

`doca_file_integrity` is created under `/tmp/build/file_integrity/src/`.

Recompiling File Integrity Application Only

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
```

Info

`doca_file_integrity` is created under `/tmp/build/file_integrity/src/`.

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
```

Info

`doca_file_integrity` is created under `/tmp/build/file_integrity/src/`.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application.

Application Code Flow

1. Parse application argument.
    1. Initialize the arg parser resources and register DOCA general parameters.

    ```
    doca_arg_init();
    ```

    2. Register file integrity application parameters.

```
register_file_integrity_params();
```

3. Parse application parameters.

```
doca_argp_start();
```

2. Set endpoint attributes.

```
set_endpoint_properties();
```

1. Set maximum message size of 4032 bytes.
2. Set number of maximum messages allowed per connection.

3. Create Comm Channel endpoint.

```
doca_comm_channel_ep_create();
```

1. Create endpoint for client/server.

4. Create SHA context.

```
doca_sha_create();
```

1. 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();
```

    1. Free all application resources.

References

- `/opt/mellanox/doca/applications/file_integrity/src`
- `/opt/mellanox/doca/applications/file_integrity/bin/file_integrity`

NVIDIA DOCA Firewall Application Guide

This guide provides an example of firewall implementation on top of NVIDIA® BlueField® DPU .

Introduction

A firewall application is a network security application that leverages the DPU's hardware capability to monitor incoming and outgoing network traffic and allow or block packets based on a set of preconfigured rules.

The firewall application is based on DOCA Flow gRPC library, used for remote programming of the DPU's hardware.

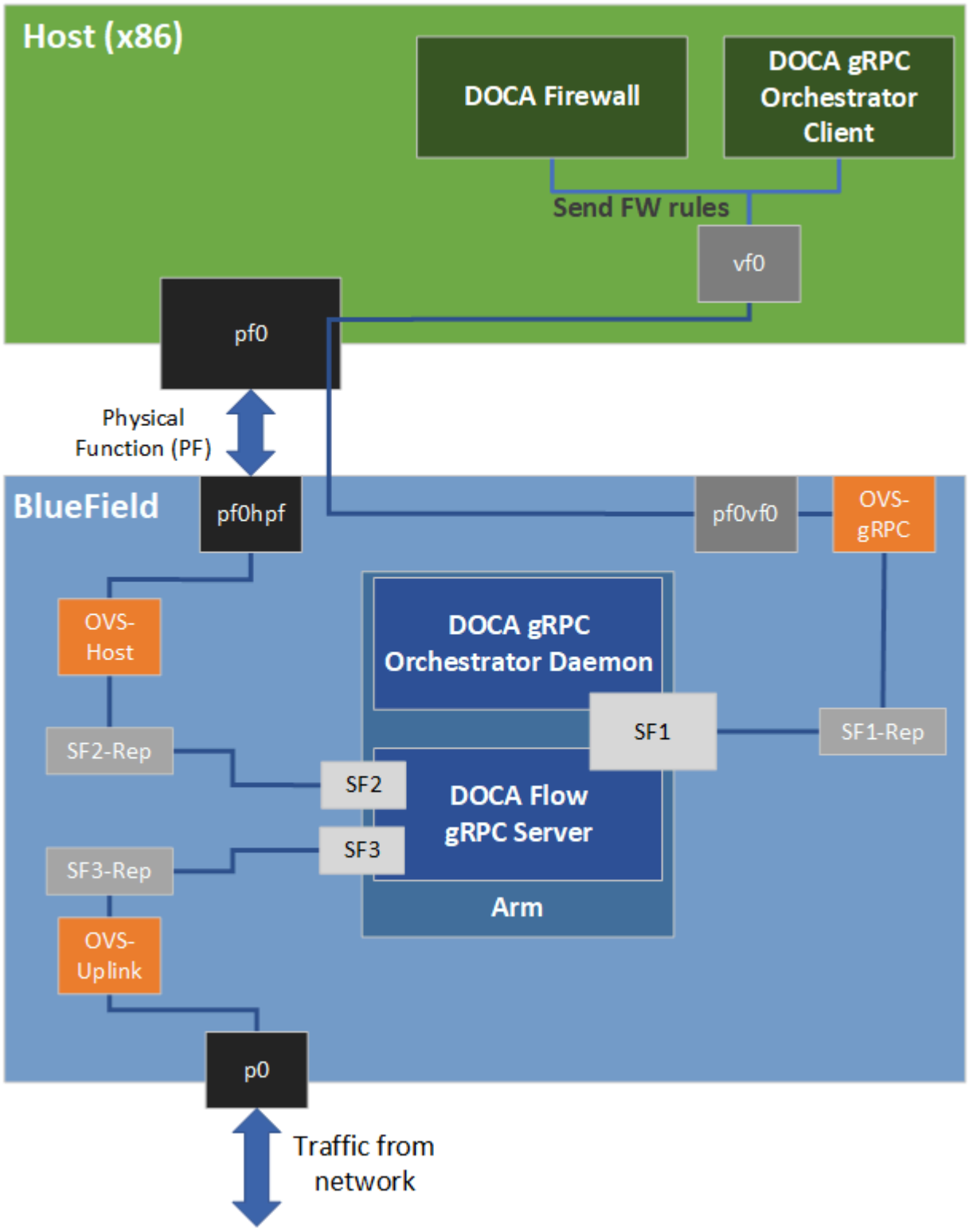
The firewall can operate in two modes:

- Static mode – the firewall application gets 5-tuple traffic from the user with a JSON file for packets to be dropped. The packets that do not match any of the 5-tuple are forwarded by a hairpin pipe.

- Interactive mode – the user can add rules from the command line in real time to execute different firewall rules

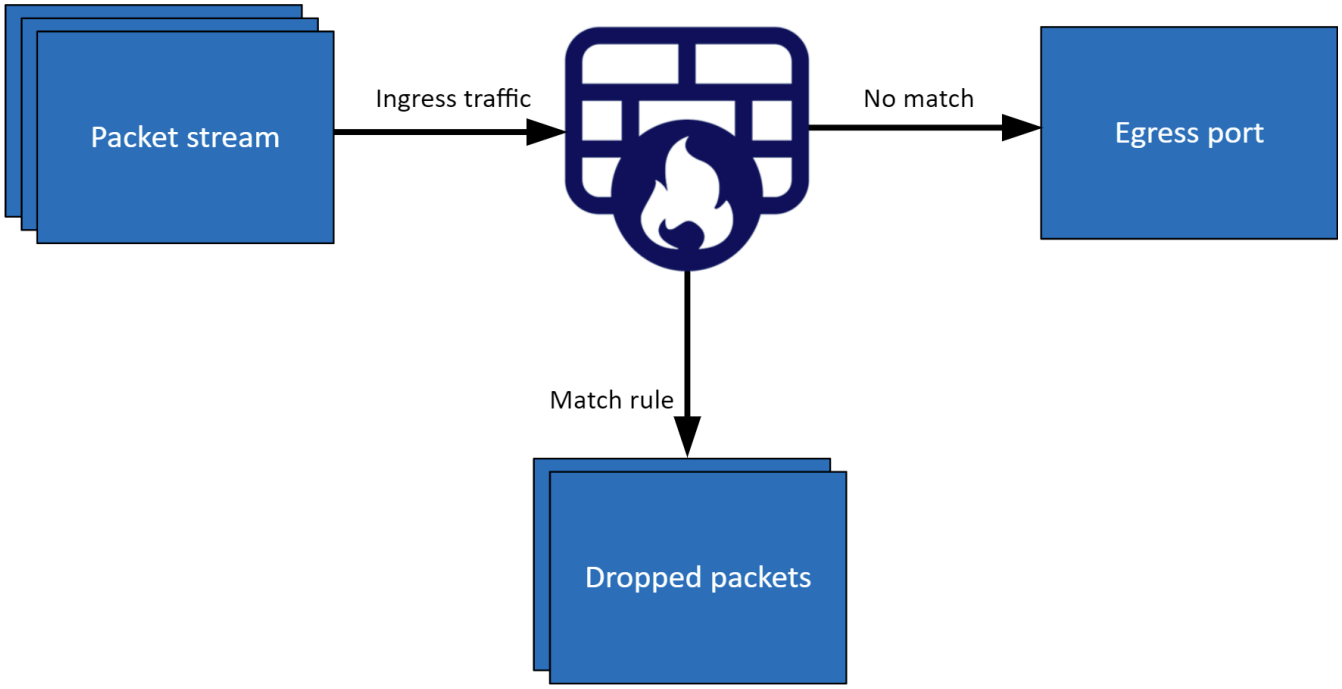
System Design

The firewall application is designed to run on the host and to use DOCA Flow gRPC client to send instructions to a server that runs on the BlueField DPU instance. The DPU intercepts ingress traffic from the wire and either drops it or forwards it to the egress port using a hairpin. The decision is made using traffic classification.

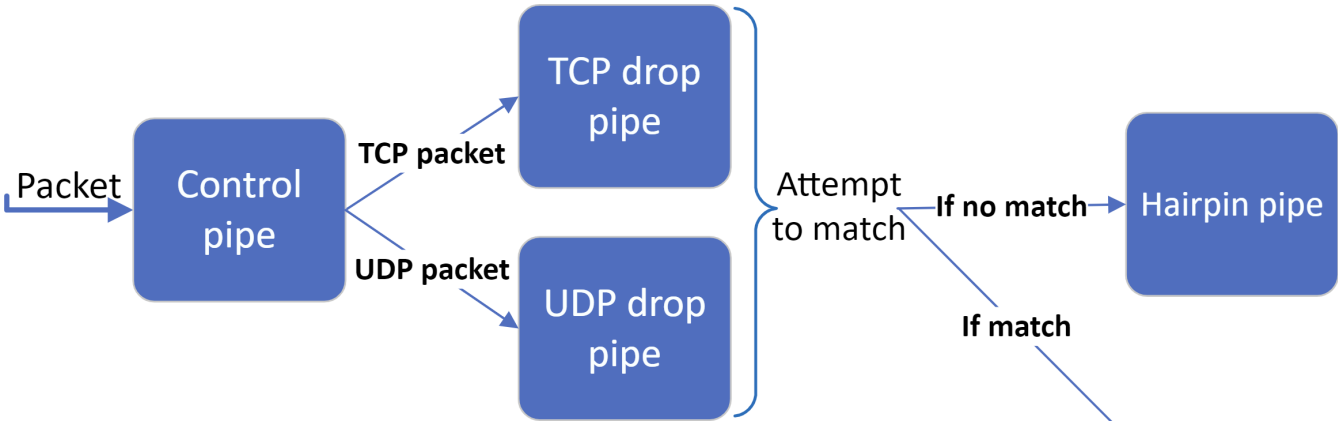


Application Architecture

The firewall runs on top of DOCA Flow gRPC to classify packets.



Static Mode



1. The firewall application builds 4 pipes for each port: One control pipe, two drop pipes, and a hairpin pipe.

2. The drop pipes match only 5-tuple traffic with specific source and destination IPs and source and destination ports.

    - One of the drop pipes matches TCP traffic and the other matches UDP
    - The hairpin pipe matches every packet (no misses)
    - The control pipe serves as a root pipe and has two entries: The first entry forwards the TCP traffic to the TCP drop pipe, and the second entry forwards UDP traffic to the UDP drop pipe

- The hairpin pipe serves as a forwarding miss component to the drop pipes. Therefore, every received packet is checked first against the drop pipes. If there is a match, then it is dropped, otherwise, it is forwarded to the hairpin pipe and is then matched.

Interactive Mode

Running in interactive mode initializes 2 ports, creates the same pipes as in "Static Mode", and the user then adds or removes entries.

- When adding an entry, one must run commands to create a 5-tuple match beforehand
- After an entry is created successfully, the relevant ID is printed for future use

Available commands:

- `add entry port_id=<port_id>`
- `rm entry port_id=<port_id>,entry_id=[entry_id]`
- `port pipes flush port_id=[port_id]`
- `port pipes dump port_id=[port_id],file=[file_name]`
- `create entry_match [field=value,…]`

5-tuple match struct fields:

Fields	Field Options
`outer.src_ip_addr`	
`outer.dst_ip_addr`	
`outer.l4_type_ext`	`tcp`, `udp`
`outer.tcp_src_port`	
`outer.tcp_dst_port`	
`outer.udp_src_port`	

Fields	Field Options
`outer.udp_dst_port`	

The following is an example for creating a pipe and adding an entry:

```
create entry_match
outer.src_ip_addr=192.168.105.2,outer.dst_ip_addr=192.168.105.3,outer.l4_t
add entry port_id=0,pipe_queue=0
```

DOCA Libraries

This application leverages the following DOCA library:

- DOCA Flow

Refer to its respective programming guide for more information.

Running the Application

Installation

Refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

Prerequisites

The DOCA Firewall application makes use of DOCA's gRPC infrastructure, and specifically the DOCA Flow gRPC server on the BlueField. For information about the required configuration steps, refer to NVIDIA DOCA gRPC Infrastructure User Guide.

Application Execution

The firewall application is provided in both source and binary forms, and the binary is located under `/opt/mellanox/doca/applications/firewall/bin/doca_firewall`.

1. Application usage instructions:

```
Usage: doca_firewall [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
  -g, --grpc-address ip_address[:port]    Set the IP address
for the grpc server

Program Flags:
  -m, --mode                                Set running mode
{static, interactive}
  -r, --firewall-rules <path>               Path to the JSON
file with 5-tuple rules when running with static mode
```

Info

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
/opt/mellanox/doca/applications/firewall/bin/doca_firewal
-h
```

Info

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the host:

    1. Interactive mode:

    ```
    /opt/mellanox/doca/applications/firewall/bin/doca_firewall
    --grpc-address 192.168.101.2 -l 50 -m interactive
    ```

    2. Static mode:

    ```
    /opt/mellanox/doca/applications/firewall/bin/doca_firewall
    --grpc-address 192.168.101.2 -l 50 -m static -r
    firewall_rules.json
    ```

3. The application also supports a JSON-based deployment mode, in which all command-line arguments are provided through a JSON file:

```
doca_firewall --json [json_file]
```

For example:

```
cd /opt/mellanox/doca/applications/firewall/bin
./doca_firewall --json ./firewall_params.json
```

> **Note**
>
> Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the PCIe addresses necessary for the deployment .

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
	`g`	`grpc-address`	Set the IP address for the gRPC server	`"grpc-address" : "0.0.0.0"`

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
Program Flags	`m`	`mode`	Set running mode {`static` or `interactive`} **Note** This flag is mandatory.	`"mode" : "interactive"`
	`r`	`firewall-rules`	Path to JSON rules file	`"firewall-rules" : "firewall_rules.json"`

Info

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications .

Recompiling the Application

In addition to providing the application in binary form, the installation also includes all of the application sources and compilation instructions so as to allow modifying the sources and recompiling 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 `/opt/mellanox/doca/applications/firewall/src` directory.

Recompiling All Applications

The applications are all defined under a single meson project, meaning that the default compilation will recompile all the DOCA applications.

To build all the applications together, including the gRPC-related applications, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_grpc_support=true
ninja -C /tmp/build
```

Info

`doca_firewall` is created under `/tmp/build/firewall/src/`.

Recompiling Firewall Application Only

To directly build only the firewall application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_grpc_support=true -
Denable_all_applications=false -Denable_firewall=true
ninja -C /tmp/build
```

Info

`doca_firewall` is created under `/tmp/build/firewall/src/`.

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_grpc_support` to `true`
   - Set `enable_firewall` to `true`
2. Run the following compilation commands :

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

Info

`doca_firewall` is created under `/tmp/build/firewall/src/`.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application .

Application Code Flow

1. Parse application argument.
    1. Initialize the arg parser resources and register DOCA general parameters.

```
doca_argp_init();
```

    2. Register firewall application parameters.

```
register_firewall_params();
```

    3. Parse application parameters.

```
doca_argp_start();
```

2. Firewall initialization.

```
firewall_ports_init();
```

1. Create a new gRPC channel and initialize a stub.
2. Initialize DOCA Flow and DOCA Flow ports.

3. Create control, TCP, UDP, and hairpin pipes for both ports.

```
firewall_pipes_init();
```

4. Configure firewall rules.

Mode	Procedure

Static	1. Initialize drop packets array from the input JSON file:
	`init_drop_packets();`
	2. Add firewall drop rules parsed from JSON file:
	`firewall_add_drop_rules();`

Interactive	1. Initialize the firewall's interactive command line:
	`interactive_cmdline();`
	2. Free allocated resources.
	`interactive_mode_cleanup();`

5. Firewall clean-up.

```
firewall_ports_destroy();
```

1. Destroy all DOCA Flow resources.

6. Arg parser destroy.

```
doca_argp_destroy();
```

References

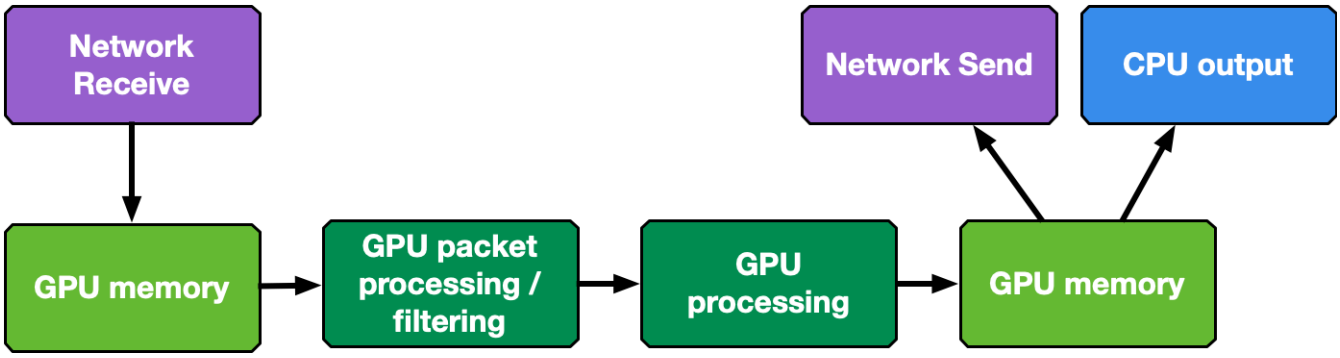
- `/opt/mellanox/doca/applications/firewall/src`
- `/opt/mellanox/doca/applications/firewall/bin/firewall_params.json`
- `/opt/mellanox/doca/applications/firewall/bin/firewall_rules.json`

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.

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.

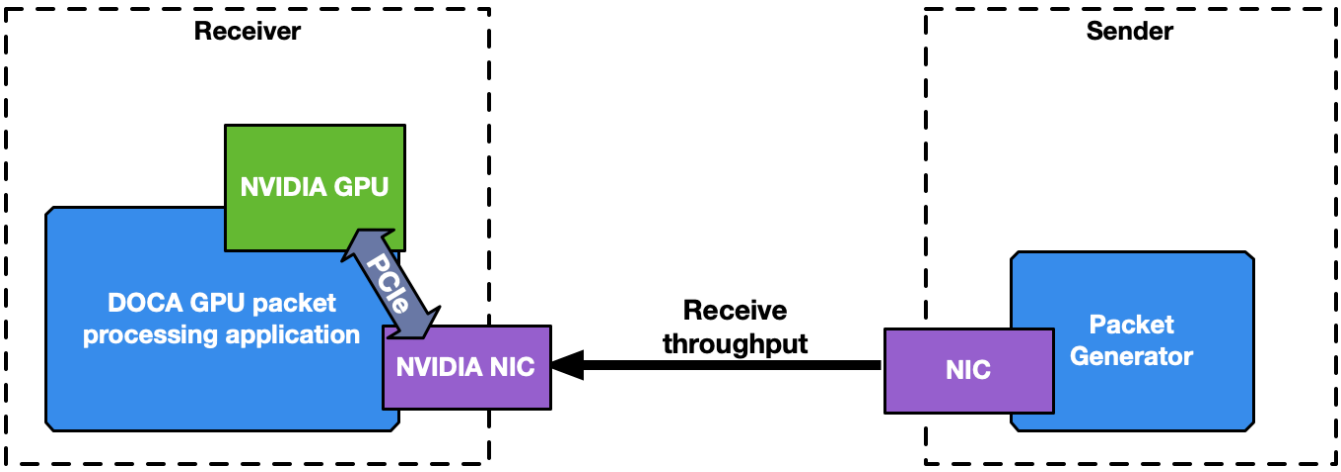


The type of data processing heavily depends on the use case. The goal of this application is to provide a basic layout to reuse in the most common use cases of being able to receive, differentiate and manage the following types of network traffic in multiple queues: UDP, TCP and ICMP.

This application is an enhancement of the use cases presented in this NVIDIA blog post about DOCA GPUNetIO.

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.

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.

Tip

It is highly recommended to use more than one receive queue for 100Gb/s or higher network traffic throughput.

ICMP Network Traffic

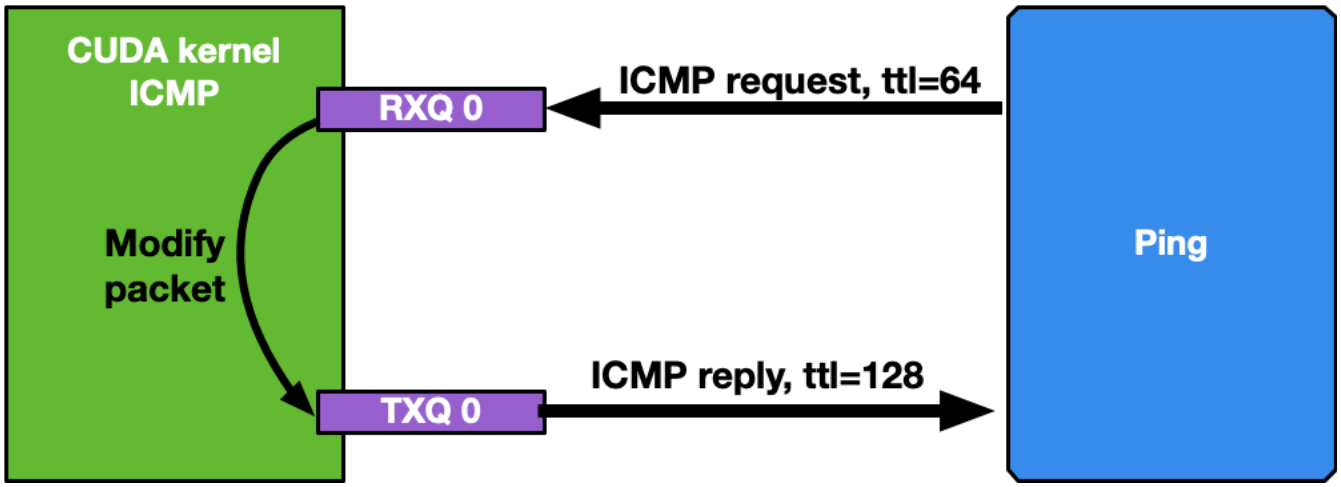
If the network interface used for the application has an IP address, it is possible to ping that interface. ICMP packets are received by a dedicated CUDA kernel (file `gpu_kernels/receive_icmp.cu`) which:

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`.

Info

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......
```

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:

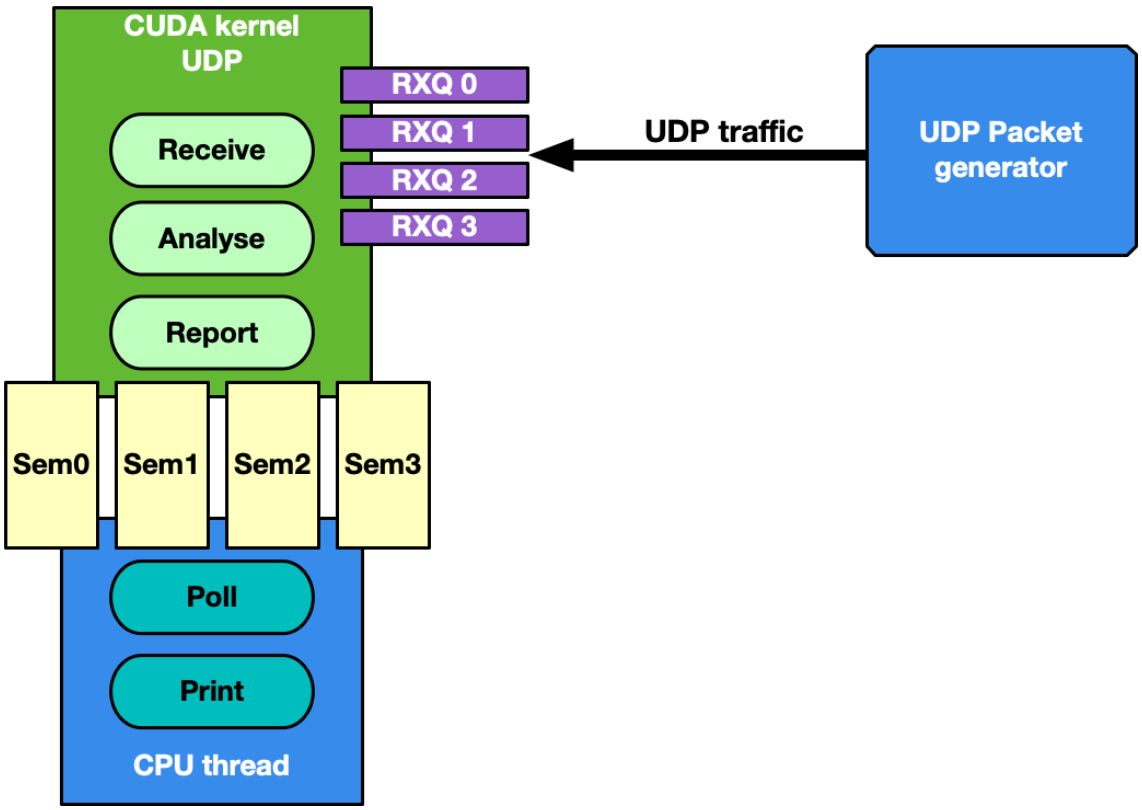
1. Receive packets using the DOCA GPUNetIO CUDA block-level function `doca_gpu_dev_eth_rxq_receive_block` .
2. Each CUDA thread works on a subset of received packets.
3. DOCA buffer containing the packet is retrieved.

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.

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.

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).

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`.

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.

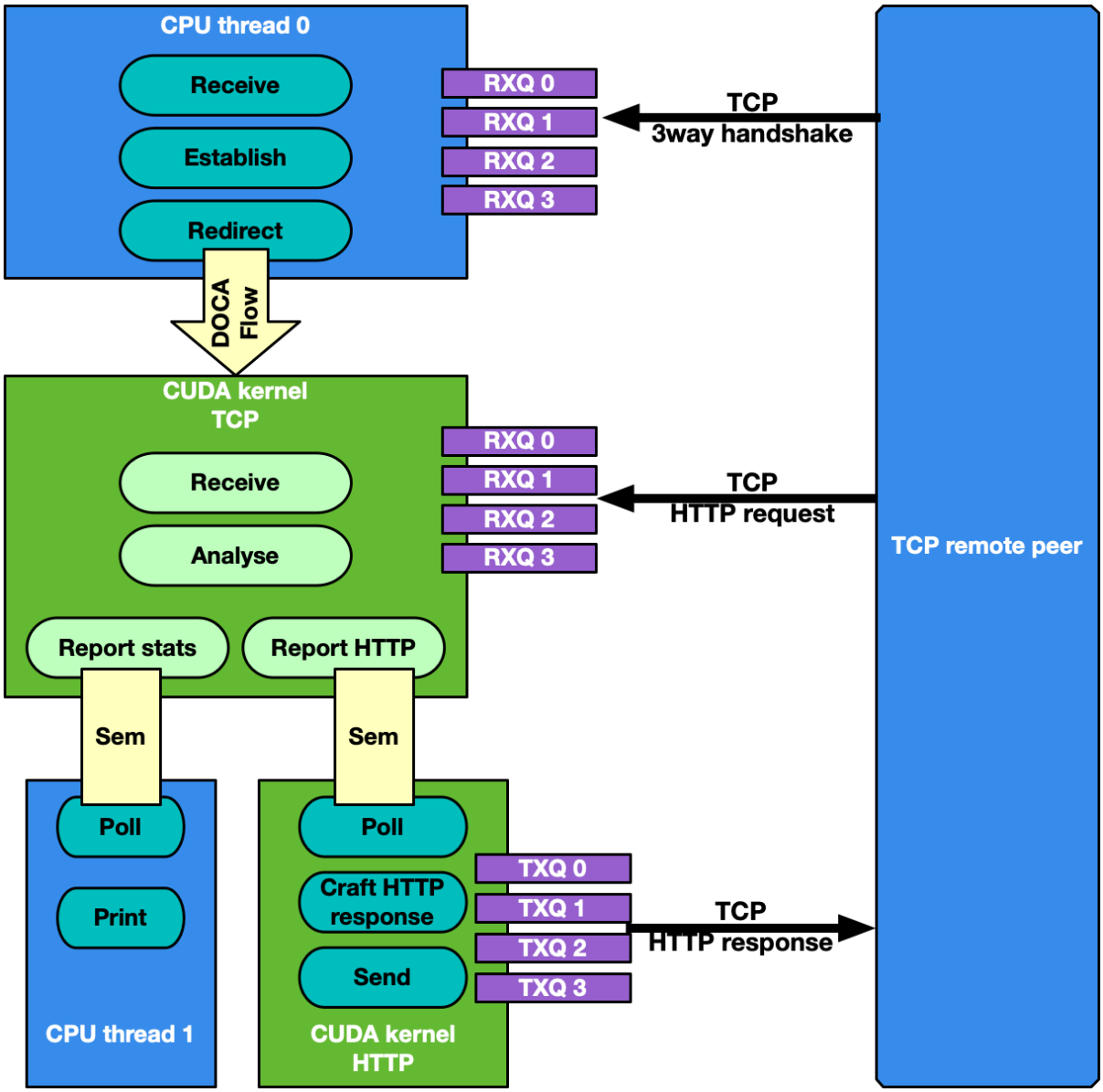
If the request is about `index.html` or `contacts.html`, the CUDA kernel replies with the appropriate HTML page using a `200 OK` code. For all other requests, the it returns a "Page not found" and `404 Error` code.

HTTP response packets are sent by this second HTTP CUDA kernel using DOCA GPUNetIO.

> **Note**
>
> Care must be taken to maintain TCP sequence/ack numbers in the packet headers.

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
```

DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA GPUNetIO
- DOCA Ethernet
- DOCA Flow

Refer to their respective programming guide for more information.

Dependencies

The following dependencies must be installed and configured:

- CUDA Toolkit - version 12.1 or newer
- DOCA's GPU packages (requires CUDA and not installed by default)
    - For Ubuntu/Debian:

```
$ sudo apt install -y doca-gpu doca-gpu-dev
```

    - For CentOS/RedHat:

```
$ sudo yum install -y doca-gpu doca-gpu-devel
```

- `gdrcopy`
- `gdrdrv` kernel module – active and running on the system
- `nvidia-peermem` kernel module – active and running on the system

Running the Application

Installation

Refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

Prerequisites

- NIC correctly configured for direct PCIe connection with the GPU
- Environment variable necessary for `gdrcopy`:

```
$ export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/opt/mellanox/gdrcopy/src
```

- Allocate 4 hugepages of 1GB (DOCA Flow and DPDK requirement):

```
$ sudo vim /etc/default/grub

# Add hugepages size and number along with other options you have in the CMDLINE
# GRUB_CMDLINE_LINUX_DEFAULT="default_hugepagesz=1G hugepagesz=1G hugepages=4"
$ sudo update-grub
$ sudo reboot

# After rebooting, check huge pages info
$ grep -i huge /proc/meminfo
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
HugePages_Total:      16
HugePages_Free:       15
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:    1048576 kB
Hugetlb:        16777216 kB
```

Application Execution

The GPU packet processing application is provided in both source and binary forms. The binary is located under `/opt/mellanox/doca/applications/gpu_packet_processing/bin/doca_gpu_pa`.

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=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:
  -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
```

Info

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
/opt/mellanox/doca/applications/gpu_packet_processing/bir
-h
```

Info

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the host:

    1. 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
```

Note

Refer to section "Running DOCA Application on Host" in the NVIDIA DOCA Virtual Functions User Guide.

Command Line Flags

Flag Type	Short Flag	Long Flag	Description
General flags	`h`	`help`	Prints a help synopsis
	`v`	`version`	Prints 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`	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

Flag Type	Short Flag	Long Flag	Description
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.

Info

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications .

Recompiling the Application

In addition to providing the application in binary form, the installation also includes all of the application sources and compilation instructions so as to allow modifying the sources

and recompiling 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 `/opt/mellanox/doca/applications/gpu_packet_processing/src` directory.

Recompiling All Applications

The applications are all defined under a single meson project, so the default compilation recompiles all the DOCA applications.

To build all the applications together, including the GPU-enabled applications, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_gpu_support=true
ninja -C /tmp/build
```

Info

`doca_gpu_packet_processing` is created under `/tmp/build/gpu_packet_processing/src/`.

Recompiling GPU Packet Processing Application Only

To directly build only the GPU packet processing application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_gpu_support=true -
Denable_all_applications=false -Denable_gpu_packet_processing=true
ninja -C /tmp/build
```

Info

`doca_gpu_packet_processing` is created under `/tmp/build/gpu_packet_processing/src/`.

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_support` to `true`
   - 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
```

Info

`doca_gpu_packet_processing` is created under `/tmp/build/gpu_packet_processing/src/`.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application .

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();
```

References

- `/opt/mellanox/doca/applications/gpu_packet_processing/src`

NVIDIA DOCA IPsec Security Gateway Application Guide

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

Note

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.

Introduction

Note

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 and DOCA IPsec libraries.

The application demonstrates how to insert rules to create an IPsec tunnel.

Note

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:

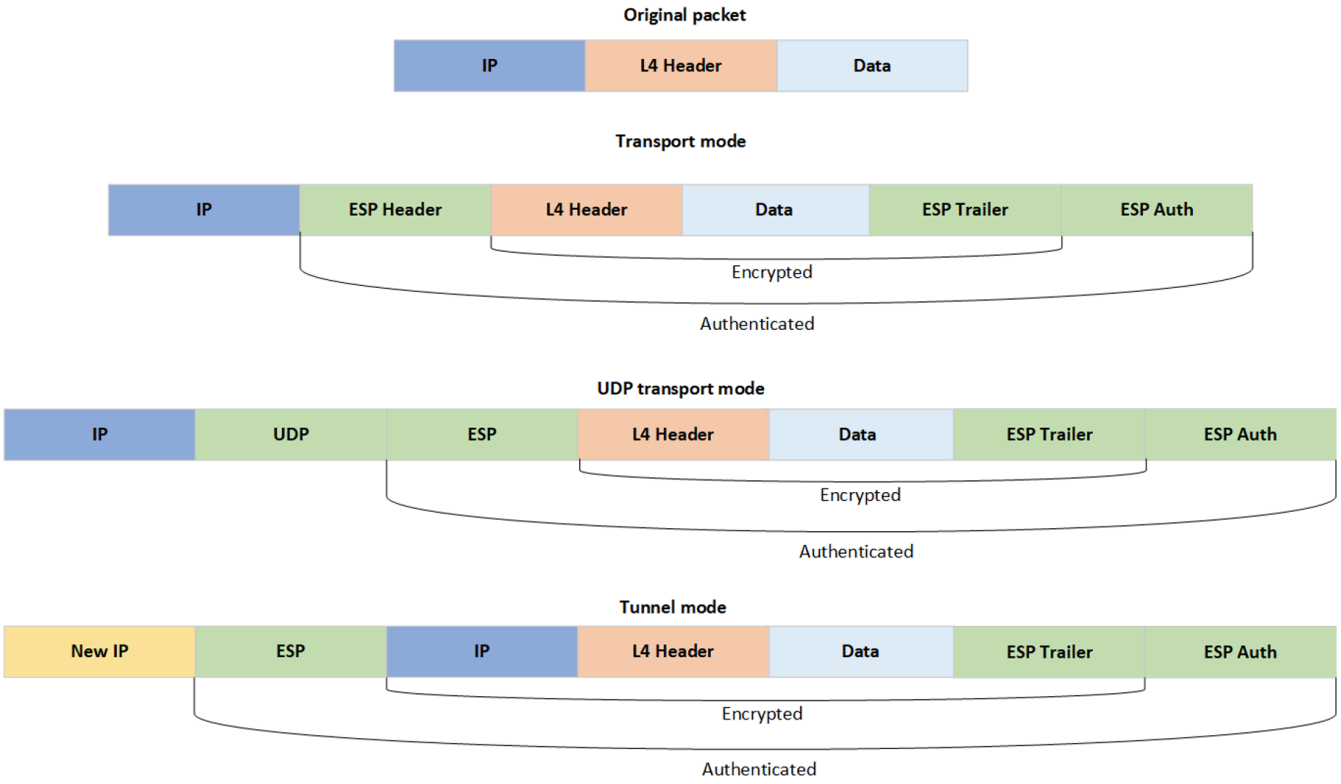
- Static configuration – (default) receives a fixed list of rules for IPsec encryption and decryption

> **Note**
>
> 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

> **Note**
>
> 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.

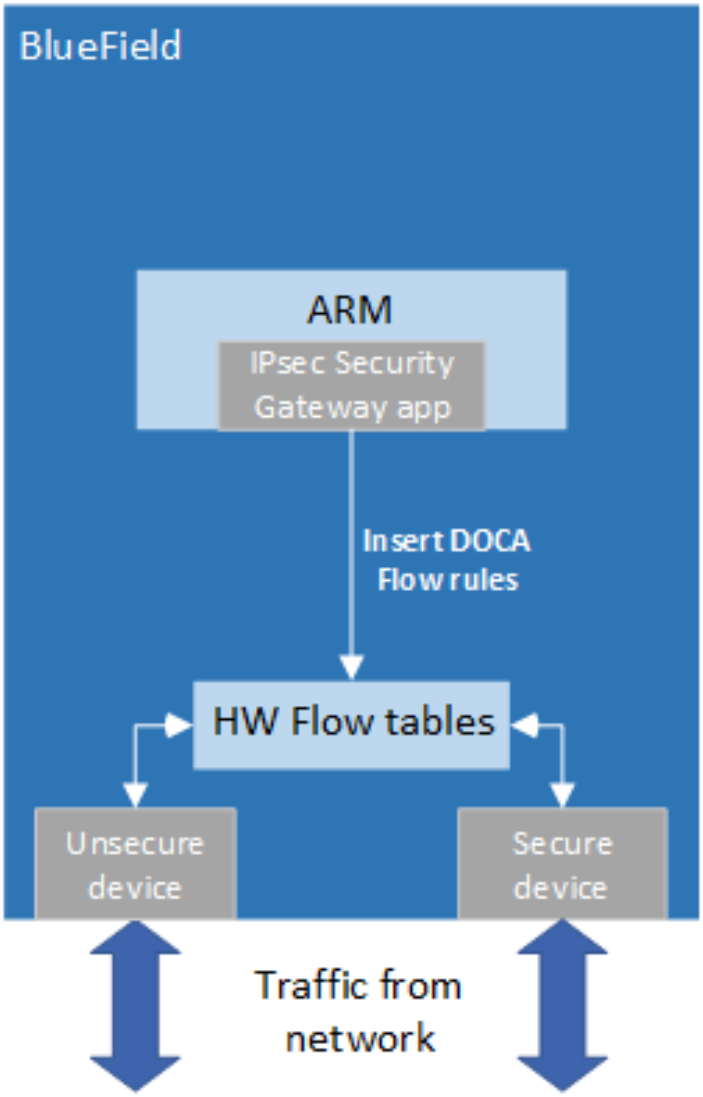


System Design

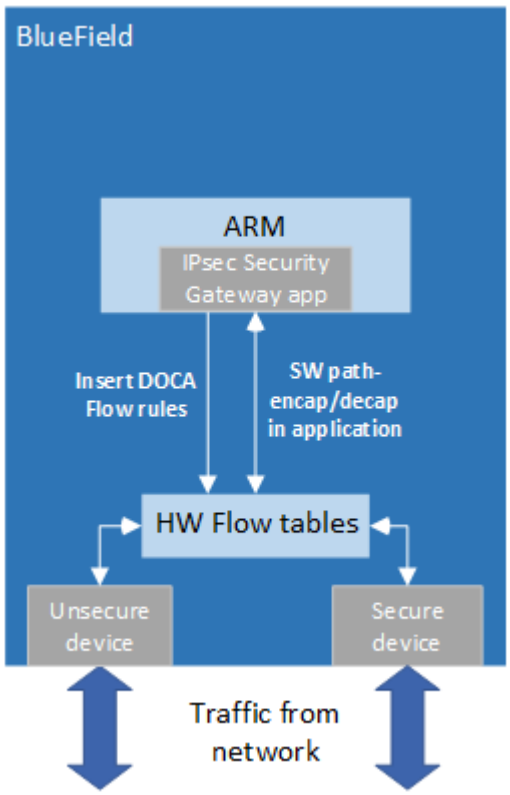
DOCA IPsec Security Gateway is designed to run with 2 ports, secured and unsecured:

- 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

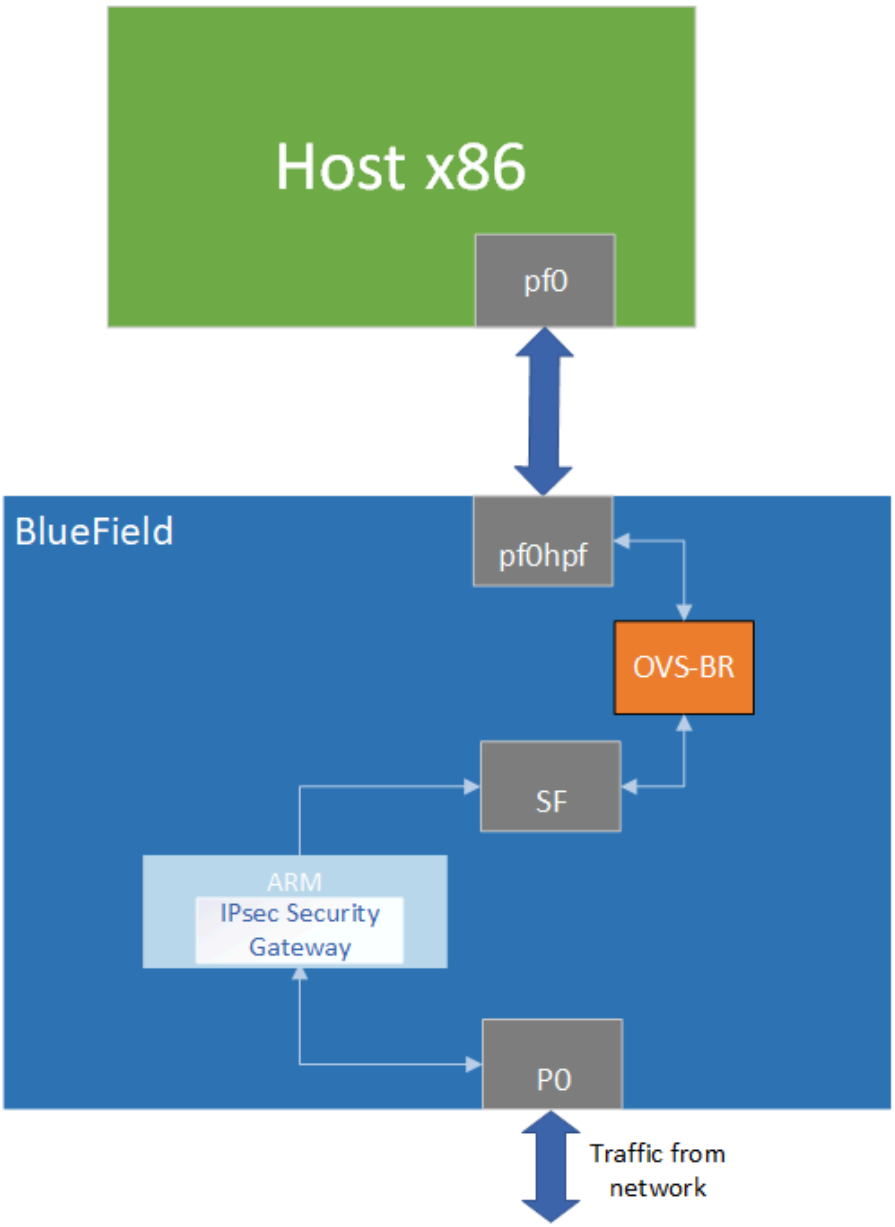
Example packet path for hardware (HW) offloading:



Example packet path for partial software processing (handling encap/decap in software):

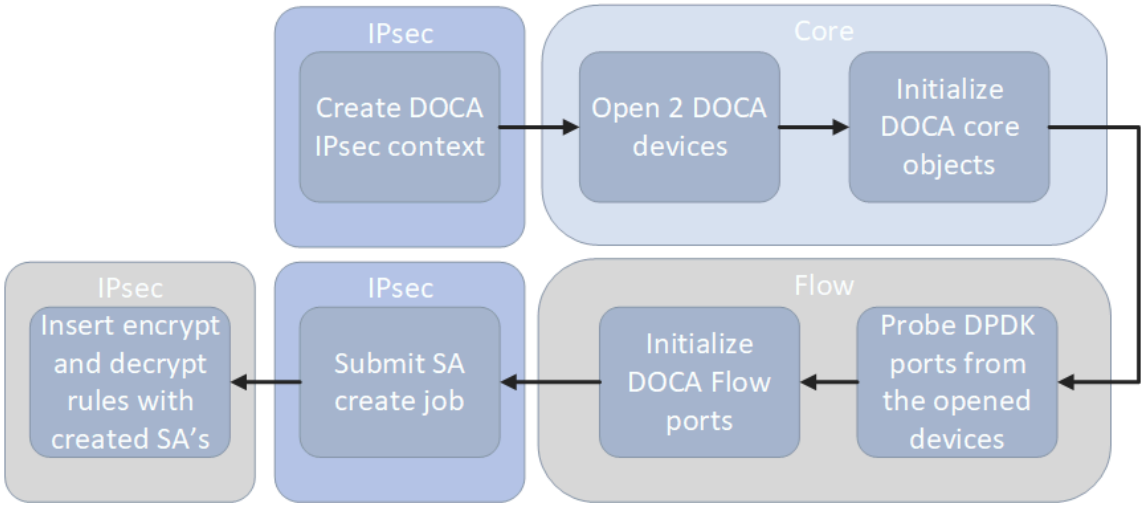


Using the application with SF:



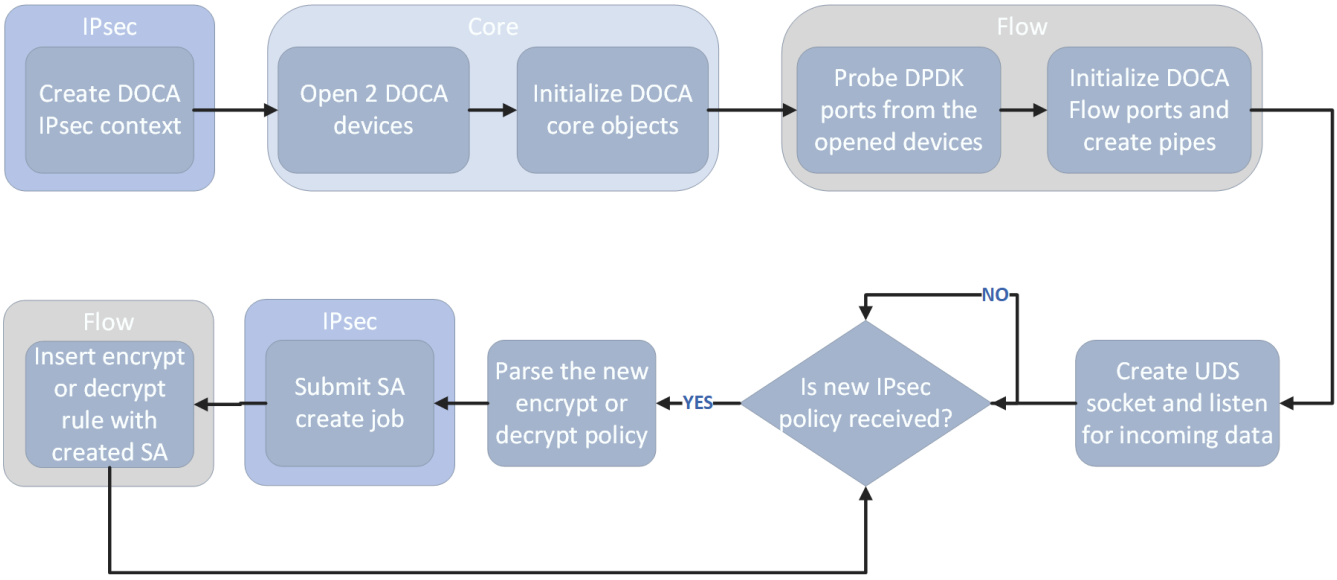
Application Architecture

Static Configuration



1. Create IPsec library context.
2. Open two DOCA devices, one for the secured port and another for the unsecured port.
3. Initialize the DOCA work queue.
4. With the open DOCA devices, the application probes DPDK ports and initializes DOCA Flow and DOCA Flow ports accordingly.
5. On the created ports, build DOCA Flow pipes.
6. In a loop according to the JSON rules:
    1. Create DOCA IPsec SA for the new rule.
    2. Insert encrypt or decrypt rule to DOCA Flow pipes.

Dynamic Configuration



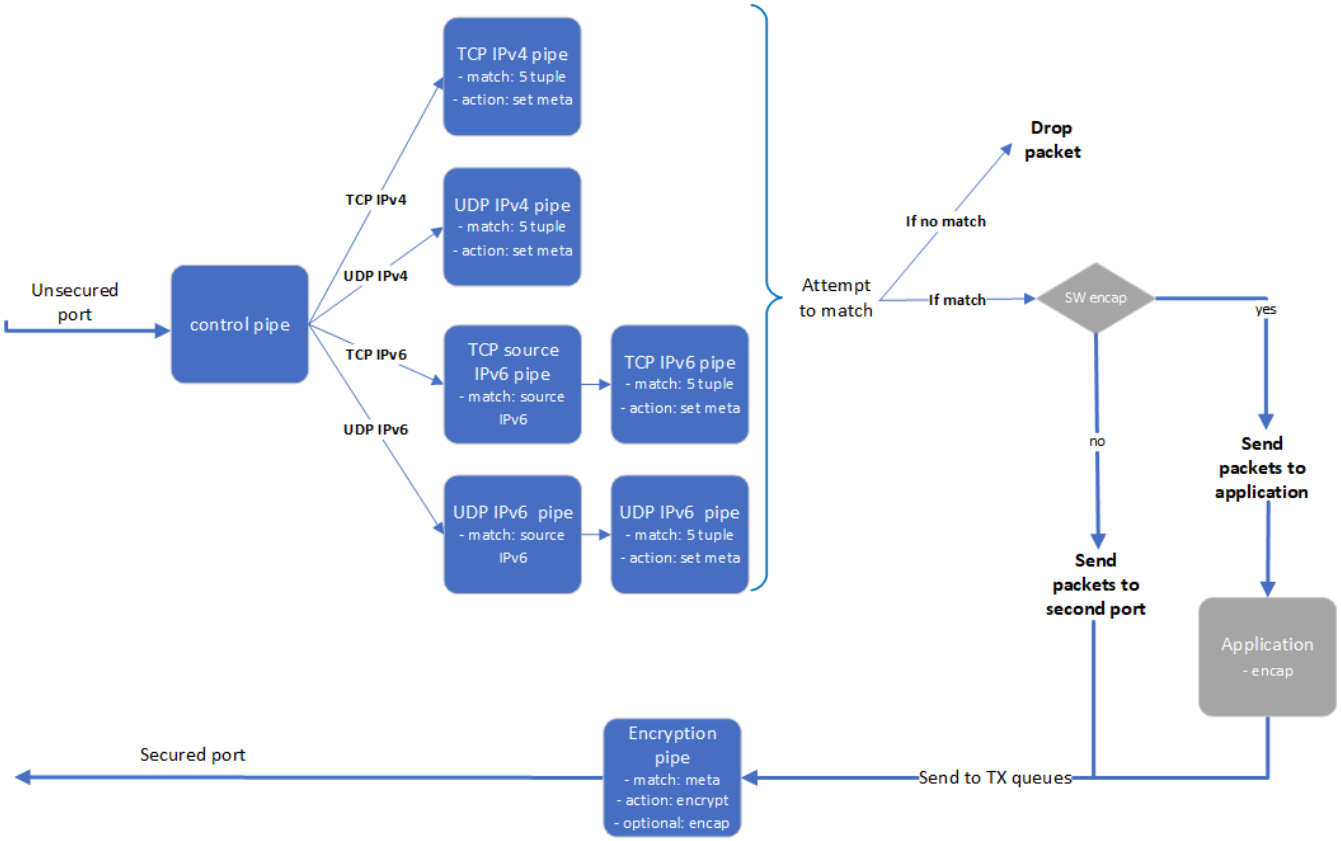
1. Create IPsec library context.
2. Open two DOCA devices, one for the secured port and another for the unsecured port.
3. Initialize the DOCA work queue.
4. With the open DOCA devices, the application probes DPDK ports and initializes DOCA Flow and DOCA Flow ports accordingly.
5. On the created ports, build DOCA Flow pipes.
6. Create UDS socket and listen for incoming data.
7. While waiting for new IPsec policies to be received in a loop, if a new IPsec policy is received:
    1. Parse the policy whether it is an encryption or decryption rule.
    2. Create DOCA IPsec SA for the new rule.
    3. Insert encrypt or decrypt rule to DOCA Flow pipes.

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.

VNF Mode

Encryption



1. The application builds 8 pipes for encryption. Control pipe as root with four entries that match L3 and L4 types and forward the traffic to the relevant pipes.

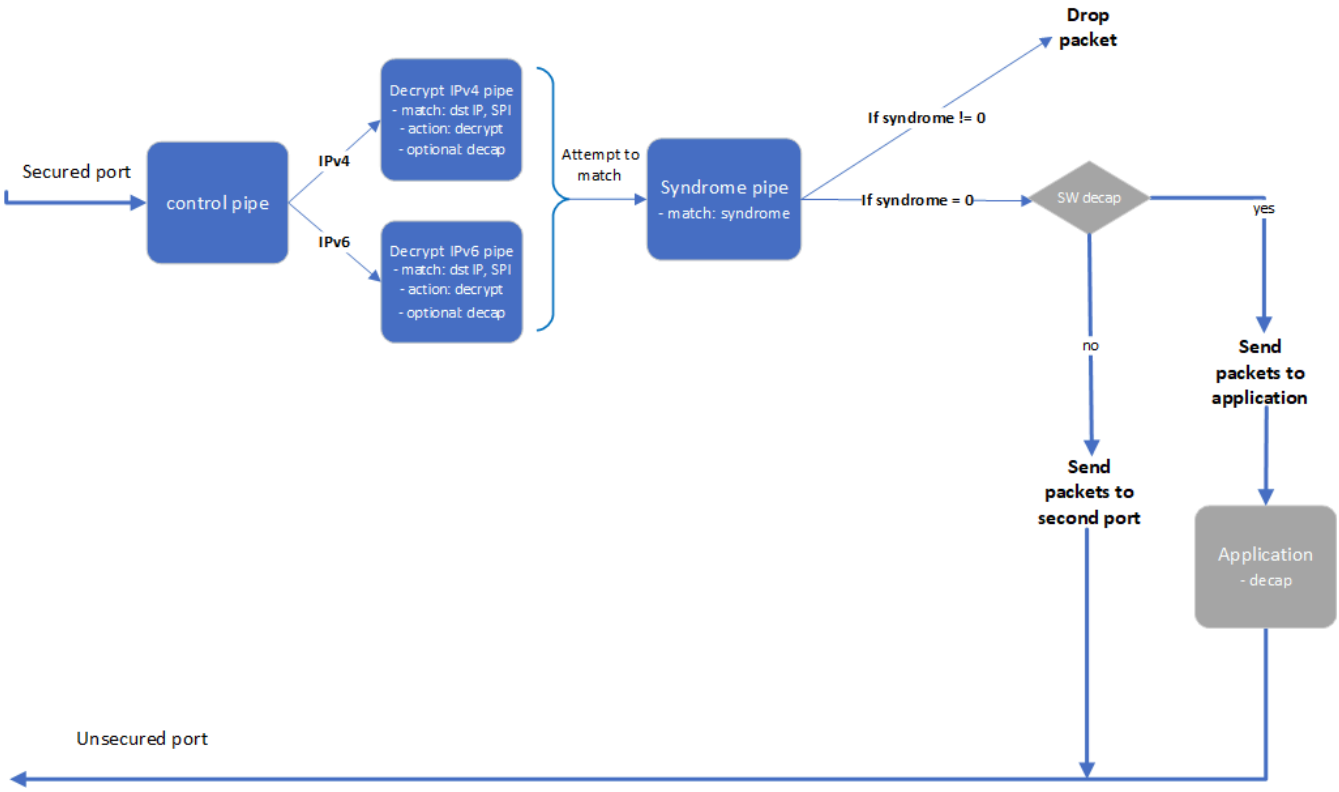
    1. IPv6 pipes – match the source IP address and forward the traffic to a pipe that matches 5-tuple excluding the source IP.

    2. In the 5-tuple match pipes set action of "set meta data", the metadata would be the rule's index in the JSON file.

    3. The matched packet is forwarded to the second port.

2. In the secured egress domain, there is an encryption pipe that 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.

Decryption

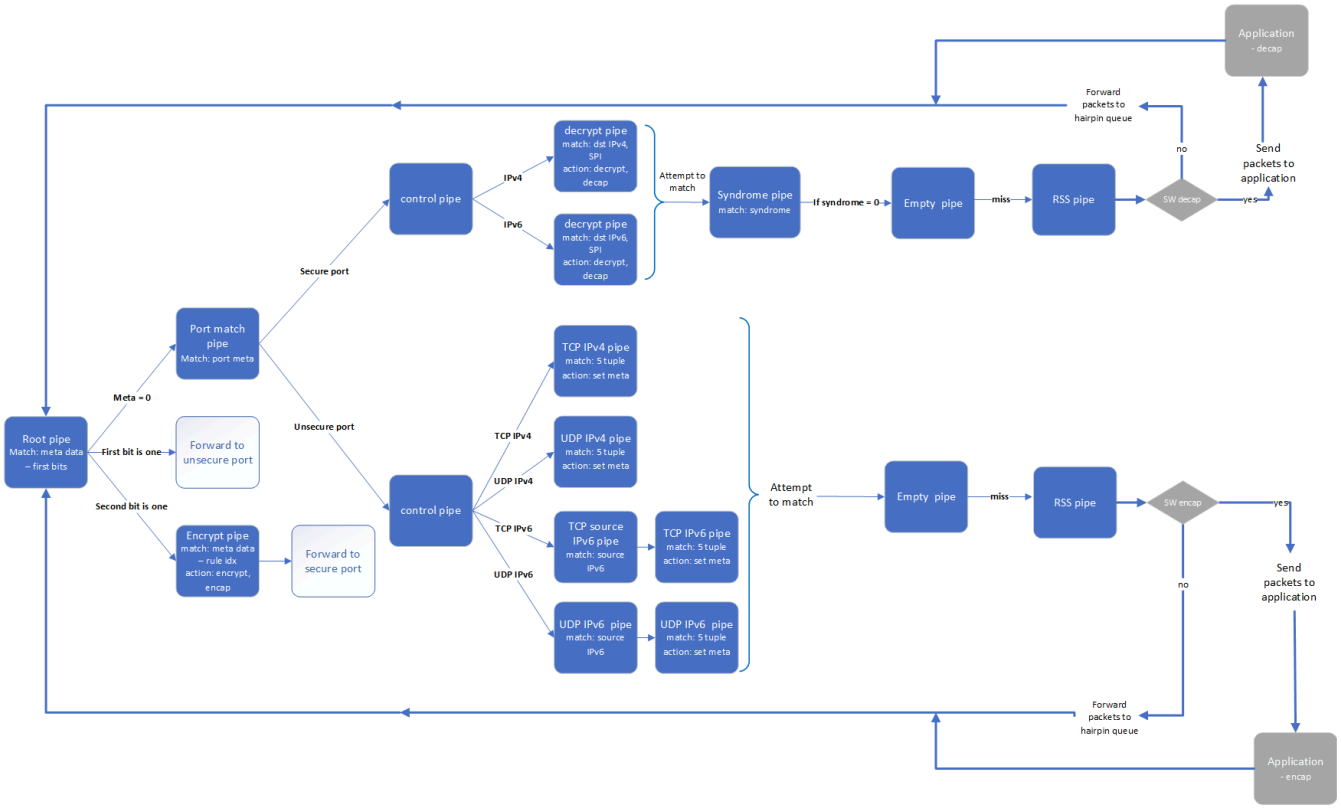


1. The application builds 4 pipes for decryption. Control pipe as root with two entries that match L3 type and forward the traffic to the relevant decrypt pipe.

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 syndrome pipe and, if the syndrome is non-zero, the packets are dropped. Otherwise, the packets are forwarded to the second port.

Switch Mode



In switch mode, a root pipe matches the first 2 most significant bits (MSBs) to decide what the next pipe is:

- Metadata is 0 – packet arrives and continues to pipe that matches on the port's meta. Based on the port, the packet passes through almost the same path as VNF mode and the metadata is set in the 2 MSBs. Afterwards, the packet moves to pipes that send the packets to the root pipe.
- First bit is 1 – packet finishes the decrypt path and must be sent to the unsecure port.
- Second 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.

DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Flow

- DOCA IPsec

Refer to their respective programming guide for more information.

Running the Application

Installation

Refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software .

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
```

> **Note**
>
> 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. In such cases, the guiding principal is to allocate 4GB of RAM, and instead of allocating 2048 pages, one should allocate the matching amount (8 pages):
>
> ```
> sudo echo 8 > /proc/sys/vm/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
```

Note

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
```

Application Execution

The IPsec Security Gateway application is provided in both source and binary forms. The binary is located under `/opt/mellanox/doca/applications/ipsec_security_gw/bin/doca_ipsec_secu`.

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=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, --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
```

Info

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
/opt/mellanox/doca/applications/ipsec_security_gw/bin/doc
-h
```

Info

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the BlueField or host:

   - Static Configuration:

```
cd /opt/mellanox/doca/applications/ipsec_security_gw/bin
./doca_ipsec_security_gw -s 03:00.0 -u 03:00.1 -c
./ipsec_security_gw_config.json -m transport
```

Note

Both the PCIe address identifiers (`-s` and `-u` flags) should match the addresses of the desired PCIe devices.

   - Dynamic Configuration:

```
cd /opt/mellanox/doca/applications/ipsec_security_gw/bin
./doca_ipsec_security_gw -s 03:00.0 -u 03:00.1 -c
./ipsec_security_gw_config.json -m transport -i
/tmp/rules_socket
```

> **Note**
>
> 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

```
cd /opt/mellanox/doca/applications/ipsec_security_gw/bin
./doca_ipsec_security_gw --json
./ipsec_security_gw_params.json
```

> **Note**
>
> Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the PCIe addresses necessary for the deployment.

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	`c`	`config`	Path to JSON file with configurations	`"config" : security_gateway_config.`
	`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"`

Info

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

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
config	switch	bool	True for running DOCA Flow in sw Default is false (VNF mode).
	esp_header_offload	string	Decap and encap offloading: bot , decap, or none. Default is bo (offloading both encap and decap
	sw_sn_inc_enable	bool	Increments sequence number of software if set to true. Default **Note** Available only if esp_header_offload is decap or none.
	sw_antireplay_enable	bool	Enables anti-replay mechanism in if set to true. Default is false. A only if esp_header_offload is none. **Note** Window size is 64. Not ESN. Supports non-zero sn_initial.
	sn_initial	uint	Initial sequence number for ESP h Used also when sw_antireplay is true. Default is 0.

Section	Field	Type	Description
encrypt_rules	ip-version	int	Source and destination IP version Optional. Default is 4.
	src-ip	string	Source IP to match
	dst-ip	string	Destination IP to match
	protocol	string	L4 protocol: TCP or UDP
	src-port	int	Source port to match
	dst-port	int	Destination port to match
	encap-ip-version	int	Encap IP version: 4 or 6. Optional; 4.
	encap-dst-ip	string	Encap destination IP. Mandatory f mode only.
	spi	int	SPI integer to set in the ESP heac
	key	string	Key for creating the SA (in hex for
	key_type	int	Key size: 128 or 256. Optional; def
	salt	int	Salt value for creating the SA. Default is 6
	icv_length	int	ICV length value: 8, 12, or 16. Default is 16

Section	Field	Type	Description
decrypt_rules	ip-version	int	Destination IP version: 4 or 6. Opt default is 4.
	dst-ip	string	Destination IP to match
	inner-ip-version	int	Inner IP version. Mandatory for tu only. Optional; default is 4.
	spi	int	SPI to match in the ESP header
	key	string	Key for creating the SA (in hex for
	key_type	int	Key size: 128 or 256. Optional; def
	salt	int	Salt value for creating the SA. Default is 6,
	icv_length	int	ICV length value: 8, 12, or 16. Default is 16

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.

> **Note**
>
> 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 */
```

```
};
```

Note

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

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications .

Recompiling the Application

In addition to providing the application in binary form, the installation also includes all of the application sources and compilation instructions so as to allow modifying the sources and recompiling 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 `/opt/mellanox/doca/applications/ipsec_security_gw/src` directory.

Recompiling All Applications

The applications are all defined under a single meson project, so the default compilation recompiles all the DOCA applications.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

Info

`doca_ipsec_security_gw` is created under `/tmp/build/ipsec_security_gw/src/`.

Recompiling IPsec Security Gateway Application Only

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
```

Info

`doca_ipsec_security_gw` is created under `/tmp/build/ipsec_security_gw/src/`.

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
```

> **Info**
>
> `doca_ipsec_security_gw` is created under `/tmp/build/ipsec_security_gw/src/`.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application .

Application Code Flow

1. Parse application argument.
    1. Initialize arg parser resources and register DOCA general parameters.

```
doca_argp_init();
```

2. Register the application's parameters.

```
register_ipsec_security_gw_params();
```

3. Parse the arguments.

```
doca_argp_start();
```

1. 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();
```

1. Open DOCA devices with input PCIe addresses / interface names.
2. Probe DPDK port from each opened device.

5. Initialize and start DPDK ports.

```
dpdk_queues_and_ports_init();
```

1. Initialize DPDK ports, including mempool allocation.
2. Initialize hairpin queues if needed.
3. Binds hairpin queues of each port to its peer port.

6. Initialize objects for DOCA IPsec library.

```
ipsec_security_gw_ipsec_ctx_create();
```

1. Create IPsec library context.
2. Create DOCA Progress Engine.

7. Initialize DOCA Flow.

```
ipsec_security_gw_init_doca_flow();
```

1. Initialize DOCA Flow library.
2. Find the indices of the DPDK-probed ports and start DOCA Flow ports with them.

8. Insert rules.

1. Insert encryption rules.

```
ipsec_security_gw_insert_encrypt_rules();
```

2. Insert decryption rules.

```
ipsec_security_gw_insert_decrypt_rules();
```

9. Wait for traffic.

```
ipsec_security_gw_wait_for_traffic();
```

1. wait in a loop until the user terminates the program.

10. IPsec security gateway cleanup:

1. DOCA Flow cleanup; destroy initialized ports.

```
doca_flow_cleanup();
```

2. SA destruction.

```
ipsec_security_gw_destroy_sas();
```

3. IPsec objects destruction.

```
ipsec_security_gw_ipsec_ctx_destroy();
```

4. Destroy DPDK ports and queues.

```
dpdk_queues_and_ports_fini();
```

5. DPDK finish.

```
dpdk_fini();
```

Calls `rte_eal_destroy()` to destroy initialized EAL resources.

6. Arg parser destroy.

```
doca_argp_destroy()
```

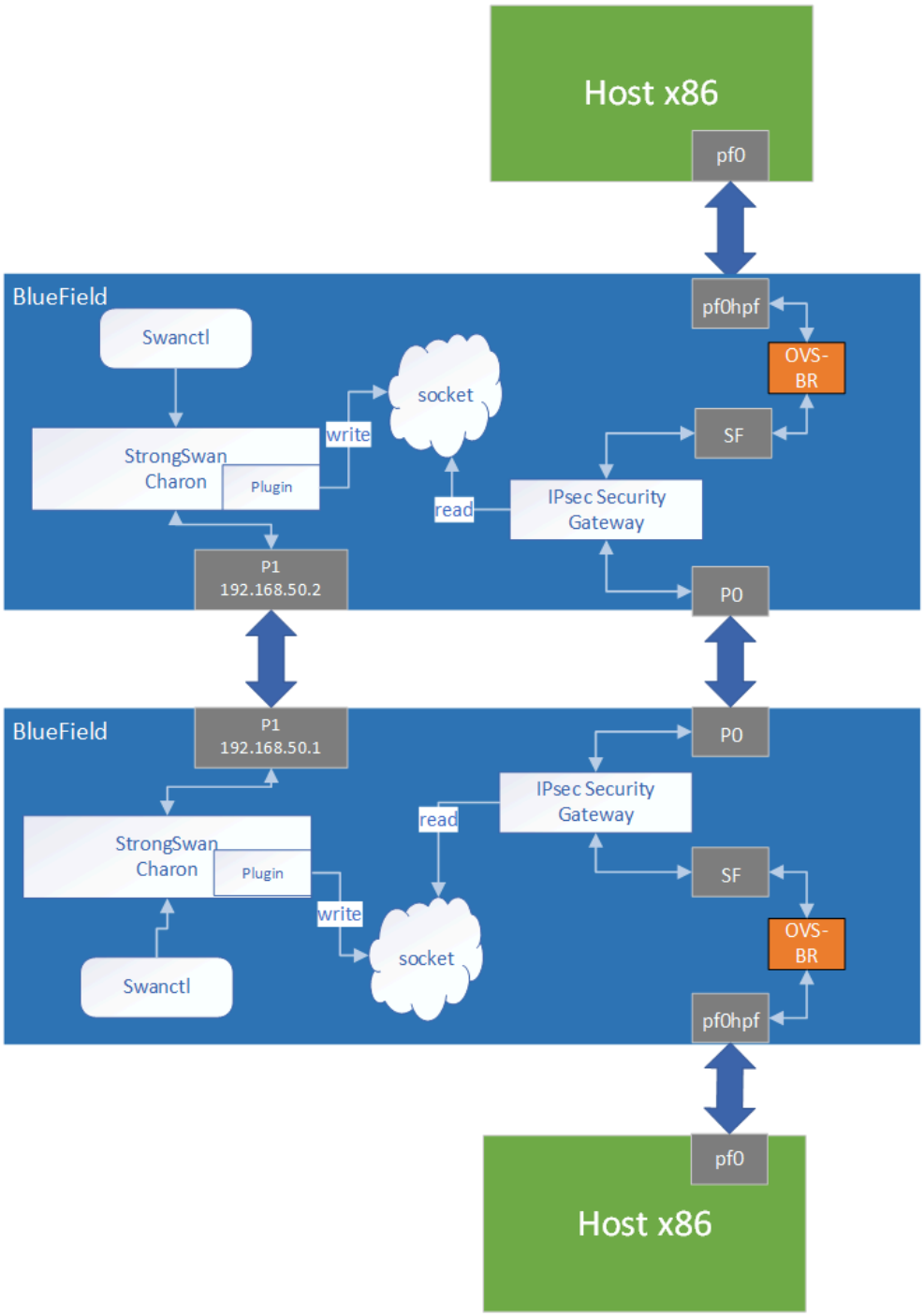
Keying Daemon Integration (StrongSwan)

strongSwan is a keying daemon that uses the Internet Key Exchange Version 2 (IKEv2) protocol to establish SAs between two peers. s trongSwan 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 .

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:

    1. On BF1:

    ```
    ip addr add 192.168.50.1/24 dev p1
    ```

    2. On BF2:

    ```
    ip addr add 192.168.50.2/24 dev p1
    ```

    > **Note**
    >
    > 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
    ```

> **Note**
>
> 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
    }
}
```

Note

`local_ts` and `remote_ts` must have a netmask of /32 for IPv4 addresses and /128 for IPv6 addresses.

Note

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

ESP Proposal	Algorithm Type Including ICV Length	Key Size
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

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
```

> **Note**
>
> 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
}
```

Note

You must provide the application with this path as well.

3. Restart the strongSwan server:

```
systemctl restart strongswan-starter.service
```

Note

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.

> **Note**
>
> 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>
```

> **Info**
>
> In the example above, the child's name is `bf`.

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-5.9.6 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-starter.service
```

References

- `/opt/mellanox/doca/applications/ipsec_security_gw/src`
- `/opt/mellanox/doca/applications/ipsec_security_gw/bin/ipsec_secur`

NVIDIA DOCA NAT Application Guide

This document provides a NAT implementation on top of NVIDIA® BlueField® DPU.

Introduction

The Network Address Translation (NAT) reference application leverages the DPU's hardware capability to switch packets with local IP addresses to global ones and vise versa.

The NAT application is based on the DOCA Flow API used for the programming of the DPU's hardware.

NAT can operate in three modes:

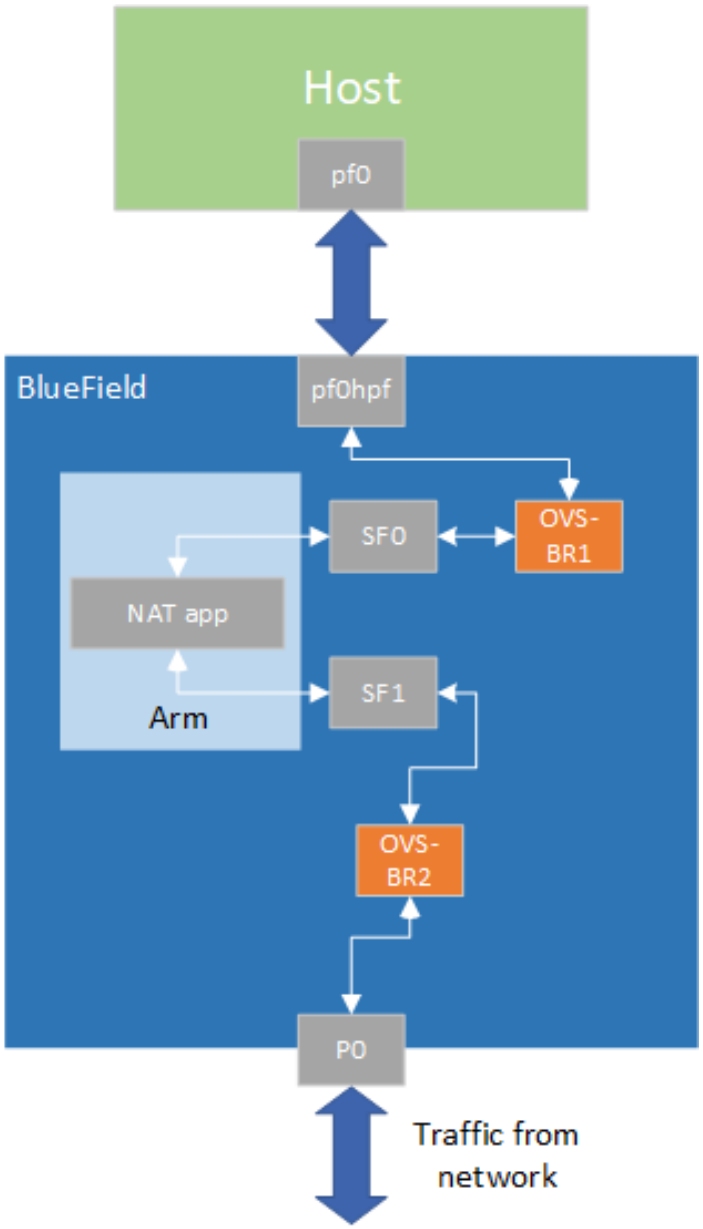
- Static mode – application gets pairs of local IP address and global IP address from the user using a JSON file

- Dynamic mode – user provides pool of global IP addresses that can be used. The application should pick one address from the pool for new local area network (LAN) IP address and use it. Once the session closes, the addresses are returned to the pool.

- PAT mode (DNS offload) – the user provides one global address to use. In addition, the user provides mapping between the local port address to the global port. For each packet, the local address is replaced with the global one and ports are replaced according to mapping table.

System Design

The NAT application is design to run on the DPU. The DPU intercepts ingress traffic from both wire and host, switches the relevant IP address and port according to data configured by the user, and forwards it to the egress port.



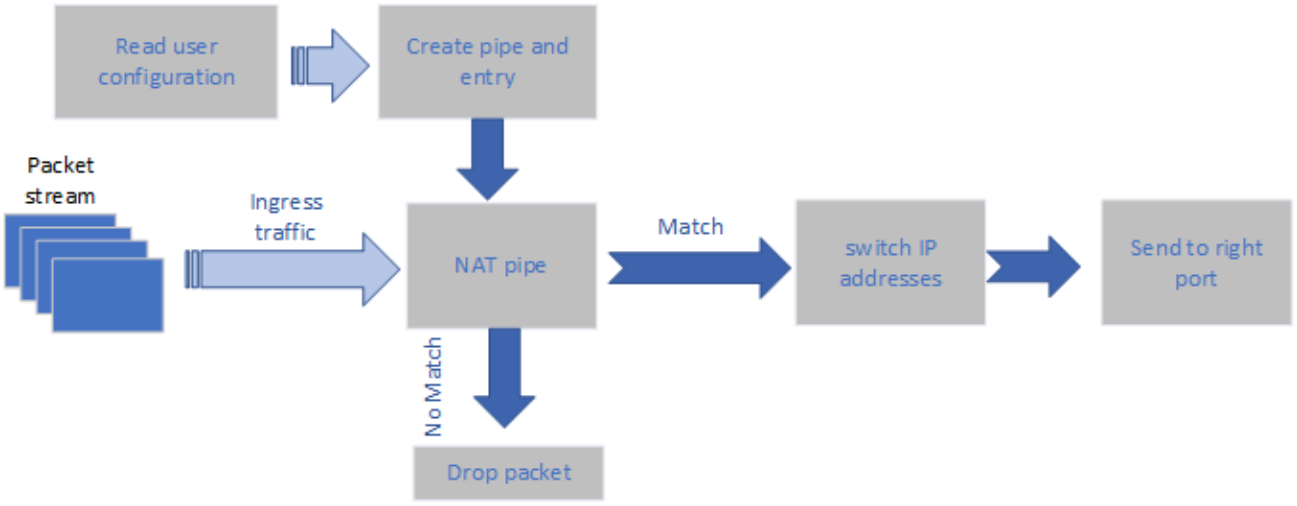
Application Architecture

NAT runs on the DPU to classify packets.

The app should be configured using a JSON file which includes the operation mode.

Static Mode

For static mode, the JSON file should include pairs of local and global IP addresses. No change for ports in this mode.

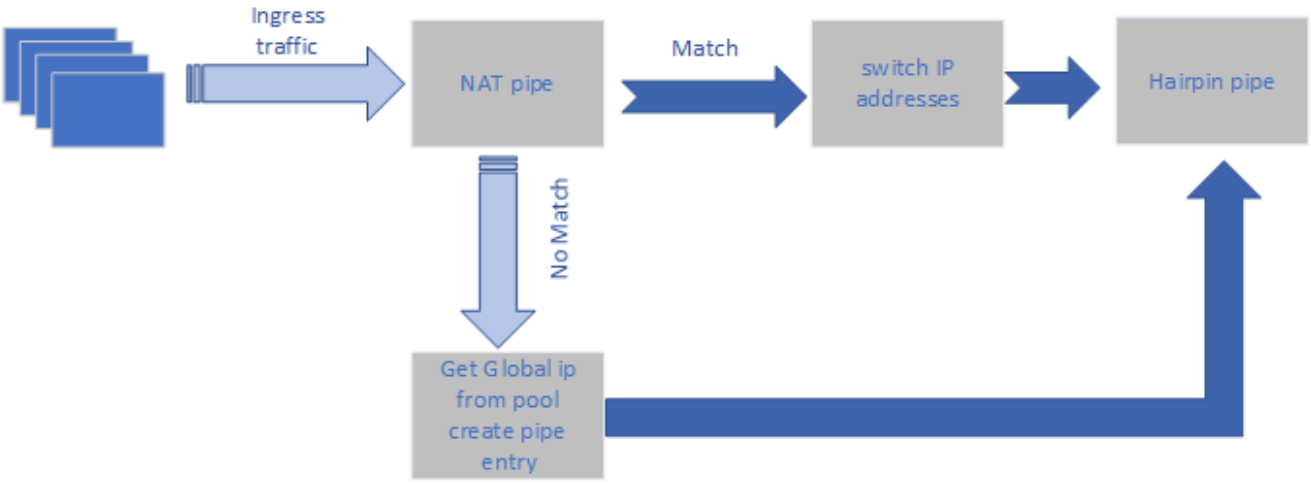


Dynamic Mode

The user must provide a pool of global IP addresses to use. The application allocates a global address to every miss in the pipe (new local address).

If no more global addresses are available in the pool, the user gets an error message and the packet is sent as is.

The application performs a callback to remove the matching of global and local IPs and returns the address to the pool.

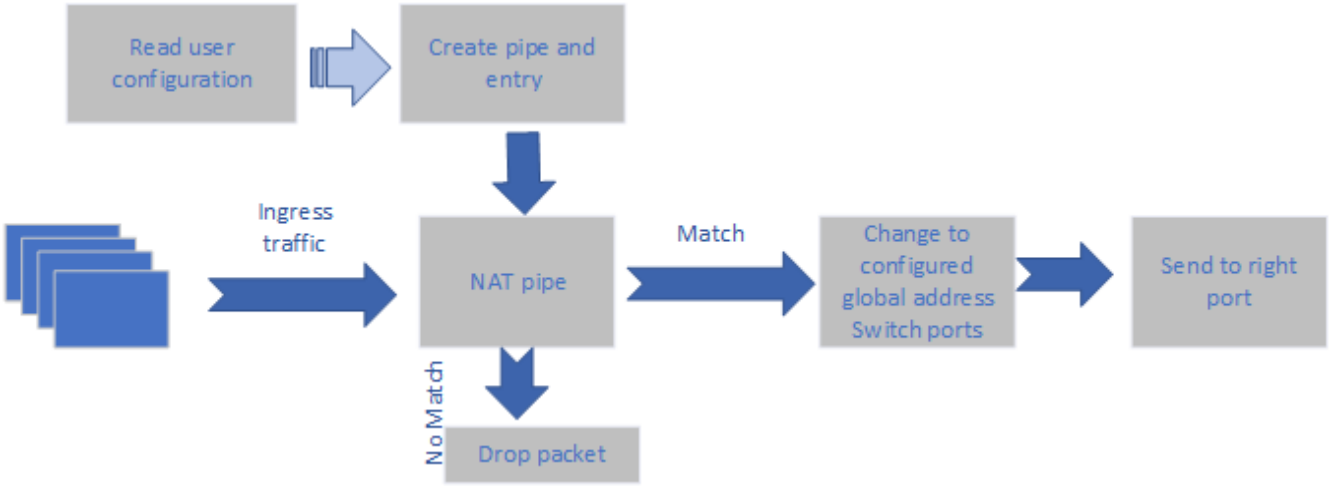


PAT (NAT Offload) Mode

The user provides a global address to replace all local addresses in the user LAN.

The user provides a matching of local IP and port to global port.

The application changes the local IP of every match to the global IP provided by the user and updates the port number according to user configuration.



DOCA Libraries

This application leverages the following DOCA library:

- DOCA Flow

Refer to its respective programming guide for more information.

Running the Application

Installation

Refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software .

Prerequisites

The NAT 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
```

Note

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. In such cases, the guiding principal is to allocate 4GB of RAM, and instead of allocating 2048 pages, one should allocate the matching amount (8 pages):

```
sudo echo 8 > /proc/sys/vm/nr_hugepages
```

Application Execution

The NAT application is provided in both source and binary forms, The binary is located under `/opt/mellanox/doca/applications/nat/bin/doca_nat`.

1. Application usage instructions:

```
Usage: doca_nat [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=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:
  -m, --mode <mode>                   set NAT mode
  -r, --nat-rules <path>              Path to the JSON file
with NAT rules
  -lan, --lan-intf <lan intf>         name of LAN interface
  -wan, --wan-intf <wan intf>         name of wan interface
```

Info

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
/opt/mellanox/doca/applications/nat/bin/doca_nat
-- -h
```

Info

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the BlueField:

```
/opt/mellanox/doca/applications/nat/bin/doca_nat -a
auxiliary:mlx5_core.sf.4,dv_flow_en=2 -a
auxiliary:mlx5_core.sf.5,dv_flow_en=2 -- -m static -r
/opt/mellanox/doca/applications/nat/bin/nat_static_rules.json
-lan sf3 -wan sf4
```

Note

SFs must be enabled according to NVIDIA BlueField DPU Scalable Function User Guide.

Note

The flag `-a auxiliary:mlx5_core.sf.4,dv_flow_en=2 -a auxiliary:mlx5_core.sf.5,dv_flow_en=2` is mandatory for proper usage of the application. Modifying this flag results in unexpected behavior as only 2 ports are supported. The SF number is arbitrary and configurable.

> **Note**
>
> The SF numbers must match the identifiers of the configured SFs.

3. The application also supports a JSON-based deployment mode, in which all command-line arguments are provided through a JSON file:

```
doca_nat --json [json_file]
```

For example:

```
cd /opt/mellanox/doca/applications/nat/bin
./doca_nat --json ./nat_params.json
```

> **Note**
>
> Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the PCIe addresses necessary for the deployment.

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", "hws" : true}, {"device" : "sf", "id" : "5", "hws" : true},]

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`	`mode`	Set NAT mode	`"mode" : "static"`
	`r`	`nat-rules`	Path to the JSON file with NAT rules	`"nat-rules" : "nat_static_rules.json"`
	`lan`	`Lan-intf`	Name of LAN interface	`"lan-intf" : "sf3"`
	`wan`	`Wan-intf`	Name of WAN interface	`"wan-intf" : "sf4"`

Info

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications .

Recompiling the Application

In addition to providing the application in binary form, the installation also includes all of the application sources and compilation instructions so as to allow modifying the sources and recompiling 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 `/opt/mellanox/doca/applications/nat/src` directory.

Recompiling All Applications

The applications are all defined under a single meson project, so the default compilation recompiles all the DOCA applications.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

Info

`doca_nat` is created under `/tmp/build/nat/src/`.

Recompiling NAT Application Only

To directly build only the NAT application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_nat=true
ninja -C /tmp/build
```

> **Info**
>
> `doca_nat` is created under `/tmp/build/nat/src/`.

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_nat` to `true`
2. Run the following compilation commands :

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> **Info**
>
> `doca_nat` is created under `/tmp/build/nat/src/`

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application .

Application Code Flow

1. Parse application argument.
    1. Initialize arg parser resources and register DOCA general parameters.

        ```
        doca_argp_init();
        ```

    2. Register NAT application.

        ```
        register_nat_params()
        ```

    3. Parse the arguments.

        ```
        doca_argp_start();
        ```

        1. Parse DPDK flags and invoke handler for calling the `rte_eal_init()` function.
        2. 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();
```

1. Initialize DPDK ports, including mempool allocation.
2. Initialize hairpin queues if needed.
3. Bind hairpin queues of each port to its peer port.

4. NAT initialization.

```
nat_init();
```

1. DOCA Flow and DOCA Flow port initialization.

5. Init user configuration rules into app structure.

```
parsing_nat_rules();
```

6. Init pipes and entry according to rules.

```
nat_pipes_init();
```

7. Wait for signal to end application.
8. NAT destroy.

```
nat_destroy();
```

9. DPDK ports and queues destruction.

```
dpdk_queues_and_ports_fini();
```

10. DPDK finish.

```
dpdk_fini();
```

    1. Calls `rte_eal_destroy()` to destroy initialized EAL resources.

11. Arg parser destroy.

```
doca_argp_destroy();
```

References

- `/opt/mellanox/doca/applications/nat/src`
- `/opt/mellanox/doca/applications/nat/bin/nat_params.json`

DOCA NVMe-oF RDMA Target Reference Application Guide

Instructions: Please replace all labels/tags such as with the relevant content from your library/driver name, samples names, etc.

All the text that is marked in RED should be replaced with actual content for the specific guide.

This document provides a <application name> implementation on top of the NVIDIA® BlueField® DPU.

Introduction

A storage target is any application that can access data storage. This includes hard drives, SSDs, network-attached storage (NAS), and cloud storage services.

It serves storage access requests from local or remote initiators.

To allow the storage target to receive remote requests (over the network), the NVMe (Non-Volatile Memory Express) over fabrics (NVMeOF) protocol was introduced.

NVMe over Fabrics (NVMe-oF) is a protocol specification designed to extend the capabilities of NVMe storage across network fabrics (for example, RoCEv1/IB RDMA and NVMeTCP).

NVMe-oF extends the concept of storage targets by enabling NVMe commands to be executed over a network, rather than being limited to direct-attached storage.

Storage target application that supports NVMe-oF is a heavy application that consumes a substantial number of CPU cores.

To improve latency, free CPU resources and save power, an offload solution is required. Starting from BF3, the target application shall be offloaded on the DPA.

To abstract the communication between the target offload application (in our case SPDK) integration with the underlying offload accelerator and to ease the integration,

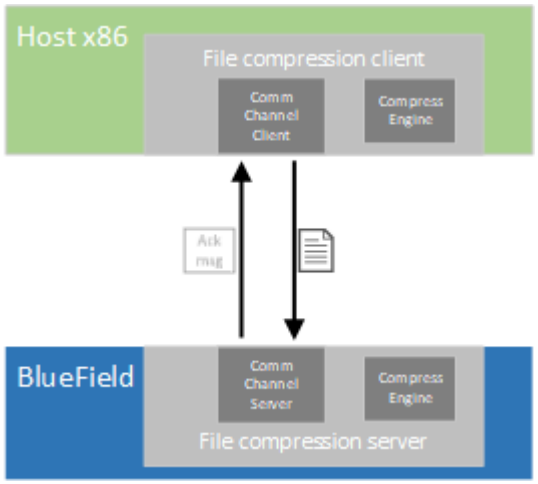
a new DOCA library is introduced (DOCA_STA). The library shall expose a public API that shall be used by the target application (DOCA application) for control-plane and data-plane handling.

While a software-based storage target accesses the NVMe drives directly over the PCIe, the DPA storage target offload accelerator accesses the NVMe drives

over PCIe-P2P (peer-to-peer) topology. To allow such access special kernel configuration is required.

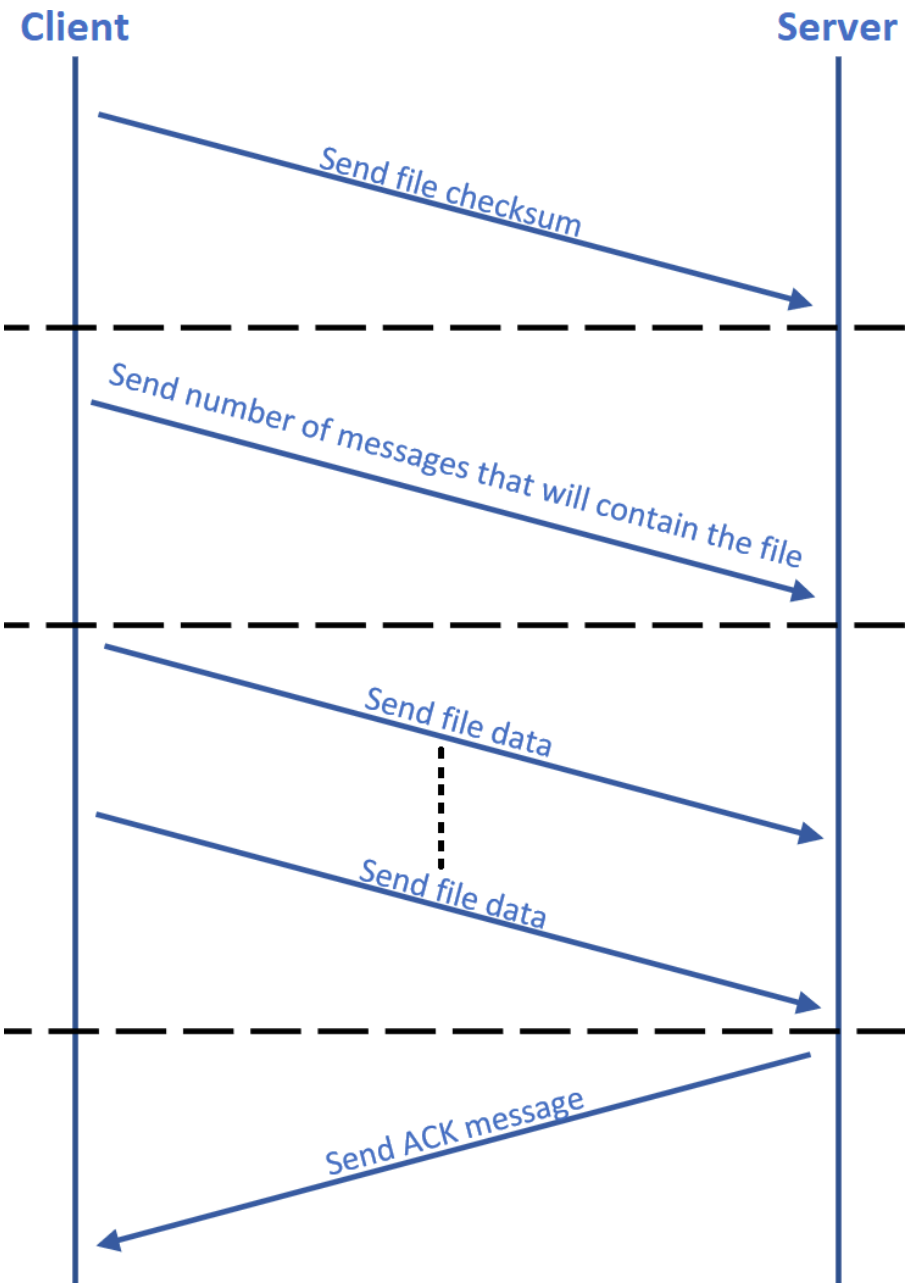
System Design

The file compression application client runs on the host and the server runs on the DPU.



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.

DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Compress - Programming Guide
- DOCA Comm Channel - Programming Guide

For additional information about the used DOCA libraries, please refer to the respective programming guides.

Dependencies

- NVIDIA BlueField-3 DPU is required.
- Ubuntu 18.04/20.04/22.04 hosts (x86)
- Open MPI version 4.1.5rc2 or greater (included in DOCA's installation)

Compiling the Application

> **Info**
>
> 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.

> **Tip**
>
> 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/`.

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
```

> **Info**
>
> `doca_file_compression` is created under `/tmp/build/file_compression/`.

Compiling Only the Current Application

1. To directly build only the file compression application: **(REPLACE WITH FLAG FOR YOUR APP)**

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -
Denable_file_compression=true
ninja -C /tmp/build
```

> **Info**
>
> `doca_file_compression` is created under `/tmp/build/file_compression/`.

2. 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_compression` to `true`

    2. 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
```

> **Info**
>
> `doca_file_compression` is created under `/tmp/build/file_compression/`.

Troubleshooting

Please refer to the NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the compilation of the DOCA applications.

Running the Application

Prerequisites

<huge pages and stuff are declared here>

1. The IPsec security gateway application is based on DOCA Flow. Therefore, the user is required to allocate huge pages.

```
$ echo '1024' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
$ sudo mkdir /mnt/huge
$ sudo mount -t hugetlbfs -o pagesize=2M nodev /mnt/huge
```

Application Execution

The file compression application is provided in source form, hence a compilation is required before the application can be executed.

1. Application usage instructions **(REPLACE WITH REAL HELP AS PRINTED FROM YOUR APP)** :

```
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=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
  -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
```

For additional information, please refer to the Command Line Flags section below.

> **Note**
>
> The above usage printout can be printed to the command line using the `-h` (or `--help`) options: **(REPLACE WITH REAL PATH OF YOUR APP)**
>
> ```
> ./doca_file_compression -h
> ```

2. CLI example for running the application on **BlueField**: **(REPLACE WITH REAL PATH & COMMAND OF YOUR APP)**

```
./doca_file_compression -p 03:00.0 -r 3b:00.0 -f received.txt
```

> **Note**
>
> Both the DOCA Comm Channel device PCI address (`03:00.0`) and the DOCA Comm Channel device representor PCI address (`3b:00.0`) should match the addresses of the desired PCI devices.

3. CLI example for running the application on the **host**: **(REPLACE WITH REAL PATH & COMMAND OF YOUR APP)**

```
./doca_file_compression -p 3b:00.0 -f send.txt
```

> **Note**
>
> The DOCA Comm Channel device PCI address (`3b:00.0`) should match the address of the desired PCI 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: **(REPLACE WITH REAL PATH & COMMAND OF YOUR APP)**

```
./doca_file_compression --json ./file_compression_params.json
```

> **Note**
>
> Before execution, please ensure that the used JSON file contains the correct configuration parameters, and especially the desired PCI addresses needed for the deployment.

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	`"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`	For client – path to the file to be sent For server – path to write the file into **Note** This is a mandatory flag.	`"file" : "/tmp/data.txt"`
	`p`	`pci-addr`	Comm Channel DOCA device PCIe address **Note** This is a mandatory flag.	`"pci-addr" : 03 : 00.1`
	`r`	`rep-pci`	Comm Channel DOCA device representor PCIe address **Note** 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.

Troubleshooting

Please refer to the NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation or execution of the DOCA applications.

Application Code Flow

1. Parse application argument.
    1. Initialize arg parser resources and register DOCA general parameters.

        ```
        doca_argp_init();
        ```

    2. Register file compression application parameters.

        ```
        register_file_compression_params();
        ```

    3. Parse the arguments.

        ```
        doca_argp_start();
        ```

        1. Parse app parameters.
2. Set endpoint attributes.

    ```
    set_endpoint_properties();
    ```

    1. Set maximum message size of 4080 bytes.
    2. Set maximum number of messages allowed.

3. Create comm channel endpoint.

```
doca_comm_channel_ep_create();
```

    1. 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();
```

    1. Free all application resources.

6. Arg parser destroy.

```
doca_argp_destroy()
```

References

- `/opt/mellanox/doca/applications/file_compression/`
- `/opt/mellanox/doca/applications/file_compression/file_compression`

NVIDIA DOCA PCC Application Guide

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

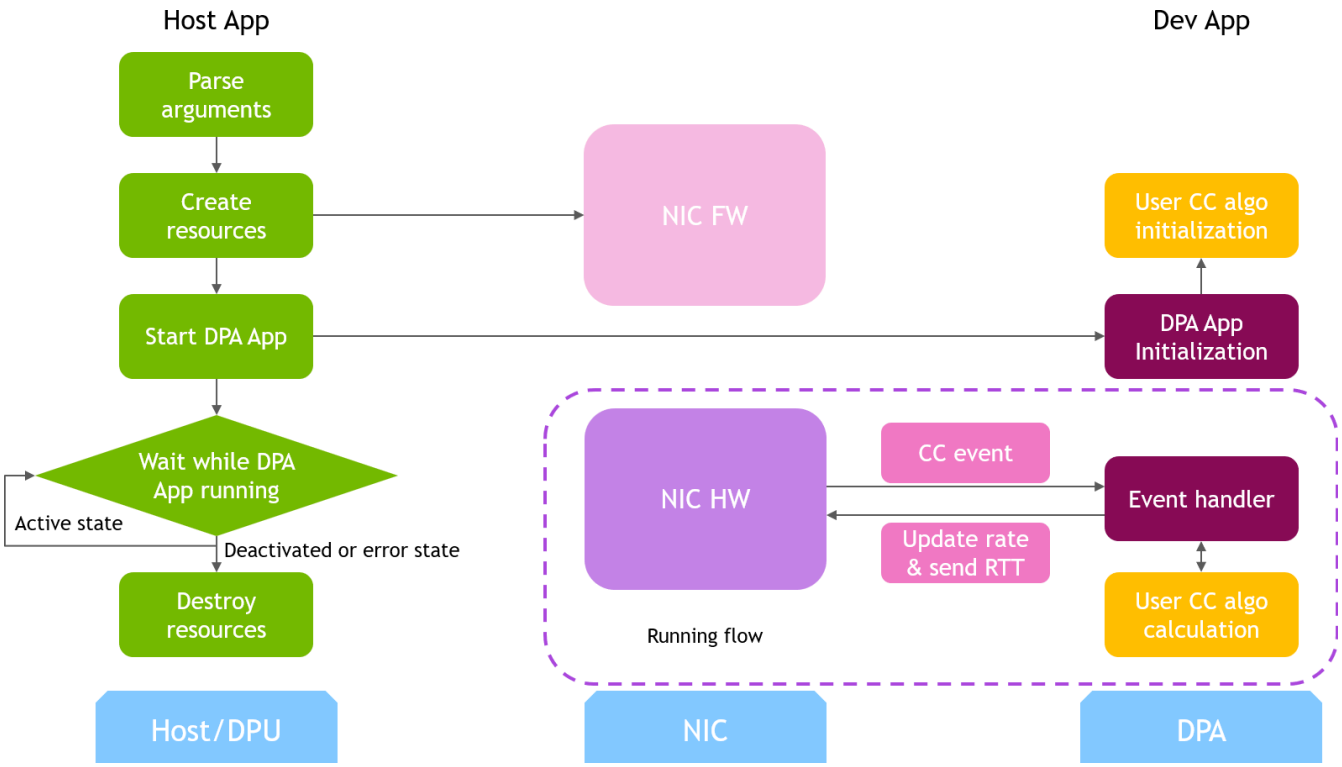
Introduction

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, 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.

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. It is mainly for CC event handler. 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 updated rate value and a flag to sent RTT request.

The host/Arm application sends command to NIC firmware when allocating or destroying resources. CC events are generated by NIC 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.

Application Architecture

```
/opt/mellanox/doca/applications/pcc/src
    host
         pcc.c
         pcc_core.c
         pcc_core.h
    device
         algo
              rtt_template.h
              rtt_template_algo_params.h
              rtt_template_ctxt.h
              rtt_template.c
         pcc_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_dev_main.c` – callbacks for user CC algorithm initialization, user CC algorithm calculation, algorithm parameter change notification

- `device/algo/*` – user CC algorithm reference template. Put user algorithm code here.

DOCA Libraries

This application leverages the following DOCA library:

- DOCA PCC

Refer to its respective programming guide for more information.

Dependencies

- NVIDIA BlueField-3 DPU is required
- Firmware 32.38.1000 and higher
- MFT 4.25 and higher

Running the Application

Installation

Refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

Prerequisites

Enable `USER_PROGRAMMABLE_CC` in `mlxconfig`:

```
mlxconfig -y -d /dev/mst/mt41692_pciconf0 set
USER_PROGRAMMABLE_CC=1
```

R eset firmware or power cycle the host to apply the configuration change.

Note

Make sure to perform graceful shutdown before power cycling the host.

Application Execution

The PCC application is provided in both source and binary forms. The binary is located under `/opt/mellanox/doca/applications/pcc/bin/doca_pcc`.

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=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 names>       IB device name that
supports PCC (mandatory).
  -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.
  -p, --pcc-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.
```

Info

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
/opt/mellanox/doca/applications/pcc/bin/doca_pcc -h
```

Info

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the BlueField or the host:

```
/opt/mellanox/doca/applications/pcc/bin/doca_pcc -d mlx5_0
```

Note

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:

```
cd /opt/mellanox/doca/applications/pcc/bin
./doca_pcc --json ./pcc_params.json
```

Note

Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the PCIe addresses necessary for the deployment.

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`	Sets the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70 **Info** 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

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
Program flags	`d`	`device`	IB device name that supports PCC	`"device" : ""`
	`w`	`wait-time`	(Optional) In seconds, the duration of the DOCA PCC wait. Negative values mean infinity.	`"wait-time" : -1`
	`p`	`pcc-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 NIC while all others are for data path CC event handling.	`"pcc-threads" : "184 185 186 200 201 202 216 217 218 232 233 234"`

Info

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

Recompiling the Application

In addition to providing the application in binary form, the installation also includes all of the application sources and compilation instructions so as to allow modifying the sources and recompiling 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 `/opt/mellanox/doca/applications/pcc/src` directory.

Recompiling All Applications

The applications are all defined under a single meson project, so the default compilation recompiles all the DOCA applications.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

Info

`doca_pcc` is created under `/tmp/build/pcc/src/`.

Recompiling PCC Application Only

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
```

Info

`doca_pcc` is created under `/tmp/build/pcc/src/`.

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
```

Info

`doca_pcc` is created under `/tmp/build/pcc/src/`.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application.

Application Code Flow

This section lists the application's configuration flow, explaining the different DOCA function calls and wrappers.

1. Parse application argument.
    1. Initialize arg parser resources and register DOCA general parameters.

        ```
        doca_argp_init();
        ```

    2. Register PCC application parameters.

        ```
        register_pcc_params();
        ```

    3. Parse the arguments.

        ```
        doca_argp_start();
        ```

        1. Parse DOCA flags.
        2. Parse DOCA PCC parameters.
2. PCC initialization.

```
pcc_init();
```

1. Open DOCA device that supports PCC.
2. Create DOCA PCC context.
3. Configure affinity of threads handling CC events.

3. Start DOCA PCC.

```
doca_pcc_start();
```

1. Create PCC process and other resources.
2. Trigger initialization of PCC on device.
3. Register the PCC in the NIC 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();
```

1. 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 NIC firmware and should be destroyed

State	Description
`DOCA_PCC_PS_ERROR = 3`	The process is in error state and should be destroyed

2. Wait on process events from the device.

5. PCC destroy.

```
doca_pcc_destroy();
```

1. Destroy PCC resources. The process stops handling PCC events.

2. Close DOCA device.

6. Arg parser destroy.

```
doca_argp_destroy()
```

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

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 – 3 `algo_nu` algo is av index • Text – alg descripti • `sl_bitm` – indicate device su `sl_bitm`
`0x1`	Enable algorithm	Set		`sl_bitmask` `trace_en` `counter_en`	N/A
`0x2`	Disable algorithm	Set		N/A	N/A
`0x3`	Get algorithm enabling status	Get		N/A	• Value: ◦ 0 – ◦ 1 – • `sl_bitm` field allo specific the bitm • `sl_bitm` – indicate device su `sl_bitm`
`0x4`	Get number of parameters	Get		N/A	• Value – n of algo

cmd_type	Description	Method	Index	Input (in --set)	Output
`0x5`	Get parameter information	Get	`algo_slot` `algo_param_index`	N/A	• `param_` default v • `param_` value of • `param_` value of • prm – ◦ 0: r ◦ 1: r ◦ 2: r may usi clea
`0x6`	Get parameter value	Get		N/A	• Value – p
`0x7`	Get and clear parameter	Get		N/A	• Value – p
`0x8`	Set parameter value	Set		Parameter value	N/A

cmd_type	Description	Method	Index	Input (in --set)	Output
`0xA`	Bulk get parameters	Get	`algo_slot`	N/A	• `text_l` num x 4 • `text[0` param va
`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_l` counter • `text[0` counter
`0xD`	Bulk get and clear counters	Get		N/A	• `text_l` counter • `text[0` counter
`0xE`	Get number of counters	Get		N/A	• Value – n of algo
`0xF`	Get counter information	Get	`algo_slot` `algo_param_index`	N/A	• `param_` value of • prm – ◦ 0: r ◦ 1: r ◦ 2: r may "ge con

cmd_type	Description	Method	Index	Input (in --set)	Output
`0x10`	Get algorithm info array	Get	N/A	N/A	• `text_l` slot initia • `text[0` 32-bit `a` if no algo available index

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"
```

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"
```

References

- `/opt/mellanox/doca/applications/pcc/src`
- `/opt/mellanox/doca/applications/pcc/bin/pcc_params.json`

NVIDIA DOCA Secure Channel Application Guide

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

Introduction

The DOCA Secure Channel reference application leverages the DOCA Comm Channel API which creates a secure, network independent communication channel between the host and the NVIDIA BlueField DPU.

Comm channel allows the host to control services on the DPU, activate certain offloads, or exchange messages using client-server framework.

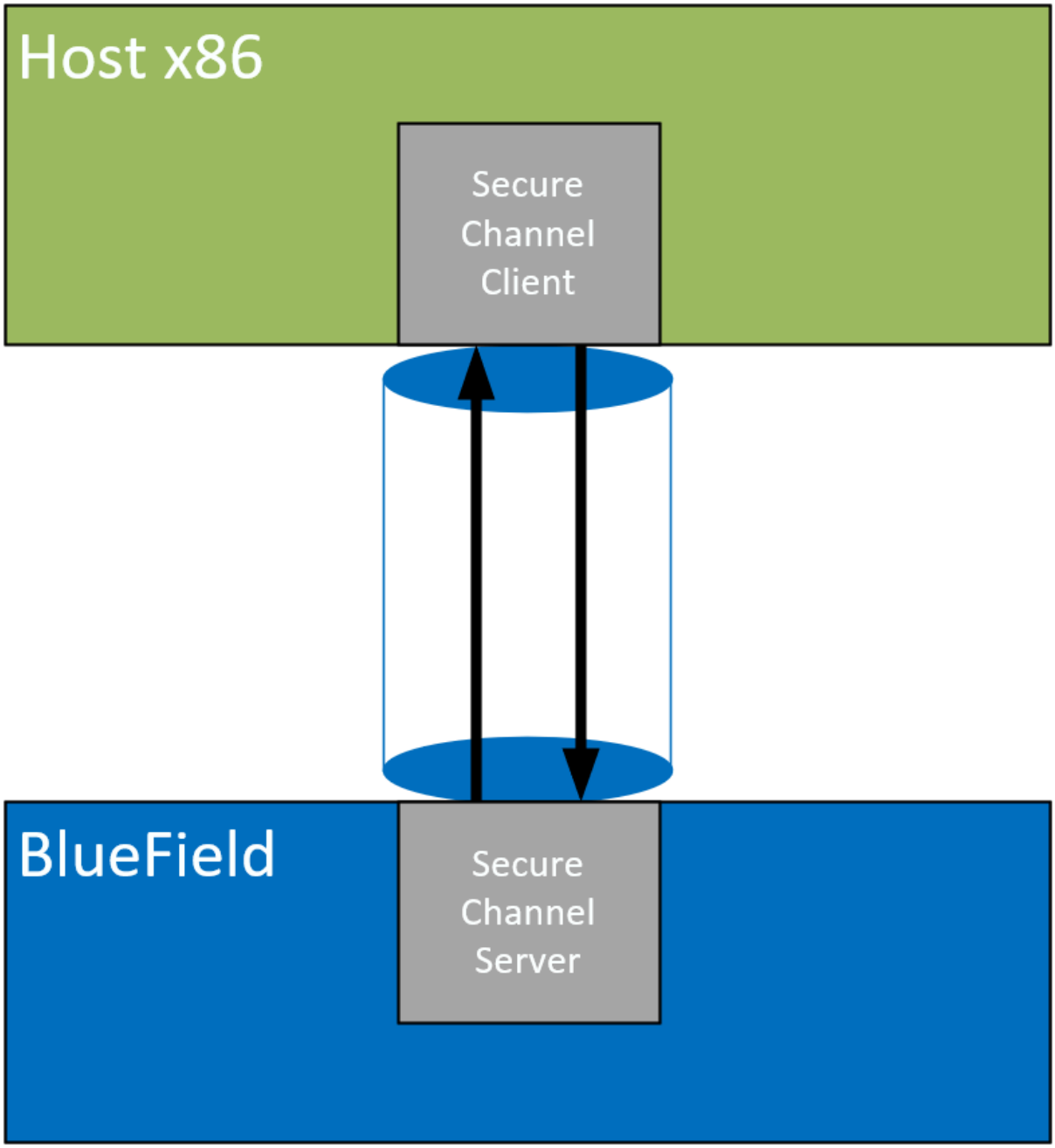
The client (host) side is able to communicate only with one server at a time while the server side is able to communicate with multiple clients.

The API allows communication between any PF/VF/SF on the host to the server located on the DPU.

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.

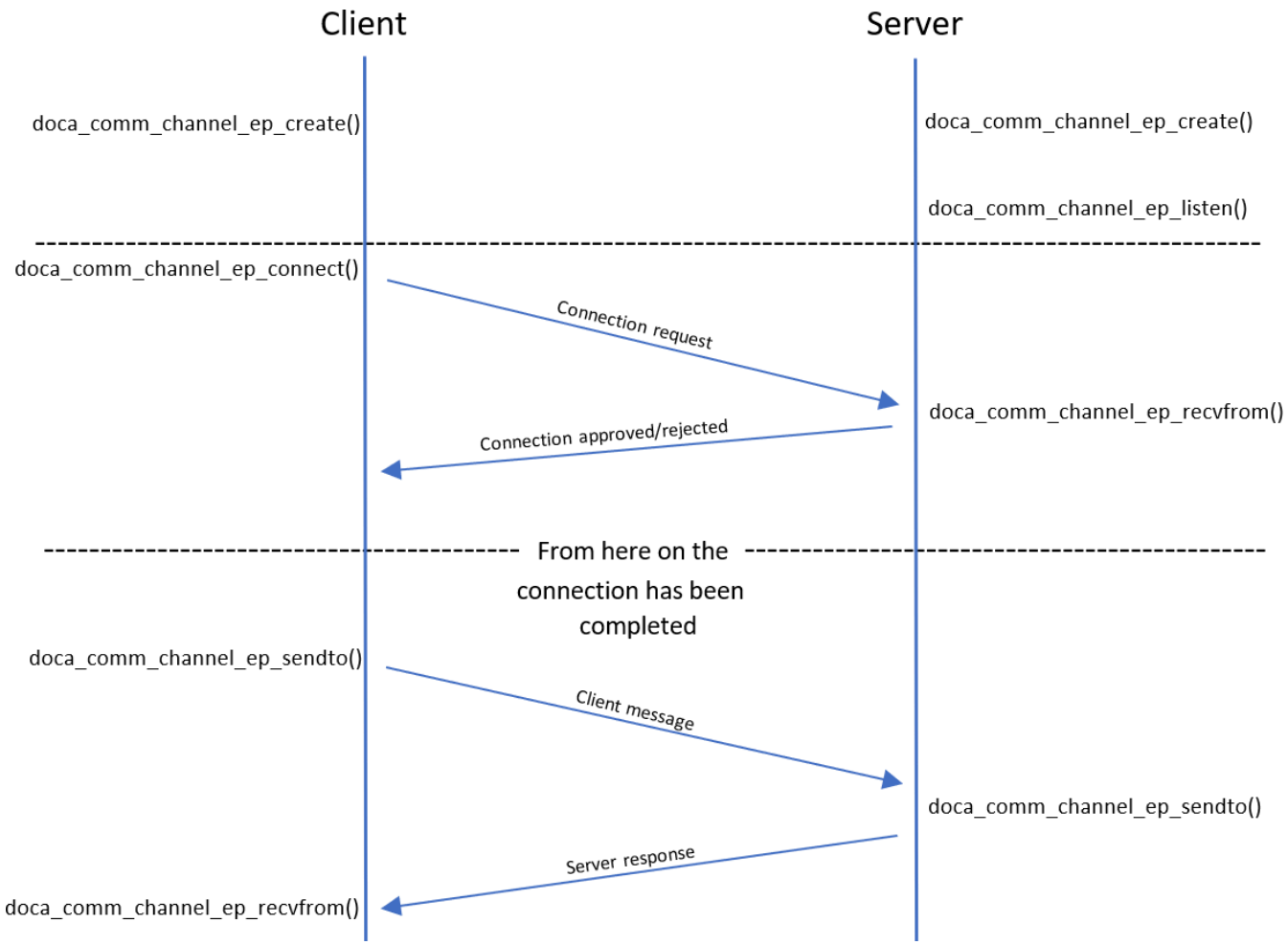
System Design

A secure channel application runs on client mode (host) and server mode (DPU). Once a channel is open, messages can flow from both sides.



Application Architecture

The secure channel application runs on top of the DOCA Comm Channel API. Full connection flow between the client and the server is illustrated in the following:



1. Both sides initiate `create()`.
2. Server listens and waits for new connections.
3. Server initiates `recvfrom()` to indicate it is ready to exchange messages
4. Client executes `connect()` to server and starts connection initialization.
5. Client sends first message to server.
6. Server sends a response.

DOCA Libraries

This application leverages the following DOCA library:

- DOCA Comm Channel

Refer to its respective programming guide for more information.

Running the Application

Installation

Refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

Application Execution

The secure channel application is provided in both source and binary forms. The binary is located under `/opt/mellanox/doca/applications/secure_channel/bin/doca_secure_channe`.

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=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, --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)
```

Info

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
/opt/mellanox/doca/applications/secure_channel/bin/doca_s
-h
```

Info

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the BlueField:

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

Note

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:

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

> **Note**
>
> 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_secure_channel --json [json_file]
```

For example:

```
cd /opt/mellanox/doca/applications/secure_channel/bin
./doca_secure_channel --json ./sc_params.json
```

> **Note**
>
> Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the PCIe addresses necessary for the deployment.

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	`s`	`msg-size`	Message size in bytes **Note** This is a mandatory flag.	`"msg-size" : 128`
	`n`	`num-msgs`	Number of messages to send on both sides **Note** This is a mandatory flag.	`"num-msgs" : 256`
	`p`	`pci-addr`	DOCA Comm Channel device PCIe address **Note** 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 Comm Channel device representor PCIe address **Note** This is a mandatory flag only on the DPU.	`"rep-pci" : b1 : 00.1`

Info

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications .

Recompiling the Application

In addition to providing the application in binary form, the installation also includes all of the application sources and compilation instructions so as to allow modifying the sources and recompiling 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 `/opt/mellanox/doca/applications/secure_channel/src` directory.

Recompiling All Applications

The applications are all defined under a single meson project, so the default compilation recompiles all the DOCA applications.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> **Info**
>
> `doca_secure_channel` is created under `/tmp/build/secure_channel/src/`.

Recompiling 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
```

Note

`doca_secure_channel` will be created under `/tmp/build/secure_channel/src/`.

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
```

Info

`doca_secure_channel` is created under `/tmp/build/secure_channel/src/`.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application .

Application Code Flow

1. Parse application argument.
    1. Initialize the arg parser resources and register DOCA general parameters.

        ```
        doca_argp_init();
        ```

    2. Register secure channel application parameters.

        ```
        register_secure_channel_params();
        ```

    3. Parse application parameters:

        ```
        doca_argp_start();
        ```

2. Run main logic.

    ```
    sc_start();
    ```

    1. Initiate synchronization mechanism between send and receive threads.
    2. Initiate Comm Channel endpoint.
    3. Server side starts listening for new connections and client side connects to server.

4. Initiate signal masking and epoll file descriptor.

5. Start send and receive threads. Both threads share the same Comm Channel so each one must "lock" the channel before any send/receive operation.

6. Send thread prints total number of messages successfully sent.

7. Once Ctrl+C is entered in the shell, receive thread prints the total number of messages successfully received.

8. Close and destroy resources.

References

- `/opt/mellanox/doca/applications/secure_channel/src`
- `/opt/mellanox/doca/applications/secure_channel/bin/sc_params.json`

NVIDIA DOCA Simple Forward VNF Application Guide

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

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.

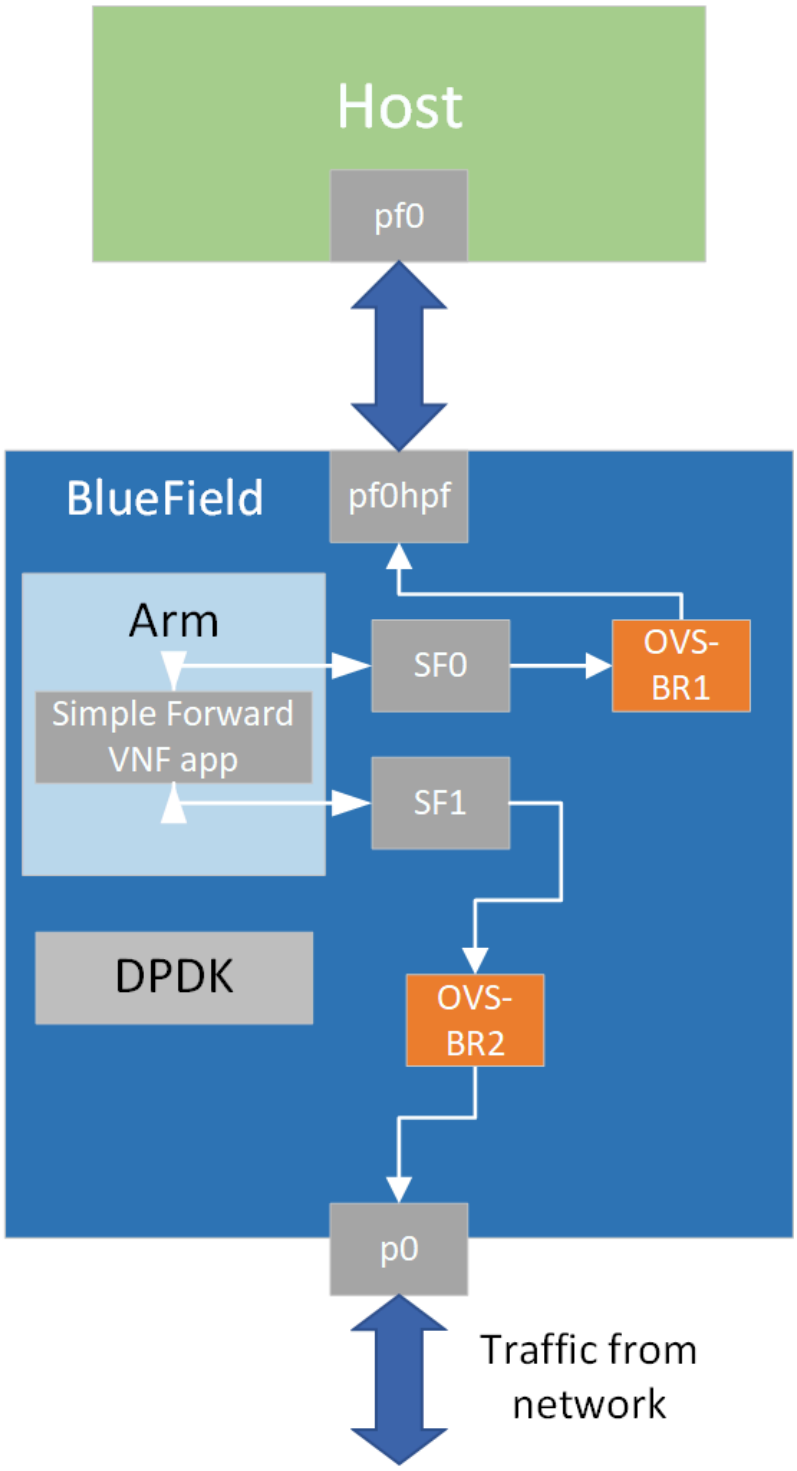
For every packet received on an RX queue on a given port, DOCA Simple Forward checks the packet's key, which consists of a 5-tuple. If it finds that the packet matches an existing flow, then it does not create a new one. Otherwise, a new flow is created with a FORWARDING component. Finally, the packet 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.

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.



Application Architecture

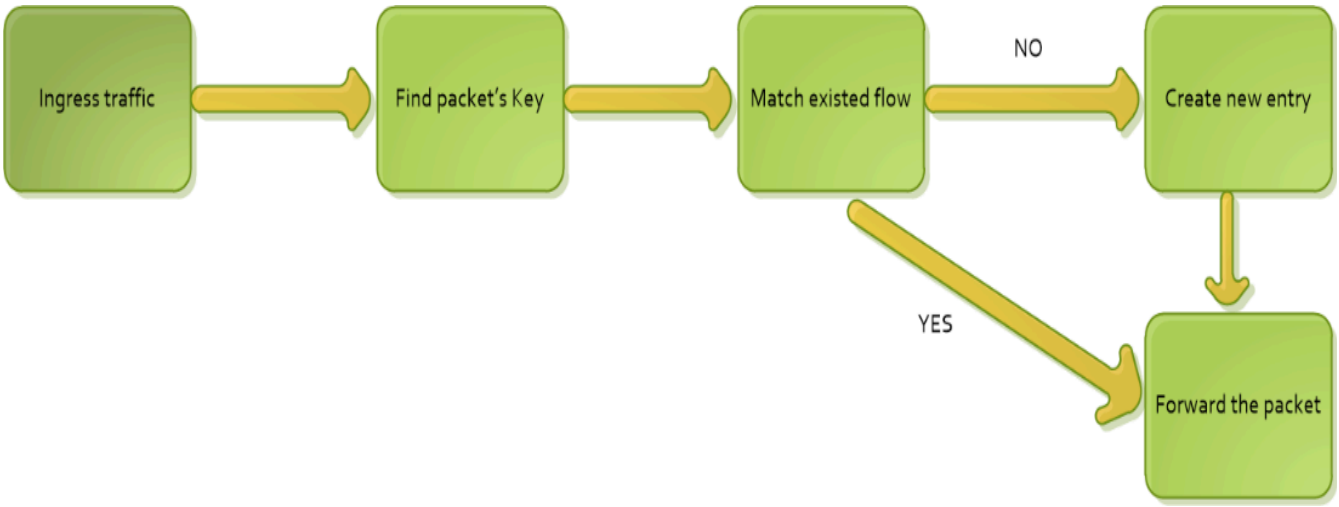
Simple forward first initializes DPDK, after which the application handles the incoming packets.

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.

DOCA Libraries

This application leverages the following DOCA library:

- DOCA Flow

Refer to its respective programming guide for more information.

Running the Application

Installation

Refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

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.

    1. Set FLEX profile number to 3 from the DPU.

    ```
    sudo mlxconfig -d <pcie_address> s
    FLEX_PARSER_PROFILE_ENABLE=3
    ```

    2. Reset the firmware from the host side by performing graceful shutdown and power cycling.

    ```
    ipmitool power cycle
    ```

    Info

    Resetting the firmware can be done from the DPU 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
```

> **Note**
>
> 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. In such cases, the guiding principal is to allocate 4GB of RAM, and instead of allocating 2048 pages, one should allocate the matching amount (8 pages):
>
> ```
> sudo echo 8 > /proc/sys/vm/nr_hugepages
> ```

Application Execution

The simple forward application is provided in both source and binary forms. The binary is located under `/opt/mellanox/doca/applications/simple_fwd_vnf/bin/doca_simple_fwd_vn`.

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=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:
  -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
```

Info

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
/opt/mellanox/doca/applications/simple_fwd_vnf/bin/doca_s
-- -h
```

Info

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the BlueField:

```
/opt/mellanox/doca/applications/simple_fwd_vnf/bin/doca_simple
-a auxiliary:mlx5_core.sf.4 -a auxiliary:mlx5_core.sf.5 -- -l
60
```

Note

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.

Note

The flag `-a auxiliary:mlx5_core.sf.4 -a auxiliary:mlx5_core.sf.5` is mandatory for proper usage of the application.

1. Modifying this flag results unexpected behavior as only 2 ports are supported.
2. The SF number is arbitrary and configurable.

Note

The SF numbers must match the desired SF devices.

3. CLI example for running the application on the host:

```
/opt/mellanox/doca/applications/simple_fwd_vnf/bin/doca_simple
-a 04:00.3 -a 04:00.4 -- -l 60
```

Note

The device identifiers must match the desired network devices.

> **Info**
>
> 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:

```
cd /opt/mellanox/doca/applications/simple_fwd_vnf/bin
./doca_simple_fwd_vnf --json ./simple_fwd_params.json
```

> **Note**
>
> Before execution, ensure that the used JSON file contains the correct configuration parameters, and especially the PCIe addresses necessary for the deployment.

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} ,]

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	`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`

Info

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications .

Recompiling the Application

In addition to providing the application in binary form, the installation also includes all of the application sources and compilation instructions so as to allow modifying the sources and recompiling 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 `/opt/mellanox/doca/applications/simple_fwd_vnf/src` directory.

Recompiling All Applications

The applications are all defined under a single meson project, so the default compilation recompiles all the DOCA applications.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

Info

`doca_simple_fwd_vnf` is created under `/tmp/build/simple_fwd_vnf/src/`.

Recompiling 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
```

Info

`doca_simple_fwd_vnf` is created under `/tmp/build/simple_fwd_vnf/src/`.

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
```

Info

`doca_simple_fwd_vnf` is created under `/tmp/build/simple_fwd_vnf/src/`.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application .

Application Code Flow

1. Parse application argument.
    1. Initialize arg parser resources and register DOCA general parameters.

```
doca_argp_init();
```

    2. Register application parameters.

```
register_simple_fwd_params();
```

    3. Parse the arguments.

```
doca_argp_start();
```

1. Parse DPDK flags and invoke handler for calling the `rte_eal_init()` function.
2. 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();
```

   1. Initialize DPDK ports.
   2. Create mbuf pool using `rte_pktmbuf_pool_create`.
   3. Driver initialization – use `rte_eth_dev_configure` to configure the number of queues.
   4. Rx/Tx queue initialization – use `rte_eth_rx_queue_setup` and `rte_eth_tx_queue_setup` to initialize the queues.
   5. Rx hairpin queue initialization – use `rte_eth_rx_hairpin_queue_setup` to initialize the queues.
   6. Start the port using `rte_eth_dev_start`.

4. Simple forward initialization.

```
simple_fwd_init();
```

1. `simple_fwd_create_ins` – create flow tables using `simple_fwd_ft_create`.

2. `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();
```

   1. Receive packets using `rte_eth_rx_burst` in a loop.

   2. Process packets using `simple_fwd_process_offload`.

   3. 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();
```

   1. Parse the packet's `rte_mbuf` using `simple_fwd_pkt_info`.

   2. 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.

References

- `/opt/mellanox/doca/applications/simple_fwd_vnf/src`
- `/opt/mellanox/doca/applications/simple_fwd_vnf/bin/simple_fwd_par`

NVIDIA DOCA Switch Application Guide

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

Introduction

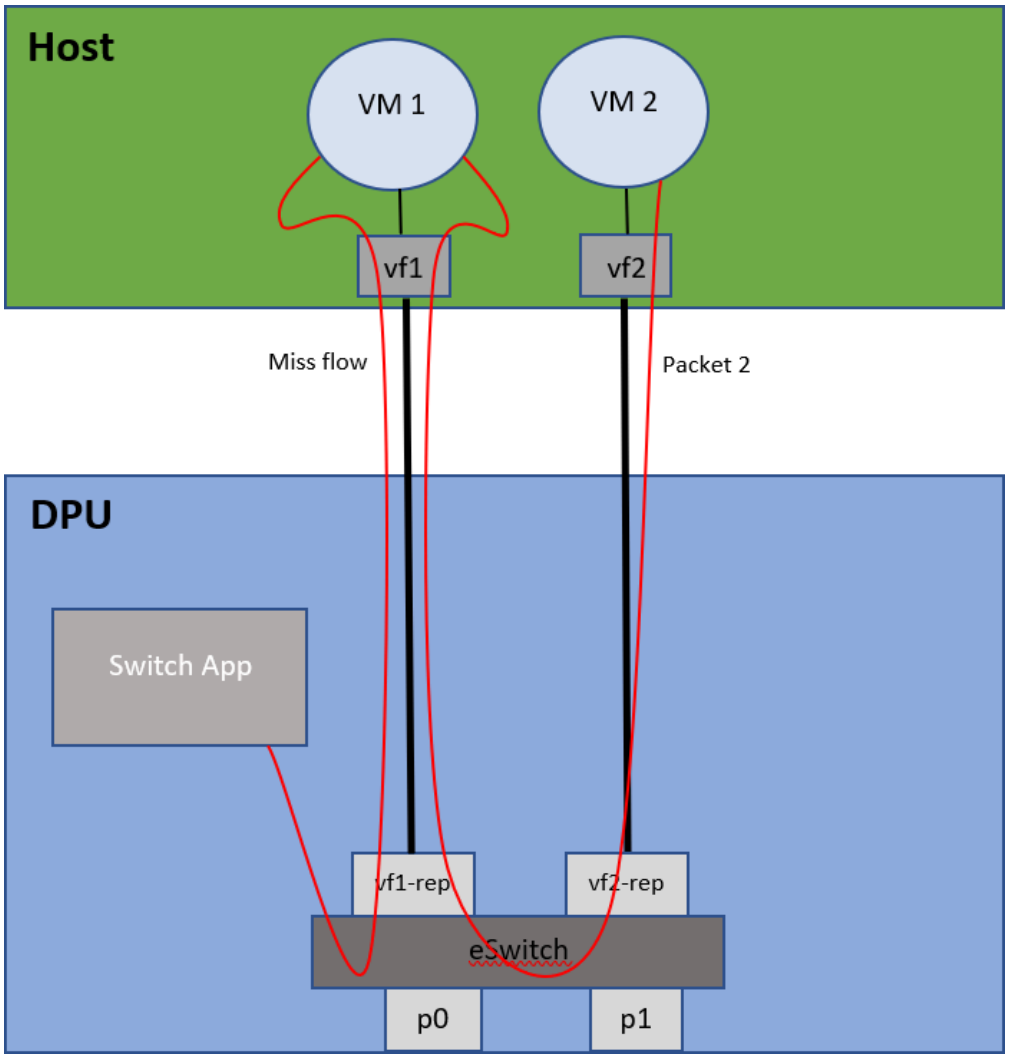
DOCA Switch is a network application that leverages the DPU's hardware capability for internal switching between representor ports on the DPU.

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.

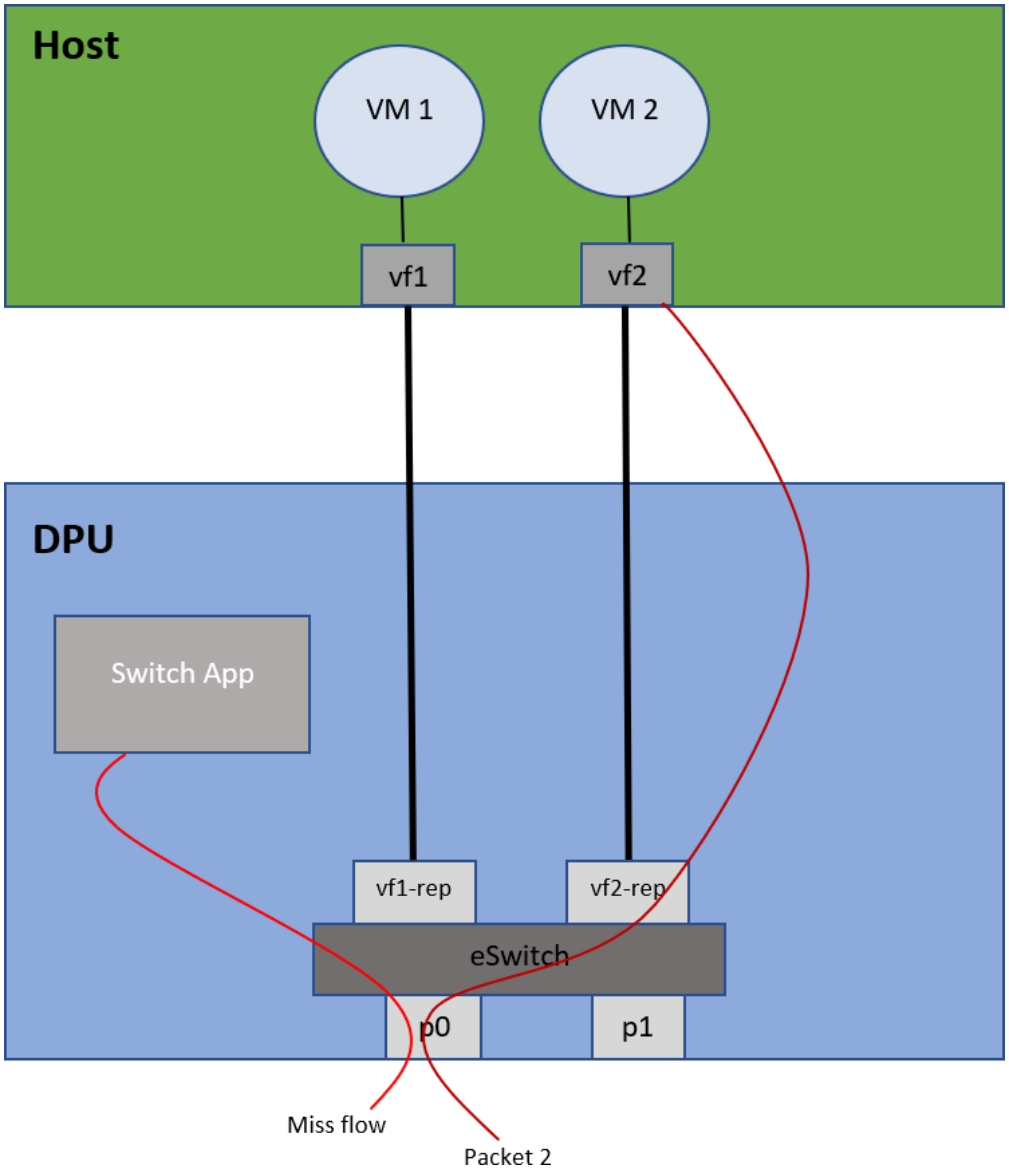
System Design

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:



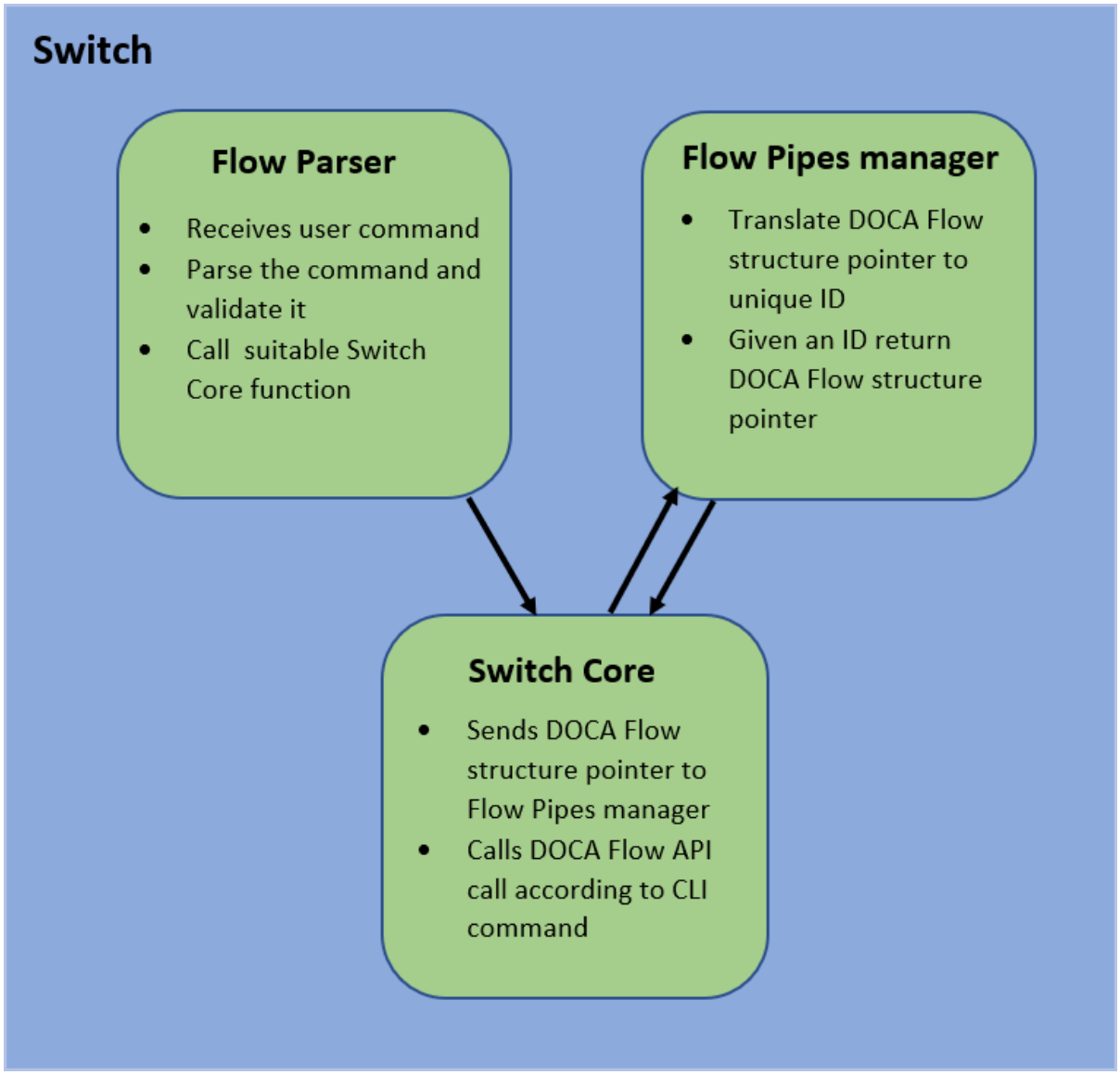
Traffic flow from a physical port to a VM on the host:



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



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

DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Flow

Refer to its respective programming guide for more information.

Running the Application

Installation

Refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

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
```

Note

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. In such cases, the guiding principal is to allocate 4GB of RAM, and instead of allocating 2048 pages, one should allocate the matching amount (8 pages):

```
sudo echo 8 > /proc/sys/vm/nr_hugepages
```

Application Execution

The switch application is provided in both source and binary forms, and the binary is located under `/opt/mellanox/doca/applications/switch/bin/doca_switch`.

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
```

Info

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
/opt/mellanox/doca/applications/switch/bin/doca_switch
-- -h
```

Info

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the BlueField:

```
/opt/mellanox/doca/applications/switch/bin/doca_switch -a
03:00.0,representor=[0-2],dv_flow_en=2 -- -l 60
```

Note

`dv_flow_en=2` is necessary to run the application with hardware steering.

Note

The PCIe address (`03:00.0`) should match the address of the desired PCIe device.

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`
	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

Info

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

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`	

Fields	Field Options
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	
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

Fields	Field Options
`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:

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

Recompiling the Application

In addition to providing the application in binary form, the installation also includes all of the application sources and compilation instructions so as to allow modifying the sources and recompiling the application. For more information about the applications, as well as development and compilation tips, refer to the DOCA Applications main guide.

The sources of the application can be found under the `/opt/mellanox/doca/applications/switch/src` directory.

Recompiling All Applications

The applications are all defined under a single meson project, so the default compilation recompiles all the DOCA applications.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

Info

`doca_switch` is created under `/tmp/build/switch/src/`.

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
```

Info

`doca_switch` is created under `/tmp/build/switch/src/`.

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
```

Info

`doca_switch` is created under `/tmp/build/switch/src/`.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application .

Application Code Flow

1. Parse application argument.
    1. Initialize the arg parser resources and register DOCA general parameters.

```
doca_argp_init();
```

    2. Register application parameters.

```
register_switch_params();
```

    3. Parse app parameters.

```
doca_argp_start();
```

2. Count total number of ports.

```
switch_ports_count();
```

1. 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();
```

1. Initialize DOCA Flow.
2. Create port pairs.
3. Create Flow Pipes Manger module.
4. Register an action for each relevant CLI command.

5. Initialize Flow Parser.

```
flow_parser_init();
```

1. Reset all internal Flow Parser structures.
2. Start the command line interface.
3. Receive user commands, parse them, and call the required DOCA Flow API command.
4. Close the interactive shell once a "quit" command is entered.

6. Clean Flow Parser resources.

```
flow_parser_cleanup();
```

7. Destroy Switch resources.

```
switch_destroy();
```

    1. 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();
```

    1. Call `rte_eal_destroy()` to destroy initialized EAL resources.

11. Arg parser destroy.

```
doca_argp_destroy();
```

References

- `/opt/mellanox/doca/applications/switch/src`

NVIDIA DOCA YARA Inspection Application Guide

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

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).

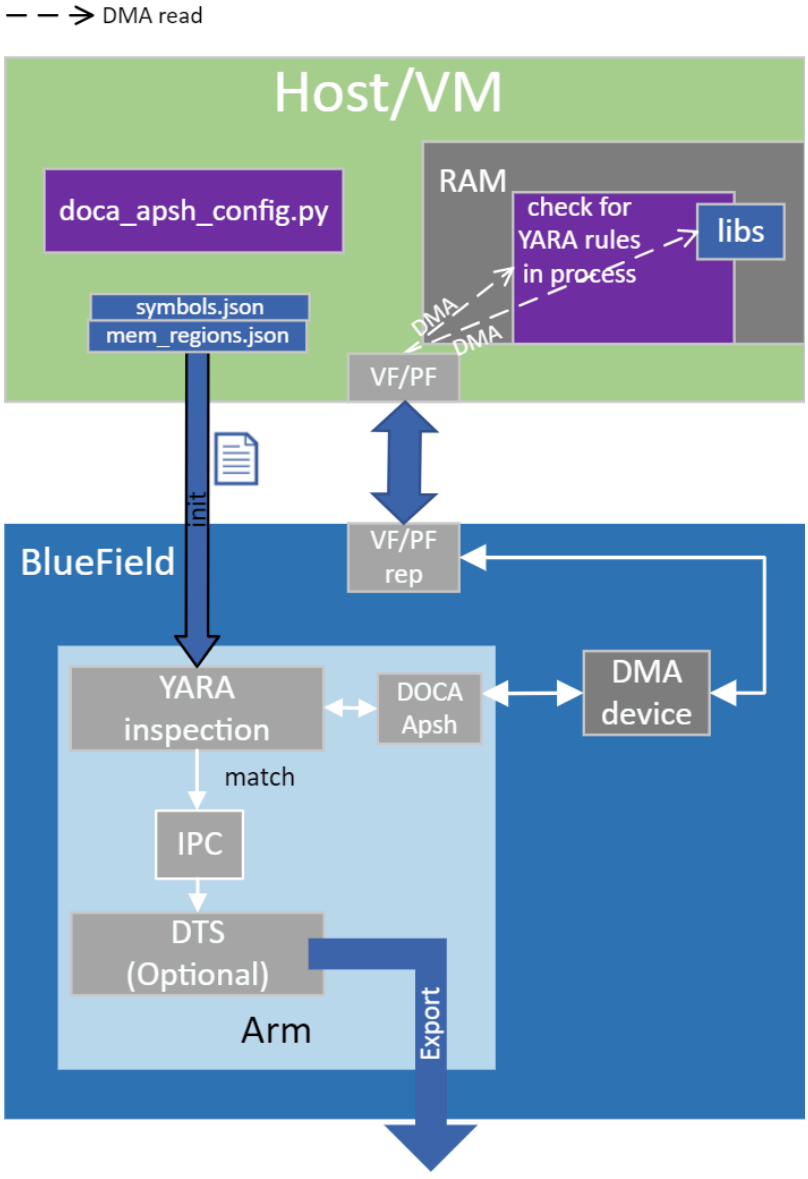
This guide describes how to build YARA inspection using the DOCA App Shield library which leverages DPU abilities such as hardware-based DMA, integrity, and more.

> **Note**
>
> As the DOCA App Shield library only supports the YARA API for Windows hosts, this application can only be used to inspect Windows hosts.

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.

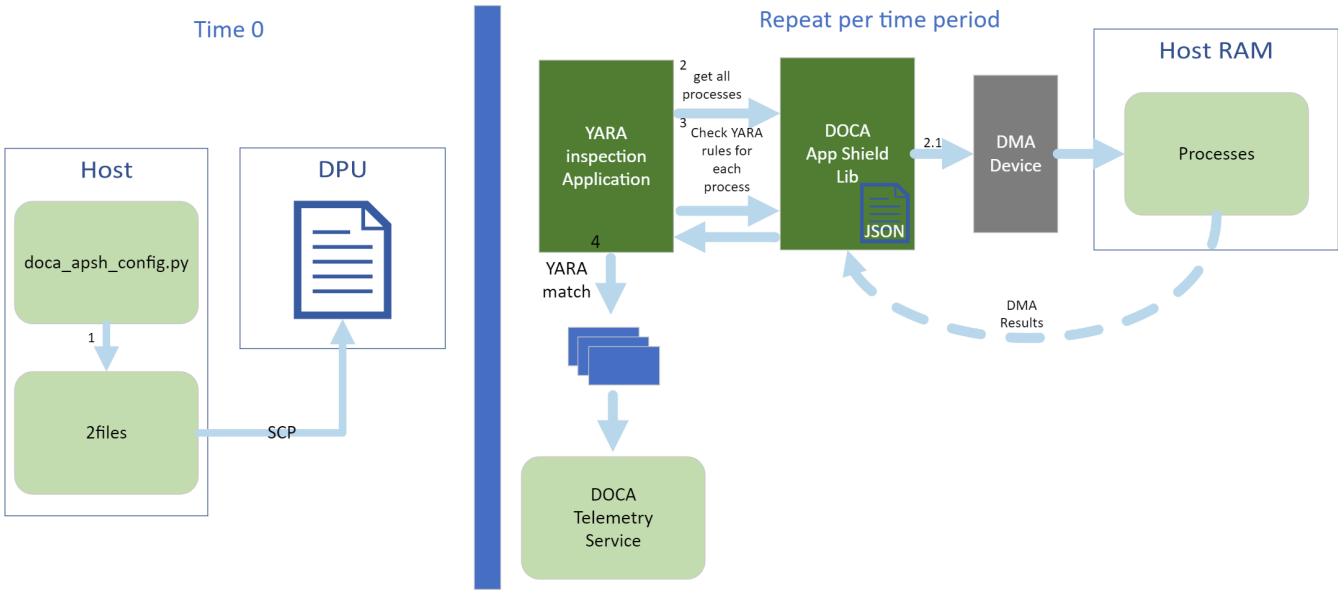


Application Architecture

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:
    1. The YARA inspection app requests a list of all apps from the DOCA App Shield library.
    2. The app loops over all processes and checks for YARA rules match using the DOCA App Shield library.
    3. 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

> **Note**
>
> 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.

DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA App Shield
- 2024-10-09_07-10-18_DOCA Telemetry

Refer to their respective programming guide for more information.

Limitations

- The application is only available on Ubuntu 22.04 environments
- The application only supports the inspection of Windows hosts

Running the Application

Installation

Refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

Prerequisites

1. Configure the BlueField's firmware

    1. 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
        ```

    2. Perform a graceful shutdown and cold boot from the host. Run:

        ```
        host> ipmitool power cycle
        ```

        Info

        These configurations can be checked 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).

Note

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=0x2a0aff8 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:

> **Note**
>
> 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.

    1. 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
```

2. 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.

> **Note**
>
> If the kernel and process `.exe` have not changed, there no need to redo this step.

Application Execution

The YARA inspection application is provided in both source and binary forms, and the binary is located under `/opt/mellanox/doca/applications/yara_inspection/bin/doca_yara_inspect`.

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=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:
  -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
```

Info

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
/opt/mellanox/doca/applications/yara_inspection/bin/doca_
-h
```

> **Info**
>
> For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the BlueField:

```
/opt/mellanox/doca/applications/yara_inspection/bin/doca_yara_
-m mem_regions.json -o symbols.json -f
MT2125X03335MLNXS0D0F0VF1 -d mlx5_0 -t 3
```

> **Note**
>
> All used identifiers (`-f` and `-d` flags) should match the identifier of the desired devices.

Command Line Flags

Flag Type	Short Flag	Long Flag	Description
General flags	`h`	`help`	Prints a help synopsis
	`v`	`version`	Prints 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`	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

Flag Type	Short Flag	Long Flag	Description
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`.
			Note
			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

Flag Type	Short Flag	Long Flag	Description
			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

Info

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications .

Recompiling the Application

In addition to providing the application in binary form, the installation also includes all of the application sources and compilation instructions so as to allow modifying the sources and recompiling 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 `/opt/mellanox/doca/applications/yara_inspection/src` directory.

Recompiling All Applications

The applications are all defined under a single meson project, so the default compilation recompiles all the DOCA applications.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

Info

`doca_yara_inspection` is created under `/tmp/build/yara_inspection/src/`.

Recompiling 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
```

Info

`doca_yara_inspection` is created under `/tmp/build/yara_inspection/src/`.

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
```

Info

`doca_yara_inspection` is created under `/tmp/build/yara_inspection/src/`.

Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application .

Application Code Flow

1. Parse application argument.
    1. Initialize arg parser resources and register DOCA general parameters.

        ```
        doca_argp_init();
        ```

    2. Register application parameters.

        ```
        register_apsh_params();
        ```

    3. Parse the arguments.

        ```
        doca_argp_start();
        ```

2. Initialize DOCA App Shield lib context.
    1. Create lib context.

        ```
        doca_apsh_create();
        ```

    2. Set DMA device for lib.

```
open_doca_device_with_ibdev_name();
doca_apsh_dma_dev_set();
```

3. Start the context

```
doca_apsh_start();
apsh_system_init();
```

3. Initialize DOCA App Shield lib system context handler.

    1. Get the representor of the remote PCIe function exposed to the system.

    ```
    open_doca_device_rep_with_vuid();
    ```

    2. 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();
```

    1. Initialize a new telemetry schema.

2. Register YARA type event.
3. Set up output to file (in addition to default IPC).
4. Start the telemetry schema.
5. Initialize and start a new DTS source with the `gethostname()` name as source ID.

5. Loop until YARA rule is matched.
    1. Get all processes from the host.

```
doca_apsh_processes_get();
```

    2. 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_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();
```

References

- `/opt/mellanox/doca/applications/yara_inspection/src`

DOCA Tools

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

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.

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

Tools

Comm Channel Admin Tool

CLI name: `doca_comm_channel_admin_tool`

The Comm Channel Admin Tool is used to monitor Comm Channel services and connections on both BlueField and the host.

DPA EU Management Tool

CLI name: `dpaeumgmt`

The DPA execution unit management 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.

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.

FlexIO Build

CLI name: `build_flexio_device.sh`

The FlexIO Build tool is used to build and compile FlexIO device code into a static library.

It is designed to generate a host library that encapsulating DPA execution. This tool relies on DPACC.

PCC Counter

CLI name: `pcc_counters.sh`

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.

Socket Relay

CLI name: `doca_socket_relay`

DOCA Socket Relay 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.

NVIDIA DOCA Comm Channel Admin Tool

This document describes the Comm Channel Admin Tool, used to monitor Comm Channel services, connections, etc.

Introduction

Comm Channel Admin Tool is used to monitor Comm Channel services and connections on both the DPU and the host. For more information on the DOCA Comm Channel library, refer to the DOCA Comm Channel – Deprecated.

Prerequisites

NVIDIA® BlueField®-2 firmware version 24.35.1012 or higher.

Description

On the DPU, Comm Channel Admin Tool can show the user which services are up alongside additional information about them:

- `service_name` – the name of the service
- `transport_type` – UD or DC
- `allowed_vhca_id` – which `vhca_id` is allowed to connect to this service
- `service_qpn_or_dct` –
    - UD transport type – indicates QP number of the service

    - DC transport type – indicates DC number of the service

- `num_connected_client` – indicates the number of current connected clients
- `max_num_connected_client` – indicates the maximum number of connected clients

Furthermore, users may query specific services according to the service name to get information about the connected clients. For every connected client the tool shows:

- `connection_id` – unique identifier for the connection
- `qpn` – QP number (or DCT) used for the connection

On the host, the tool can show the user the name of the active services on the DPU. Furthermore, it can show information about the active connections between the host and the DPU:

- `service_name` – the name of the service
- `transport_type` – UD or DC
- `my_qpn_or_dci` –
    - UD transport type – indicates QPN of the connection initiator
    - DC transport type – the field indicates the DCI of the connection initiator
- `connection_type` – `CONNECT_BY_SERVICE_ID` or `CONNECT_BY_VHCA_ID`
- `dst_vhca_id` – `vhca_id` of the target side of the connection. Valid only when the connection type is `CONNECT_BY_VHCA_ID`.

Execution

To execute the DOCA Comm Channel Admin Tool, run:

```
/opt/mellanox/doca/tools/doca_comm_channel_admin_tool
```

Afterwards, users get an interactive CI that awaits a command:

```
Comm-Channel-Admin-Tool >>
```

The commands and their flags can be obtained by writing help:

```
Comm-Channel-Admin-Tool >> help
Comm-Channel-Admin-Tool CLI Commands:
     service, s          To be used on the DPU
         --all, -a                              Show all
services
         --service-name, -sn [service_name]     Show specific
service and its connected clients
     connection, c       To be used on the host
         --all, -a                              Show all
connections
         --service-name, -sn [service_name]     Show specific
connection according to service name

     discovery, d        To be used on the host
         --all, -a                              Show all active
services
         --service-name, -sn [service_name]     Show specific
active service according to service name

     quit            Exit tool
```

On the DPU, to see all services, users may use the `service` (or `s`) command with the flag `--all` (or `-a`). For example:

```
Comm-Channel-Admin-Tool >> service --all
2022-09-28 09:54:28,410 - Comm-Channel-Admin-Tool - INFO - On
/dev/mst/mt41686_pciconf0:
Active services:
+----------------------+------------------+------------------+-
--------------------+-----------------------+-----------------
----------+
| service_name          | transport_type   |   allowed_vhca_id |
service_qpn_or_dct |   num_connected_client |
max_num_connected_client |
+======================+==================+==================+===
| secure_channel_server | DC               |                0 |
4548 |                     1 |                        512 |
+----------------------+------------------+------------------+-
--------------------+-----------------------+-----------------
----------+
2022-09-28 09:54:28,886 - Comm-Channel-Admin-Tool - INFO - On
/dev/mst/mt41686_pciconf0.1:
No active services
```

To query a specific service and see its connected clients, users may use the `--serivce-name` (or `-sn`) flag while providing the `service_name`. For example:

```
Comm-Channel-Admin-Tool >> service --service-name
secure_channel_server
2022-09-28 09:56:16,335 - Comm-Channel-Admin-Tool - INFO - On
/dev/mst/mt41686_pciconf0:
+----------------------+-----------------+------------------+-
--------------------+-----------------------+-----------------
----------+
| service_name         | transport_type   |   allowed_vhca_id |
service_qpn_or_dct |   num_connected_client |
max_num_connected_client |
+======================+=================+==================+===
| secure_channel_server | DC              |                0 |
4548 |                     1 |                        512 |
+----------------------+-----------------+------------------+-
--------------------+-----------------------+-----------------
----------+

Connected clients:
+-----------------+-----------------+
|  connection_id  | transport_type   |
+=================+=================+
|               0 |            4547 |
+-----------------+-----------------+
2022-09-28 09:56:16,809 - Comm-Channel-Admin-Tool - INFO - On
/dev/mst/mt41686_pciconf0.1:
No active service by name secure_channel_server
```

On the host, to see all active services, users may use the `discovery` (or `d`) command with `--all` flag (or `-a`). For example:

```
Comm-Channel-Admin-Tool >> discovery --all
2022-09-28 12:58:42,201 - Comm-Channel-Admin-Tool - INFO - On
/dev/mst/mt41686_pciconf0:
Active services:
+----------------------+
| service_name         |
+======================+
| secure_channel_server |
+----------------------+
2022-09-28 12:58:42,632 - Comm-Channel-Admin-Tool - INFO - On
/dev/mst/mt41686_pciconf0.1:
No active services
```

Users may also filter to show only specific services according to the `service_name` by using the `--serivce-name` (or `-sn`) flag while providing the `service_name`.

On the host, to see all active connections, users may use the `connection` command (or `c`) with `--all` flag (or `-a`). For example:

```
Comm-Channel-Admin-Tool >> connection --all
2022-09-28 13:01:54,420 - Comm-Channel-Admin-Tool - INFO - On
/dev/mst/mt41686_pciconf0:
Active connections:
+----------------------+-----------------+----------------+---
-------------------+--------------+
| service_name         | transport_type   |   my_qpn_or_dci |
connection_type      |   dst_vhca_id |
+======================+=================+================+====
| secure_channel_server | DC               |              71 |
CONNECT_BY_SERVICE_ID |            0 |
+----------------------+-----------------+----------------+---
-------------------+--------------+
2022-09-28 13:01:54,860 - Comm-Channel-Admin-Tool - INFO - On
/dev/mst/mt41686_pciconf0.1:
No active Connections
```

Users may also filter to show only specific connections according to `service_name` by using the `--serivce-name` (or `-sn`) flag while providing the `service_name`.

Note

If a new service or connection is added, there is no need to restart the tool. After each command, the tool always shows the latest services and connections.

NVIDIA DOCA PCC Counter Tool

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

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.

Prerequisites

DOCA 2.2.0 and higher.

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.

> **Info**
>
> Refer to NVIDIA BlueField DPU 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

Execution

To use the PCC Counter:

1. Initialize all supported hardware counters. Run:

```
sudo ./pcc_counters.sh set /dev/mst/mt41692_pciconf0
```

> **Info**
>
> Counters are zeroed after each `set` command.

2. Query all supported hardware counters. Run:

```
sudo ./pcc_counters.sh query /dev/mst/mt41692_pciconf0
```

> **Info**
>
> 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
```

NVIDIA DOCA Socket Relay

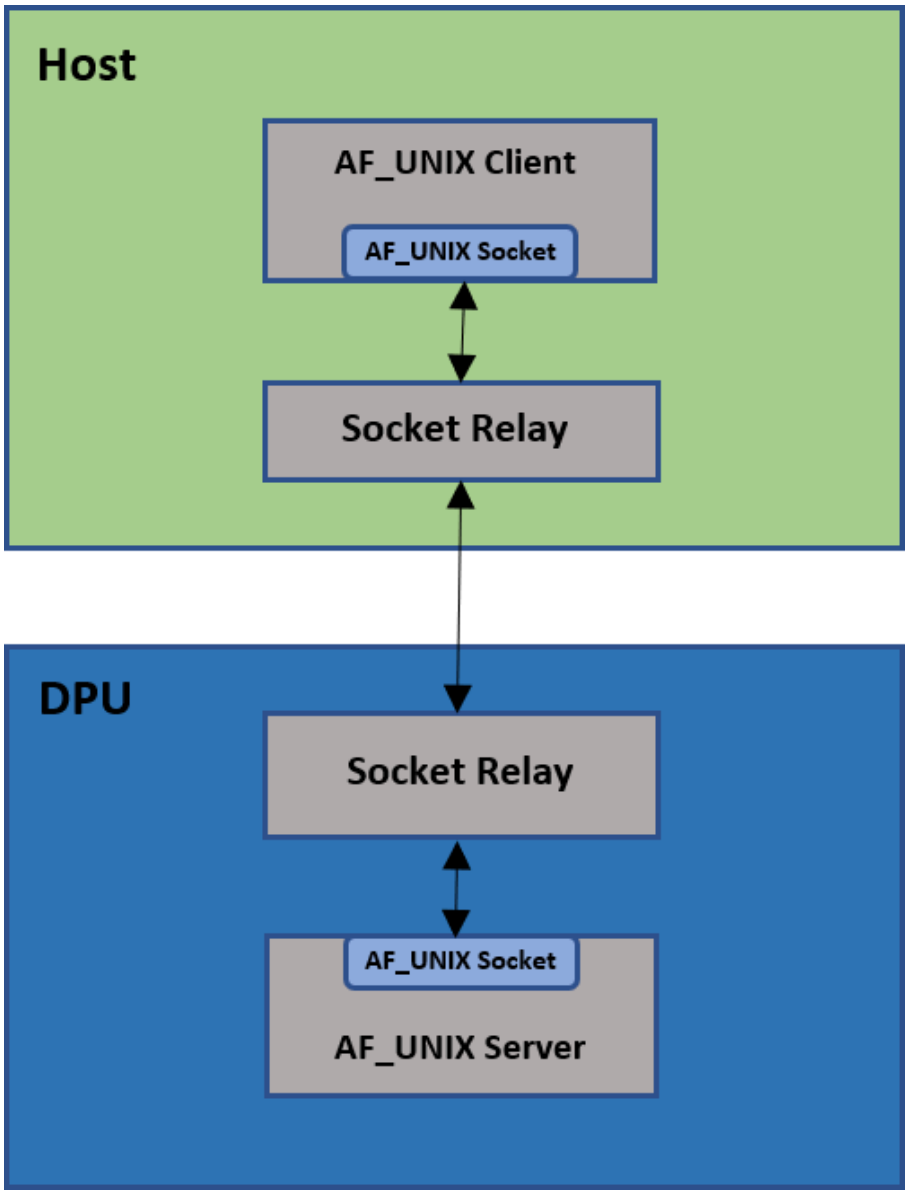
This document describes DOCA Socket Relay architecture, usage, etc.

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 Comm Channel.

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

Prerequisites

Windows 10 build 17063 is the minimal Windows version to run DOCA Socket Relay on a Windows host.

Dependencies

NVIDIA® BlueField®-2 firmware version 24.35.1012 or higher.

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
```

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
	`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

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
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. **Note** This flag is mandatory.	`"socket" : "/tmp/uds-server.socket"`
	`n`	`cc-name`	Comm Channel service name **Note** This flag is mandatory.	`"cc-name" : sr_channel`
	`p`	`pci-addr`	DOCA Comm Channel device PCIe address **Note** This flag is mandatory.	`"pci-addr" : b1 : 00.1`

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
	`r`	`rep-pci`	DOCA Comm Channel device representor PCIe address **Note** This flag is available and mandatory only on the DPU.	`"rep-pci" : b1 : 02.2`

NVIDIA DOCA Ngague

NOTE THAT THIS CONFLUENCE PAGE IS NOT READY YET!

Contents:

This document provides instructions on the usage of the ngague tool.

Introduction

ngauge is a tool for probing NIC HW counters, and storing them in an HDF5 format , together with the relevant metadata, for later processing. In addition, the progress and measurement summary are displayed graphically, on a CLI.

Supported hardware are BlueField-3, ConnectX-7, and above.

Prerequisites

NVIDIA® BlueField®-3, ConnectX®-7, and above with firmware version xx.43.1000 or higher, and fwctl driver.

Info

To install the fwctl driver (for host only, for DPU it's already installed), search for a package with "fwctl" and install the package you find.

On deb-based distros, use `apt-cache search fwctl` and for RPM-based distros use `dnf search fwctl`

NOTE: On Ubuntu **20.04** the `fwctl` driver is not loaded automatically, and one needs to `modprobe mlx5_fwctl` after every reboot.

Description

All the configurations are done in the input YAML file.

Start by copying a sample configuration from `/usr/share/doc/ngauge/examples/settings.`

The device to run on should be configured as the PCI address (*e.g.* 0000:03:00.0):

```
device: "0000:03:00.0"
```

The data output path is configured like so (path and prefix to the output file - both are **mandatory**):

```
output:
  path: /path/to/output/directory
  prefix: "ngauge_data_"
```

In the example above, the output will be saved like so: `/path/to/output/directory/ngauge_data_<DATE>_<TIME>.h5`. The explicit output name will be printed after each run.

Run parameters (the most useful of them is the sampling period!) are configured like so:

```
params:
  mode: repetitive  # [repetitive, single]
  period_us: 1e2
```

In the example above, "1e2" means 100 µs. Numbers in decimal or scientific notation are accepted.

The counters to measure are configured like so. The only mandatory configuration for a counter is the Data ID. All the other configurations are optional.

```
counters:
  - id: 0x1020000100000000
    desc: RX bytes port 0
    unit: RX port
    accumulating: false
    normalizer: time  # Normalizer, if present, must be either 'time'
or a number.
```

You can find all supported performance counters in this link: Supported Data IDs

Tip

You may want to install **doca-telemetry-utils** - a tool which can generate counter IDs to be used to configure ngauge. Do it like so: `sudo apt-get install doca-telemetry-utils` or `sudo dnf install doca-telemetry-utils`.

Then run `doca_telemetry_utils -h` for help, and `doca_telemetry_utils get-counters` to get the list of available counters.

Parsing output

A sample plugin, named *simple-plot*, will be installed in `/usr/share/doc/ngauge/examples/plugins`.

This plugin is a basic demonstration of how you can open the output HDF5 file with the data in it and plot it. Besides plotting, many types of analyses can be done on these data. The sample plugin is just a rudimentary demonstration.

```
Usage: /usr/share/doc/ngauge/examples/plugins/simple_plot.py
<ngauge output .h5 file> <counter ID> [<counter ID> ...]
```

Tip

If you only want to plot the results of the very last run, and your output directory is `/tmp` (the default) then you can use this expression to always represent the last results, instead of copy-pasting the file name every time): `"$(ls -1 /tmp/ngauge_data_*.h5 | tail -n1)"` .

Info

The sample plugins are just examples, and should not be considered integral parts of the ngauge tool. therefore you may need to install the dependencies to tun them, such as NumPy, H5py, Matplotlib, plotext, and others separately.

Execution

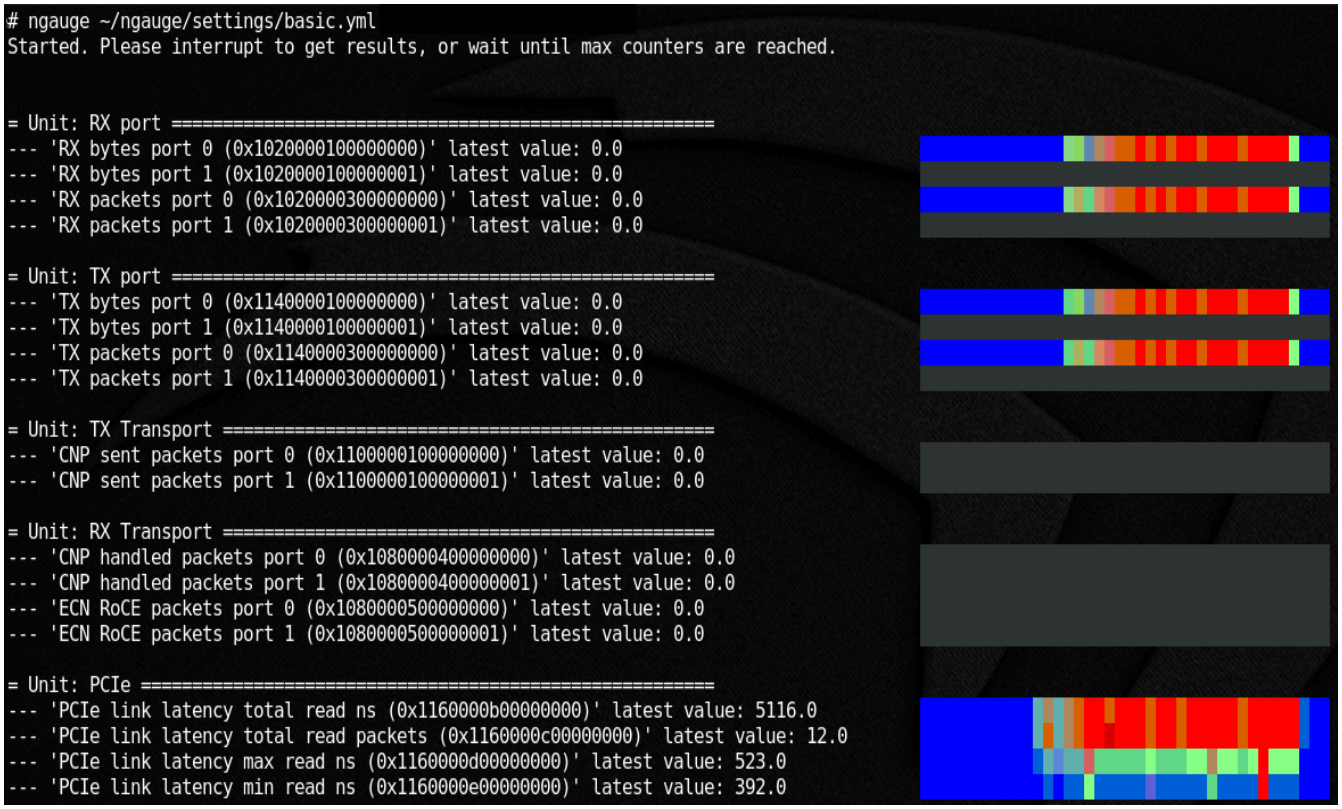
To run ngague:

```
Usage: ngague <configuration YAML file>
```

The output will be saved in an HDF5 file (.h5) in the path that you have specified in the configuration YAML.

During the run you will see progress bars for each counter, just as in the image below. The colors symbolize the following:

- Blue - low values (relatively to the other values of the same counter).
- Red - high values (relatively to the other values of the same counter).
- Any color between blue and red - intermediate values.
- Solid gray bars mean that the values of this counter did not change at all during the run.



DOCA Services

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

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.

For information on the deployment of the services, refer to the NVIDIA BlueField DPU Container Deployment Guide.

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.

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.

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.

Profiling

As mentioned in the NVIDIA BlueField DPU 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.

Services

Container Deployment

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

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.

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).

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.

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 E-VPN extension to enable multi-tenant cloud.

At its core, HBN is the Linux networking acceleration driver of the DPU, Netlink to a DOCA daemon which seamlessly accelerates Linux networking using hardware programming APIs.

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.

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

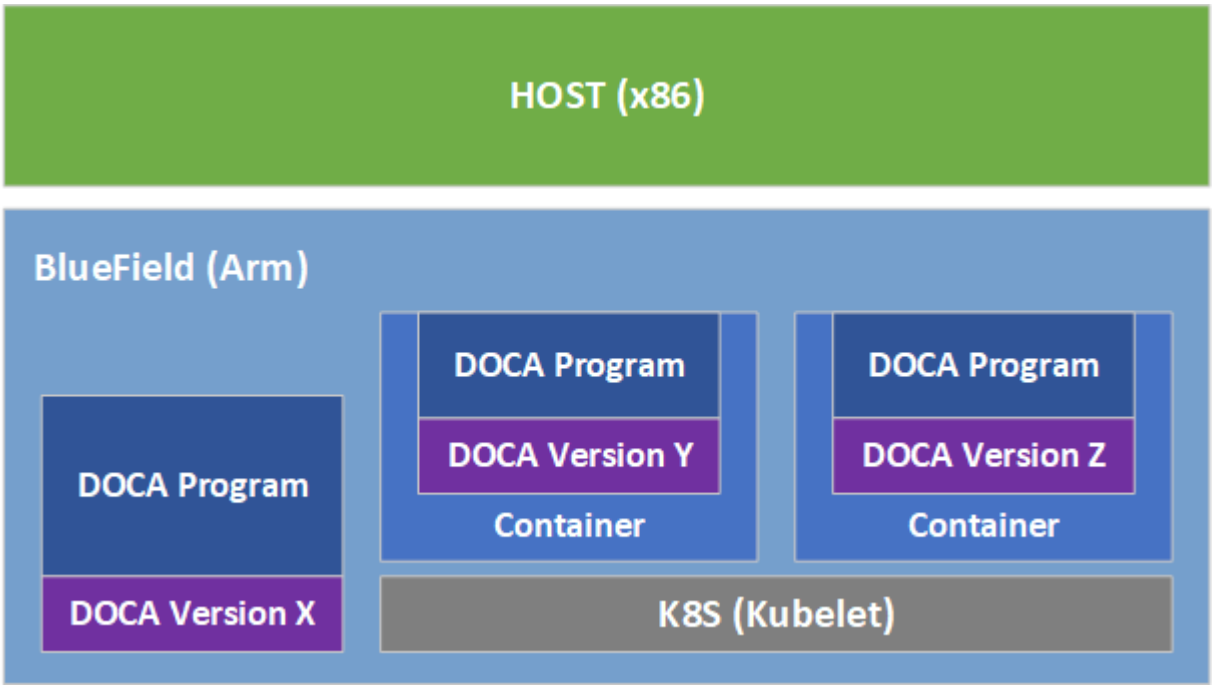
NVIDIA BlueField DPU Container Deployment Guide

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

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.

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

> **Note**
>
> Container deployment based on **standalone** Kubelet, as presented in this guide, is currently in **alpha version** and is subject to change in future releases.

Container Deployment

Deploying containers on top of the BlueField DPU requires the following setup sequence:

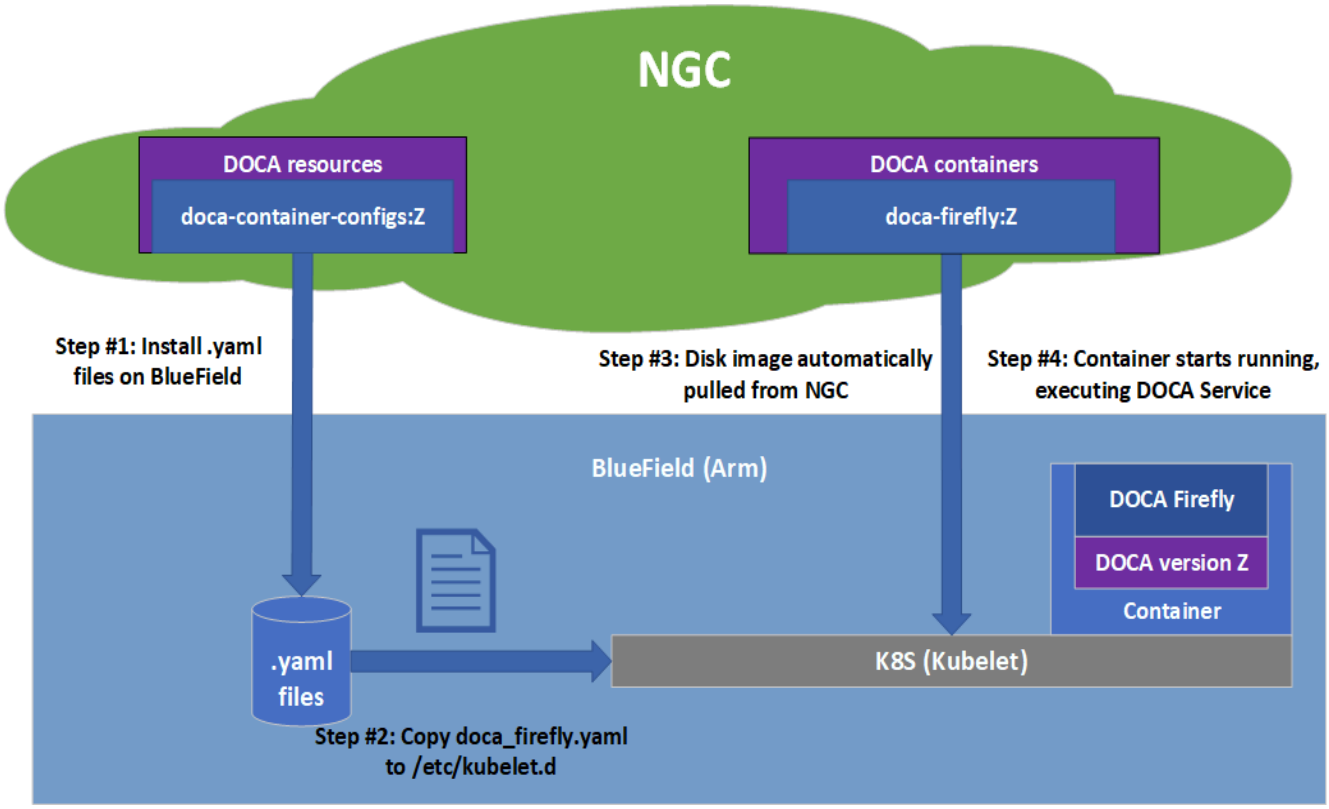
1. Pull the container `.yaml` configuration files.

2. Modify the container's `.yaml` configuration file.

3. Deploy the container. The image is automatically pulled from NGC.

Some of the steps only need to 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 application recognition (AR) container as an example.



Pull Container YAML Configurations

> **Note**
>
> 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.

Pulling the latest resource version can be done using the following command:

```
# Pull the entire resource as a *.zip file
wget --content-disposition
https://api.ngc.nvidia.com/v2/resources/nvidia/doca/doca_container_
-O doca_container_configs_2.5.0v1.zip
# Unzip the resource
unzip -o doca_container_configs_2.5.0v1.zip -d
doca_container_configs_2.5.0v1
```

More information about additional versions can be found in the NGC resource page.

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.

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`).

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.

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
```

> **Info**
>
> 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.

Info

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                     3.2
2a060e2e7101d       251kB
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
```

Info

For additional information and guides on using `crictl`, refer to the Kubernetes documentation.

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>
```

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.

> **Info**
>
> If more troubleshooting is required, refer to the matching section in the Troubleshooting Guide.

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.

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:
    1. Huge pages are allocated as required per container.
    2. Both the amount and size of pages match the requirements precisely.
2. Once new 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`).

Advanced Troubleshooting

Manual Execution from Within Container - Debugging

> **Note**
>
> 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.

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

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
```

Note

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>
```

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
```

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_2.tar
    enable_offline_containers.sh
```

This image is imported by default during boot as part of the automatic activation of DOCA Telemetry Service (DTS).

> **Note**
>
> 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.

In versions prior to DOCA 4.2.0, this image can be pulled and imported as follows:

- Exporting the image:

```
docker pull k8s.gcr.io/pause:3.2
docker save k8s.gcr.io/pause:3.2 > docker_pause_3_2.tar
```

- Importing the image:

```
ctr --namespace k8s.io image import docker_pause_3_2.tar
crictl images
IMAGE                                TAG                 IMAGE
ID            SIZE
k8s.gcr.io/pause                     3.2
2a060e2e7101d       487kB
```

DOCA Services for Host

A subset of the DOCA services are 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.

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):

    1. Visit the NGC page of the desired container.

    2. Under the "Tags" menu, select the desired tag and click the paste icon so it is copied to the clipboard.

    3. 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
    ```

    Note

    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:

    1. 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
    ```

    > **Info**
    >
    > For more information, refer to Docker's official documentation.

    2. The specific deployment command for each DOCA service is listed in their respective deployment guide.

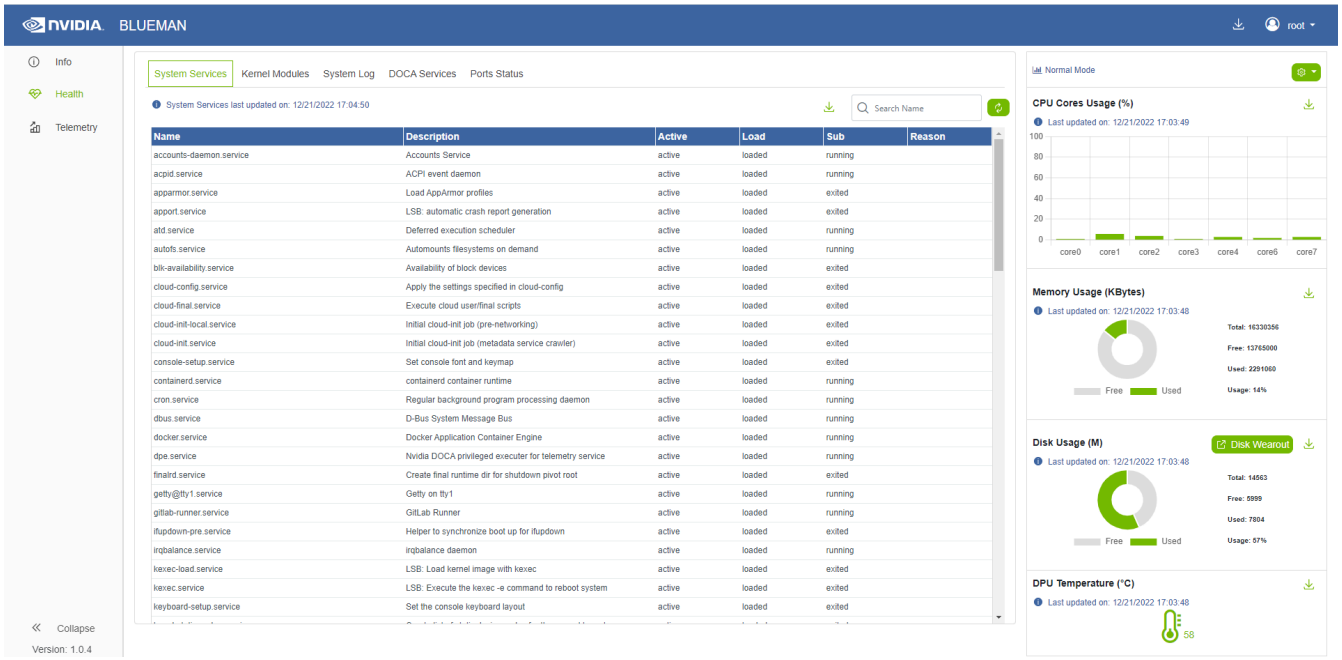
NVIDIA DOCA BlueMan Service Guide

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

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.



Requirements

- BlueField image version 3.9.3.1 or higher
- DTS and the DOCA Privileged Executer (DPE) daemon must be up and running

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
```

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
```

Service Deployment

For information about the deployment of DOCA containers on top of the BlueField DPU, refer to the NVIDIA DOCA Container Deployment Guide.

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.

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
```

Enabling BlueMan Service

Using Script

Run `bring_up_doca_blueman_service.sh`:

```
$ chmod +x
/opt/mellanox/doca/services/blueman/bring_up_doca_blueman_service.s
$
/opt/mellanox/doca/services/blueman/bring_up_doca_blueman_service.s
```

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/
```

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
```

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

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.



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."

    1. Restart the DPE daemon:

        ```
        $ systemctl restart dpe.service
        ```

    2. 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, v erify 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.

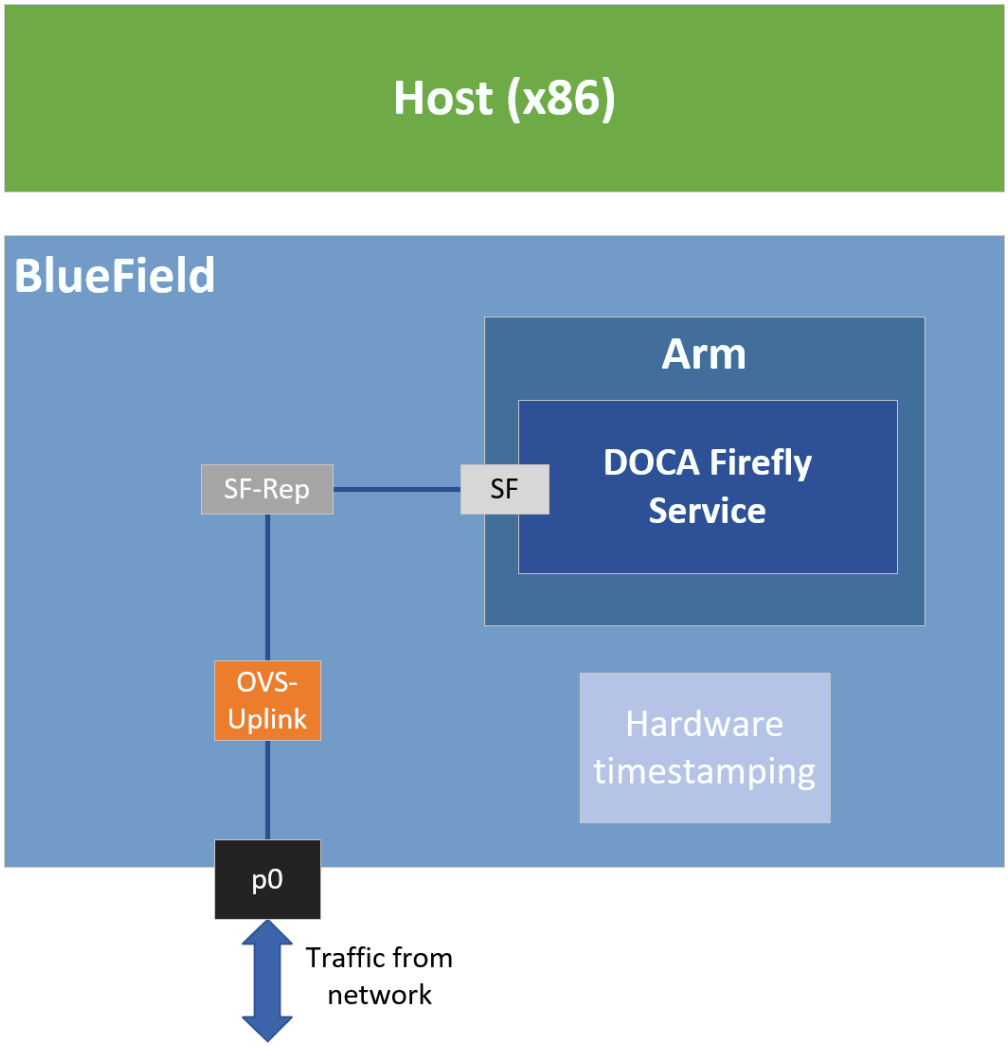
NVIDIA DOCA Firefly Service Guide

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

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.



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.

Firmware Version

Firmware version must be 24.34.1002 or higher.

BlueField BSP Version

Supported BlueField image versions are 3.9.0 and higher.

Embedded Mode

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)
```

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.

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 `1.3.0`.

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.

Note

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
```

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")

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.

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.

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 .

   > **Note**
   >
   > 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
```

Note

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
```

Separated Mode

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)
```

Setting Up Network Interfaces for Separated Mode

1. Make sure that 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
```

Host-based Deployment

Host-based deployment requires the same configuration described under section "Separated Mode".

Service Deployment

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.

Note

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.

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:1.3.0-doca2.5.0-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

Configuration

All modules within the service have configuration files that allow customizing various settings, both general and PTP-related.

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`).

> **Note**
>
> 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.

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`.

Note

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).

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` , `PHC2SYS`, SYNCE, `MONITOR`
- `SECTION` – the section in the config file that the parameter should be placed in

Note

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

> **Note**
>
> 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"
```

> **Note**
>
> 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).

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 (as mentioned below). 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
```

Note

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.

Description

Providers

DOCA Firefly Service uses the following third-party providers to provide time syncing services:

- Linuxptp - Version v4.1
  - `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

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`

> **Note**
>
> 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"
```

Note

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".

Profiles

DOCA Firefly Service includes profiles which represent common use cases for the Firefly service that provide a different default configuration per profile:

Profiles	Default	Media	Custom
Purpose	Any user that requires PTP	Media productions	Custom configuration for a dedicated user scenario
PTP	Enabled	Enabled	No default. Enable/disable should be set by the user.
PTP profile	PTP default profile	SMPTE 2059-2	Set by the user
PTP Client/Server[a]	Both	Client-only	Set by the user
PHC2SYS	Enabled	Enabled	No default. Enable/disable should be set by the user.
PPS (in/out)	Enabled	Enabled	No default. Enable/disable should be set by the user.

Profiles	Default	Media	Custom
PTP Monitor	Disabled	Disabled	No default. Enable/disable should be set by the user.
SyncE	Disabled	Disabled	No default. Enable/disable should be set by the user.

Note

[a] 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.

Outputs

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 1.3.0
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     - [+] 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     - [+] 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
```

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.

> **Note**
>
> 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>`.

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
```

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
```

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)
```

Note

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"
```

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
```

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
```

> **Note**
>
> 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".

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.

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

Note

`phc2sys` must use the same `domainNumber` setting used by `ptp4l`. If the same `domainNumber` is not set by the user, Firefly does that automatically.

Note

`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
```

SYNCE

Note

The SyncE module is supported at alpha level.

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.

Note

SyncE is currently only supported over network interfaces of the DPU's physical functions (i.e., `p0` or `p1`).

PTP Monitoring

Note

Monitoring is still in beta phase. There will be updates to the API in the near future.

PTP monitoring 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`.

Note

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.

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`.

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):

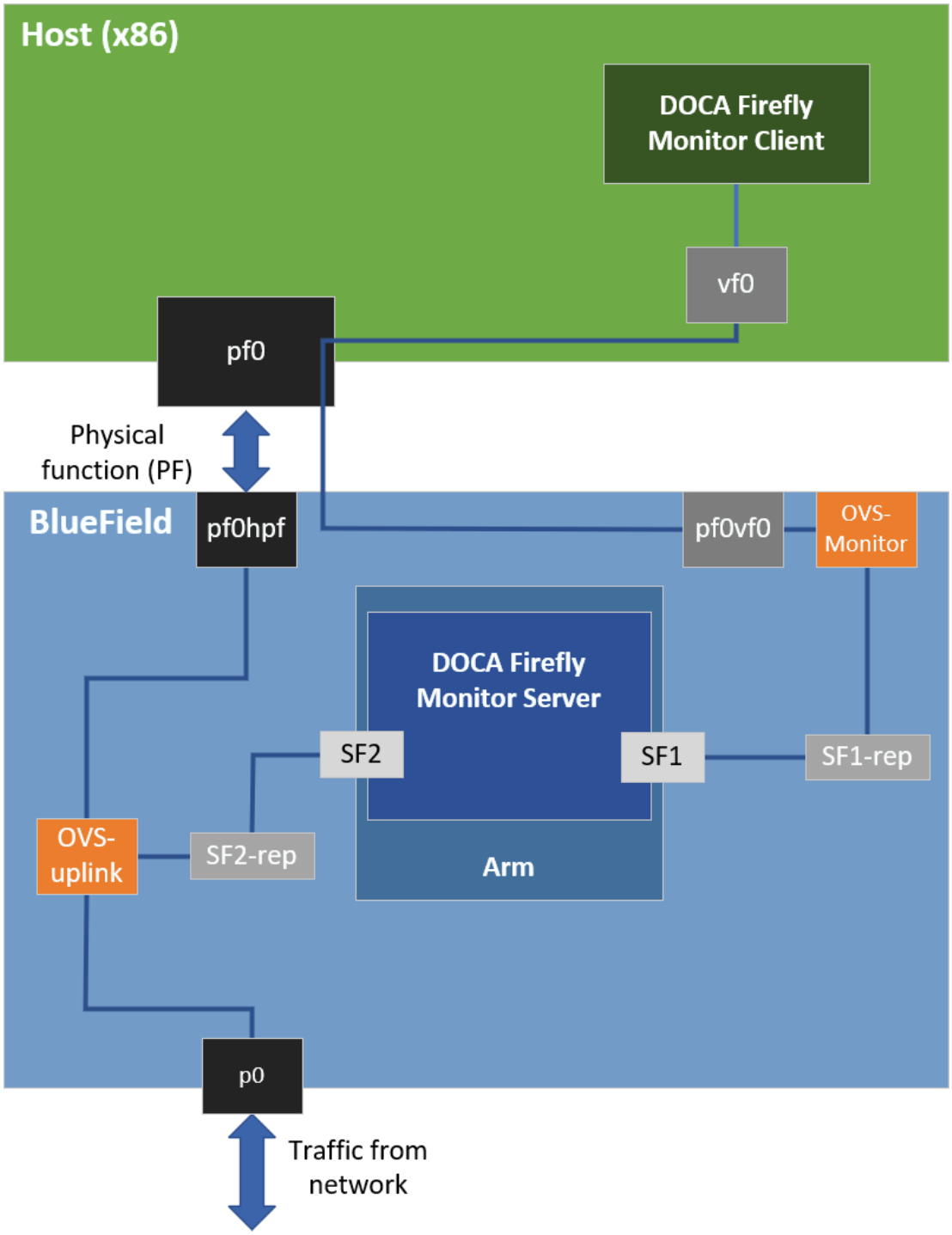
```
...
UtcOffset:                  37
...
ptp_time (TAI):             Thu Sep  7 11:22:50 2023
ptp_time (UTC adjusted):    Thu Sep  7 11:22:13 2023
system_time (UTC):          Thu Sep  7 11:22:13 2023
```

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.

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 NVIDIA DOCA gRPC Infrastructure User Guide.



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.

Monitor Client

All the required files for the monitor client are available under the service's dedicated installation directory:

- Linux installations – `/opt/mellanox/doca/services/firefly`
    - Example command line for executing the compiled monitor client from a Linux host:

```
$
/opt/mellanox/doca/services/firefly/bin/doca_firefly_monit
-g <ip-address-for-the-monitoring-server>
```

    - Example command line for executing the python-based monitor client from a Linux host:

```
$ export
PYTHONPATH=${PYTHONPATH}:/opt/mellanox/grpc/python3/lib
$
/opt/mellanox/doca/services/firefly/bin/doca_firefly_monit
<ip-address-for-the-monitoring-server>
```

> **Note**
>
> Reference source files and the `.proto` file used for Firefly's monitor are placed under `firefly/src/monitor`.

- Windows installation – `C:\Program Files\Mellanox\DOCA\SDK\firefly`
  - Example command line for executing the python-based monitor client from a Windows host:
    - Installing required pip packages:

```
$ pip3 install grpcio protobuf click
```

    - Running the client:

```
$ C:\Program
Files\Mellanox\DOCA\SDK\firefly\bin\doca_firefly_monit
<ip-address-for-the-monitoring-server>
```

VLAN Tagging

DOCA Firefly natively supports VLAN-tagging-enabled network interfaces.

Separated Mode

The name of the VLAN-enabled network interface should be the one passed through the YAML file in the `PTP_INTERFACE` field.

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
```

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"
```

Note

The monitoring feature is supported for multiple interfaces only when the `clientOnly` configuration is enabled.

Note

Automatic mode (`-a`) for `phc2sys` is not supported when working with multiple interfaces. It is recommended to disable `phc2sys` in this mode.

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 the to container, the finalized configuration file can be found under the `/tmp` directory using the same filename as the original configuration file.

> **Info**
>
> More information regarding the configuration files can be found under section "Ensuring and Debugging Correctness of Config File".

Pod is Marked as "Ready" and No Container is Listed

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
```

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
```

Custom Config File is Not Found

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
...
```

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.

Profile is Not Supported

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
...
```

Solution

Verify that the profile selected in the YAML file matches one of the supported profiles as listed in the profiles table.

> **Note**
>
> The profile name is case sensitive. The name must be specified in lower-case letters.

PPS Capability is Missing

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
...
```

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.

Timed Out While Polling for Tx Timestamp

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
```

Info

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.

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".

PTP Profile Default Config Files

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
```

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
```

Firefly Modules Configuration Options

PTP Monitor

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
```

Configuration Options

- `report_interval` – once in how many milliseconds should the monitor publish a report to all defined output providers (standard output, gRPC clients, etc.)? Default is 1 second (1000 milliseconds).
- `doca_logging_level` – logging level for the module, based on DOCA's logging levels (default is 50):
    - 10=DISABLE
    - 20=CRITICAL
    - 30=ERROR
    - 40=WARNING
    - 50=INFO
    - 60=DEBUG

NVIDIA DOCA Flow Inspector Service Guide

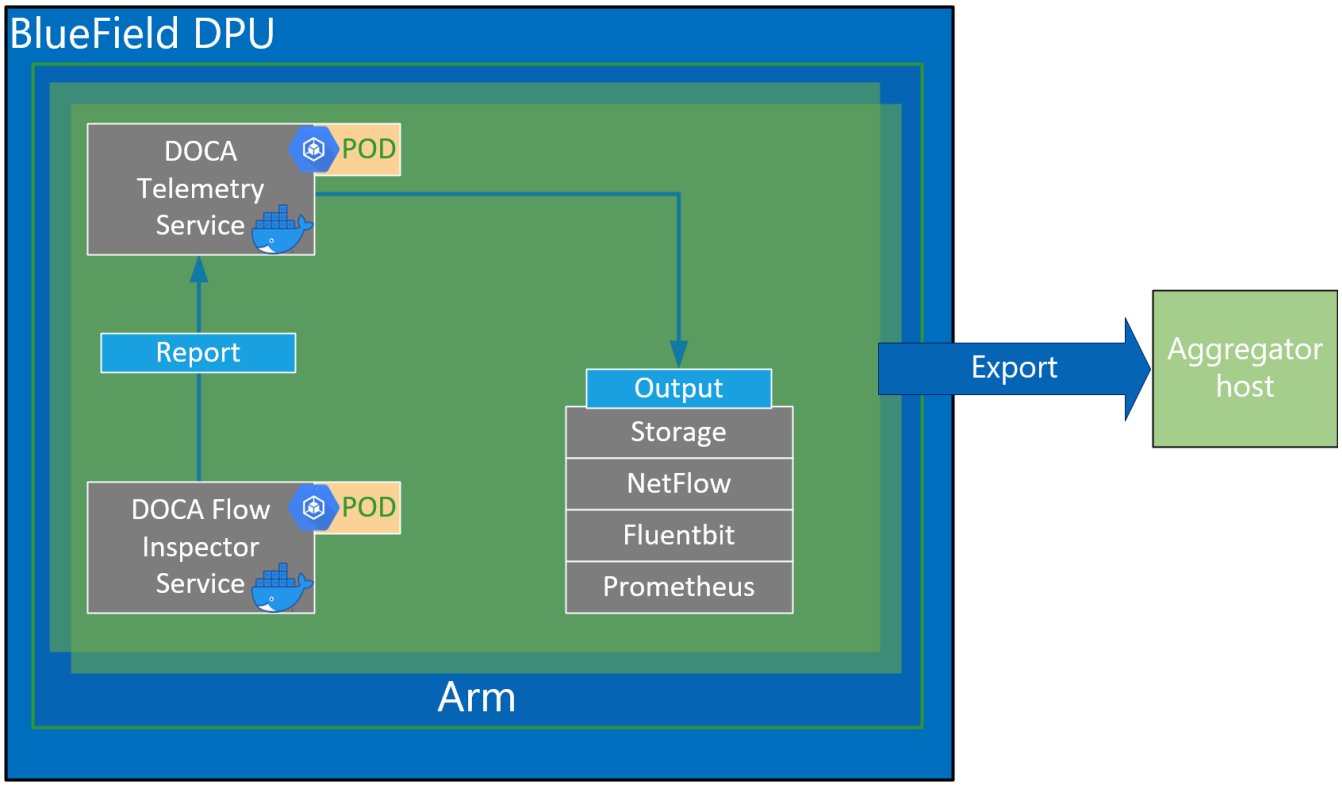
This guide provides instructions on how to use the DOCA Flow Inspector service container on top of NVIDIA® BlueField® DPU.

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 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.



Service Flow

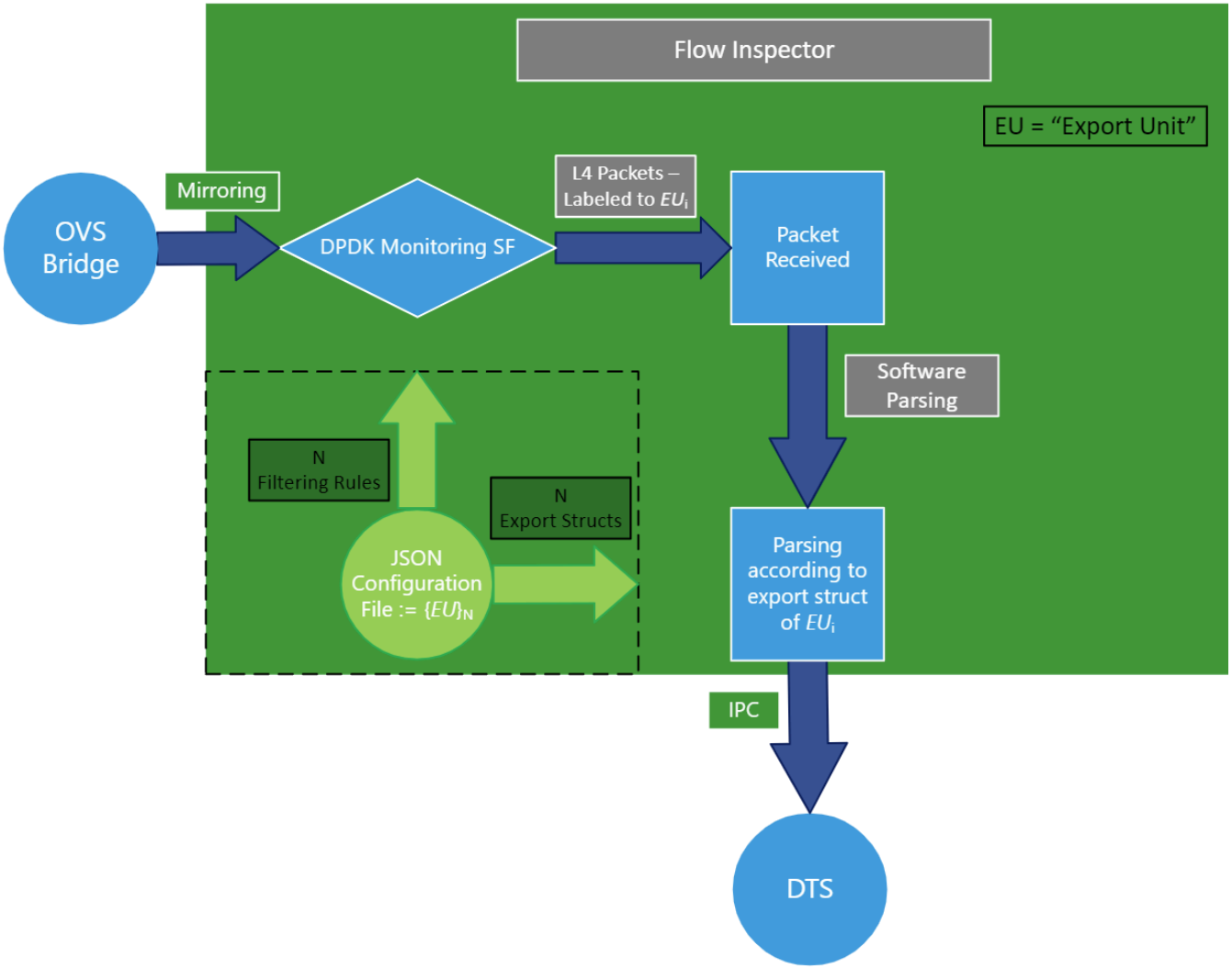
The DOCA Flow Inspector receives a configuration file in a JSON format which includes which of the mirrored packets should be filtered and which information should be sent to DTS for inspection.

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.

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_
   ```

   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:

1. 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=@p1 \
           -- set bridge ovsbr1 mirrors=@m
```

2. 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=@p1 \
           -- 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=@p1 \
           -- 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
```

> **Note**
>
> The designated SF must be created as a trusted function. Additional details can be found in the NVIDIA BlueField DPU Scalable Function User Guide.

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.

> **Note**
>
> 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.

Configuration

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>
        ]
}
```

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"]
                                }
                        }
                ]
}
```

Note

If a packet header contains L4 ports or L4 protocol which are not specified in any filter, they are filtered out.

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

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
```

Note

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:

Note

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.

Note

`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"
}
```

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.

Pod is Marked as "Ready" and No Container is Listed

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
```

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>.
```

Pod is Not Listed

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
```

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), ]"
```

NVIDIA DOCA HBN Service Guide

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

Release Notes

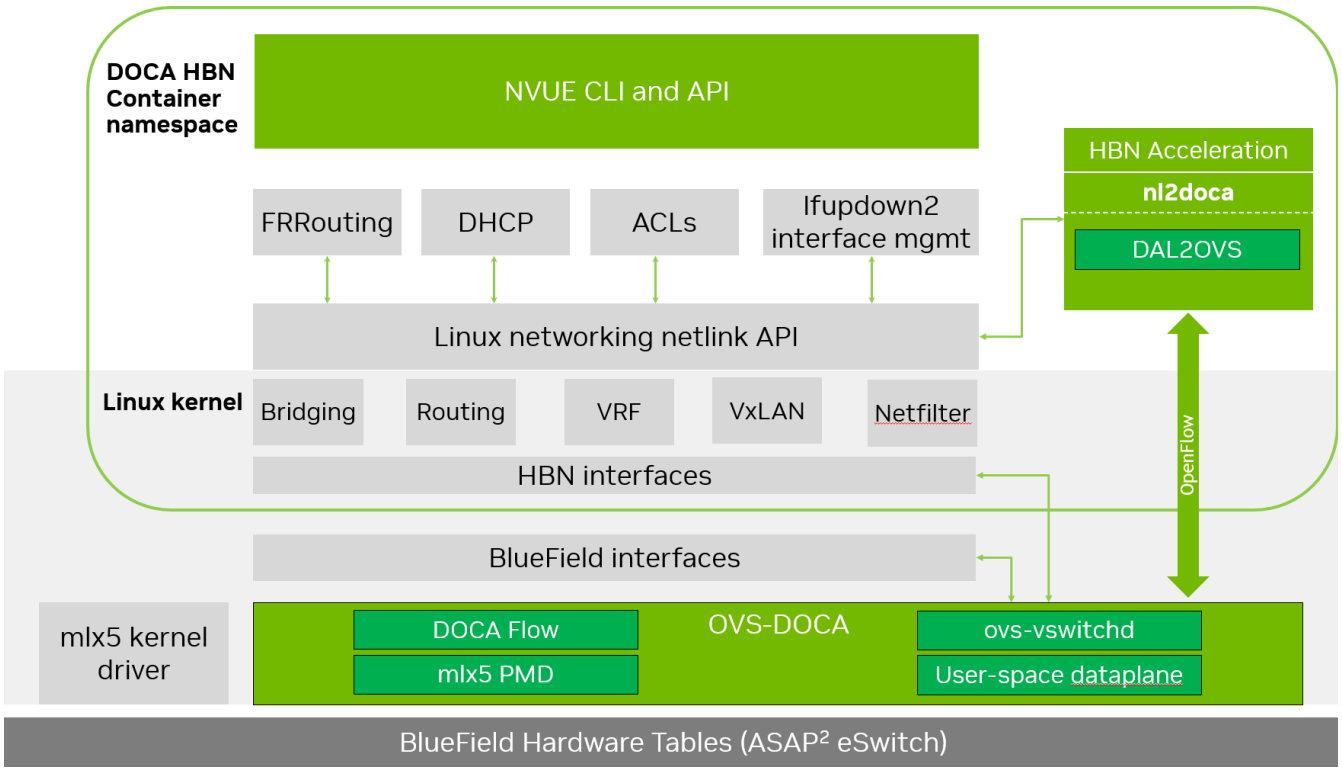
For the release notes of HBN 2.0.0, please refer to "HBN Service Release Notes".

HBN Overview

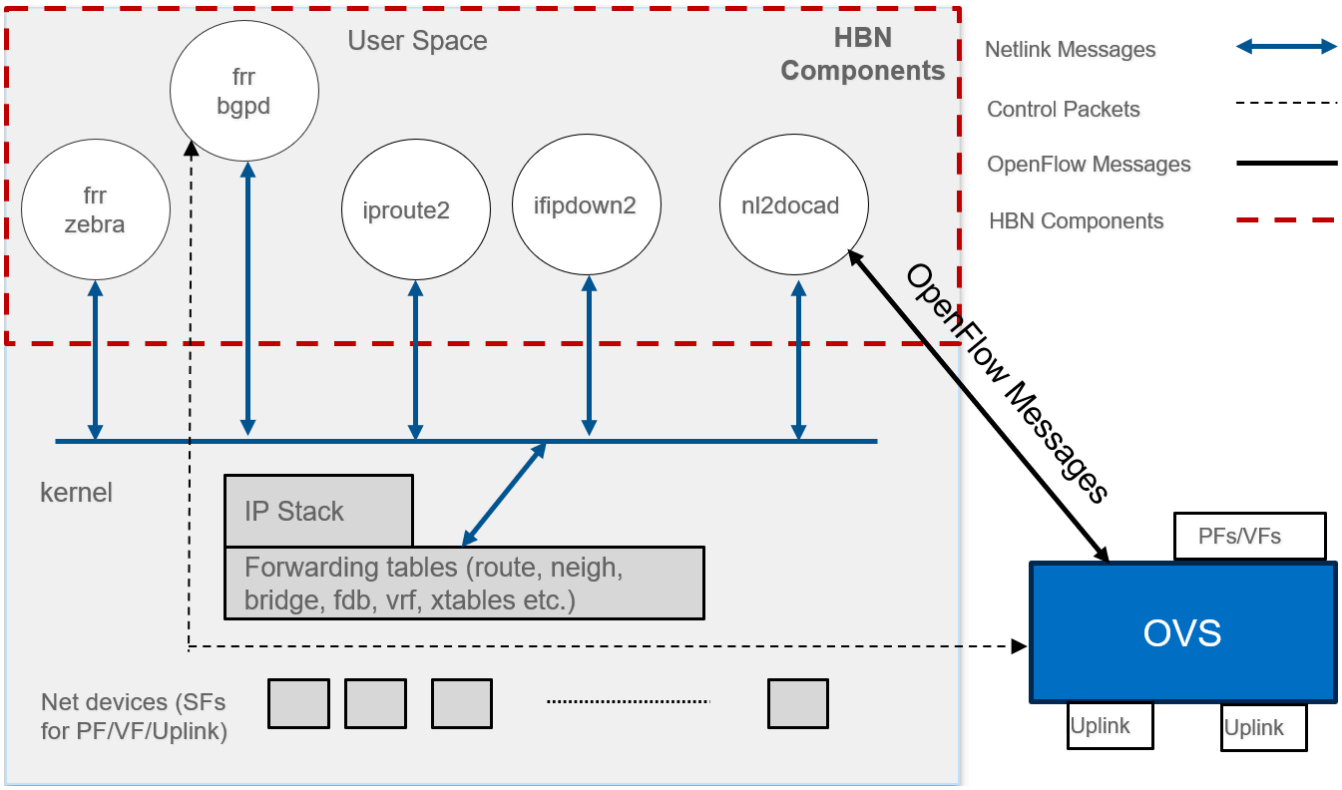
Host-based networking (HBN) is a DOCA service that enables the network architect to design a network purely on L3 protocols, enabling routing to run on the server-side of the network by using the DPU as a BGP router. The EVPN extension of BGP, supported by HBN, extends the L3 underlay network to multi-tenant 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 the DPU. At the core of HBN is the Linux networking DPU acceleration driver. Netlink to DOCA daemon, or nl2docad, implements the DPU acceleration driver. nl2docad seamlessly accelerates Linux networking using DPU hardware programming APIs.

The driver mirrors the Linux kernel routing and bridging tables into the DPU hardware 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 DPU 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 the DPU hardware tables. nl2docad also offloads these flows to eSwitch.

Service Deployment

Preparing DPU for HBN Deployment

HBN requires service function chaining (SFC) to be activated on the DPU before running the HBN service container. SFC allows for additional services/containers to be chained to HBN and provides additional data manipulation capabilities.

The following subsections provide additional information about SFC and instructions on enabling it during DPU BFB installation.

Service Function Chaining

The diagram below shows the fully detailed default configuration for HBN with Service Function Chaining (SFC).

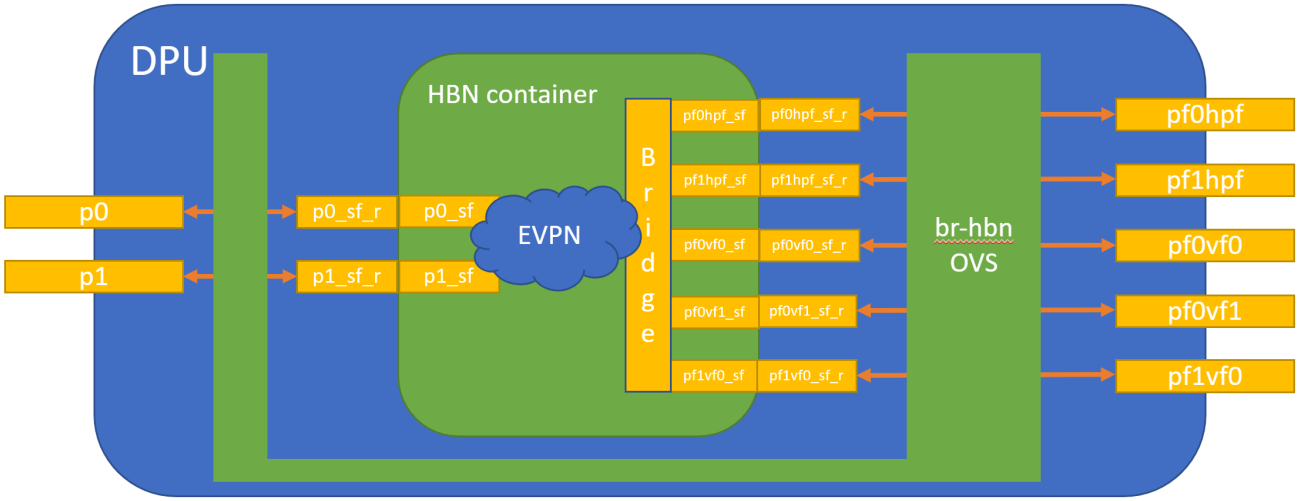
In this setup, the HBN container is configured to use sub-function ports (SFs) instead of the actual uplinks, PFs and VFs. To illustrate, for example:

- Uplinks – use `p0_sf` instead of `p0`

- PF – use `pf0hpf_sf` instead of `pf0hpf`

- VF – use `pf0vf0_sf` 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 the DPU with HBN enabled. This indirection layer allows other services to be chained to existing SFs and provide additional functionality to transit traffic.



Enabling SFC for HBN Deployment

Deployment from BFB

DPU installation should follow the *NVIDIA DOCA Installation Guide for Linux*.

1. Make sure link type is set to ETH in step 5 of the "Installing Software on Host" section in the *NVIDIA DOCA Installation Guide for Linux*.

2. Add the following parameters to the `bf.cfg` configuration file:

```
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>
```

Deployment from 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_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
EOF
```

`/etc/bf.cfg` is sourced by the BFB `install.sh` script.

Note

It is recommended to verify the accuracy of the DPU'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.

HBN Service Container Deployment

HBN service is available on NGC, NVIDIA's container catalog. Service-specific configuration steps and deployment instructions can be found under the service's container page. Make sure to follow the instructions in the NGC page to verify that the container is running properly.

For information about the deployment of DOCA containers on top of the BlueField DPU, refer to NVIDIA DOCA Container Deployment Guide.

HBN Default Deployment Configuration

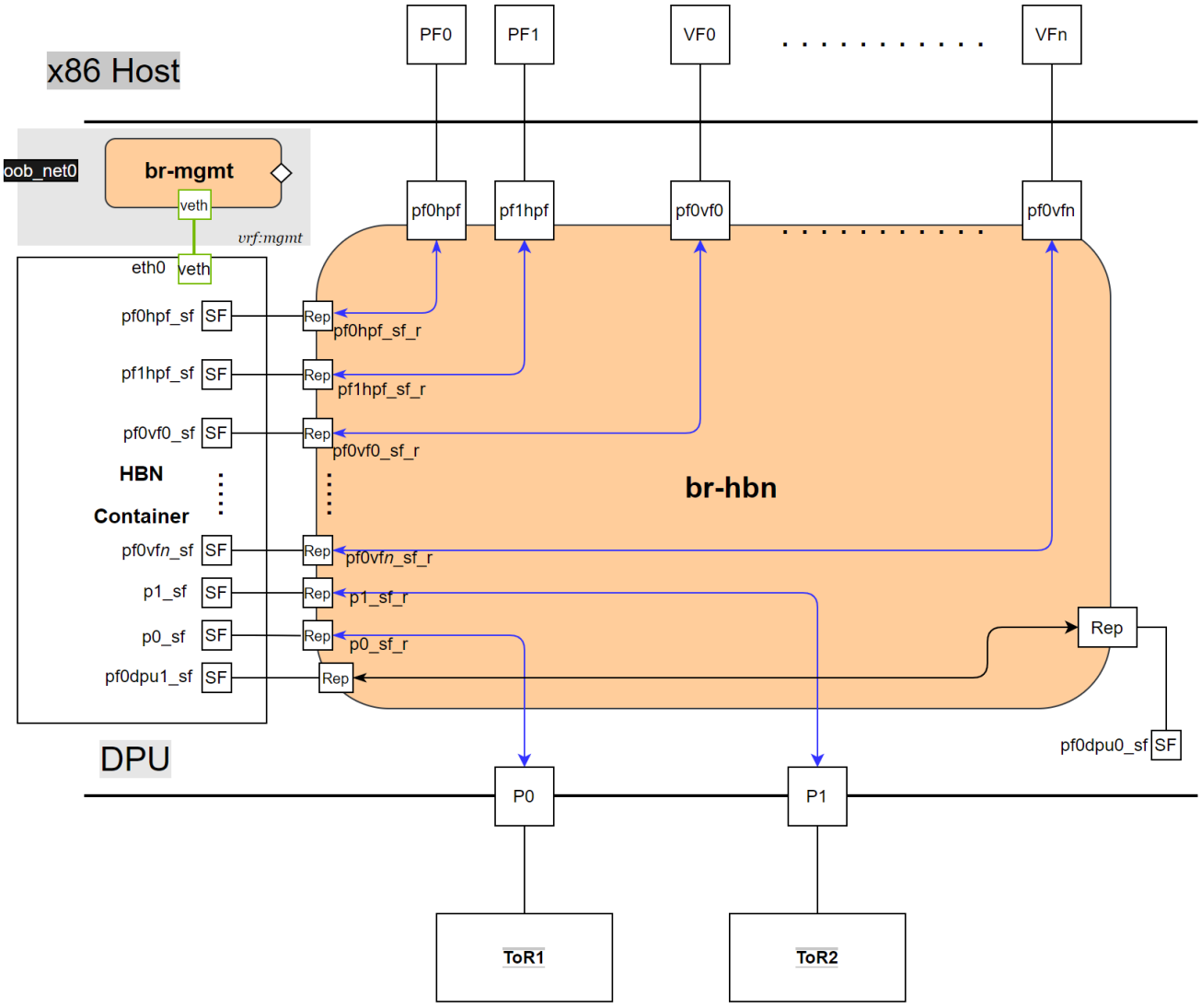
The HBN service comes with four types of configurable interfaces:

- Two uplinks (`p0_sf`, `p1_sf`)
- Two PF port representors (`pf0hpf_sf`, `pf1hpf_sf`)
- User-defined number of VFs (i.e., `pf0vf0_sf`, `pf0vf1_sf`, ..., `pf1vf0_sf`, `pf1vf1_sf`, ...)
- One interface to connect to services running on the DPU, outside of the HBN container (`pf0dpu1_sf`)

The `*_sf` 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 DPU.

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, `*_sf_r` representors).



`pf0dpu1_sf` is a special interface for HBN to connect to services running on the DPU. Its counterpart `pf0dpu0_sf` is located outside the HBN container. See section "Connecting to Services on DPU" for deployment considerations when using the `dpu1_sf` interface in HBN.

`eth0` is equivalent to the `oob_net0` interface in the HBN container. It is part of the management VRF of the container. It is not configurable via NVUE and does not need any configuration from the user. See section "MGMT VRF in HBN Container" for more details on this interface and the management VRF.

HBN Deployment Considerations

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_sf` 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_sf` 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 the DPU, outside of the HBN container, to sync the state. The daemon listens to netlink notifications for interfaces and transfers the state to SFs.

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 the DPU 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, execute `systemctl restart systemd-networkd`.

Connecting to Services on DPU

`pf0dpu1_sf` can be used by HBN to connect to services running on the DPU. Its counterpart, `pf0dpu0_sf`, is located outside the HBN container.

Traffic between the DPU and the outside world is not hardware-accelerated in the HBN container when using a native L3 connection over `pf0dpu0_sf` / `pf0dpu1_sf`. To get hardware-acceleration, configure `pf0dpu1_sf` in the HBN container with `bridge-access` over an SVI.

Disabling DPU 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.

> **Note**
>
> 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 the DPU.

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).

Configuration

General Network Configuration

Flat Files Configuration

Add network interfaces and FRR configuration files to the DPU to achieve the desired configuration:

- `/etc/network/interfaces`

  > **Note**
  >
  > Refer to NVIDIA® Cumulus® Linux documentation for more information.

- `/etc/frr/frr.conf`; `/etc/frr/daemons`

  > **Note**
  >
  > Refer to NVIDIA® Cumulus® Linux documentation for more information.

NVUE Configuration

This section assumes familiarity with NVIDIA user experience (NVUE) Cumulus Linux documentation. The following subsections, only expand on DPU-specific aspects of NVUE.

NVUE Service

HBN installs NVUE by default and enables NVUE service at boot.

NVUE REST API

HBN enables REST API by default.

Users may run the cURL commands from the command line. Use the default HBN username `nvidia` and password `nvidia`.

To change the default password of the `nvidia` user or add additional users for NVUE access, refer to section "NVUE User Credentials".

REST API example:

```
curl -u 'nvidia:nvidia' --insecure
https://10.188.108.58:8765/nvue_v1/interface/p0
{
  "ip": {
    "address": {
      "30.0.0.1/24": {}
    }
  },
  "link": {
    "auto-negotiate": "on",
    "duplex": "full",
    "fec": "auto",
    "mac": "b8:ce:f6:a8:83:9a",
    "mtu": 9216,
    "speed": "100G",
    "state": {
      "up": {}
    },
    "stats": {
      "carrier-transitions": 13,
      "in-bytes": 0,
      "in-drops": 0,
      "in-errors": 0,
      "in-pkts": 0,
      "out-bytes": 14111,
      "out-drops": 0,
      "out-errors": 0,
      "out-pkts": 161
    }
  },
  "pluggable": {
    "identifier": "QSFP28",
    "vendor-name": "Mellanox",
    "vendor-pn": "MCP1600-C00AE30N",
    "vendor-rev": "A4",
```

```
    "vendor-sn": "MT2105VB02844"
  },
  "type": "swp"
}
```

Note

For information about using the NVUE REST API, refer to the NVUE API documentation .

NVUE CLI

For information about using the NVUE CLI, refer to the NVUE CLI documentation

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
```

Note

`nv config apply startup` applies the yaml configuration saved at `/etc/nvue.d/`.

Note

`nv config save` saves the running configuration to `/etc/nvue.d/startup.yaml`.

NVUE User Credentials

The preconfigured default user credentials are as follows:

Username	nvidia
Password	nvidia

NVUE user credentials can be added post installation. This functionality is enabled by the HBN startup script by using the `--username` and `--password` script switches. For example:

```
./hbn-dpu-setup.sh -u newuser -p newpassword
```

After executing this script, respawn the container or start the `decrypt-user-add` script:

```
supervisorctl start decrypt-user-add
decrypt-user-add: started
```

The script creates a user on the HBN container:

```
cat /etc/passwd | grep newuser
newuser:x:1001:1001::/home/newuser:/bin/bash
```

NVUE Interface Classification

Interface	Interface Type	NVUE Type
`p0_sf`	Uplink representor	swp
`p1_sf`	Uplink representor	swp
`lo`	Loopback	loopback
`pf0hpf_sf`	Host representor	swp
`pf1hpf_sf`	Host representor	swp
`pf0vfx_sf` (where `x` is 0-255)	VF representor	swp
`pf1vfx_sf` (where `x` is 0-255)	VF representor	swp

Configuration Persistence

The following directories are mounted from the host DPU to the HBN container and are persistent across HBN restarts and DPU reboots:

Host DPU Mount Point	HBN Container Mount Point
Configuration Files Mount Pints	

Host DPU Mount Point	HBN Container Mount Point
`/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 Files Mount Points	
`/var/lib/hbn/var/support/`	`/var/support/`
`/var/log/doca/hbn/`	`/var/log/hbn/`

SR-IOV Support

Creating VFs on Host Server

The first step to use SR-IOV is to create VFs on the host server. VFs can be created using the following command:

```
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 on DPU

VFs created on the host must have corresponding SF representors on the DPU side. For example:

- `ens1f0vf0` is the first VF from the first host representor; this interface is created on the host server
- `pf0vf0` is the corresponding VF representor to `ens1f0vf0`; this interface is on the DPU and automatically created at the same time as `ens1f0vf0` is created
- `pf0vf0_sf` is the corresponding SF for `pf0vf0` which is used by HBN

The creation of the SF representor for VFs is done ahead of time when installing the BFB, see section "Enabling SFC for HBN Deployment" to see how to select how many SFs to create ahead of time.

The SF representors for VFs (i.e., `pfXvfY`) are pre-mapped to work with the corresponding VF representors when these are created with the command from section "Creating VFs on Host Server".

Management VRF

Two management VRFs are setup for HBN with SFC:

- The first management VRF is outside the HBN container on the DPU. 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 `*_sf` interfaces.

MGMT VRF on Host DPU

The management (mgmt) VRF is enabled by default when the DPU is deployed with SFC (see section "Enabling SFC for HBN Deployment"). 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 in HBN Container

Inside the HBN container, a separate mgmt VRF is present. Similar commands as those listed under section "MGMT VRF on Host DPU" can be used to query management routes.

The `*_sf` 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 DPU `oob_net0` interface, ultimately using the DPU 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 Host DPU

On the host DPU, 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
```

> **Note**
>
> The original version of these services (without `@mgmt`) are not used and must not be started.

Running New Service in MGMT VRF

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 host DPU (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
```

HBN Configuration Examples

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_sf
iface p0_sf

auto p1_sf
iface p1_sf
```

FRR configuration files would also be present under `/etc/frr/` but no configuration would be enabled.

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.

> **Info**
>
> For TCP and UDP packets, it also includes source port and destination port.

ECMP Configuration

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_sf weight 1 onlink <<<<<
via uplink p0_sf
        nexthop via 169.254.0.1 dev p1_sf weight 1 onlink <<<<<
via uplink p1_sf
```

> **Info**
>
> HBN supports up to 16 paths for ECMP.

Sample NVUE Configuration

```
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_sf,pf1hpf_sf 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_sf remote-as external
nv set vrf default router bgp neighbor p0_sf type unnumbered
nv set vrf default router bgp neighbor p0_sf address-family ipv4-
unicast enable on
nv set vrf default router bgp neighbor p0_sf address-family ipv6-
unicast enable on
nv set vrf default router bgp neighbor p1_sf remote-as external
nv set vrf default router bgp neighbor p1_sf type unnumbered
nv set vrf default router bgp neighbor p1_sf address-family ipv4-
unicast enable on
```

```
nv set vrf default router bgp neighbor p1_sf address-family ipv6-
unicast enable on
```

Sample Flat Files Configuration

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_sf
iface p0_sf

auto p1_sf
iface p1_sf

auto pf0hpf_sf
iface pf0hpf_sf
        bridge-access 100

auto pf1hpf_sf
iface pf1hpf_sf
        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_sf pf1hpf_sf
    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_sf interface remote-as external
 neighbor p0_sf advertisement-interval 0
 neighbor p0_sf timers 3 9
 neighbor p0_sf timers connect 10
 neighbor p1_sf interface remote-as external
 neighbor p1_sf advertisement-interval 0
 neighbor p1_sf timers 3 9
 neighbor p1_sf 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_sf activate
  neighbor p1_sf activate
  maximum-paths 64
```

```
  maximum-paths ibgp 64
 exit-address-family
!
line vty
!
end
```

BGP Peering with Host

HBN supports the ability to establish a BGP session between the host and DPU and allow the host to announce arbitrary route prefixes through the DPU 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.

> **Note**
>
> 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

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.

> **Note**
>
> 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

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 bfd 3 1000 1000
 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_sf interface peer-group fabric
 neighbor p1_sf interface peer-group fabric
 !
 address-family ipv4 unicast
        neighbor dpu_host activate
 !
 address-family ipv6 unicast
        neighbor dpu_host activate
```

Sample Configuration on Host Running FRR

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 interface 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:c15c:d06: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
```

L2 EVPN with BGP and ECMP

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.

Single VXLAN Device

With a single VXLAN device, a set of 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_sf pf1hpf_sf
    bridge-vids 10 20
    bridge-pvid 1
```

Note

Users may not use a combination of single and traditional VXLAN devices.

Sample NVUE Configuration on DPU

The following is a sample NVUE configuration which has L2-VNIs (`2000`, `2001`) for EVPN bridging on DPU.

```
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_sf peer-group fabric
nv set vrf default router bgp neighbor p0_sf type unnumbered
nv set vrf default router bgp neighbor p1_sf peer-group fabric
nv set vrf default router bgp neighbor p1_sf 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:c15c:d06:f00d::4/128
nv set interface lo ip address 27.0.0.4/32
nv set interface lo type loopback
nv set interface p0_sf,p1_sf,pf0hpf_sf,pf1hpf_sf type swp
nv set interface pf0hpf_sf bridge domain br_default access 200
nv set interface pf1hpf_sf 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 on HBN

The following is a sample flat files configuration which has L2-VNIs (`vx-2000`, `vx-2001`) for EVPN bridging on DPU.

This file is located at `/etc/network/interfaces`:

```
auto lo
iface lo inet loopback
    address 2001:c15c:d06:f00d::4/128
    address 27.0.0.4/32
    vxlan-local-tunnelip 27.0.0.4

auto p0_sf
iface p0_sf

auto p1_sf
iface p1_sf

auto pf0hpf_sf
iface pf0hpf_sf
    bridge-access 200

auto pf1hpf_sf
iface pf1hpf_sf
    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_sf pf1hpf_sf 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"
```

This 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_sf interface peer-group fabric
neighbor p1_sf 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
```

Sample Switch Configuration for EVPN

The following is a sample NVUE config for underlay switches (NVIDIA® Spectrum® with Cumulus Linux) for EVPN use case.

It assumes that the uplinks on DPUs 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:c15c:d06: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
```

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.

ACL Ordering

ACL ordering ensures that the order in which ACLs are executed in DPU hardware is the same as the order in which the ACLs are configured. In general, IPv4 ACLs should be configured before IPv6 ACLs which in turn should be configured before L2 ACLs. ACLs should be configured in the following order:

1. IPv4 header match fields + UDP header match fields

2. IPv4 header match fields + TCP header match fields

3. IPv4 header match fields + ICMP header match fields

4. IPv4 header match fields

5. IPv6 header match fields + UDP header match fields

6. IPv6 header match fields + TCP header match fields

7. IPv6 header match fields + ICMP header match fields

8. IPv6 header match fields

9. Ethernet header match fields

Stateless ACLs

HBN supports configuration of stateless ACLs for IPv4 packets, IPv6 packets, and Ethernet 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_sf` 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_sf` of the DPU in the inbound direction:

```
root@hbn01-host01:~# nv set interface pf0hpf_sf 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_sf` 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_sf` of the DPU in the outbound direction:

```
root@hbn01-host01:~# nv set interface pf0hpf_sf 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_sf`:

```
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_sf` of the DPU in the inbound direction:

```
root@hbn01-host01:~# nv set interface pf0hpf_sf 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_sf`:

```
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_sf` of the DPU in the outbound direction:

```
root@hbn01-host01:~# nv set interface pf0hpf_sf acl acl6_egress
outbound
root@hbn01-host01:~# nv config apply
```

NVUE L2 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_sf`:

```
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_sf acl acl3_ingress
inbound
```

Bind the ingress MAC ACL to host representor port `pf0hpf_sf` of the DPU in the inbound direction:

```
root@hbn01-host01:~# nv set interface pf0hpf_sf 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_sf`:

```
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_sf` of the DPU in the outbound direction:

```
root@hbn01-host01:~# nv set interface pf0hpf_sf 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 the DPU followed by invoking `cl-acltool -i`:

```
[iptables]
## ACL acl1_ingress in dir inbound on interface pf0hpf_sf ##
-t filter -A FORWARD -i pf1vf1_sf -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 the DPU followed by invoking `cl-acltool -i`:

```
[iptables]
## ACL acl2_egress in dir outbound on interface pf0hpf_sf ##
-t filter -A FORWARD -o pf0hpf_sf -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 the DPU followed by invoking `cl-acltool -i`:

```
[ip6tables]
## ACL acl5_ingress in dir inbound on interface pf0hpf_sf ##
-t filter -A FORWARD -i pf0hpf_sf -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 the DPU followed by invoking `cl-acltool -i`:

```
[ip6tables]
## ACL acl6_egress in dir outbound on interface pf0hpf_sf ##
-t filter -A FORWARD -o pf0hpf_sf -s 48:2034::80:9 -p tcp -j
ACCEPT
```

Flat Files L2 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 the DPU followed by invoking `cl-acltool -i`:

```
[ebtables]
## ACL acl3_ingress in dir inbound on interface pf0hpf_sf ##
-t filter -A FORWARD -i pf0hpf_sf -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 DPU followed by invoking `cl-acltool -i`:

```
[ebtables]
## ACL acl4_egress in dir outbound on interface pf0hpf_sf ##
-t filter -A FORWARD -o pf0hpf_sf -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
```

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 permit initiation of connections from within the network to outside the network and allow only replies to the initiated connections from outside the network.

HBN supports stateful ACL configuration for IPv4 traffic.

Stateful ACLs can be applied for routed traffic (north-south traffic) or bridged traffic (east-west traffic). Stateful ACLs applied for routed traffic are called "L3 stateful ACLs" and for bridged traffic are called "L2 stateful ACLs". Currently, NVUE-based configuration is supported only for L3 stateful ACLs (L2 stateful ACLs must be configured using flat-file configuration).

Stateful ACLs in HBN are disabled 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 Examples for L3 Stateful ACLs

The following is an example of allowing HTTP (TCP) connection originated by the host where the DPU 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 host representor of the DPU and the associated VLAN's 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 host representor of the DPU and the associated VLAN's SVI in the outbound direction.

In this example, the host representor on the DPU is `pf0hpf_sf` and it is part of VLAN 101 (SVI interface is `vlan101`).

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 host representor of the DPU and the associated VLAN's SVI interface in the inbound direction:

```
root@hbn03-host00:~# nv set interface pf0hpf_sf,vlan101 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 host representor of the DPU and the associated VLAN's SVI interface in the outbound direction:

```
root@hbn03-host00:~# nv set interface pf0hpf_sf,vlan101 acl
allow_tcp_resp_from_server outbound
root@hbn03-host00:~# nv config apply
```

Note

If virtual router redundancy (VRR) is set, L3 stateful ACLs must be bound to all the related SVI interfaces. For example, if VRR is configured on SVI `vlan101` as follows in the `/etc/network/interfaces` file:

```
auto vlan101
iface vlan101
    address 45.3.1.2/24
    address-virtual 00:00:5e:00:01:01 45.3.1.1/24
    vlan-raw-device br_default
    vlan-id 101
```

With this configuration, two SVI interfaces, `vlan101` and `vlan101-v0` would be created in the system:

```
root@hbn03-host00:~# ip -br addr show | grep
vlan101
vlan101@br_default UP             45.3.1.2/24
fe80::204:4bff:fe8a:f100/64
vlan101-v0@vlan101 UP             45.3.1.1/24
metric 1024 fe80::200:5eff:fe00:101/64
```

In this case, stateful ACLs must be bound to both SVI interfaces (`vlan101` and `vlan101-v0`). In the stateful ACL described in the current section, the binding would be:

```
root@hbn03-host00:~# nv set interface
pf0hpf_sf,vlan101,vlan101-v0 acl
allow_tcp_conn_from_host inbound
root@hbn03-host00:~# nv set interface
pf0hpf_sf,vlan101,vlan101-v0 acl
allow_tcp_resp_from_server outbound
root@hbn03-host00:~# nv config apply
```

Flat Files (cl-acltool) Examples for L3 Stateful ACLs

For the example described under section "NVUE Examples for L3 Stateful ACLs", 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 DPU 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 host representor of the DPU and the associated VLAN's SVI followed by invoking `cl-acltool -i`:

```
[iptables]
## ACL allow_tcp_conn_from_host in dir inbound on interface
pf1vf7_sf ##
-t mangle -A PREROUTING -p tcp -d 11.11.11.11/32 --dport 80 -
m conntrack --ctstate EST,NEW -m connmark ! --mark 9998 -j
CONNMARK --set-mark 9999
-t filter -A FORWARD -i pf1vf7_sf -p tcp -d 11.11.11.11/32 --
dport 80 -m conntrack --ctstate EST,NEW -j ACCEPT

## ACL allow_tcp_conn_from_host in dir inbound on interface
vlan118 ##
-t filter -A FORWARD -i vlan118 -p tcp –d 11.11.11.11/32--
dport 80 -m conntrack --ctstate EST,NEW -j ACCEPT
```

Note

A mangle table rule must be configured with CONNMARK action. The CONNMARK values (`-j CONNMARK --set-mark <value>`) for ingress ACL rules are protocol dependent: 9999 for TCP, 9997 for UDP, and 9995 for ICMP.

2. Configure an egress ACL rule matching with TCP and CONNTRACK state of ESTABLISHED, RELATED under `/etc/cumulus/acl/policy.d/stateful_acl.rules` file with the egress interface as the host representor of the DPU and the associated VLAN's SVI followed by invoking `cl-acltool -i`:

```
[iptables]
## ACL allow_tcp_resp_from_server in dir outbound on
interface pf1vf7_sf ##
-t mangle -A PREROUTING -p tcp -s 11.11.11.11/32 --sport 80 -
m conntrack --ctstate EST -j CONNMARK --set-mark 9998
-t filter -A FORWARD -o pf1vf7_sf -p tcp -m conntrack --
ctstate EST,REL -j ACCEPT

## ACL allow_tcp_resp_from_server in dir outbound on
interface vlan118 ##
-t filter -A FORWARD -o vlan118 -p tcp -m conntrack --ctstate
EST,REL -j ACCEPT
```

Note

A mangle table rule must be configured with CONNMARK action. The CONNMARK values (`-j CONNMARK --set-mark <value>`) for egress ACL rules are protocol dependent: 9998 for TCP, 9996 for UDP, and 9994 for ICMP.

Flat Files (cl-acltool) Examples for L2 Stateful ACLs

For the same example cited above (HTTP server at IP address 192.168.5.5 accessible over bridged 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`.

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 host representor of the DPU and the associated VLAN's SVI followed by invoking `cl-acltool -i`:

```
[iptables]
## ACL allow_tcp_conn_from_host in dir inbound on interface
pf1vf7_sf
-t mangle -A PREROUTING -p tcp -d 192.168.5.5/32 --dport 80 -
m conntrack --ctstate EST,NEW -m connmark ! --mark 9998 -j
CONNMARK --set-mark 9999
-t filter -A FORWARD -m physdev --physdev-in pf1vf7_sf -p tcp
-d 192.168.5.5/32 --dport 80 -m conntrack --ctstate EST,NEW -
j ACCEPT

## ACL allow_tcp_conn_from_host in dir inbound on interface
vlan118 ##
-t filter -A FORWARD -i vlan118 -p tcp –d 192.168.5.5/32--
dport 80 -m conntrack --ctstate EST,NEW -j ACCEPT
```

Note

A mangle table rule must be configured with CONNMARK action. The CONNMARK values (`-j CONNMARK --set-mark <value>`) for ingress ACL rules are protocol dependent: 9999 for TCP, 9997 for UDP, and 9995 for ICMP.

2. Configure an egress ACL rule matching with TCP and CONNTRACK state of ESTABLISHED, RELATED under `/etc/cumulus/acl/policy.d/stateful_acl.rules` file with the egress interface as the host representor of the DPU and the associated VLAN's SVI followed by invoking `cl-acltool -i`:

```
[iptables]
## ACL allow_tcp_resp_from_server in dir outbound on
interface pf1vf7_sf ##
-t mangle -A PREROUTING -p tcp -s 192.168.5.5/32 --sport 80 -
m conntrack --ctstate EST -j CONNMARK --set-mark 9998
-t filter -A FORWARD -m physdev --physdev-out pf1vf7_sf -p
tcp -m conntrack --ctstate EST,REL -j ACCEPT

## ACL allow_tcp_resp_from_server in dir outbound on
interface vlan118 ##
-t filter -A FORWARD -o vlan118 -p tcp -m conntrack --ctstate
EST,REL -j ACCEPT
```

Note

A mangle table rule must be configured with CONNMARK action. The CONNMARK values (`-j CONNMARK --set-mark <value>`) for egress ACL rules are protocol dependent: 9998 for TCP, 9996 for UDP, and 9994 for ICMP.

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.

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 the DPU in `/var/lib/hbn/etc/supervisor/conf.d/`. The DPU 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_sf`, `p1_sf`, 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_sf
nv set service dhcp-relay default interface p1_sf
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_sf -i p1_sf -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_sf -i p1_sf -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_sf` and `p1_sf`.

```
nv set service dhcp-relay6 default interface downstream vlan482
nv set service dhcp-relay6 default interface upstream p0_sf
nv set service dhcp-relay6 default interface upstream p1_sf
```

Flat Files Example

```
[program: isc-dhcp-relay6-default]
command = /usr/sbin/dhcrelay --nl -6 -d -l vlan482 -u p0_sf -u
p1_sf
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

Option	Description
`--loglevel-debug`	Debug logging located at `/var/log/syslog`

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>
```

Troubleshooting

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_sf` is missing. Run the command `ip -br link show dev p0_sf` in the DPU and inside the container to check if `p0_sf` is present or not. If its missing, make sure the firmware is upgraded to the latest version. Gracefully shutdown and power cycle the host for the new firmware to take effect.

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_sf_r`, `p0_sf`, and the remote peer of `p0`.

Generating Support Dump

HBN support dump can be generated using the `cl-support` command, inside the HBN container:

```
root@bf2:/tmp# cl-support
Please send /var/support/cl_support_bf2-s02-1-
ipmi_20221025_180508.txz to Cumulus support
```

The generated dump would be available in `/var/support` in the HBN container and would contain any process core dump as well as log files.

The `/var/support` directory is also mounted on the host DPU at `/var/lib/hbn/var/support`.

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 the DPU 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.

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
```

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_sf`, the basic offload flow should look as follows: RxPort (`pf0vf0_sf`) -> BR (Overlay) -> RTR (Underlay) -> BR (Underlay) -> TxPort (one of the uplink `p0_sf` or `p1_sf` 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_sf`):

```
Interface: pf0vf0_sf
    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_sf)
    STATS:
       pkts: 1099
       bytes: 162652
```

This will show the packet going out on the uplink.

NVUE Troubleshooting

To check the status of the NVUE daemon, run:

```
supervisorctl status nvued
```

To restart the NVUE daemon, run:

```
supervisorctl restart nvued
```

HBN Service Release Notes

The following subsections provide information on HBN service new features, interoperability, known issues, and bug fixes.

Changes and New Features

HBN 2.0.0 offers the following new features and updates:

- New hardware-accelerated dataplane based on OVS-DOCA
- Added support for HBN interoperating with SNAP storage service for NVMe-oF and NVMe block device emulation scenario (for NVIDIA® BlueField®-2 only)

Supported Platforms and Interoperability

Supported BlueField Networking Platforms

HBN 2.0.0 has been validated on the following NVIDIA BlueField Networking Platforms:

- BlueField-2 DPU platforms:
    - 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 DPU platforms:
    - 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

Note

Single-port BlueField platforms are currently not supported with HBN.

Supported BlueField OS

HBN 2.0.0 supports DOCA 2.5.0 (BSP 4.5.0) on Ubuntu 22.04 OS.

Verified Scalability Limits

HBN 2.0.0 has been tested to sustain the following maximum scalability limits:

Limit	BlueField-2	BlueField-3	Comments
VTEP peers (DPUs per control plane) in the fabric	2k	2k	Number of DPUs (VTEPs) within a single overlay fabric (reachable in the underlay)
VNIs/overlay networks in the fabric	18	18	Total number of L2 VNIs in the fabric (max VNIs = max VF + max PF)
DPUs per VNI/overlay network	3, 2000	3, 2000	Total number of DPUs configured with the same VNI (3 real DPUs, 2000 emulated VTEPs)
Tenants (L3 VNIs) per server	8	8	Maximum number of tenants on the same host server
VM/pods per server	16	16	Maximum number of IP addresses advertised by EVPN in DPU
Maximum number of L3 LPM routes (underlay)	256	256	IPv4 or IPv6 underlay LPM routes per DPU

Limit	BlueField-2	BlueField-3	Comments
Maximum number of EVPN type-2 entries	4K	4k	Remote overlay MAC/IP entries for compute peers stored on a single DPU (L2 EVPN use case)
Maximum number of EVPN type-5 entries	16K	16K	Remote overlay L3 LPM entries for compute peers stored on a single DPU (L3 EVPN use case) **Info** Supported at beta level.
Maximum number of PFs	2	2	Total number of PFs visible to the host
Maximum number of VFs	16	16	Total number of VFs created on the host

Known Issues

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

Reference	Description
3705894	Description: In an EVPN Symmetric Routing scenario, IPv6 traffic is not hardware offloaded. It is only IPv6 traffic that is routed using L3VNIs to remote hosts that is affected.
	Workaround: N/A
	Keyword: EVPN; IPv6
	Reported in HBN version: 2.0.0
3378928	Description: When an interface is brought down or deleted (e.g., an SVI deletion), the routes learned over that interface, though removed from kernel, are not notified to netlink. Hence, these routes are still present in nl2doca and consequently in the FDB. If upon raising an interface these older routes are not newly installed, then those stale routes in nl2doca remain until nl2doca is restarted or a suggested workaround is applied.
	Workaround: Resync netlink cache with kernel. `echo 1 > /cumulus/nl2docad/ctrl/netlink/resync`
	Keyword: Container
	Reported in HBN version: 2.0.0
3519324	Description: The DOCA HBN container takes about 1 minute longer to spawn, as compared to previous HBN release (1.4.0).
	Workaround: N/A
	Keyword: Container
	Reported in HBN version: 1.5.0
3605486	Description: When the DPU boots up after issuing a "reboot" command from the DPU itself, some host-side interfaces may remain down.
	Workaround: N/A
	Keyword: Reboot
	Reported in HBN version: 1.5.0

3547103	Description: IPv6 stateless ACLs are not supported.
	Workaround: N/A
	Keyword: IPv6 ACL
	Reported in HBN version: 1.5.0
3339304	Description: Statistics for hardware-offloaded traffic are not reflected on SFs inside an HBN container.
	Workaround: Look up the stats using `ip -s link show` on PFs outside of the HBN container. PFs would show Tx/Rx stats for traffic that is hardware-accelerated in the HBN container.
	Keyword: Statistics; container
	Reported in HBN version: 1.4.0
3352003	Description: NVUE show, config, and apply commands malfunction if the `nvued` and `nvued-startup` services are not in the `RUNNING` and `EXITED` states respectively.
	Workaround: N/A
	Keyword: NVUE commands
	Reported in HBN version: 1.3.0
3168683	Description: If many interfaces are participating in EVPN/routing, it is possible for the routing process to run out of memory.
	Workaround: Have a maximum of 8 VF interfaces participating in routing/VXLAN.
	Keyword: R outing; memory
	Reported in HBN version: 1.2.0
3219539	Description: TC rules are programmed by OVS to map uplink and host representor ports to HBN service. These rules are ageable and can result in packets needing to get software forwarded periodically to refresh the rules.
	Workaround: The timeout value can be adjusted by changing the OVS parameter `other_config : max-idle` as documented here. The shipped default value is 10000ms (10s).
	Keyword: SFC; aging
	Reported in HBN version: 1.2.0

3184745	**Description: The command `nv show interface <intf> acl` does not show correct information if there are multiple ACLs bound to the interface.**
	Workaround: Use the command `nv show interface <intf>` to view the ACLs bound to an interface.
	Keyword: ACLs
	Reported in HBN version: 1.2.0
3158934	**Description: Deleting an NVUE user by removing their password file and restarting the `decrypt-user-add` service on the HBN container does not work.**
	Workaround: Either respawn the container after deleting the file, or delete the password file corresponding to the user by running `userdel -r username`.
	Keyword: User deletion
	Reported in HBN version: 1.2.0
3185003	**Description: When a packet is encapsulated with a VXLAN header, it adds extra bytes which may cause the packet to exceed the MTU of link. Typically, the packet would be fragmented but its silently dropped and no fragmentation happens.**
	Workaround: Make sure that the MTU on the uplink port is always 50 bytes more than host ports so that even after adding VXLAN headers, ingress packets do not exceed the MTU.
	Keyword: MTU; VXLAN
	Reported in HBN version: 1.2.0
3184905	**Description: On VXLAN encapsulation, the DF flag is not propagated to the outer header. Such a packet may be truncated when forwarded in the kernel, and it may be dropped when hardware offloaded.**
	Workaround: Make sure that the MTU on the uplink port is always 50 bytes more than host ports so that even after adding VXLAN headers, ingress packets do not exceed the MTU.
	Keyword: VXLAN
	Reported in HBN version: 1.2.0

3188688	**Description: When stopping the container using the command `crictl stop` an error may be reported because the command uses a timeout of 0 which is not enough to stop all the processes in the HBN container.**
	Workaround: Pass a timeout value when stopping the HBN container by running: `crictl stop --timeout 60 <hbn-container>`
	Keyword: Timeout
	Reported in HBN version: 1.2.0
3129749	**Description: The same ACL rule cannot be applied in both the inbound and outbound direction on a port.**
	Workaround: N/A
	Keyword: ACLs
	Reported in HBN version: 1.2.0
3126560	**Description: The system's time zone cannot be modified using NVUE in the HBN container.**
	Workaround: The timezone can be manually changed by symlinking the `/etc/localtime` file to a binary time zone's identifier in the `/usr/share/zoneinfo` directory. For example: `sudo ln -sf /usr/share/zoneinfo/GMT /etc/localtime`
	Keyword: Time zone; NVUE
	Reported in HBN version: 1.2.0
3118204	**Description: Auto-BGP functionality (where the ASN does not need to be configured but is dynamically inferred by the system based on the system's role as a leaf or spine device) is not supported on HBN.**
	Workaround: If BGP is configured and used on HBN, the BGP ASN must be manually configured.
	Keyword: BGP
	Reported in HBN version: 1.2.0

3233088	Description: Since checksum calculation is offloaded to the hardware (not done by the kernel), it is expected to see an incorrect checksum in the tcpdump for locally generated, outgoing packets. BGP keepalives and updates are some of the packets that show such incorrect checksum in tcpdump.
	Workaround: N/A
	Keyword: BGP
	Reported in HBN version: 1.2.0
2821785	Description: MAC addresses are not learned in the hardware but only in software. This may affect performance in pure L2 unicast traffic.
	Workaround: N/A
	Keyword: MAC; L2
	Reported in HBN version: 1.3.0
3017202	Description: Due to disabled backend foundation units, some NVUE commands return `500 INTERNAL SERVER ERROR`/`404 NOT FOUND`. These commands are related to features or subsystems which are not supported on HBN.
	Workaround: N/A
	Keyword: Unsupported NVUE commands
	Reported in HBN version: 1.3.0
2828838	Description: NetworkManager and other services not directly related to HBN may display the following message in syslog: `"netlink: read: too many netlink events. Need to resynchronize platform cache"` The message has no functional impact and may be ignored.
	Workaround: N/A
	Keyword: Error
	Reported in HBN version: 1.3.0

Bug Fixes

The following table lists the known issues which have been fixed for this release of HBN.

Reference	Description
3610971	Description: The output of the command `nv show interface` does not display information about VRFs, VXLAN, and bridge.
	Fixed in HBN version: 2.0.0
3378928	Description: Service functions (`*_sf`) inside the HBN container are UP at container start irrespective of their presence/absence in the `/etc/network/interfaces` file. But once any of them are added to /e/n/i and later taken off from /e/n/i, they stay DOWN unless added back to /e/n/i.
	Fixed in HBN version: 2.0.0
3452914	Description: IPv6 OOB connectivity from the HBN container stops working if the br-mgmt interface on the DPU goes down. When going down, the br-mgmt interface loses its IPv6 address, which is used as the gateway address for the HBN container. If the br-mgmt interface comes back up, its IPv6 address is not added back and IPv6 OOB connectivity from the HBN container will not work
	Fixed in HBN version: 1.5.0
3191433	Description: ECMP selection for the underlay path uses the ingress port and identifies uplink ports via round robin. This may not result in uniform spread of the traffic.
	Fixed in HBN version: 1.4.0
3049879	Description: When reloading (`ifreload`) an empty `/etc/network/interfaces` file, the previously created interfaces are not deleted.
	Fixed in HBN version: 1.4.0
3284607	Description: When an ACL is configured for IPv4 and L4 parameters (protocol tcp/udp, source, and destination ports) match, the ACL also matches IPv6 traffic with the specified L4 parameters.
	Fixed in HBN version: 1.4.0
3282113	Description: Some DPUs experience an issue with the clock settings after installing a BlueField OS in an HBN setting in which the date reverts back to "Thu Sep 8, 2022".
	Fixed in HBN version: 1.4.0

3354029	**Description: If interfaces on which BGP unnumbered peering is configured are not defined in the `/etc/network/interfaces` configuration file, BGP peering does not get established on them.**
	Fixed in HBN version: 1.4.0

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.

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

> **Note**
>
> 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).

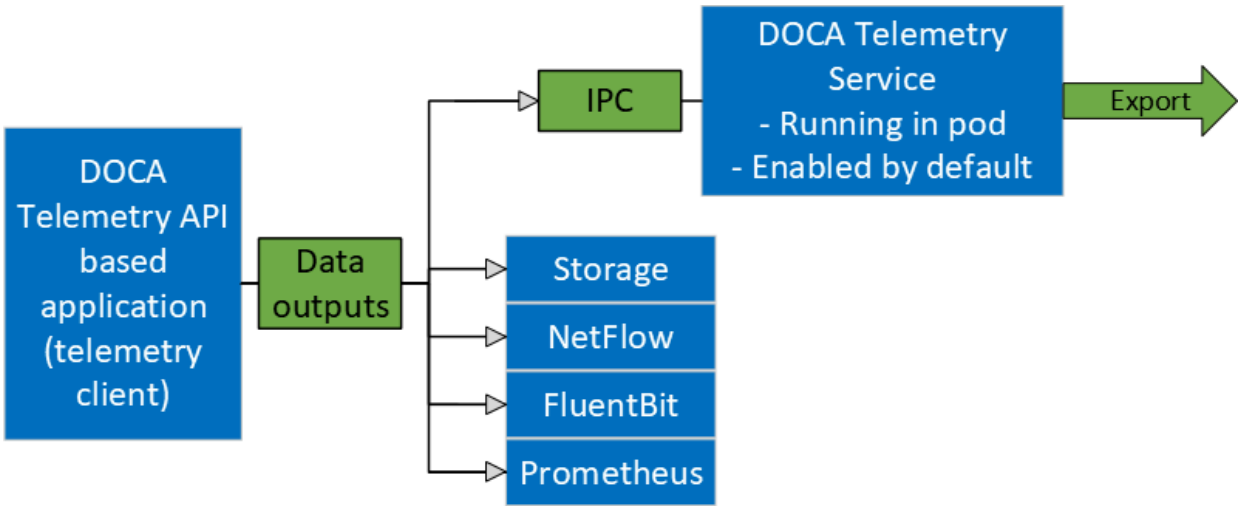
DTS allows exporting NetFlow packets when data is collected from the DOCA Telemetry NetFlow API client application. NetFlow exporter is enabled from `dts_config.ini` by setting NetFlow collector IP/address and port.

Service Deployment

Available Images

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 2024-10-09_07-10-18_DOCA Telemetry library.



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 .

Info

For more information about the deployment of DOCA containers on top of the BlueField DPU, refer to NVIDIA DOCA Container Deployment Guide.

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

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:

```
export DTS_IMAGE=nvcr.io/nvidia/doca/doca_telemetry
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"
```

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_so
\
              -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
```

Note

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`

Deployment with Grafana Monitoring

Refer to section "Deploying with Grafana Monitoring".

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.

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.

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"]
```

Note

The host and port shown above are just an example. See section Fluent Bit to learn about manual configuration.

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.

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.

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
```

> **Note**
>
> More information about telemetry providers can be found under the Providers section.

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 the means by which 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
```

Enabling Data Write

Uncomment the following line in `dts_config.ini`:

```
#output=/data
```

Note

Changes in `dts_config.ini` force the main DTS process to restart in 60 seconds to apply the new settings.

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 2024-10-09_07-10-18_DOCA Telemetry.

Description

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.

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
```

Note

`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
```

> **Note**
>
> 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
```

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
```

> **Note**
>
> DPE server should be active before changing the `dts_config.ini` file. See section "Remote Collection" for details.

Ethtool Counters

Ethtool counters is the generated list of counters which corresponds to Ethtool utility. Counters are generated on a per-device basis. See this community post for more information on mlx5 ethtool counters.

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

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
```

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.

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
```

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.

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
```

> **Note**
>
> DPE server should be active before changing the `dts_config.ini` file. See section "Remote Collection" for details.

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
```

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
```

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
```

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)

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.

Note

`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
```

Note

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"
}
…
```

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
```

> **Note**
>
> The default port for Prometheus can be changed in `dts_config.ini`.

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
```

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
```

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

> **Note**
>
> All `.exp` files are disabled by default if not configured by `initContainer` entry point through `.yaml` file.

> **Note**
>
> 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.

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.

> **Note**
>
> Use `#` to comment a configuration line.

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,...}}
```

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"`.

> **Note**
>
> 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.
```

NetFlow Exporter

NetFlow exporter must be used when data is collected as NetFlow packets from the telemetry client applications. In this case, DOCA Telemetry 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.

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).

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`.

DPE Configuration File

DPE can be configured by the user. This section covers the syntax and implications of its configuration file.

> **Note**
>
> 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]
```

Note

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`.

Note

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.

Deploying with Grafana Monitoring

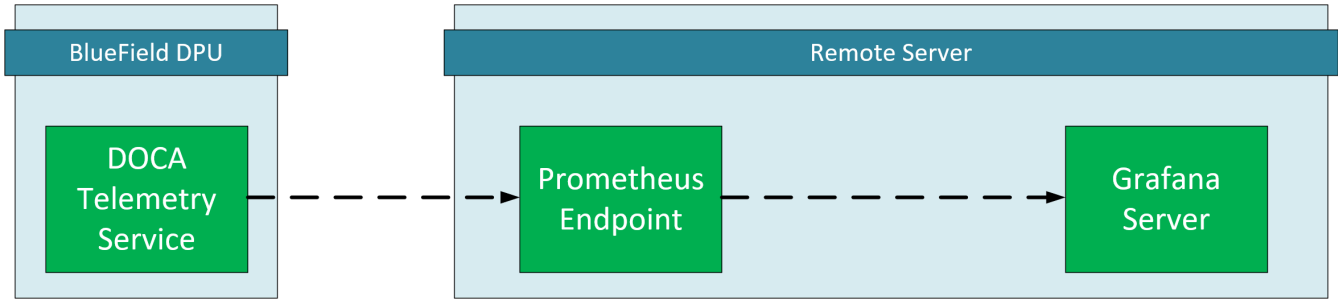
This chapter provides an overview and deployment configuration of DOCA Telemetry Service with Grafana .

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.

Grafana Deployment Configuration



DTS Configuration (DPU Side)

DTS will be configured to export the sysfs counter using the Prometheus plugin.

> **Note**
>
> Sysfs is used as an example, other counters are available. Please refer to the NVIDIA DOCA Telemetry Service Guide for more information.

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
```

Note

In this example, the Prometheus plugin exports data on localhost port 9100, this is an arbitrary value and can changed.

Note

DTS must be restarted to apply changes.

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"
```

> **Tip**
>
> Prometheus services are available as Docker images. Please refer to Using Docker in Prometheus' Installation guide.

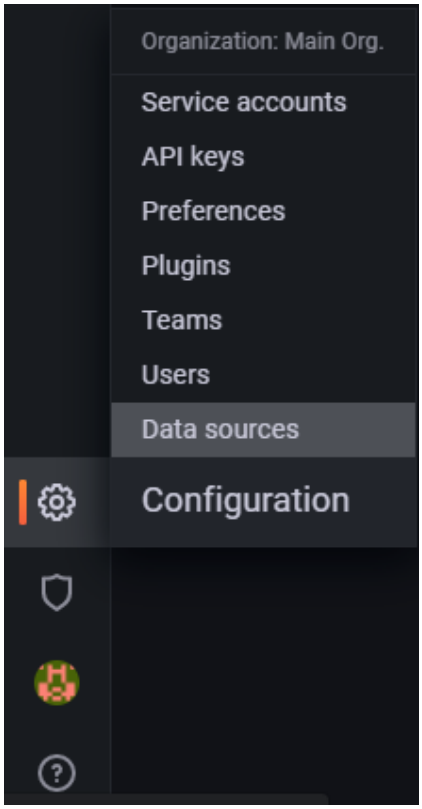
Grafana Configuration (Remote Server)

Please download and install Grafana for your platform.

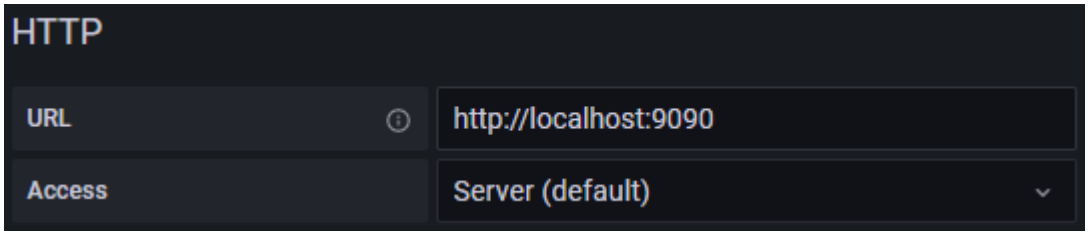
1. Setup Grafana. Please refer to Install Grafana guide in Grafana documentation.
2. Log into the Grafana dashboard at http://localhost:3000.

> **Note**
>
> 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.



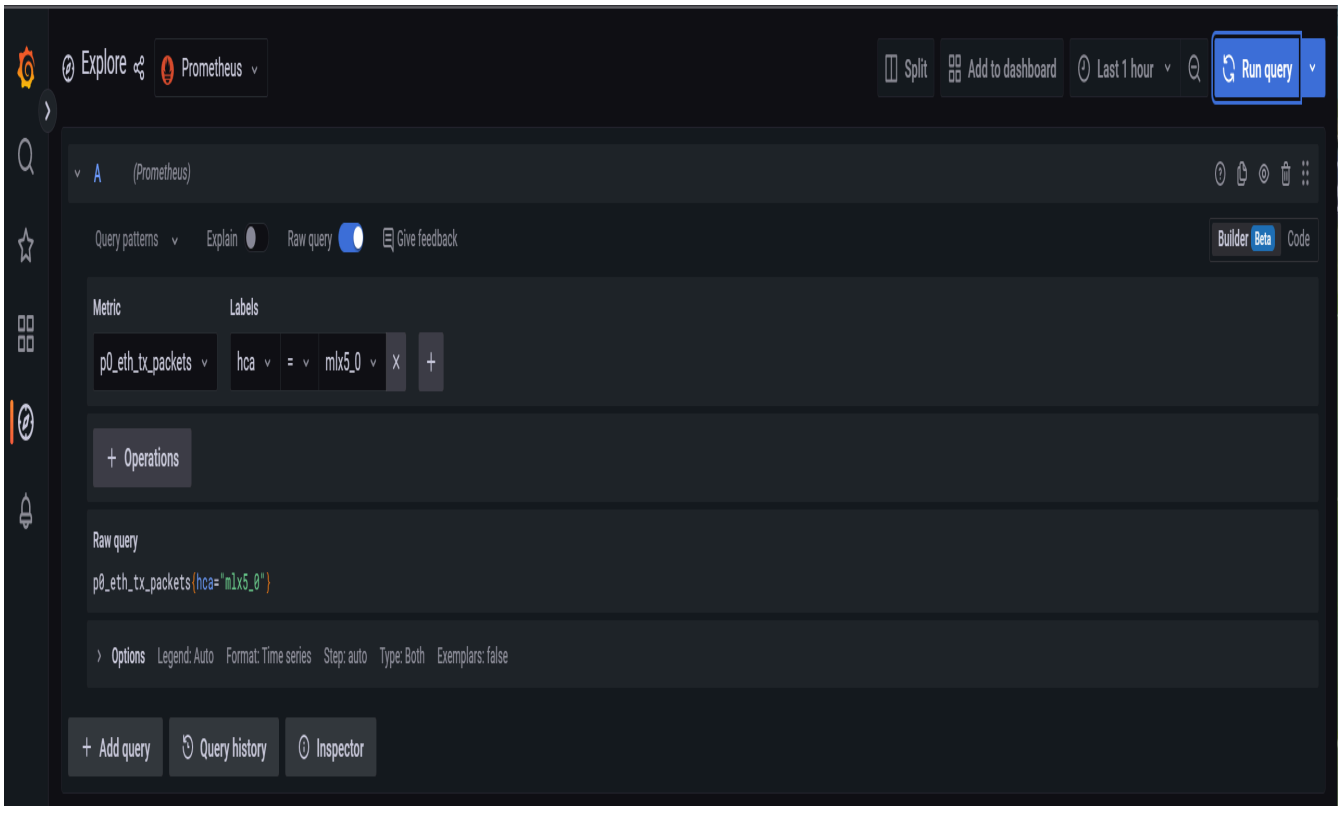
> **Note**
>
> The Prometheus server's default listen port is 9090. Prometheus and Grafana are both running on the same server, thus the address is localhost.

5. Save and test.

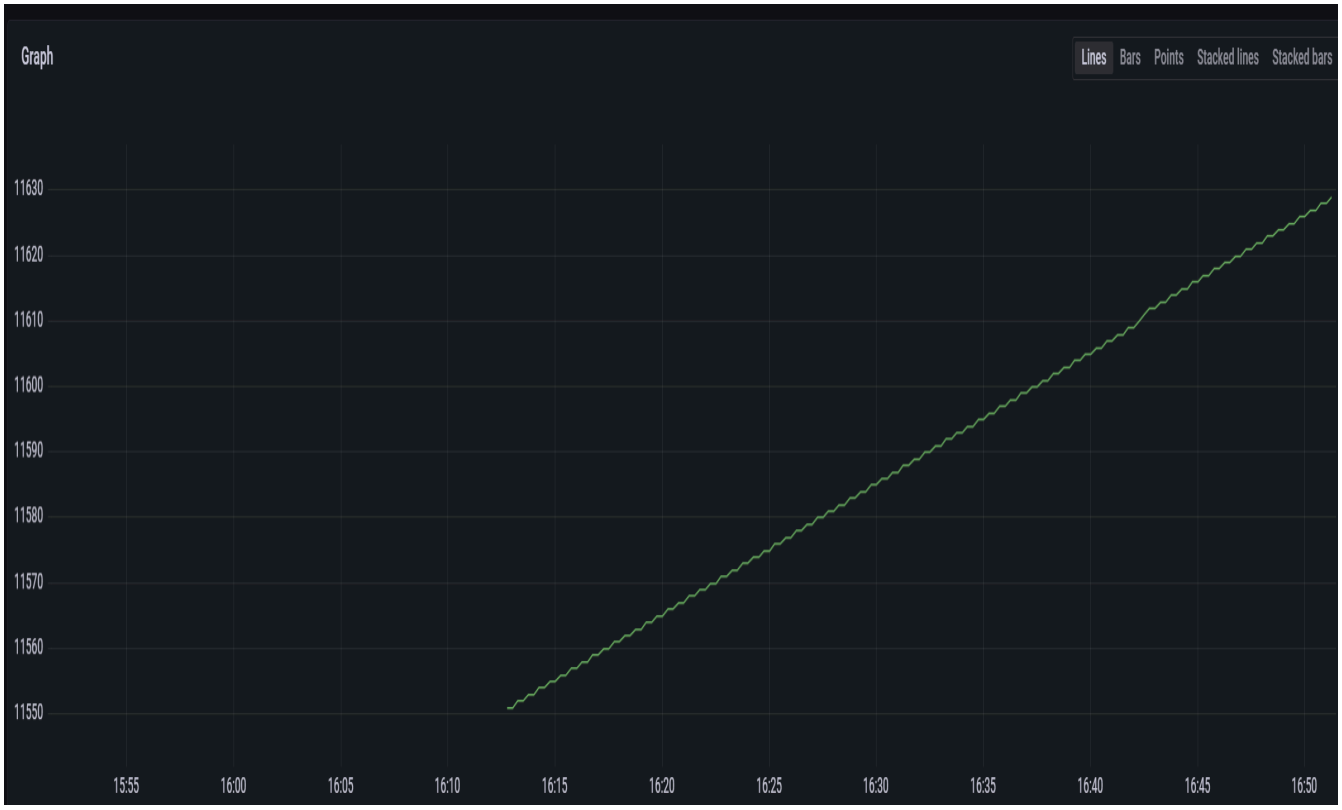
Exploring Telemetry Data

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.



Graph display after selecting a metric and specifying a label to filter by:



Troubleshooting

On top of the Troubleshooting section in the NVIDIA DOCA Container Deployment Guide, here are additional troubleshooting tips for DTS:

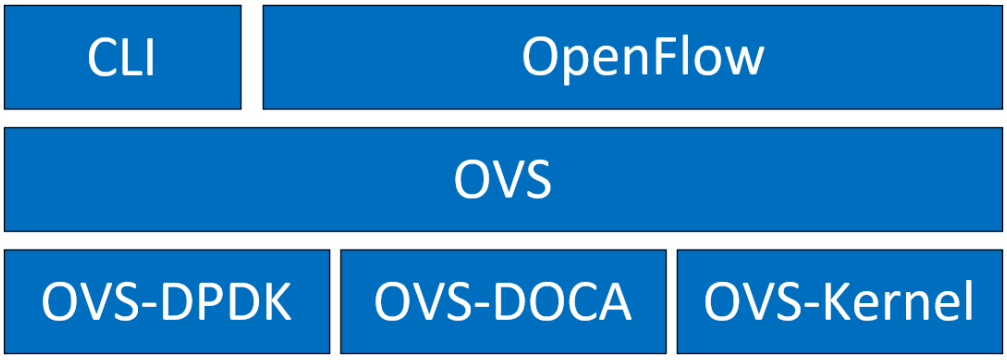
- For general troubleshooting, refer to the NVIDIA DOCA Troubleshooting Guide.
- If the pod's state fails to be marked as "Ready", refer to `/var/log/syslog`.
- Check if the service is configured to write data to the disk as this may cause the system to run out of disk space.
- If a PIC bus error occurs, 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
"
```

OpenvSwitch Offload

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 OVS architecture extends the traditional OVS-DPDK and OVS-Kernel data-path offload interfaces, introducing OVS-DOCA as an additional implementation. OVS-DOCA, built upon NVIDIA's networking API, preserves the same interfaces as OVS-DPDK and OVS-Kernel while utilizing the DOCA Flow library. Unlike the other modes, OVS-DOCA 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.

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

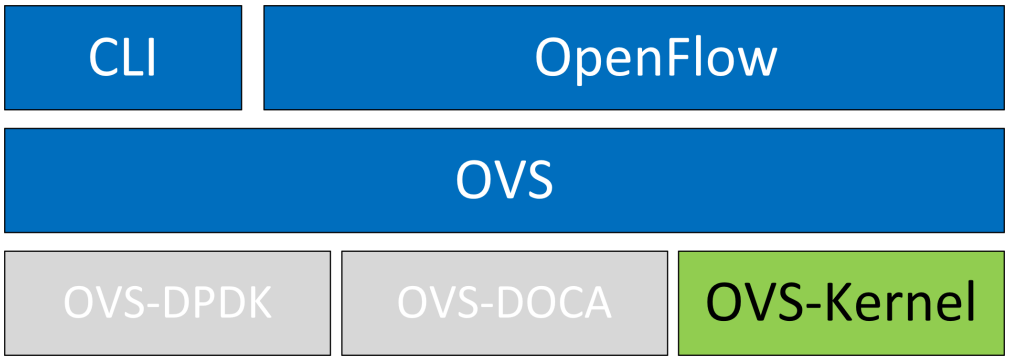
subsystem), an alternate virtualization technology implements full virtio emulation within the DPU, enabling the host server to communicate with the DPU as a software virtio device.

- 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.

OVS Hardware Offloads Configuration

OVS-Kernel Hardware Offloads

OVS-Kernel is the default OVS flavor enabled on your NVIDIA device.



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
```

> **Note**
>
> 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.

> **Note**
>
> Before changing the mode, make sure that all VFs are unbound.

> **Info**
>
> 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
```

Switchdev Performance Tuning

Switchdev tuning improves its performance.

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).

Troubleshooting SMFS

mlx5 debugfs supports presenting Software Steering resources. `dr_domain` including its tables, matchers and rules. The interface is read-only.

> **Note**
>
> New steering rules cannot be inserted/deleted w hile 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.

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. **Note** 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
```

Note

This mode must be set prior to moving to switchdev.

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
```

Note

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.

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
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=e4:11:22:3
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.

> **Note**
>
> Users can also verify offloaded packets by adding `type=offloaded` to the command. For example:
>
> ```
> ovs-appctl dpctl/dump-flows type=offloaded
> ```

OVS Performance Tuning

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
```

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
```

> **Note**
>
> TC policy should only be used for debugging purposes.

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
```

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.

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
```

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
```

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]
   ```

> **Note**
>
> 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
```

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.

SR-IOV VF LAG Configuration on ASAP2

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
```

Note

Once the SR-IOV VF LAG is configured, all VFs of the two PFs become part of the bond and behave as described above.

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
```

1. 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
```

2. 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
```

3. 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):

  1. 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
```

  2. 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
```

Classification Fields (Matches)

OVS-Kernel supports multiple classification fields which packets can fully or partially match.

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=00:02:10:40:10:0
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
```

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,h
```

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
```

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
```

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
```

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(+
```

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
```

Supported Actions

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=00:02:10:40:10:0
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
```

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=00:02:10:40:10:0
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
```

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=00:02:10:40:10:0
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
```

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
```

Info

For a GRE IPv4 tunnel, use `type=gre`. For a GRE IPv6 tunnel, use `type=ip6gre`. For a Geneve tunnel, use `type=geneve`.

Note

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
```

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

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:b
\
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=e4:11:22:33:44:5
\
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
```

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
```

Header Rewrite

This action allows for modifying packet fields.

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=00:02:10:40:10:0
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
```

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
```

> **Note**
>
> IPv4 and IPv6 header rewrite is only supported with match on UDP/TCP/ICMP protocols.

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
```

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
```

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
```

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=enp8s0f0
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
```

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
```

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
```

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
```

Note

TC supports up to 32 actions.

Note

If header rewrite is used, then all destinations should have the same header rewrite.

Note

If VLAN push/pop is used, then all destinations should have the same VLAN ID and actions.

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
```

Note

A userspace application is needed to process the sampled packet from the kernel. An example is available on Github.

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
```

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`

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
```

Note

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
```

Representor Metering

Info

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

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_typ
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.

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 nvconig 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:

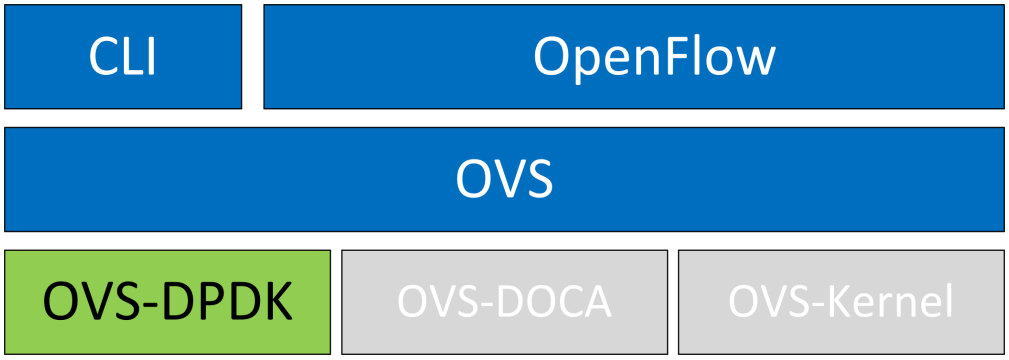
   ```
   echo multiport_esw >
   /sys/class/net/enp8s0f0/compat/devlink/lag_port_select_mode
   echo multiport_esw >
   /sys/class/net/enp8s0f1/compat/devlink/lag_port_select_mode
   ```

   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
```

OVS-DPDK Hardware Offloads



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
```

> **Note**
>
> 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
```

> **Note**
>
> 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
```

> **Info**
>
> 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=0000:88:00.0,representor=[0]
```

Where `representor=[0-N]`.

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.

Configuring VXLAN Encap/Decap Offloads

> **Note**
>
> 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
```

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
```

SR-IOV VF LAG

To configure OVS-DPDK SR-IOV VF LAG:

1. Enable SR-IOV on the NICs:

   ```
   mlxconfig -d <PCI> set SRIOV_EN=1
   ```

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>
```

> **Info**
>
> 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
```

VirtIO Acceleration Through VF Relay: Software and Hardware vDPA

Note

Hardware vDPA is enabled by default. In case 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.

Note

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>
```

Note

`vdpa-max-queues` is an optional field. When the user wants to configure 32 vDPA ports, the maximum queues number is limited to 8.

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,representor=[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
```

Note

To configure SW vDPA, add `options:vdpa-sw=true` to the command.

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,representor=[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
```

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=o
</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=9216
```

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
```

E2E Cache

> **Note**
>
> 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
```

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>
```

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
```

Note

Refer to the OVS user manual for more information.

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
```

Note

Currently sFlow for OVS-DPDK is supported without CT.

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.

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 -0 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 -0 openflow13 add-flow $bridge
"table=0,actions=meter:$id,normal"
```

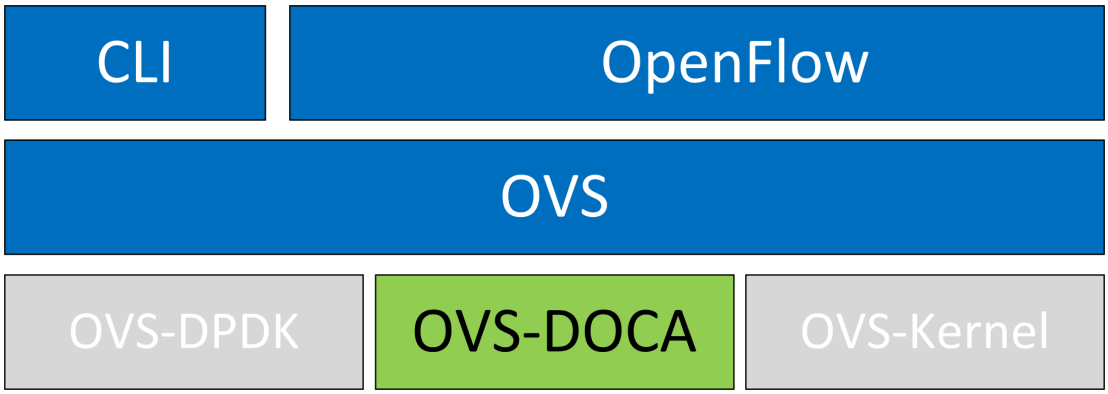
3. View the meter statistics:

```
ovs-ofctl -O openflow13 meter-stats $bridge meter=$id
```

4. For more information, refer to official OVS documentation.

OVS-DOCA Hardware Offloads

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), and northbound API 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.

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
```

> **Note**
>
> 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
```

> **Note**
>
> 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
```

> **Info**
>
> 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 pf -- set Interface pf type=dpdk
options:dpdk-devargs=0000:88:00.0,dv_flow_en=2,dv_xmeta_en=4
```

> **Info**
>
> OVS-DOCA uses DPDK ports and configuration. Note the different dpdk-devargs parameters.

10. Add representor to OVS:

```
ovs-vsctl add-port br0-ovs representor -- set Interface
representor type=dpdk options:dpdk-
devargs=0000:88:00.0,representor=[<vf-
number>],dv_flow_en=2,dv_xmeta_en=4
```

> **Note**
>
> Note that `<vf-number>` must be replaced by the number of the VF.

11. Optional configuration:

    1. To set port MTU, run:

```
ovs-vsctl set interface pf mtu_request=9000
```

> **Note**
>
> OVS restart is required for changes to take effect.

    2. To set VF/SF MAC, run:

```
ovs-vsctl add-port br0-ovs representor -- set Interface
representor type=dpdk options:dpdk-
devargs=0000:88:00.0,representor=[<vf-
number>],dv_flow_en=2,dv_xmeta_en=4 options:dpdk-vf-
mac=00:11:22:33:44:55
```

> **Note**
>
> Unbinding and rebinding the VFs/SFs is required for the change to take effect.

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.

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.

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.

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.

OVS-DOCA Specific vSwitch Configuration

The following configuration is particularly useful or specific to OVS-DOCA mode.

> **Info**
>
> The full list of OVS vSwitch configuration is documented in `man ovs-vswitchd.conf.db`.

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
`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
`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

Configuration	Description
`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

Configuration	Description
`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.
`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.

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

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 p0 -- set Interface p0 type=dpdk
options:dpdk-devargs=0000:03:00.0,dv_flow_en=2,dv_xmeta_en=4
```

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],dv_flow_en=2,dv_xmeta_en=4
```

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
```

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
```

SR-IOV VF LAG

To configure OVS-DOCA SR-IOV VF LAG:

1. Enable SR-IOV on the NICs:

   ```
   mlxconfig -d <PCI> set SRIOV_EN=1
   ```

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>
   ```

> **Note**
>
> 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. To work with 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,dv_flow_en=2,dv_xmeta_en=4
options:dpdk-lsc-interrupt=true
```

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,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
```

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.

2. 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
   ```

> **Info**
>
> The mode becomes operational after entering switchdev mode on both PFs.

3. 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 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
```

Offloading Geneve Encapsulation/Decapsulation

Geneve tunneling offload support includes matching on extension header.

Note

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
- Users must change firmware configuration to enable the flex parser by running the following commands:

```
mst start
mlxconfig -d <mst device> s
FLEX_PARSER_PROFILE_ENABLE=8
mlxfwreset -d <mst device> r -y
```

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 pf -- set Interface pf type=dpdk
options:dpdk-devargs=<PF PCI>,dv_flow_en=2,dv_xmeta_en=4
```

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 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>
```

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 pf -- set Interface pf type=dpdk
options:dpdk-devargs=<PF PCI>,dv_flow_en=2,dv_xmeta_en=4
```

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 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 gre0 -- set interface gre0 type=gre
options:key=<VNI> options:remote_ip=<$remote_ip_1>
options:local_ip=<$local_ip_1>
```

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.

> **Info**
>
> 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)

- The hash calculation may be different for packets going into software vs. ones that are offloaded
- Does not work concurrently with CT (i.e., configure `hw-offload-ct-size="0"` beforehand)

OVS-DOCA Known Limitations

- Only one insertion thread is supported (n-offload-threads=1)
- Only 250K connection are offloadable by default (can be configured)
- Only 8 CT zones are supported by CT offload
- Offload of IPv6 tunnels are not supported

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.
`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.

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.

OVS Inside the DPU

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
   ```

Verifying Connection from Host to BlueField

There are two SFs configured on the BlueFIeld-2 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
```

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
```

API References

This section contains the following pages:

- NVIDIA DOCA Driver APIs
- NVIDIA DOCA Library APIs

NVIDIA DOCA Driver APIs

The driver APIs for this DOCA version are available here.

NVIDIA DOCA Library APIs

The library APIs for this DOCA version are available here.

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 DOCA gRPC Infrastructure User Guide
- NVIDIA BlueField DPU Modes of Operation
- NVIDIA DOCA Switching Support
- 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

NVIDIA DOCA Glossary

Term	Description
ACS	Access control services
ATF	Arm-trusted firmware
BDF address	Bus, device, function address. This is the device's PCIe bus address to uniquely identify the specific device.

Term	Description
BFB	BlueField bootstream
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
CTX	Context
DEK	Data encryption key
DMA	Direct memory access
DOCA	DPU SDK
DPA	Data path accelerator; a n auxiliary processor designed to accelerate data-path operations
DPCP	Direct packet control plane
DPDK	Data plane development kit
DPI	Deep packet inspection
DPU	Data processing unit, the third pillar of the data center with CPU and GPU
DW	Dword
EBS	Excess burst size
ECE	Enhanced connection establishment
ECPF	Embedded CPU physical function
EIR	Excess information rate
eMMC	Embedded multi-media card
ESP	EFI system partition
ESP header	Encapsulating security payload
EU	Execution unit. HW thread; a logical DPA processing unit.

Term	Description
FIPS	Federal Information Processing Standards
FPGA	Field-programmable gate arrays
FW	Firmware
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
IR	Intermediate representation
LSO	Large send offload
LTO	Link-time optimization
MFT	Mellanox firmware tools
MLNX_OFED	Mellanox OpenFabrics Enterprise Distribution
MSB	Most significant bit
MSS	Memory subsystem
MST	Mellanox software tools
NAT	Network address translation
NIC	Network interface card

Term	Description
NIST	National Institute of Standards and Technology
NS	Namespace
OOB	Out-of-band
OS	Operating system
OVS	Open vSwitch
PBS	Peak burst size
PCIe	PCI Express; Peripheral Component Interconnect Express
PF	Physical function
PE	Progress engine
PIR	Peak information rate
PK	Platform key
PKA	Public key accelerator
POC	Proof of concept
RDMA	Remote direct memory access
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
RQ	Receive queue
RShim	Random shim
RTT	Round-trip time
RX	Receive
RXP	Regular expression processor
SA	Security association

Term	Description
SBSA	Server base system architecture
SDK	Software development kit
SF	Sub-function or scalable function
SG	Scatter-gather
SHA	Secure hash algorithm
SNAP	Storage-defined network-accelerated processing
SPDK	Storage performance development kit
SQ	Send queue
SR-IOV	Single-root IO virtualization
Sync event	Synchronization event
TAI	International Atomic Time
TIR	Transport interface receive
TIS	Transport interface send
TLS	Transport layer security
TX	Transmit
UDS	Unix domain socket
UEFI	Unified extensible firmware interface
UTC	Coordinated Universal Time
VF	Virtual function
VFE	Virtio full emulation
VM	Virtual machine
VMA	NVIDIA® Messaging Accelerator
VPI	Virtual protocol interconnect
WorkQ or workq	Work queue
WQE	Work queue elements
WR	Write
XLIO	NVIDIA® Accelerated IO

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.

Note

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.

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.

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.

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
```

> **Note**
>
> 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.

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
```

Note

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.

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.

NVIDIA DOCA DPU CLI

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

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

> **Note**
>
> 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`).

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. **Note** This command shows the speed of the network card of the DPU.

Command	Description
`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. **Note** 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).

DPU/DOCA Commands

Command	Description
`ibdev2netdev`	Displays available `mlx`
`mst`	Used to start MST ser with NVIDIA devices li
`cat /etc/mlnx-release`	Displays the full BlueF

Command	Description
`cat /etc/os-release`	Displays the details of
`ibv_devinfo`	Displays the current Ir relevant information. version.
`ipmitool power cycle`	Power cycle **Note** Prior to perfor sure to do a gr
`echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages`	DPDK setup. Allocates abstraction layer (EAL
`mlxdevm tool`	The mlxdevm tool is f `/opt/mellanox/ip` possible to create an HW address and set i 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 f unique SF number. Ex
	`/opt/mellanox/`
	`add pci/0000:03`
	`sfnum 4`

`/opt/mellanox/iproute2/sbin/mlxdevm port show`	Displays information a

Command	Description
`/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 capabil making it "trusted", ar `<sf_index>` the SF. `mlxdevm port sho` `/opt/mellanox/ function set p 02:25:f2:8d:a2`
`$ 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 command unbinds th second command bin deployment phase sh SF are configured. Th which can be obtaine
`ls /sys/bus/auxiliary/devices/mlx5_core.sf.*`	Displays additional inf their "next serial num For example, if `mlx5_` command, then runni `cat /sys/bus/auxilia` would output the sfnu
`/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 First, users must set t then should it be dele
`/opt/mellanox/iproute2/sbin/mlxdevm port help`	Displays additional inf used on created SF p
`crictl pods`	Displays currently act up to 20-30 seconds
`crictl ps`	Displays currently act
`crictl ps -a`	Displays all containers finished their executi
`crictl logs <container-id>`	Examines the logs of

Command	Description
`crictl exec -it <container-id> /bin/bash`	Attaches a shell to a r
`journalctl -u kubelet`	Examines the Kubelet to spawn.
`crictl stopp <pod-id>`	Stops a running K8S p
`crictl stop <container-id>`	Stops a running conta
`crictl rmi <image-id>`	Removes a container

NVIDIA DOCA Emulated Devices

This document describes the ability of NVIDIA® BlueField® DPU to emulate and accelerate physical and virtual host functions.

VirtIO-net Emulated Devices

Virtio-net device emulation enables users to create VirtIO-net emulated PCIe devices in the system where the NVIDIA® BlueField® DPU is connected. This is done by the virtio-net-controller software module present in the DPU. Virtio-net emulated devices allow users to hot plug up to 31 virtio-net PCIe PF Ethernet NIC devices or 504 virtio-net PCIe VF Ethernet NIC devices in the host system where the DPU is plugged in.

> **Note**
>
> Currently, VirtIO specification v1.0 is supported.

DPU software also enables users to create virtio block PCIe PF and SR-IOV PCIe VF devices. This is covered in the *NVIDIA BlueField SNAP and virtio-blk SNAP Documentation*.

VirtIO-net Controller

Virtio-net-controller is a systemd service running on the DPU, with a user interface frontend to communicate with the background service. An SF representor is created for

each virtio-net device created on the host. Virtio-net controller only uses an SF number ⩾1000. Refer to section "Scalable Functions" for more information.

> **Note**
>
> SF representor name is determined by udev rules. The default name is in the format of `<prefix><pf_num><sf_num>`. For example: `en3f0pf0sf1001`.

Each virtio-net PF/VF requires a dedicated SF and it should be reserved from mlxconfig (see section "VirtIO-net PF Device Configuration"). However, since an SF is a shared resource on the system, there may be other application-created SFs as well. In that case, `PF_TOTAL_SF` must be updated to consider those SFs. Otherwise, virtio-net is not able to create enough configured PF/VF.

> **Note**
>
> Since the controller provides hardware resources and acknowledges (ACKs) the request from the host's virtio driver, it is mandatory to reboot the host OS first and the DPU second. This also applies to reconfiguring a controller from the DPU (e.g., reconfiguring LAG); unloading the virtio-net driver from guest side is recommended.

SystemD Service

Controller systemd service is enabled by default and runs automatically if `VIRTIO_NET_EMULATION_ENABLE` is true from mlxconfig.

1. To check controller service status, run:

```
$ systemctl status virtio-net-controller.service
```

2. To reload the service, make sure to unload virtio-net/virtio-pcie drivers on host. Then run:

```
$ systemctl restart virtio-net-controller.service
```

3. To monitor log output of the controller service, run:

```
$ journalctl -u virtio-net-controller -f
```

The controller service has an optional configuration file which allows users to customize several parameters. The configuration file should be defined on the DPU at the following path `/opt/mellanox/mlnx_virtnet/virtnet.conf`.

This file is read every time the controller starts. Dynamic change of `virtnet.conf` is not supported. It is defined as a JSON format configuration file. The currently supported options are:

- `ib_dev_p0` – RDMA device (e.g., `mlx5_0`) used to create SF on port 0. This port is the EMU manager when `is_lag` is 0. Default value is `mlx5_0`.
- `ib_dev_p1` – RDMA device (e.g., `mlx5_1`) used to create SF on port 1. Default value is `mlx5_1`.
- `ib_dev_lag` – RDMA LAG device (e.g., `mlx5_bond_0`) used to create SF on LAG. Default value is `mlx5_bond_0`. This port is EMU manager when `is_lag` is 1. `ib_dev_lag` and `ib_dev_p0`/`ib_dev_p1` cannot be configured simultaneously.
- `ib_dev_for_static_pf` – the RDMA device (e.g., `mlx5_0`) which the static virtio PF is created on
- `is_lag` – specifies whether LAG is used. Note that if LAG is used, make sure to use the correct IB dev for static PF.
- `static_pf` –

- `mac_base` – base MAC address for static PFs. MACs are automatically assigned with the following pattern: `pf_mac` → `pf_0`, `pf_mac` +1→ `pf_1`, etc.

  > **Note**
  >
  > Note that the controller does not validate the MAC address (other than its length). The user must ensure MAC is valid and unique.

- `features` – virtio spec-defined feature bits for static PFs. If unsure, leave `features` out of the JSON file and a default value is automatically assigned.

- `vf` –
  - `mac_base` – base MAC address for static PFs. MACs are automatically assigned with the following pattern: `pf_mac` → `pf_0`, `pf_mac` +1→ `pf_1`, etc.
  - `features` – virtio spec-defined feature bits for static VFs. If unsure, leave `features` out of the JSON file and a default value is automatically assigned.
  - `vfs_per_pf` – number of VFs to create on each PF. This is mandatory if `mac_base` is specified.

    > **Note**
    >
    > This value does not equal `VIRTIO_NET_EMULATION_NUM_VF` in mlxconfig. `vfs_per_pf` ⩽ `VIRTIO_NET_EMULATION_NUM_VF`.

    - `qp_num` – number of QPs for each VF. If not specified, then the QP number assigned is taken from its parent PF.

- `recovery` – specifies whether recovery is enabled. If unspecified, recovery is enabled by default. To disable it, set `recovery` to 0.

- `sf_pool_percent` – determines the initial SF pool size as the percentage of `PF_TOTAL_SF` of `mlxconfig`. Valid range: [0, 100]. For instance, if the value is 5, it means an SF pool with 5% of `PF_TOTAL_SF` is created. 0 means no SF pool is reserved beforehand (default).

  > **Note**
  >
  > `PF_TOTAL_SF` is shared by all applications. User must ensure the percent request is guaranteed or else the controller will not be able to reserve the requested SFs resulting in failure.

- `sf_pool_force_destroy` – specifies whether to destroy the SF pool. When set to 1, the controller destroys the SF pool when stopped/restarted (and the SF pool is recreated if `sf_pool_percent` is not 0 when starting), otherwise it does not. Default value is 0.

For example, the following definition has all static PFs using mlx5_0 (port 0) as the data path device in a non-lag configuration:

```
{
  "ib_dev_p0": "mlx5_0",
  "ib_dev_p1": "mlx5_1",
  "ib_dev_for_static_pf": "mlx5_0",
  "is_lag": 0,
  "recovery": 1,
  "sf_pool_percent": 0,
  "sf_pool_force_destroy": 0,
  "static_pf": {
    "mac_base": "11:22:33:44:55:66",
    "features": "0x230047082b"
  },
  "vf": {
    "mac_base": "CC:48:15:FF:00:00",
    "features": "0x230047082b",
    "vfs_per_pf": 100,
    "qp_num": 4
  }
}
```

The following is an example for LAG configuration:

```
{
  "ib_dev_lag": "mlx5_bond_0",
  "ib_dev_for_static_pf": "mlx5_bond_0",
  "is_lag": 1,
  "recovery": 1,
  "sf_pool_percent": 0,
  "sf_pool_force_destroy": 0
}
```

User Frontend

To communicate with the service, a user frontend program (virtnet) is installed on the DPU. Run the following command to check its usage:

```
# virtnet -h
usage: virtnet [-h] [-v] {hotplug,unplug,list,query,modify,log}
...

Nvidia virtio-net-controller command line interface v1.0.9

positional arguments:
  {hotplug,unplug,list,query,modify,log}
                        ** Use -h for sub-command usage
    hotplug             hotplug virtnet device
    unplug              unplug virtnet device
    list                list all virtnet devices
    query               query all or individual virtnet device(s)
    modify              modify virtnet device
    log                 set log level

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
```

Note that each positional argument has its own help menu as well. For example:

```
# virtnet log -h
usage: virtnet log [-h] -l {info,err,debug}
optional arguments:
  -h, --help            show this help message and exit
  -l {info,err,debug}, --level {info,err,debug}
                        log level: info/err/debug
```

To operate a particular device, either the VUID or device index can be used to locate the device. Both attributes can be fetched from command "virtnet list". For example, to modify the MAC of a specific VF, you may run either of the following commands:

```
# virtnet modify -p 0 –v 0 device -m 0C:C4:7A:FF:22:98
```

Or:

```
# virtnet modify -u <VUID-string> device -m 0C:C4:7A:FF:22:98
```

Note

The following `modify` options require unbinding the virtio device from virtio-net driver in the guest OS:

- MAC
- MTU
- Features
- Msix_num
- max_queue_size

For example:

- On the guest OS:

```
$ echo "bdf of virtio-dev" >
/sys/bus/pci/drivers/virtio-pci/unbind
```

- On the Arm side:

```
$ virtnet modify ...
```

- On the guest OS:

```
$ echo "bdf of virtio-dev" >
/sys/bus/pci/drivers/virtio-pci/bind
```

Controller Recovery

It is possible to recover the control and data planes if communications are interrupted so the original traffic can resume.

Recovery depends on the JSON files stored in `/opt/mellanox/mlnx_virtnet/recovery` where there is a file that corresponds to each device (either PF or VF). The following is an example of the data stored in these files:

```
{
  "port_ib_dev": "mlx5_0",
  "pf_id": 0,
  "function_type": "pf",
  "bdf_raw": 26624,
  "device_type": "hotplug",
  "mac": "0c:c4:7a:ff:22:93",
  "pf_num": 0,
  "sf_num": 2000,
  "mq": 1
}
```

These files should not be modified under normal circumstances. However, if necessary, advanced users may tune settings to meet their requirements. Users are responsible for the validity of the recovery files and should only perform this when the controller is not running.

Note

Controller recovery is enabled by default and does not need user configuration or intervention unless a system reset is needed or BlueField configuration is changed (i.e., any of the mlxconfig options `PCI_SWITCH_EMULATION_NUM_PORT`, `VIRTIO_NET_EMULATION_NUM_VF`, or `VIRTIO_NET_EMULATION_NUM_PF`). To this end, the files under `/opt/mellanox/mlnx_virtnet/recovery` must be deleted.

The first time LAG is configured with a controller, recover files must be cleaned up to ensure the controller does not try to recover devices with the previous IB parent device.

Controller Live Update

Live update minimizes network interface down time by performing online upgrade of the virtio-net controller without necessitating a full restart.

To perform a live update, you must install a newer version of the controller either using the `rpm` or `deb` package (depending on the OS distro used). Run:

For Ubuntu/Debian	`dpkg --force-all -i virtio-net-controller-x.y.z-1.mlnx.aarch64.deb`
For CentOS/RedHat	`rpm -Uvh virtio-net-controller-x.y.z-1.mlnx.aarch64.rpm --force`

It is recommended to use the following command to verify the versions of the controller currently running and the one just installed:

```
virtnet version
```

If the versions that are correct, issue the following command to start the live update process:

```
virtnet update --start
virtnet update -s
```

> **Note**
>
> If an error appears regarding the "update" command not being supported, this implies that the controller version you are trying to install is too old. Reinstalling the proper version will resolve this issue.

During the update process, the following command may be used to check the update status:

```
virtnet update status
virtnet update -t
```

During the update, all existing `virtnet` commands (e.g., `list`, `query`, `modify`) are still supported. VF creation/deletion works as well.

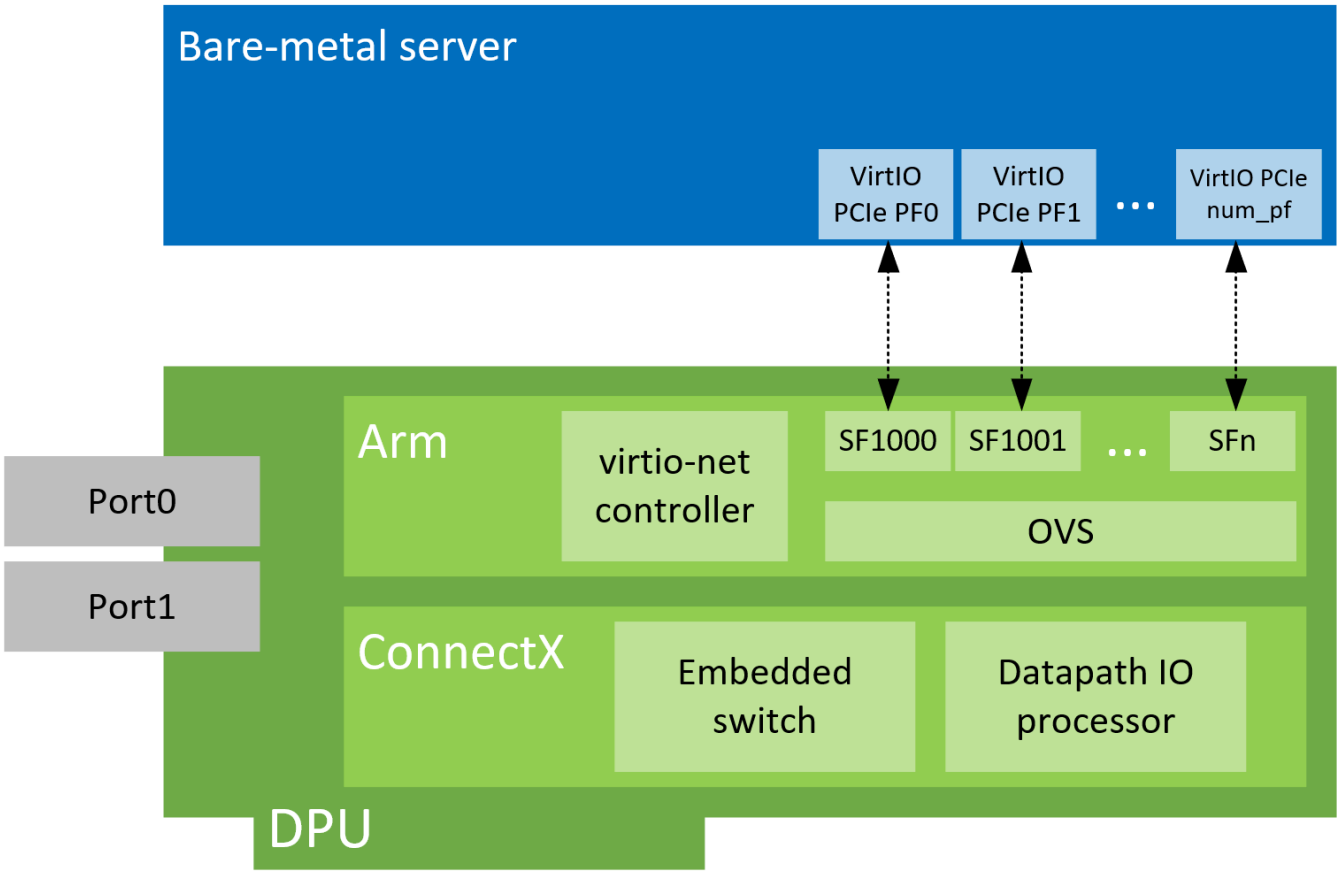
When the update process completes successfully, the command `virtnet update status` will reflect the status accordingly.

Note

If a device is actively migrating, the existing `virtnet` commands will appear as "migrating" for that specific device so that user can retry later.

VirtIO-net PF Devices

This section covers managing virtio-net PCIe PF devices using virtio-net controller.



VirtIO-net PF Device Configuration

1. Run the following command on the DPU:

```
$ mlxconfig -d /dev/mst/mt41686_pciconf0 s
INTERNAL_CPU_MODEL=1
```

2. Add the following kernel boot parameters to the Linux boot arguments:

```
pci=realloc
```

3. Cold reboot the host system.

4. Apply the following configuration on the DPU:

```
$ mst start
$ mlxconfig -d /dev/mst/mt41686_pciconf0 s PF_BAR2_ENABLE=0
PER_PF_NUM_SF=1
$ mlxconfig -d /dev/mst/mt41686_pciconf0 s \
PCI_SWITCH_EMULATION_ENABLE=1 \
PCI_SWITCH_EMULATION_NUM_PORT=16 \
VIRTIO_NET_EMULATION_ENABLE=1 \
VIRTIO_NET_EMULATION_NUM_VF=0 \
VIRTIO_NET_EMULATION_NUM_PF=0 \
VIRTIO_NET_EMULATION_NUM_MSIX=10 \
SRIOV_EN=0 \
PF_SF_BAR_SIZE=10 \
PF_TOTAL_SF=64
$ mlxconfig -d /dev/mst/mt41686_pciconf0.1 s \
PF_SF_BAR_SIZE=10 \
PF_TOTAL_SF=64
```

5. Cold reboot the host system a second time.

Creating Modern Hotplug VirtIO-net PF Device

Virtio emulated network PCIe devices are created and destroyed using virtio-net-controller application console. When this application is terminated, all created virtio-net emulated devices are hot unplugged.

1. Create a hotplug virtio-net device. Run:

```
$ virtnet hotplug -i mlx5_0 -f 0x0 -m 0C:C4:7A:FF:22:93 -t
1500 -n 3 -s 1024
```

> **Note**
>
> The maximum number of virtio-net queues is bound by the minimum of the following numbers:
>
> - `VIRTIO_NET_EMULATION_NUM_MSIX` from the command `mlxconfig -d <mst_dev> q`
> - `max_virtq` from the command `virtnet list`

This creates one hotplug virtio-net device with MAC address 0C:C4:7A:FF:22:93, MTU 1500, and 3 virtio queues with a depth of 1024 entries. This device is uniquely identified by its index. This index is used to query and update device attributes. If the device is created successfully, an output appears similar to the following:

```
{
  "bdf": "85:00.0",
  "vuid": "VNETS1D0F0",
  "id": 3,
  "sf_rep_net_device": "en3f0pf0sf2000",
  "mac": "0C:C4:7A:FF:22:93"
}
```

2. Add the representor port of the device to the OVS bridge and bring it up. Run:

```
$ ovs-vsctl add-port <bridge> en3f0pf0sf2000
$ ip link set dev en3f0pf0sf2000 up
```

   Once steps 1-3 are completed, virtio-net device should be available in the host system.

3. To query all the device configurations of virtio-net device that you created, run:

```
$ virtnet query –p 0
```

4. To list all the virtio-net devices, run:

```
$ virtnet list
```

5. To modify device attributes, for example, changing its MAC address, run:

```
$ virtnet modify -p 0 device -m 0C:C4:7A:FF:22:98
```

6. Once usage is complete, to hot-unplug a virtio-net device, run:

```
$ virtnet unplug -p 0
```

Creating Transitional Hotplug VirtIO-net PF Device

A transitional device is a virtio device which supports drivers conforming to virtio specification 1.x and legacy drivers operating under virtio specification 0.95 (i.e., legacy mode) so that servers with old Linux kernels can still utilize virtio-based technology.

1. Run the following command on the DPU:

```
$ mst start
$ mlxconfig -d /dev/mst/mt41686_pciconf0 s \
  VIRTIO_NET_EMULATION_PF_PCI_LAYOUT=1 \
  VIRTIO_EMULATION_HOTPLUG_TRANS=1
```

2. Add the following parameters to the Linux boot arguments on the guest OS (host OS or VM) side:

```
virtio_pci.force_legacy=1 intel_iommu=off
```

   Refer to the known limitations below.

3. Cold reboot the host system.

4. If `virtio_pci` is a kernel module rather than built-in from the guest OS, run the following command after both the host and DPU OSes are up:

```
modprobe –rv virtio_pci
modprobe –v virtio_pci force_legacy=1
```

5. To create a transitional hotplug virtio-net device. Run the following command on the DPU (with additional `-l` / `--legacy`):

```
$ virtnet hotplug -i mlx5_0 -f 0x0 -m 0C:C4:7A:FF:22:93 -t
1500 -n 3 -s 1024 -l
```

6. Proceed from step 2 of section "Creating Modern Hotplug VirtIO-net PF Device" for the rest of configuration.

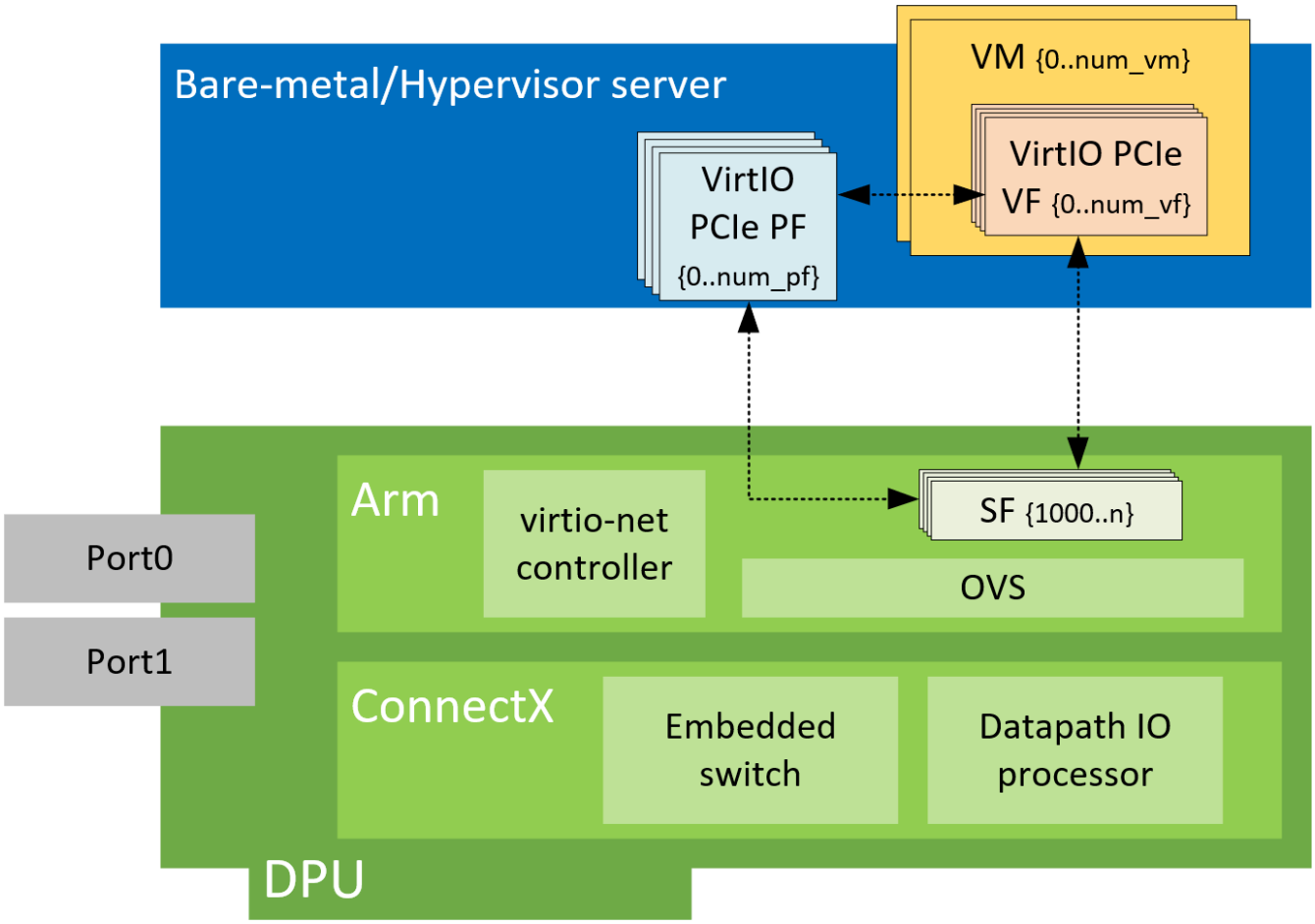
Note

Known limitations:

- AMD CPU is not supported.
- Only kernel versions 3.10 and above are supported. `intel_iommu=off` is not required for kernel 5.1 and above.
- An x86-64 system has only 64K I/O port space which is shared by all peripherals. The virtio transitional device uses I/O BAR. The hotplug device is under one PCIe bridge which is at the emulated PCIe switch downstream port. According to the PCIe specification, the granularity for the bridge I/O window is 4K bytes. If the system cannot satisfy the I/O resource demands by the emulated PCIe switch (depending on the port number of the PCIe switch), the I/O BAR allocation will fail. One hot-plug device requires one emulated PCIe switch port. Each emulated PCIe switch port takes 4K bytes of I/O space if the transitional virtio device is supported. Use `cat /proc/ioports` to check how many I/O port resources are allocated for the host bridge which contains the NIC. The number of supported hotplug transitional virtio device equals: (allocated I/O port space – 4k) / 4k.

Virtio-net SR-IOV VF Devices

This section covers managing virtio-net PCIe SR-IOV VF devices using virtio-net-controller.



Virtio-net SR-IOV VF Device Configuration

> **Note**
>
> Virtio-net SR-IOV VF is only supported with statically configured PF, hot-plugged PF is not currently supported.

1. On the DPU, make sure virtio-net-controller service is enabled so that it starts automatically. Run:

```
systemctl status virtio-net-controller.service
```

2. On the host, enable SR-IOV. Please refer to MLNX_OFED documentation under Features Overview and Configuration > Virtualization > Single Root IO Virtualization (SR-IOV) > Setting Up SR-IOV for instructions on how to do that. Make sure the parameters "`intel_iommu=on iommu=pt pci=realloc`" exist in `grub.conf` file.

3. It is recommended to add `pci=assign-busses` to the boot command line when creating more than 127 VFs. Without this option, the following errors might appear from host and the virtio driver will not probe these devices.

```
pci 0000:84:00.0: [1af4:1041] type 7f class 0xffffff
pci 0000:84:00.0: unknown header type 7f, ignoring device
```

4. Run the following command on the DPU:

```
mst start && mlxconfig -d /dev/mst/mt41686_pciconf0 s
INTERNAL_CPU_MODEL=1
```

5. Add the following kernel boot parameters to the Linux boot arguments:

```
intel_iommu=on iommu=pt pci=realloc
```

6. Cold reboot the host system.

7. Apply the following configuration on the DPU in three steps to support up to 125 VFs per PF (500 VFs in total).

    1. ```
       $ mst start && mlxconfig -d /dev/mst/mt41686_pciconf0 s
       PF_BAR2_ENABLE=0 PER_PF_NUM_SF=1
       ```
       ```

2.
```
$ mlxconfig -d /dev/mst/mt41686_pciconf0 s \
PCI_SWITCH_EMULATION_ENABLE=0 \
PCI_SWITCH_EMULATION_NUM_PORT=0 \
VIRTIO_NET_EMULATION_ENABLE=1 \
VIRTIO_NET_EMULATION_NUM_VF=126 \
VIRTIO_NET_EMULATION_NUM_PF=4 \
VIRTIO_NET_EMULATION_NUM_MSIX=4 \
SRIOV_EN=1 \
PF_SF_BAR_SIZE=8 \
PF_TOTAL_SF=508 \
NUM_OF_VFS=0
```

3.
```
$ mlxconfig -d /dev/mst/mt41686_pciconf0.1 s
PF_TOTAL_SF=1 PF_SF_BAR_SIZE=8
```

8. Cold reboot the host system.

Creating Virtio-net SR-IOV VF Devices

1. On the host, make sure the static virtio network device presents. Run:

```
# lspci | grep -i virtio
85:00.3 Network controller: Red Hat, Inc. Virtio network
device
```

2. On the host, make sure virtio_pci and virtio_net are loaded. Run:

```
# lsmod | grep virtio
```

The net device should be created:

```
# ethtool -i p7p3
driver: virtio_net
version: 1.0.0
firmware-version:
expansion-rom-version:
bus-info: 0000:85:00.3
supports-statistics: no
supports-test: no
supports-eeprom-access: no
supports-register-dump: no
supports-priv-flags: no
```

3. To create SR-IOV VF devices on the host, run:

```
# echo 2 > /sys/bus/pci/drivers/virtio-
pci/0000\:85\:00.3/sriov_numvfs
```

> **Note**
>
> When the number of VFs created is high, SR-IOV enablement may take several minutes.

2 VFs should be created from the host:

```
# lspci | grep -i virt
85:00.3 Network controller: Red Hat, Inc. Virtio network
device
85:04.5 Network controller: Red Hat, Inc. Virtio network
device
85:04.6 Network controller: Red Hat, Inc. Virtio network
device
```

4. From the DPU virtio-net controller, run the following command to get VF information.

```
# virtnet list
{
    "vf_id": 0,
    "parent_pf_id": 0,
    "function_type": "VF",
    "vuid": "VNETS0D0F2VF1",
    "bdf": "83:00.6",
    "sf_num": 3000,
    "sf_parent_device": "mlx5_0",
    "sf_rep_net_device": "en3f0pf0sf3000",
    "sf_rep_net_ifindex": 19,
    "sf_rdma_device": "mlx5_7",
    "sf_vhca_id": "0x192",
    "msix_config_vector": "0x0",
    "num_msix": 10,
    "max_queues": 4,
    "max_queues_size": 256,
    "net_mac": "5A:94:07:04:F6:1C",
    "net_mtu": 1500
},
```

You may use the pci-bdf to match the PF/VF on the host to the information showing on DPU.

To query all the device configurations of the virtio-net device of that VF, run:

```
$ virtnet query -p 0 -v 0
```

Add the corresponding SF representor to the OVS bridge and bring it up. Run:

```
# ovs-vsctl add-port <bridge> en3f0pf0sf1004
# ip link set dev en3f0pf0sf1004 up
```

Now the VF is functional.

> **Note**
>
> When port MTU (p0/p1 of the DPU) is changed after the controller is started, you must restart controller service. It is not recommended to use jumbo MTUs because that may lead to performance degradation.

5. To destroy SR-IOV VF devices on the host, run:

```
# echo 0 > /sys/bus/pci/drivers/virtio-
pci/0000\:85\:00.3/sriov_numvfs
```

Note

When the command returns from the host OS, it does not necessarily mean the controller finished its operations. Look at controller log from the DPU and make sure you see a log like below before removing virtio kernel modules or recreate VFs.

```
# virtio-net-controller[3544]: [INFO]
virtnet.c:617:virtnet_device_vfs_unload:
PF(0): Unload (4) VFs finished
```

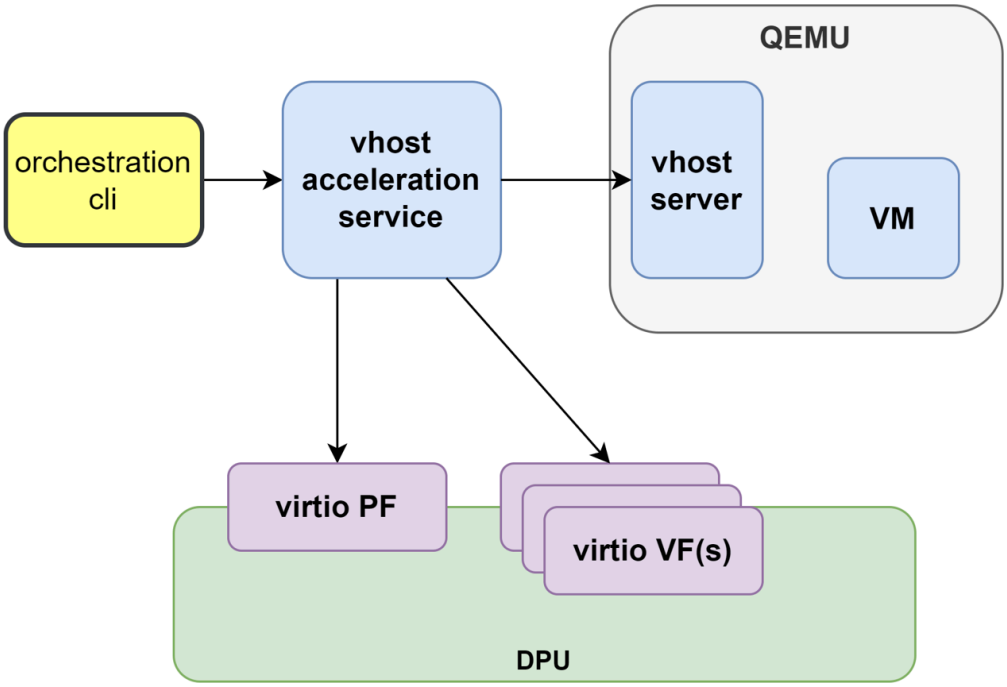
Once VFs are destroyed, created SFs from the DPU side are not destroyed but are saved into the SF pool to be reused later.

Transitional VirtIO-net VF Device Support

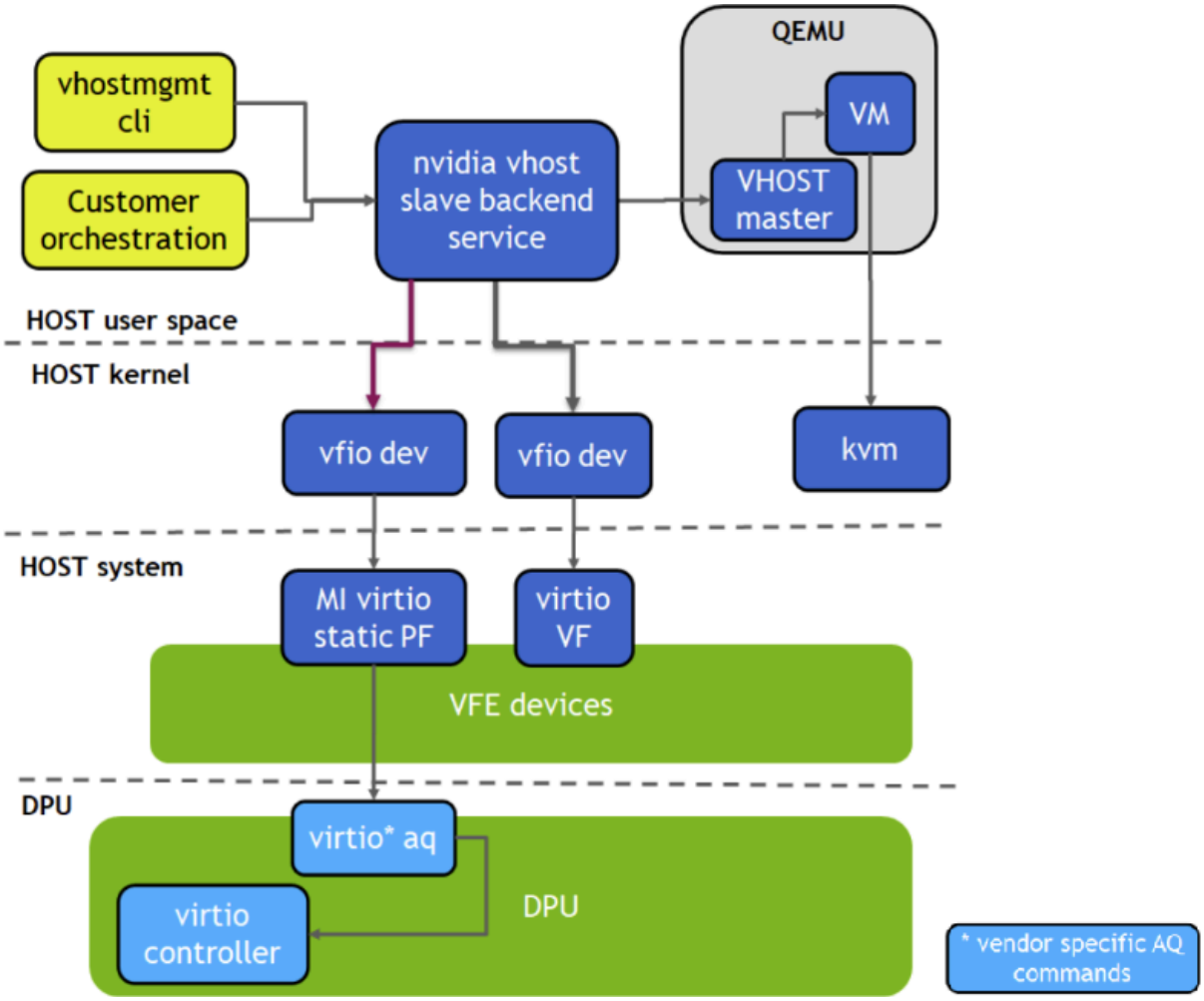
Transitional virtio-net VF devices are not currently supported.

Virtio VF PCIe Devices for vHost Acceleration

Virtio VF PCIe devices can be attached to the guest VM using vhost acceleration software stack. This enables performing live migration of guest VMs.



This section describes the steps to enable VM live migration using virtio VF PCIe devices along with vhost acceleration software.



Prerequisites

- Minimum hypervisor kernel version – Linux kernel 5.7 (for VFIO SR-IOV support)

Install vHost Acceleration Software Stack

Vhost acceleration software stack is built using open-source BSD licensed DPDK.

To install vhost acceleration software:

1. Clone the software source code.

```
git clone https://github.com/Mellanox/dpdk-vhost-vfe
```

> **Info**
>
> Latest release tag is `vfe-1.0`.

2. Build software:

```
apt-get install libev-dev
yum install -y numactl-devel libev-devel
meson build -Dexamples=vdpa
ninja -C build
```

To install QEMU:

> **Info**
>
> Upstream QEMU later then 8.1 can be used or the following QEMU.

1. Clone QEMU sources.

```
git clone https://github.com/Mellanox/qemu -b stable-8.1-presetup
```

> **Info**
>
> Latest release tag is `vfe-0.4`.

2. Build QEMU.

```
mkdir bin
cd bin
../configure --target-list=x86_64-softmmu --enable-kvm
make -j24
```

Configure vHost and DPU System

1. Set the DPU nvconfig.

```
mlxconfig -d /dev/mst/mt41686_pciconf0 s \
VIRTIO_NET_EMULATION_ENABLE=1 VIRTIO_NET_EMULATION_NUM_PF=1
VIRTIO_NET_EMULATION_NUM_VF=16 \
VIRTIO_BLK_EMULATION_ENABLE=1 VIRTIO_BLK_EMULATION_NUM_PF=1
VIRTIO_BLK_EMULATION_NUM_VF=16 \
VIRTIO_NET_EMULATION_NUM_MSIX=64
VIRTIO_BLK_EMULATION_NUM_MSIX=64 NUM_VF_MSIX=64
```

2. Cold reboot the system after above configuration.

3. Setup the hypervisor system:

    1. Configure hugepages and libvirt VM XML (see OVS Hardware Offloads Configuration for information on doing that).

    2. Add a virtio-net interface and a virtio-blk interface in VM XML.

```
<qemu:commandline>
<qemu:arg value='-chardev'/>
<qemu:arg value='socket,id=char0,path=/tmp/vfe-
net0,server=on'/>
<qemu:arg value='-netdev'/>
<qemu:arg value='type=vhost-
user,id=vdpa,chardev=char0,queues=4'/>
<qemu:arg value='-device'/>
<qemu:arg value='virtio-net-
pci,netdev=vdpa,mac=00:00:00:00:33:00,page-per-
vq=on,rx_queue_size=1024,tx_queue_size=1024,mq=on'/>
<qemu:arg value='-chardev'/>
<qemu:arg value='socket,id=char1,path=/tmp/vfe-
blk0,server=on'/>
<qemu:arg value='-device'/>
<qemu:arg value='vhost-user-blk-pci,chardev=char1,page-
per-vq=on,num-queues=4,disable-legacy=on,disable-
modern=off'/>
</qemu:commandline>
```

4. Create block device on the DPU:

```
spdk_rpc.py bdev_null_create Null0 1024 512
snap_rpc.py controller_virtio_blk_create --pf_id 0 --
bdev_type spdk mlx5_0 --bdev Null0 --num_queues 1 --admin_q -
-force_in_order
```

5. On BlueField-3 SNAP:

```
spdk_rpc.py bdev_null_create Null0 1024 512
snap_rpc.py virtio_blk_controller_create --pf_id 0 --bdev
Null0 --num_queues 1 --admin_q --force_in_order
```

Run vHost Acceleration Service

1. Bind the virtio PF devices to `vfio-pci` driver:

```
modprobe vfio vfio_pci
echo 1 > /sys/module/vfio_pci/parameters/enable_sriov

echo 0x1af4 0x1041 > /sys/bus/pci/drivers/vfio-pci/new_id
echo 0x1af4 0x1042 > /sys/bus/pci/drivers/vfio-pci/new_id

echo 0000:af:00.2 > /sys/bus/pci/drivers/vfio-pci/bind
echo 0000:af:00.3 > /sys/bus/pci/drivers/vfio-pci/bind

lspci -vvv -s 0000:af:00.3 | grep "Kernel driver"
Kernel driver in use: vfio-pci
lspci -vvv -s 0000:af:00.2 | grep "Kernel driver"
Kernel driver in use: vfio-pci
```

2. Enable SR-IOV and create a VF(s):

```
echo 1 > /sys/bus/pci/devices/0000:af:00.2/sriov_numvfs
echo 1 > /sys/bus/pci/devices/0000:af:00.3/sriov_numvfs

lspci | grep Virtio
af:00.2 Ethernet controller: Red Hat, Inc. Virtio network
device
af:00.3 Non-Volatile memory controller: Red Hat, Inc. Virtio
block device
af:04.5 Ethernet controller: Red Hat, Inc. Virtio network
device
af:05.1 Non-Volatile memory controller: Red Hat, Inc. Virtio
block device
```

3. Add a VF representor to the OVS bridge on the DPU:

```
virtnet query -p 0 -v 0 | grep sf_rep_net_device
"sf_rep_net_device": "en3f0pf0sf3000",

 ovs-vsctl add-port ovsbr1 en3f0pf0sf3000
```

4. Run the vhost acceleration software service:

```
cd dpdk-vhost-vfe
sudo ./build/app/dpdk-vfe-vdpa -a 0000:00:00.0 --log-
level=.,8 --vfio-vf-token=cdc786f0-59d4-41d9-b554-
fed36ff5e89f -- --client
```

5. Provision the virtio-net PF and VF.

```
cd dpdk-vhost-vfe

python ./app/vfe-vdpa/vhostmgmt mgmtpf -a 0000:af:00.2
# Wait on virtio-net-controller finishing handle PF FLR

# On DPU, change VF MAC address or other device options
virtnet modify -p 0 -v 0 device -m 00:00:00:00:33:00
python ./app/vfe-vdpa/vhostmgmt vf -a 0000:af:04.5 -v
/tmp/vfe-net0
```

6. Provision the virtio-blk PF and VF.

```
cd dpdk-vhost-vfe

python ./app/vfe-vdpa/vhostmgmt mgmtpf -a 0000:af:00.3
# Wait on SNAP controller to finish handling PF FLR

# On DPU, the user must create a VF device controller before
adding the VF device to the
# vhostmgmt upon pf or vf device delete from vhostmgmt, or
vhostmgmt restart:
#       For BlueField-3, the VF controller is automatically
recreated
#       For BlueField-2, the VF controller must be manually
recreated
# Use snap_rpc.py controller_list to check for controller
exsistence and create controller if it's not there
snap_rpc.py controller_virtio_blk_create mlx5_0 --pf_id 0 --
vf_id 0 --bdev_type spdk --bdev Null0 --force_in_order
python ./app/vfe-vdpa/vhostmgmt vf -a 0000:af:05.1 -v
/tmp/vfe-blk0
```

Note

If the SR-IOV is disabled and reenabled, the user must re-provision the VFs.

Start the VM

```
virsh start <domain-name>
```

Simple Live Migration

Prepare two identical hosts and perform the provisioning of the virtio device to DPDK on both.

Boot the VM on one server:

```
virsh migrate --verbose --live --persistent gen-l-vrt-440-162-CentOS-7.4 qemu+ssh://gen-l-vrt-439/system --unsafe
```

Remove Device

When finished with using the virtio device, use following commands to remove them from DPDK:

```
python ./app/vfe-vdpa/vhostmgmt vf -r 0000:af:04.5
python ./app/vfe-vdpa/vhostmgmt mgmtpf -r 0000:af:00.2

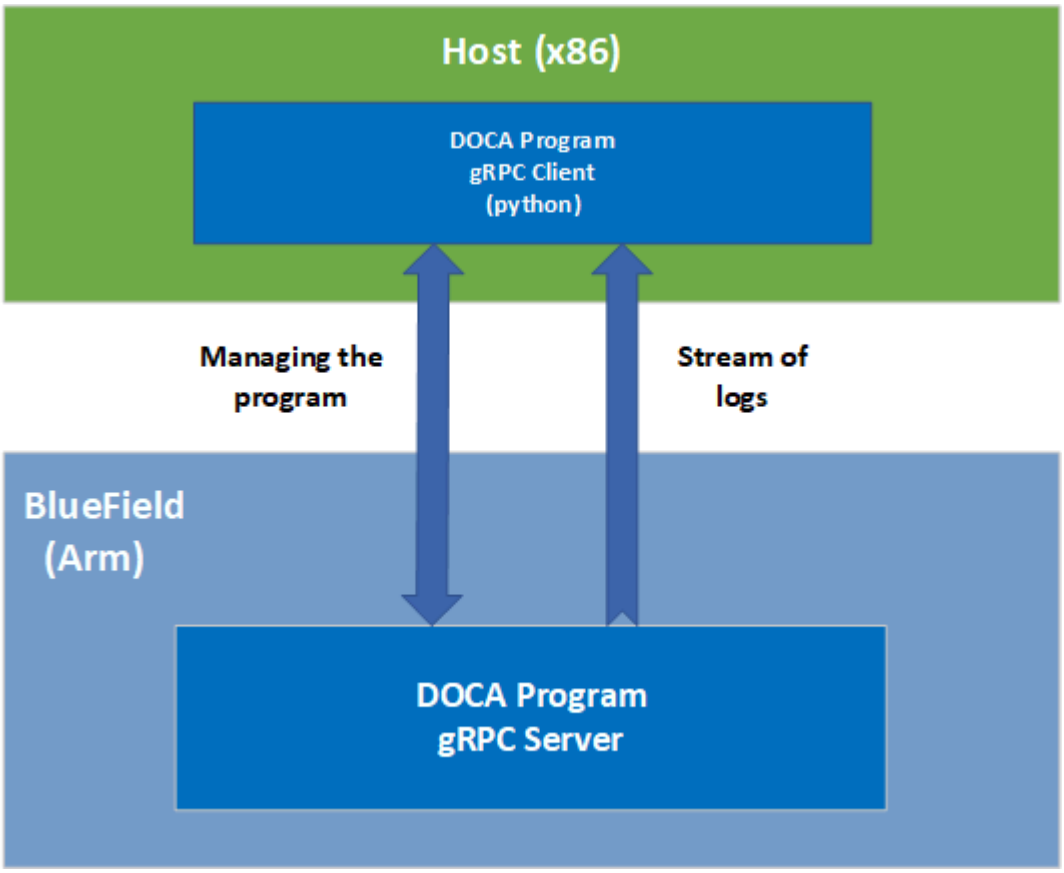
python ./app/vfe-vdpa/vhostmgmt vf -r 0000:af:05.1
python ./app/vfe-vdpa/vhostmgmt mgmtpf -r 0000:af:00.3
```

NVIDIA DOCA gRPC Infrastructure User Guide

This guide provides an overview and configuration instructions for gRPC-supported DOCA infrastructure, programs, and services for NVIDIA® BlueField® DPU.

Introduction

The recommended setup for deploying most DOCA programs is deployment on the DPU itself. However, in some cases it is useful to be able to manage and configure the programs running on top of it directly from the host (x86).



For this purpose, DOCA includes built-in gRPC support, thereby allowing a program to expose to the host its logical interface in the form of a gRPC-equivalent API. This guide elaborates on the different components that enable this support, including instructions for developers who wish to create similar gRPC extensions to their own DOCA-based programs.

Prerequisites

Refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField related software.

gRPC Overview

gRPC is Google's open-source remote procedure call (RPC) library and is the most widely used RPC solution.

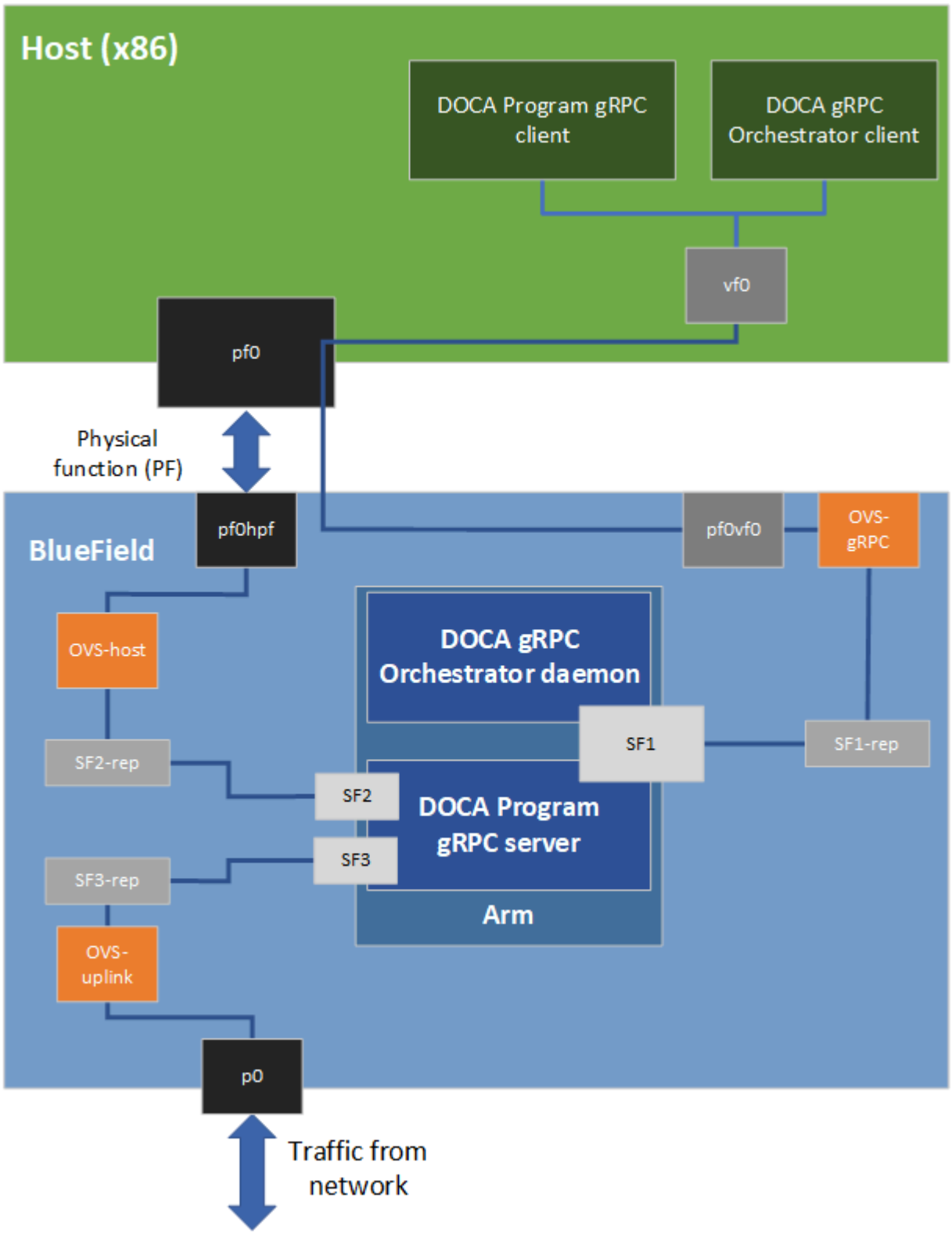
gRPC consists of two layers:

- Protobuf – Google library for semantically defining message formats
- gRPC "services" – definitions of the exposed RPC functionality

gRPC's support for different language bindings, combined with the unified protocol implementation, allows the client and server to run on different machines using different programming languages.

OVS and Connectivity

It is important to differentiate between and separate the program's data path and the gRPC management interface. The following figure contains an overview of a sample setup for a bump-on-the wire program:



As can be seen above, the program is a bump-on-the-wire, and the gRPC-related traffic flows through a separate OVS bridge connected to the host using a virtual function (VF). More information about VFs and how to configure them can be found in the NVIDIA DOCA Virtual Functions User Guide .

The architecture above allows us to associate a network address to SF1, effectively making the program's gRPC server part of an IP network with the host. This allows for an easy client-server setup, masking away the hardware details from the logical gRPC interface.

Compilation Instructions

Some of DOCA's reference applications have integrated gRPC support and require a gRPC development setup for their recompilation. DOCA's reference applications provide a `meson_options.txt` file that controls the gRPC support and that is set to "off" by default.

```
option('enable_grpc_support', type: 'boolean', value: false,
       description: 'Enable all gRPC based DOCA applications.')
```

Once a gRPC setup is installed, according to the instructions in the next section, the `meson_options.txt` file can be updated to enable gRPC support so as to allow for recompilation of the gRPC-enabled applications.

```
option('enable_grpc_support', type: 'boolean', value: true,
       description: 'Enable all gRPC based DOCA applications.')
```

Installing gRPC Development Setup

Rebuilding a gRPC-enabled DOCA application on BlueField requires a gRPC development setup which is provided as part of DOCA's development packages. Due to the lack of packaging support for gRPC, at least for C/C++ environments, gRPC is compiled from the source and can be found at `/opt/mellanox/grpc`. Upon compilation, Meson ensures that the requirements exist.

Info

To troubleshoot a requirement that is reported to be missing by Meson, refer to the NVIDIA DOCA Troubleshooting Guide.

If a different gRPC version is needed, these are Google's instructions for creating a development setup. If a different gRPC setup is used, do not forget to update the necessary environment variables to point at your installation directory.

Running gRPC-Enabled Program

Running Program Server on BlueField

The gRPC-enabled program is equivalent to the "regular" DOCA program as it:

- Requires the same configuration steps (huge pages, etc.)
- Requires the same program command line arguments

The only difference is that the program also requires one more command line argument:

```
doca_<program_name>_grpc [DPDK flags] -- [DOCA flags] [Program flags] -g/--grpc-address <ip-address[:port]>
```

- `ip-address` – IP address to be used by the gRPC server
- `port` – TCP port to be used by the server instead of the program's default gRPC port (optional)

One could also use the json configuration file as follows:

```
doca_<program_name>_grpc -j/--json <path to grpc configuration
json file>
```

For more information, refer to DOCA Arg Parser.

Running Program Client on Host

While the gRPC Python environment is already installed on the host as part of the DOCA installation, it has not been added to the default Python path so as to not clutter it. The environment variable definitions needed for using the gRPC Python environment, as needed by the client, are:

```
export PYTHONPATH=${PYTHONPATH}:/opt/mellanox/grpc/python3/lib
```

To run the Python client of the gRPC-enabled application:

```
doca_<program_name>_gRPC_client.py -d/--debug <server
address[:server port]>
```

Installing gRPC on RHEL CentOS 7.6 and Older

On RHEL/CentOS distributions 7.6 and older for x64 host, there is a known issue in the python `grpcio` package which causes the following error:

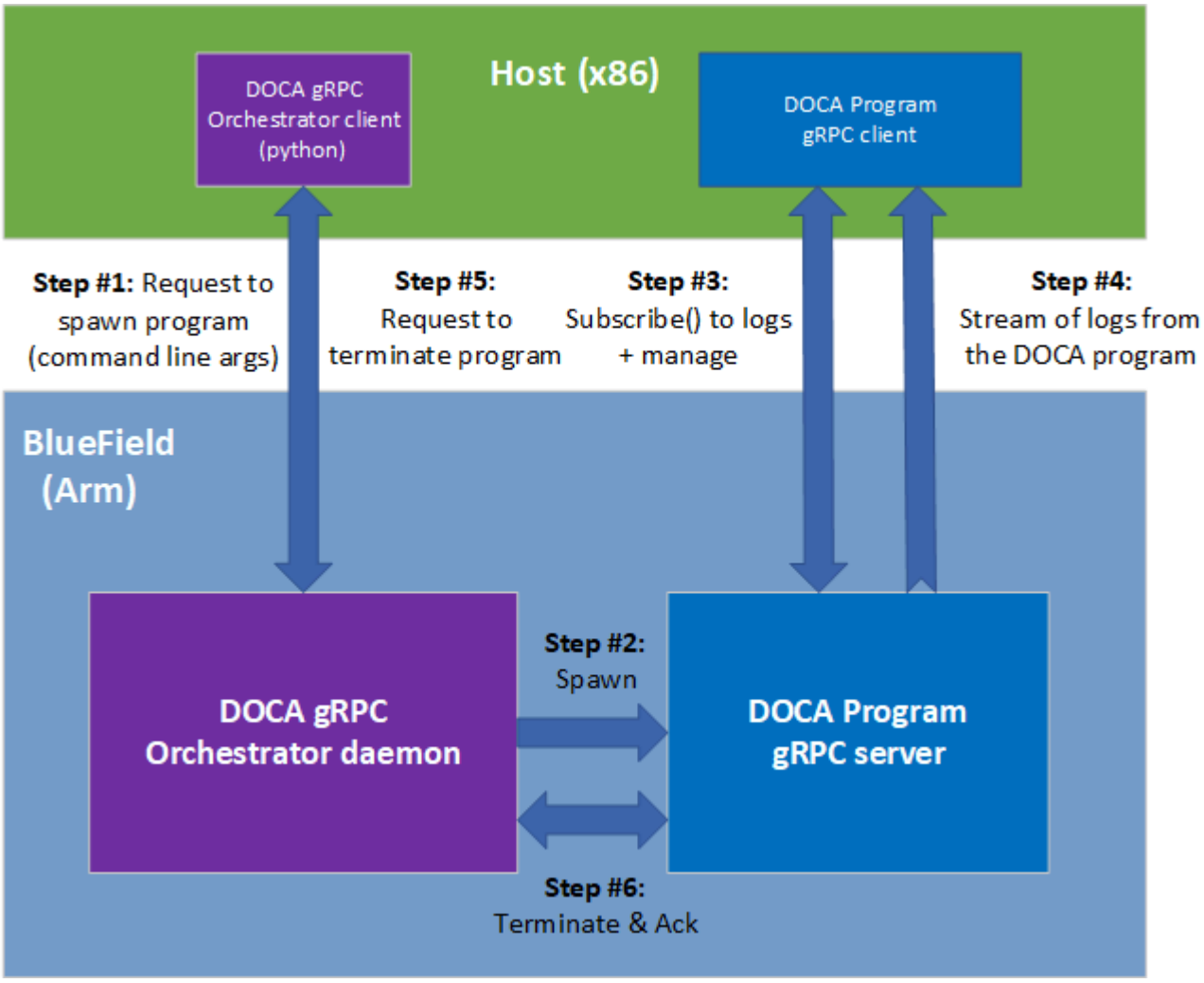
```
from grpc._cython import cygrpc as _cygrpc
ImportError:
/opt/mellanox/grpc/python3/lib/grpc/_cython/cygrpc.cpython-36m-
x86_64-linux-gnu.so: undefined symbol:
_ZSt24__throw_out_of_range_fmtPKcz
```

To fix this error, please run the following commands on the host:

```
rm -rf /opt/mellanox/grpc/python3/lib/grpc*
wget
https://files.pythonhosted.org/packages/67/3c/53cc28f04fb9bd3e8bad
1.39.0-cp36-cp36m-manylinux2010_x86_64.whl -P /tmp && unzip
/tmp/grpcio-1.39.0-cp36-cp36m-manylinux2010_x86_64.whl -d
/opt/mellanox/grpc/python3/lib
```

DOCA gRPC Orchestrator

A gRPC-enabled program must first be executed on BlueField for it to be managed from the host. This creates a bootstrapping issue that is solved by the DOCA gRPC Orchestrator, as can be seen in the following figure:



After a one-time configuration step, the DOCA gRPC Orchestrator daemons runs on BlueField, listening for incoming requests from the host to start/stop a given gRPC-enabled DOCA program.

Enabling and Configuring DOCA gRPC Orchestrator Daemon

The `doca_grpc` daemon on the DPU starts as "disabled" by default and has a one-time configuration step for enabling it:

```
# One-time only, enable the DOCA gRPC Orchestrator Daemon
systemctl enable doca_grpc.service
# One-time only, start the daemon
systemctl start doca_grpc.service
```

The daemon is controlled via a configuration file stored at `/etc/doca_grpc/doca_grpc.conf`.

> **Note**
>
> Users are requested to update the configuration file so as to reflect the desired deployment parameters. Starting the daemon using the default file "as-is" does not invoke any server instance.

The configuration file defines the configurations for every DOCA gRPC server instance that the daemon spawns, alongside the list of programs it exposes to the host. Accordingly, the file comes prepopulated with a list of all gRPC-enabled DOCA programs and can be modified to support additional proprietary programs. For more information regarding the structure of the file, read the full instructions in the file itself: `/etc/doca_grpc/doca_grpc.conf`.

Once the configuration file is modified to suit the requested deployment, the daemon must be restarted so it could pull the new configuration:

```
systemctl restart doca_grpc.service
```

Running DOCA gRPC Orchestrator Client

The DOCA gRPC Orchestrator client is located at `/opt/mellanox/doca/infrastructure/doca_grpc/orchestrator`.

Note

Being a Python client, the same Python environment variable mentioned earlier for using the application's gRPC client is needed for this client as well.

The usage instructions for the DOCA gRPC client are:

```
Usage: doca_grpc_client.py [OPTIONS] SERVER_ADDRESS COMMAND
[ARGS]...
  DOCA gRPC Client CLI tool
Options:
  -d, --debug
  --help       Show this message and exit.

Commands:
  create   Create PROGRAM_NAME [PROGRAM_ARGS]...
  destroy  Destroy PROGRAM_UID Terminate the execution of the
program...
  list     List the names of gRPC-supported program.
```

For example:

```
/opt/mellanox/doca/infrastructure/doca_grpc/orchestrator/doca_grpc.
192.168.103.2:1234 list
```

The supported commands are:

- list – prints a list of names of all DOCA gRPC-enabled programs currently supported
- create – spawns a gRPC-enabled program on the DPU based on its name and arguments

- destroy – terminates the execution of a gRPC-enabled program based on the program's UID as returned from the "create" command

The `SERVER_ADDRESS` argument is of the form `<server address[:server port]>` allowing the client to use a TCP port other than the default one if the server uses a proprietary port.

The command-specific options are shown when passing the `--help` flag to the respective command:

```
/opt/mellanox/doca/infrastructure/doca_grpc/orchestrator/doca_grpc_
192.168.103.2 destroy --help
```

The gRPC-specific command line arguments (listed under Running gRPC-Enabled Program) are only needed when invoking the gRPC-enabled program directly. When invoked through the DOCA gRPC client, the arguments must match those of the "regular" DOCA program and the gRPC-specific commands should **not** be used.

> **Note**
>
> The orchestrator client supports the option to spawn a gRPC-enabled program using a non-default port. This option is mandatory for programs that do not support a default gRPC port.

Running gRPC-Enabled DOCA Libs

All servers for gRPC-enabled DOCA libraries accept the following arguments:

```
Usage: doca_<lib_name>_grpc [DPDK Flags] -- [DOCA Flags]
DOCA Flags:
  ...
  -g, --grpc-address ip_address[:port]    Set the IP address for
the grpc server
  ...
```

Therefore, they should be invoked using the orchestrator client as follows:

```
/opt/mellanox/doca/infrastructure/doca_grpc/orchestrator/doca_grpc.
192.168.103.2 create doca_flow_grpc -a
auxiliary:mlx5_core.sf.2,dv_flow_en=2 -a
auxiliary:mlx5_core.sf.3,dv_flow_en=2 -- -l 60
```

Info

Note that the `-g` flag is not passed as it is taken care of automatically by the orchestrator client.

Running DOCA Program gRPC Client

Once the program's gRPC server is spawned on BlueField, you can connect to it directly from the host using the respective gRPC client.

If the program's gRPC server is configured to use a TCP port that is not the default port of the program, make sure to also configure the gRPC client to use the same non-default port.

NVIDIA BlueField DPU Modes of Operation

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

Introduction

The NVIDIA® BlueField® DPU has several modes of operation:

- DPU mode, or embedded function (ECPF) ownership, where the embedded Arm system controls the NIC resources and data path (**default**)
- Zero-trust mode which is an extension of the ECPF ownership with additional restrictions on the host side
- NIC mode where the DPU behaves exactly like an adapter card from the perspective of the external host

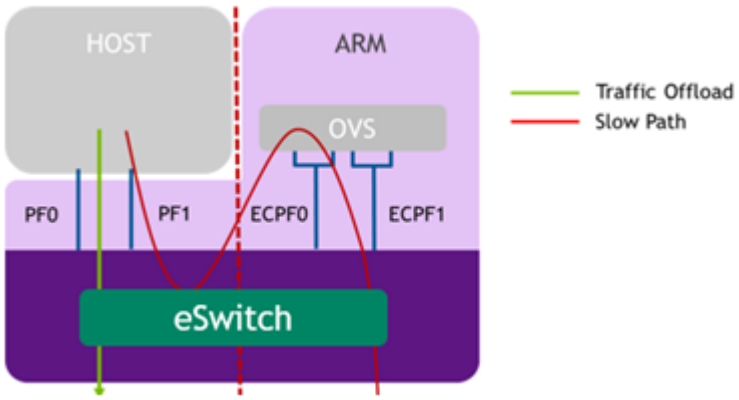
DPU Mode

This mode, known also as embedded CPU function ownership (ECPF) mode, is the default mode for BlueField DPU.

In DPU mode, the NIC resources and functionality are owned and controlled by the embedded Arm subsystem. All network communication to the host flows through a virtual switch control plane hosted on the Arm cores, and only then proceeds to the host. While working in this mode, the DPU is the trusted function managed by the data center and host administrator—to load network drivers, reset an interface, bring an interface up and down, update the firmware, and change the mode of operation on the DPU device.

A network function is still exposed to the host, but it has limited privileges. In particular:

1. The driver on the host side can only be loaded after the driver on the DPU has loaded and completed NIC configuration.
2. All ICM (Interface Configuration Memory) is allocated by the ECPF and resides in the DPU's memory.
3. The ECPF controls and configures the NIC embedded switch which means that traffic to and from the host (DPU) interface always lands on the Arm side.



When the server and DPU are initiated, the networking to the host is blocked until the virtual switch on the DPU is loaded. Once it is loaded, traffic to the host is allowed by default.

There are two ways to pass traffic to the host interface: Either using representors to forward traffic to the host (every packet to/from the host would be handled also by the network interface on the embedded Arm side) or push rules to the embedded switch which allows and offloads this traffic.

In DPU mode, OpenSM must be run from the DPU side (not the host side). Also, management tools (e.g., sminfo, ibdev2netdev, ibnetdiscover) can only be run from the DPU side (not from the host side).

Zero-trust Mode

Zero-trust mode is a specialization of DPU mode which implements an additional layer of security where the host system administrator is prevented from accessing the DPU from the host. Once zero-trust mode is enabled, the data center administrator should control the DPU entirely through the Arm cores and/or BMC connection instead of through the host.

For security and isolation purposes, it is possible to restrict the host from performing operations that can compromise the DPU. The following operations can be restricted individually when changing the DPU host to zero-trust mode:

- Port ownership – the host cannot assign itself as port owner
- Hardware counters – t he host does not have access to hardware counters
- Tracer functionality is blocked
- RShim interface is blocked
- Firmware flash is restricted

Enabling Zero-trust Mode

To enable host restriction:

1. Start the MST service.
2. Set zero-trust mode. From the Arm side, run:

```
$ sudo mlxprivhost -d /dev/mst/<device> r --disable_rshim --
disable_tracer --disable_counter_rd --disable_port_owner
```

> **Note**
>
> Graceful shutdown and p ower cycle are required if any `--disable_*` flags are used.

Disabling Zero-trust Mode

To disable host restriction, set the mode to privileged. Run:

```
$ sudo mlxprivhost -d /dev/mst/<device> p
```

The configuration takes effect immediately.

Note

Graceful shutdown and p ower cycle are required when reverting to privileged mode if host restriction has been applied using any `--disable_*` flags.

NIC Mode

In this mode, the DPU behaves exactly like an adapter card from the perspective of the external host.

Note

The following instructions presume the DPU to operate in DPU mode. If the DPU is operating in zero-trust mode, please return to DPU mode before continuing.

NIC Mode for BlueField-3

Note

When BlueField-3 is configured to operate in NIC mode, Arm OS will not boot.

NIC mode for BlueField-3 saves power, improves device performance, and improves the host memory footprint.

Configuring NIC Mode on BlueField-3 from Linux

Enabling NIC Mode from Linux

Before moving to NIC mode, make sure you are operating in DPU mode by running:

```
host/dpu> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 -e q
```

The output should have `INTERNAL_CPU_MODEL= EMBBEDDED_CPU(1)` and `EXP_ROM_UEFI_ARM_ENABLE = True (1)` (default).

To enable NIC mode from DPU mode:

1. Run the following on the host or Arm:

   ```
   host/dpu> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s
   INTERNAL_CPU_MODEL=1 INTERNAL_CPU_OFFLOAD_ENGINE=1
   ```

2. Perform a graceful shutdown and power cycle the host.

Disabling NIC Mode from Linux

To return to DPU mode from NIC mode:

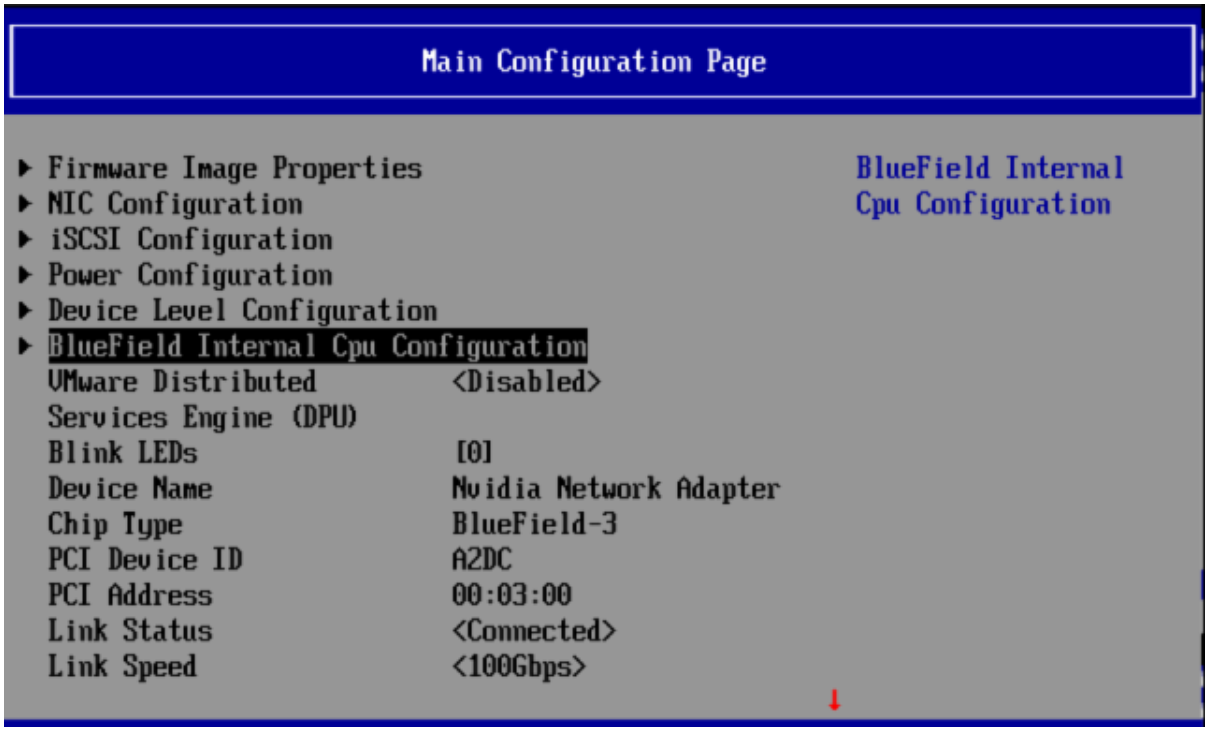
1. Run the following on the host:

   ```
   host> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s
   INTERNAL_CPU_MODEL=1 INTERNAL_CPU_OFFLOAD_ENGINE=0
   ```

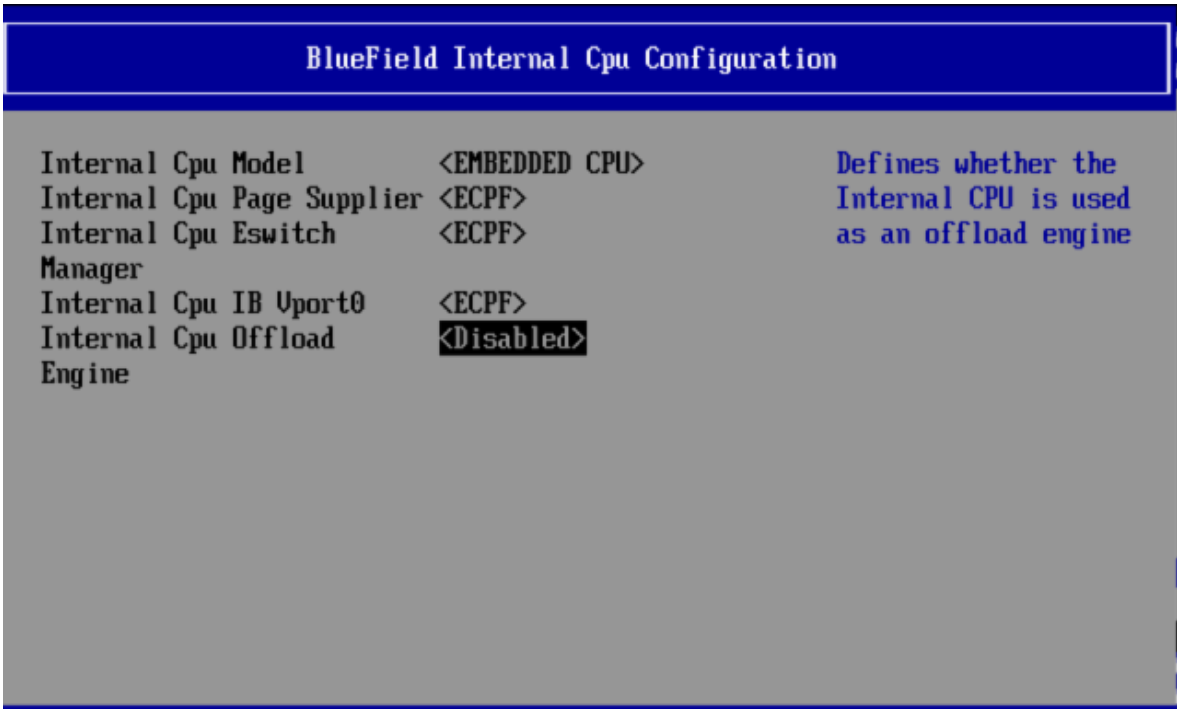
2. Perform a graceful shutdown and power cycle the host.

Configuring NIC Mode on BlueField-3 from UEFI

1. Access the Arm UEFI menu by pressing the Esc button twice
2. Select "Device Manager".
3. Select "Network Device List".
4. Select the network device that presents the uplink (i.e., select the device with the uplink MAC address).
5. Select "NVIDIA Network adapter - $<uplink-mac>".
6. Select "BlueField Internal Cpu Configuration".



   - To enable NIC mode, set "Internal Cpu Offload Engine" to "Disabled".
   - To switch back to DPU mode, set "Internal Cpu Offload Engine" to "Enabled".



Updating ATF and UEFI in BlueField-3 NIC Mode

Once in NIC mode, updating ATF and UFEI can be done using `preboot-install.bfb` by running:

```
# bfb-install --bfb <BlueField-BSP>.bfb --rshim rshim0
```

NIC Mode for BlueField-2

In this mode, the ECPFs on the Arm side are not functional but the user is still able to access the Arm system and update `mlxconfig` options.

Note

When NIC mode is enabled, the drivers and services on the Arm are no longer functional.

Enabling NIC Mode on BlueField-2

To enable NIC mode from DPU mode:

1. Run the following from the x86 host side:

```
$ mst start
$ mlxconfig -d /dev/mst/<device> s INTERNAL_CPU_MODEL=1 \
INTERNAL_CPU_PAGE_SUPPLIER=1 \
INTERNAL_CPU_ESWITCH_MANAGER=1 \
INTERNAL_CPU_IB_VPORT0=1 \
INTERNAL_CPU_OFFLOAD_ENGINE=1
```

Note

To restrict RShim PF (optional), make sure to configure `INTERNAL_CPU_RSHIM=1` as part of the `mlxconfig` command.

2. Perform a graceful shutdown and power cycle the host.

Note

Multi-host is not supported when the DPU is operating in NIC mode.

Note

To obtain firmware BINs for BlueField-2 devices, please refer to the BlueField-2 firmware download page.

Disabling NIC Mode on BlueField-2

To change from NIC mode back to DPU mode:

1. Install and start the RShim driver on the host.
2. Disable NIC mode. Run:

```
$ mst start
$ mlxconfig -d /dev/mst/<device> s INTERNAL_CPU_MODEL=1 \
INTERNAL_CPU_PAGE_SUPPLIER=0 \
INTERNAL_CPU_ESWITCH_MANAGER=0 \
INTERNAL_CPU_IB_VPORT0=0 \
INTERNAL_CPU_OFFLOAD_ENGINE=0
```

> **Note**
>
> If `INTERNAL_CPU_RSHIM=1`, then make sure to configure `INTERNAL_CPU_RSHIM=0` as part of the `mlxconfig` command.

3. Perform a graceful shutdown and power cycle the host.

NVIDIA DOCA Switching Support

The NVIDIA® BlueField® DPU family delivers the flexibility to accelerate a range of applications while leveraging ConnectX-based network controllers hardware-based offloads with unmatched scalability, performance, and efficiency.

OVS configuration involves the definition and creation of representors, virtualization, optional bridge configuration, etc. The following subsections provide information and steps for setting up each of these software components.

This section contains the following pages:

- DPU Kernel Representors Model
- Virtio Acceleration through Hardware vDPA
- Bridge Offload
- DPU Functional Diagram
- Link Aggregation
- Controlling Host PF and VF Parameters

DPU Kernel Representors Model

> **Note**
>
> This model is only applicable when the DPU 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.

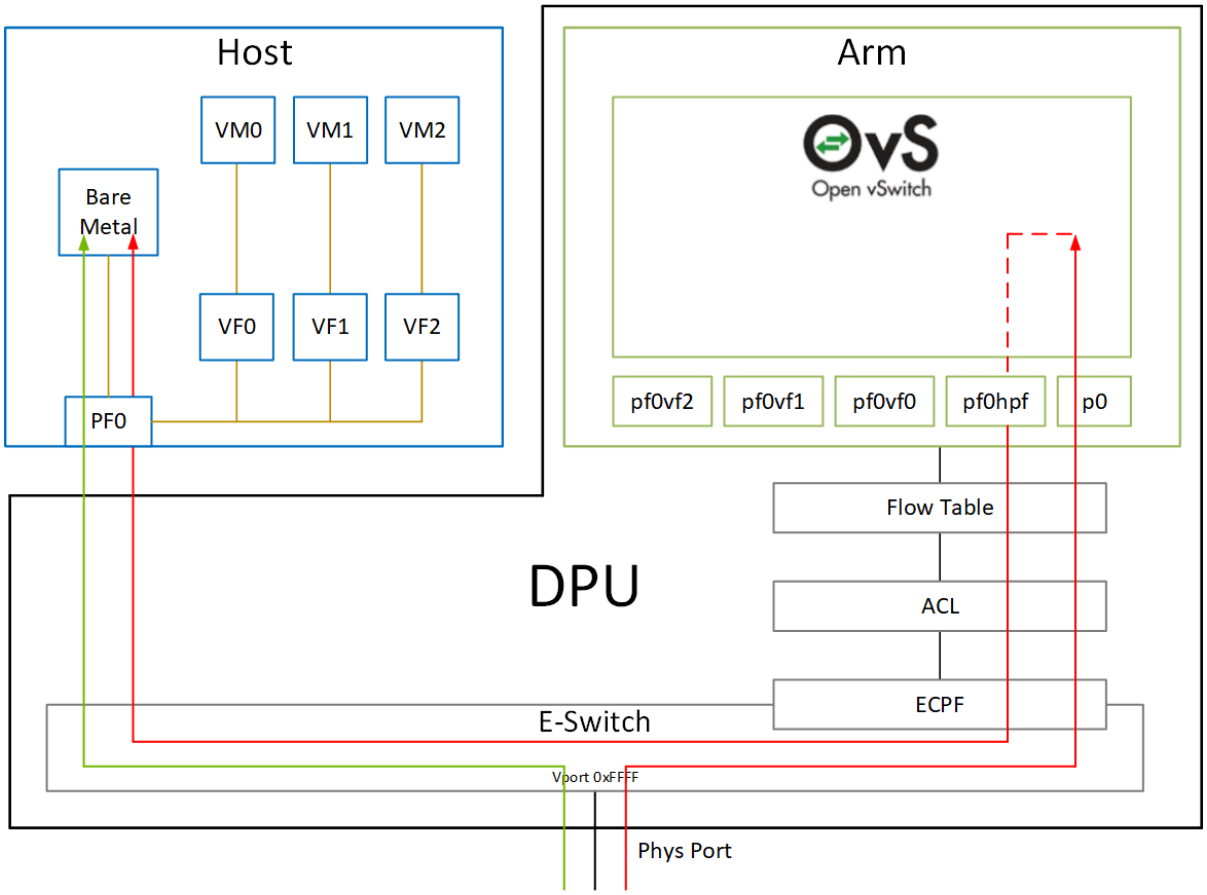
2. Serve as the channel to configure the embedded switch with rules to the corresponding represented function.

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 PCI 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.

Note

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.

Virtio Acceleration through Hardware vDPA

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
```

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:

   1. Change the top line to:

      ```
      <domain type='kvm'
      xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
      ```

   2. 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>
```

3. 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>
```

4. 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>
```

5. 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>
```

6. 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
   page-per-vq=on,rx_queue_size=1024,tx_queue_size=1024'/>
</qemu:commandline>
```

> **Note**
>
> 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.

Running Hardware vDPA

Note

Hardware vDPA supports switchdev mode only.

1. Create the ASAP^2 environment:
    1. Create the VFs.
    2. Enter switchdev mode.
    3. 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>
```

Note

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.

Bridge Offload

> **Note**
>
> Bridge offload is supported switchdev mode only.

> **Note**
>
> 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.

Basic Configuration

1. Initialize the ASAP^2 environment:
    1. Create the VFs.
    2. 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
```

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
```

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".

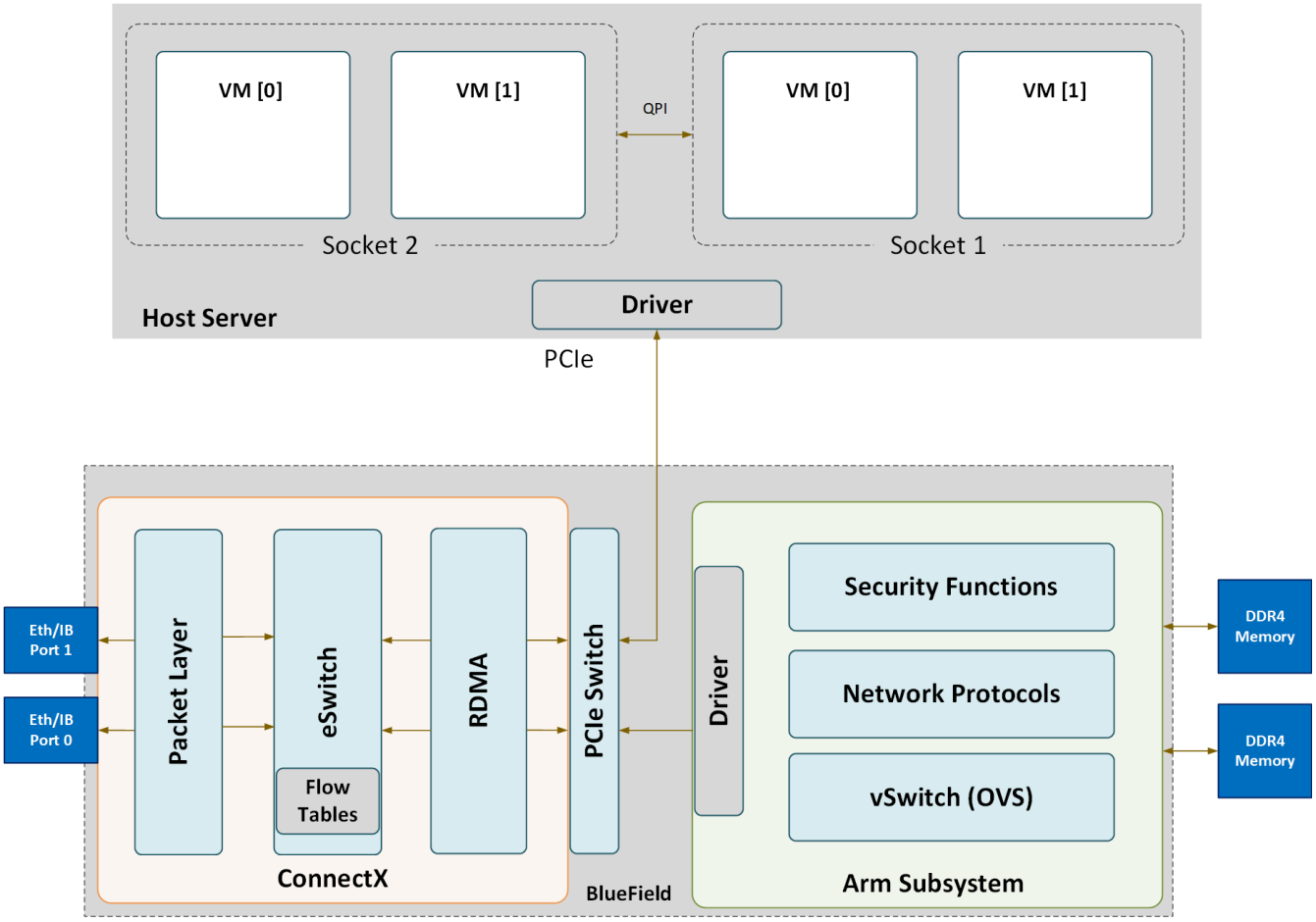
To add a bonding net device to bridge:

```
ip link set bond0 master bridge0
```

For further information on interacting with Linux bridge via iproute2 bridge tool, refer to man 8 bridge.

DPU Functional Diagram

The following is a functional diagram of the NVIDIA® BlueField® DPU.



For each one of the BlueField DPU network ports, there are 2 physical PCIe networking functions exposed:

1. To the embedded Arm subsystem
2. To the host over PCIe

The mlx5 drivers and their corresponding software stacks must be loaded on both hosts (Arm and the host server). The OS running on each one of the hosts would probe the drivers. BlueField-2 network interfaces are compatible with NVIDIA® ConnectX®-6 Dx and higher. BlueField-3 network interfaces are compatible with ConnectX-7 and higher.

The same network drivers are used both for BlueField and the ConnectX NIC family.

Link Aggregation

Network bonding enables combining two or more network interfaces into a single interface. It increases the network throughput, bandwidth and provides redundancy if one of the interfaces fails.

NVIDIA ® BlueField ® DPU has an option to configure network bonding on the Arm side in a manner transparent to the host. Under such configuration, the host would only see a single PF.

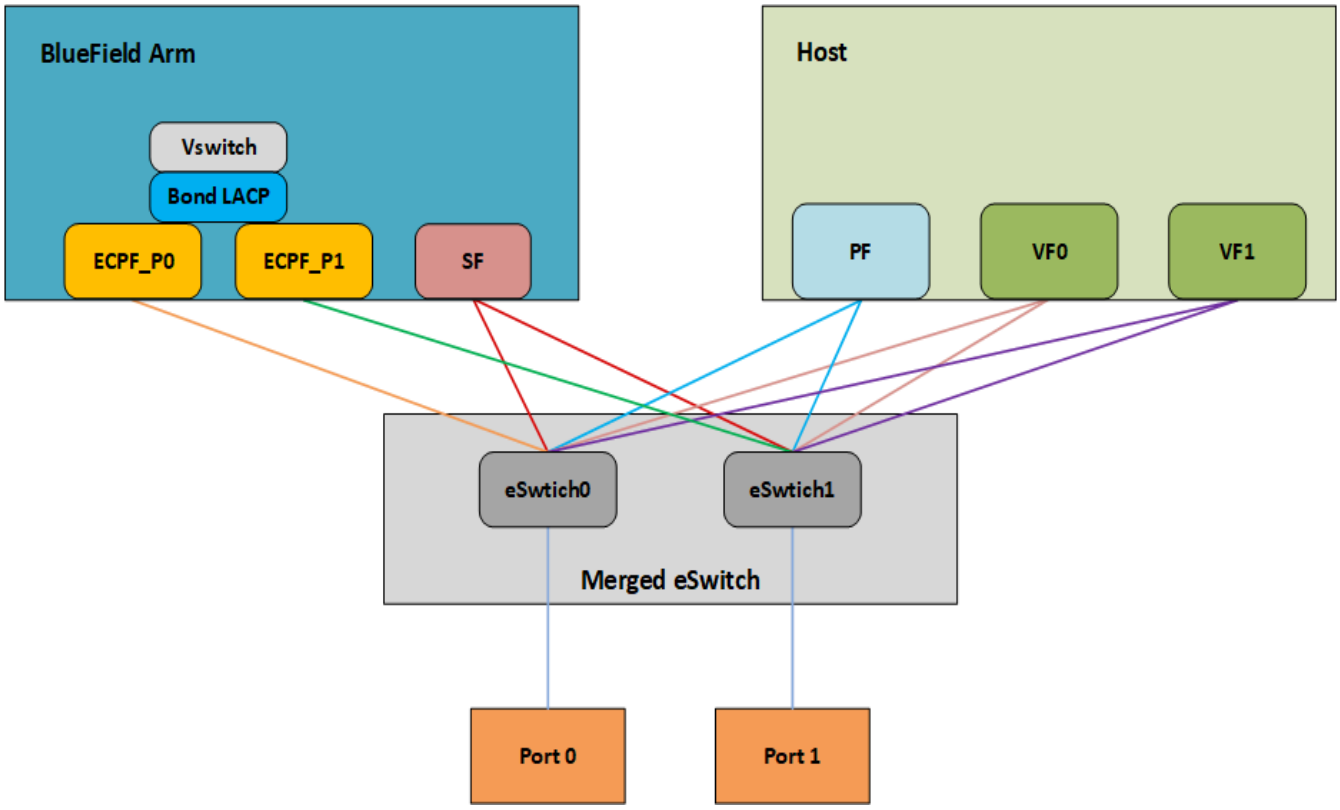
Note

This functionality is supported when the DPU is set in embedded function ownership mode for both ports.

Note

While LAG is being configured (starting with step 2 under section "LAG Configuration"), traffic cannot pass through the physical ports.

The diagram below describes this configuration:



LAG Modes

Two LAG modes are supported on BlueField:

- Queue Affinity mode
- Hash mode

Queue Affinity Mode

In this mode, packets are distributed according to the QPs.

1. To enable this mode, run:

```
$ mlxconfig -d /dev/mst/<device-name> s
LAG_RESOURCE_ALLOCATION=0
```

   Example device name: `mt41686_pciconf0`.

2. Add/edit the following field from `/etc/mellanox/mlnx-bf.conf` as follows:

```
LAG_HASH_MODE="no"
```

3. Perform a graceful shutdown and system power cycle.

Hash Mode

In this mode, packets are distributed to ports according to the hash on packet headers.

> **Note**
>
> For this mode, prerequisite steps 3 and 4 are not required.

1. To enable this mode, run:

```
$ mlxconfig -d /dev/mst/<device-name> s
LAG_RESOURCE_ALLOCATION=1
```

   Example device name: `mt41686_pciconf0`.

2. Add/edit the following field from `/etc/mellanox/mlnx-bf.conf` as follows:

```
LAG_HASH_MODE="yes"
```

3. Perform a graceful shutdown and system power cycle.

Prerequisites

1. Set the LAG mode to work with.
2. (Optional) Hide the second PF on the host. Run:

```
$ mlxconfig -d /dev/mst/<device-name> s HIDE_PORT2_PF=True
NUM_OF_PF=1
```

Example device name: `mt41686_pciconf0`.

> **Note**
>
> This step necessitates a system power cycle. If not performed, the second physical interface will still be visible to the host, but it will not be functional. This step has no effect on LAG configuration or functionality on the Arm side.

3. Delete any installed Scalable Functions (SFs) on the Arm side.
4. Stop the driver on the host side. Run:

```
$ systemctl stop openibd
```

5. The uplink interfaces (`p0` and `p1`) on the Arm side must be disconnected from any OVS bridge.

LAG Configuration

1. Create the bond interface. Run:

```
$ ip link add bond0 type bond
$ ip link set bond0 down
$ ip link set bond0 type bond miimon 100 mode 4
xmit_hash_policy layer3+4
```

Note

While LAG is being configured (starting with the next step), traffic cannot pass through the physical ports.

2. Subordinate both the uplink representors to the bond interface. Run:

```
$ ip link set p0 down
$ ip link set p1 down
$ ip link set p0 master bond0
$ ip link set p1 master bond0
```

3. Bring the interfaces up. Run:

```
$ ip link set p0 up
$ ip link set p1 up
$ ip link set bond0 up
```

The following is an example of LAG configuration in Ubuntu:

```
# cat /etc/network/interfaces

# interfaces(5) file used by ifup(8) and ifdown(8)
# Include files from /etc/network/interfaces.d:
source /etc/network/interfaces.d/*
auto lo
iface lo inet loopback
#p0
auto p0
iface p0 inet manual
        bond-master bond1
#
#p1
auto p1
iface p1 inet manual
        bond-master bond1
#bond1
auto bond1
iface bond1 inet static
        address 192.168.1.1
        netmask 255.255.0.0
        mtu 1500
        bond-mode 2
        bond-slaves p0 p1
        bond-miimon 100
        pre-up (sleep 2 && ifup p0) &
        pre-up (sleep 2 && ifup p1) &
```

As a result, only the first PF of the DPU would be available to the host side for networking and SR-IOV.

> **⚠ Warning**
>
> When in shared RQ mode (enabled by default), the uplink interfaces (`p0` and `p1`) must always stay enabled. Disabling them will break LAG support and VF-to-VF communication on same host.

For OVS configuration, the bond interface is the one that needs to be added to the OVS bridge (interfaces `p0` and `p1` should not be added). The PF representor for the first port (`pf0hpf`) of the LAG must be added to the OVS bridge. The PF representor for the second port (`pf1hpf`) would still be visible, but it should not be added to OVS bridge. Consider the following examples:

```
ovs-vsctl add-br bf-lag
ovs-vsctl add-port bf-lag bond0
ovs-vsctl add-port bf-lag pf0hpf
```

> **⚠ Warning**
>
> Trying to change bonding configuration in Queue Affinity mode (including bringing the subordinated interface up/down) while the host driver is loaded would cause FW syndrome and failure of the operation. Make sure to unload the host driver before altering DPU bonding configuration to avoid this.

Note

When performing driver reload (`openibd restart`) or reboot, it is required to remove bond configuration and to reapply the configurations after the driver is fully up. Refer to steps 1-4 of "Removing LAG Configuration".

Removing LAG Configuration

1. If Queue Affinity mode LAG is configured (i.e., `LAG_RESOURCE_ALLOCATION=0`):
    1. Delete any installed Scalable Functions (SFs) on the Arm side.
    2. Stop driver (openibd) on the host side. Run:

        ```
        systemctl stop openibd
        ```

2. Delete the LAG OVS bridge on the Arm side. Run:

    ```
    ovs-vsctl del-br bf-lag
    ```

    This allows for later restoration of OVS configuration for non-LAG networking.

3. Stop OVS service. Run:

    ```
    systemctl stop openvswitch-switch.service
    ```

4. Run:

```
ip link set bond0 down
modprobe -rv bonding
```

As a result, both of the DPU's network interfaces would be available to the host side for networking and SR-IOV.

5. For the host to be able to use the DPU ports, make sure to attach the ECPF and host representor in an OVS bridge on the Arm side. Refer to "Virtual Switch on DPU" for instructions on how to perform this.

6. Revert from `HIDE_PORT2_PF`, on the Arm side. Run:

```
mlxconfig -d /dev/mst/<device-name> s HIDE_PORT2_PF=False
NUM_OF_PF=2
```

7. Restore default LAG settings in the DPU's firmware. Run:

```
mlxconfig -d /dev/mst/<device-name> s
LAG_RESOURCE_ALLOCATION=DEVICE_DEFAULT
```

8. Delete the following line from `/etc/mellanox/mlnx-bf.conf` on the Arm side:

```
LAG_HASH_MODE=...
```

9. Perform a graceful shutdown and system power cycle.

LAG on Multi-host

Only LAG hash mode is supported with BlueField multi-host.

LAG Multi-host Prerequisites

1. Enable LAG hash mode.

2. Hide the second PF on the host. Run:

```
$ mlxconfig -d /dev/mst/<device-name> s HIDE_PORT2_PF=True
NUM_OF_PF=1
```

3. Make sure NVME emulation is disabled:

```
$ mlxconfig -d /dev/mst/<device-name> s
NVME_EMULATION_ENABLE=0
```

   Example device name: `mt41686_pciconf0`.

4. The uplink interfaces (`p0` and `p4`) on the Arm side, representing port0 and port1, must be disconnected from any OVS bridge. As a result, only the first PF of the DPU would be available to the host side for networking and SR-IOV.

LAG Configuration on Multi-host

1. Create the bond interface. Run:

```
$ ip link add bond0 type bond
$ ip link set bond0 down
$ ip link set bond0 type bond miimon 100 mode 4
xmit_hash_policy layer3+4
```

2. Subordinate both the uplink representors to the bond interface. Run:

```
$ ip link set p0 down
$ ip link set p4 down
$ ip link set p0 master bond0
$ ip link set p4 master bond0
```

3. Bring the interfaces up. Run:

```
$ ip link set p0 up
$ ip link set p4 up
$ ip link set bond0 up
```

4. For OVS configuration, the bond interface is the one that must be added to the OVS bridge (interfaces p0 and p4 should not be added). The PF representor, `pf0hpf`, must be added to the OVS bridge with the bond interface. The rest of the uplink representors must be added to another OVS bridge along with their PF representors. Consider the following examples:

```
ovs-vsctl add-br br-lag
ovs-vsctl add-port br-lag bond0
ovs-vsctl add-port br-lag pf0hpf
ovs-vsctl add-br br1
ovs-vsctl add-port br1 p1
ovs-vsctl add-port br1 pf1hpf
ovs-vsctl add-br br2
ovs-vsctl add-port br2 p2
ovs-vsctl add-port br2 pf2hpf
ovs-vsctl add-br br3
ovs-vsctl add-port br3 p3
ovs-vsctl add-port br3 pf3hpf
```

Note

When performing driver reload (`openibd restart`) or reboot, you must remove bond configuration from NetworkManager, and to reapply the configurations after the driver is fully up.

Removing LAG Configuration on Multi-host

Refer to section "Removing LAG Configuration".

NVIDIA DOCA with OpenSSL

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

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.

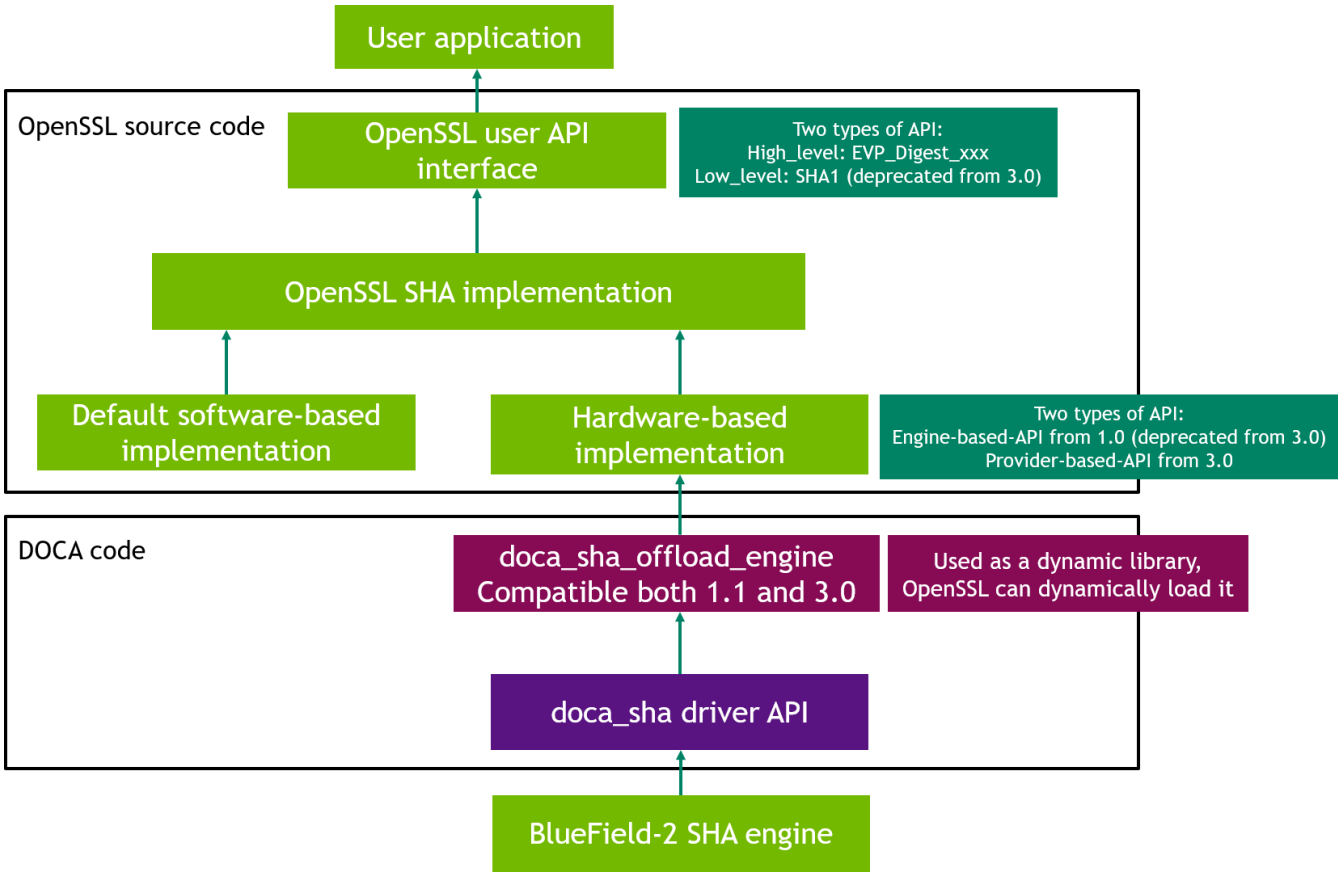
Prerequisites

- Hardware-based `doca_sha` engine which can be verified by calling `doca_sha_get_hardware_supported()`
- Installed OpenSSL version ⩾ 1.1.1

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.



Capabilities and Limitations

- Only one-shot OpenSSL SHA is supported
- The maximum message length ⩽ 2GB, the same as `doca_sha` library

OpenSSL Command Line Verification

Verify that the engine can be loaded:

```
$ openssl engine dynamic -pre NO_VCHECK:1 -pre
SO_PATH:${DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_
-pre LOAD -vvv -t -c
(dynamic) Dynamic engine loading support
[Success]:
SO_PATH:${DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_
[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_
-engine_impl
```

- For SHA-256:

```
$ echo "hello world" | openssl dgst -sha256 -engine
{DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_sha_
-engine_impl
```

- For SHA-512:

```
$ echo "hello world" | openssl dgst -sha512 -engine
{DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_sha_
-engine_impl
```

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_
```

- 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_
-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_
-async_jobs 256 -multi 8
```

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);
```

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.

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).

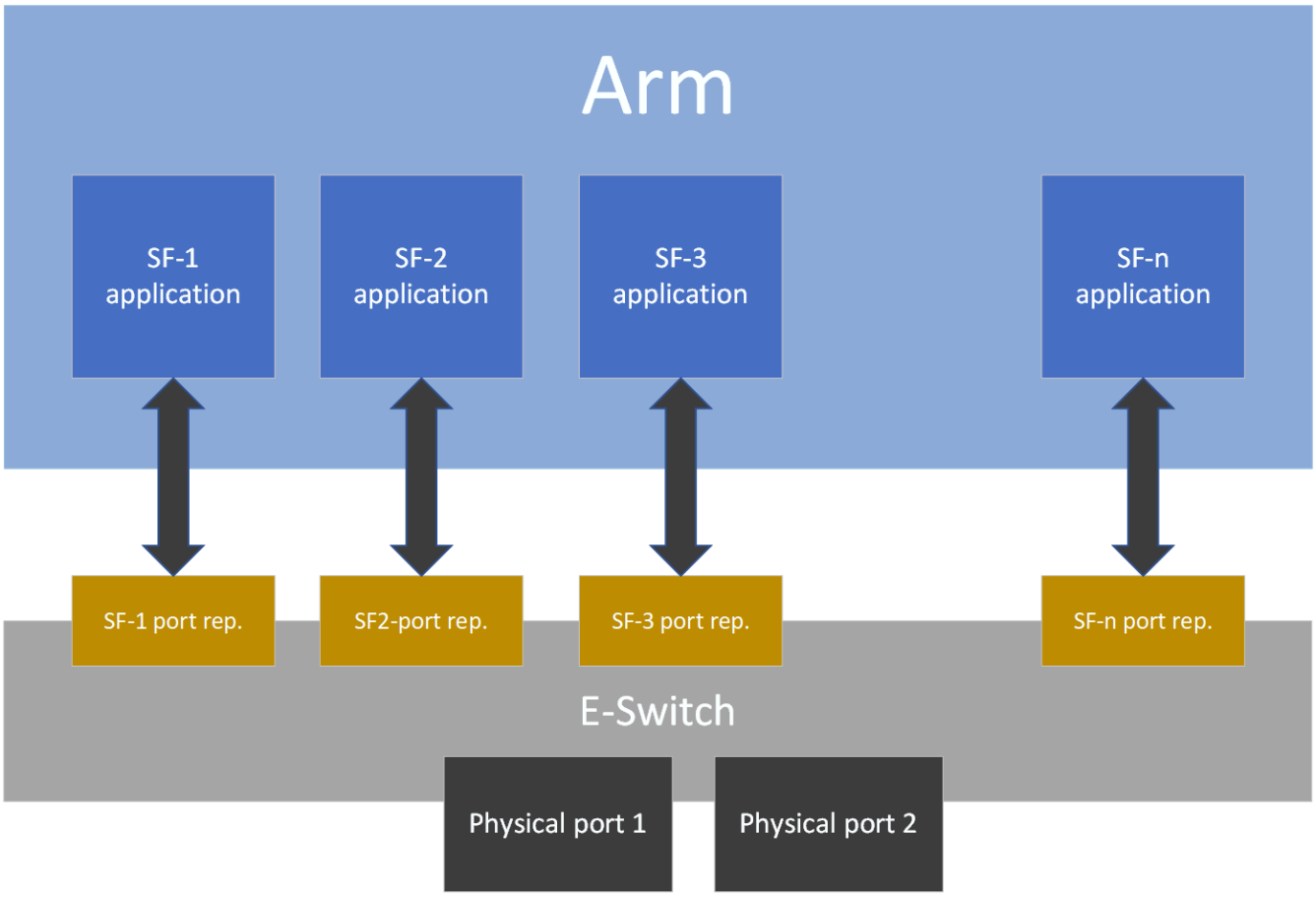
The PF provides control over the creation and allocation of new VFs. VFs share the device's underlying hardware and PCIe. A key feature of the SR-IOV specification is that VFs are very lightweight so that many of them can be implemented in a single device.

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.



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.
A cold reboot is needed for the effect to take place.
```

SF Configuration

To use an SF, a 3-step setup sequence must be followed first:

1. Create.
2. Configure.
3. Deploy.



These steps can be performed using `mlxdevm` tool.

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>
   ```

   Note

   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
```

> **Note**
>
> 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
```

Note

`<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
```

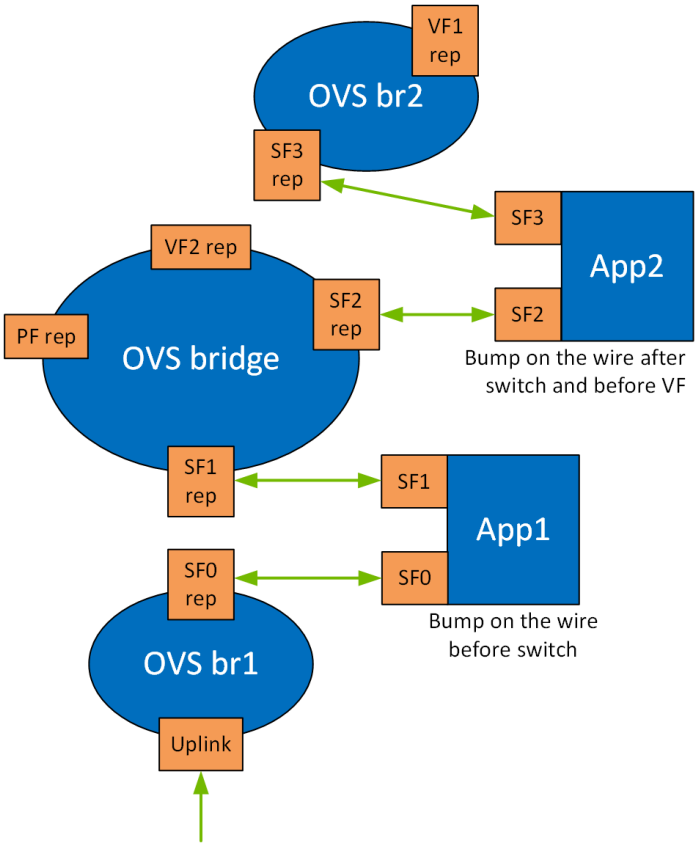
To delete the SF port representor, run:

```
/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.

> **Note**
>
> 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.

1. 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
```

2. 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.

3. Delete existing OVS bridges (optional).

   For example, run the following command to delete an OVS bridge called `ovsbr1`:

   ```
   ovs-vsctl del-br ovsbr1
   ```

4. 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
   ```

5. 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"
```

Note

The interface might be down by default. Remember to `ifconfing` the interface to "up" status.

> **Note**
>
> 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,sft_en=1 -a
auxiliary:mlx5_core.sf.5,sft_en=1 -- [application_flags]
```

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.

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

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.

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.

Note

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).

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).

kTLS Offload Flow in High Level

> **Note**
>
> 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.

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.

Prerequisites

All commands in this section should be performed on host (not on BlueField) unless stated otherwise.

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
....
```

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.

> **Note**
>
> 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 configur ations and recompile, or build a new kernel .

> **Note**
>
> Follow the build instructions provided with the kernel provider.

Schematic flow for building a Linux kernel:

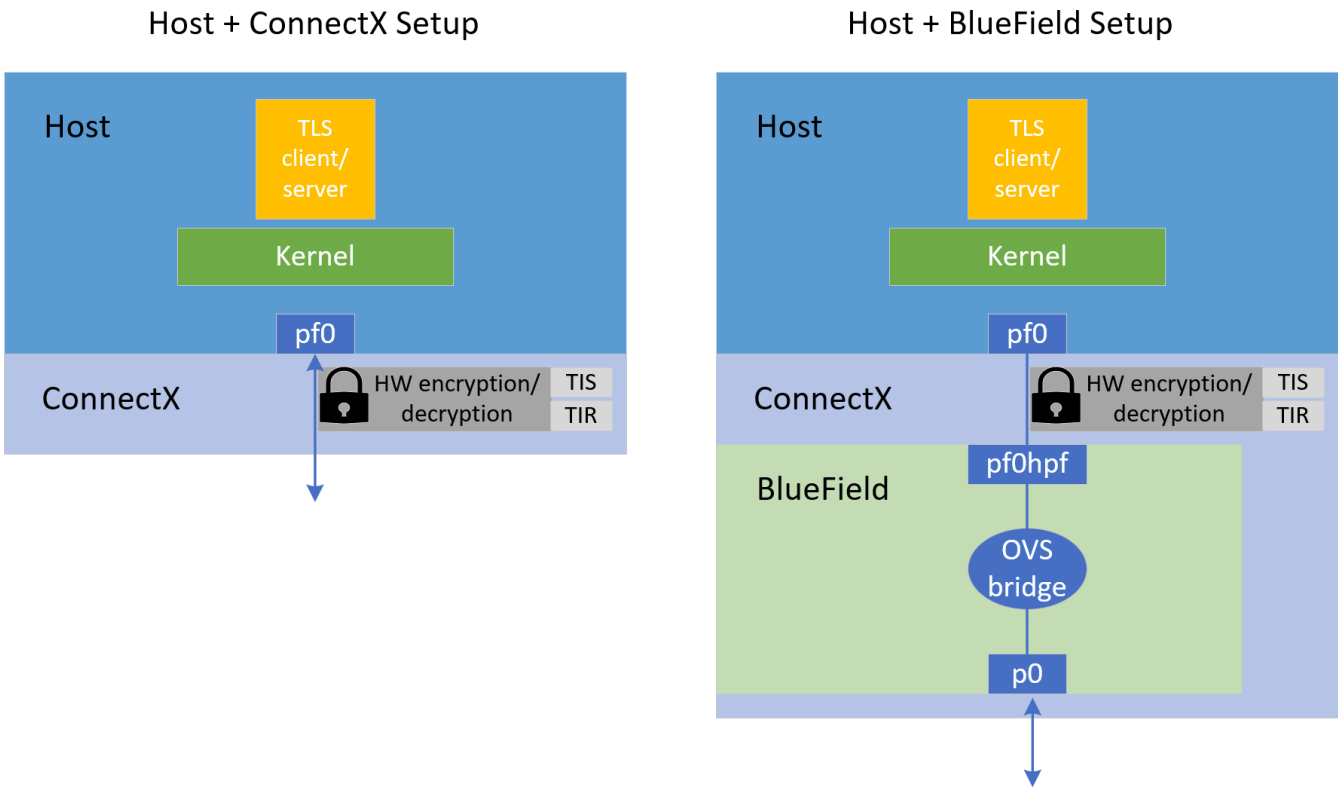
1. Enter the Linux kernel directory downloaded (usually in `/usr/src/`):

```
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
```

2. Update the grub to the new configured kernel then reboot.

Configurations and Useful Commands

TLS Setup



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.

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]
  ```

> **Note**
>
> `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>
```

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).

> **Note**
>
> 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.

Common Use Cases

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.

> **Note**
>
> 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.

> ⚠ **Warning**
>
> 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`.

Note

Installing a new OpenSSL requires recompiling user tools that were configured over OpenSSL (e.g., Nginx).

Note

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.

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.

Prerequisites

Refer to the OpenSSL section for setting OpenSSL.

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
```

> **Note**
>
> 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_conf_command Options KTLS;
    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.

Stopping Nginx

```
host> pkill nginx
```

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>
```

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
```

Note

Testing the kTLS offload (with or without hardware offload) is in the same manner as mentioned in section "Testing kTLS". TBD

Testing Offload via OpenSSL

This chapter demonstrates how to test the kTLS hardware offload.

Note

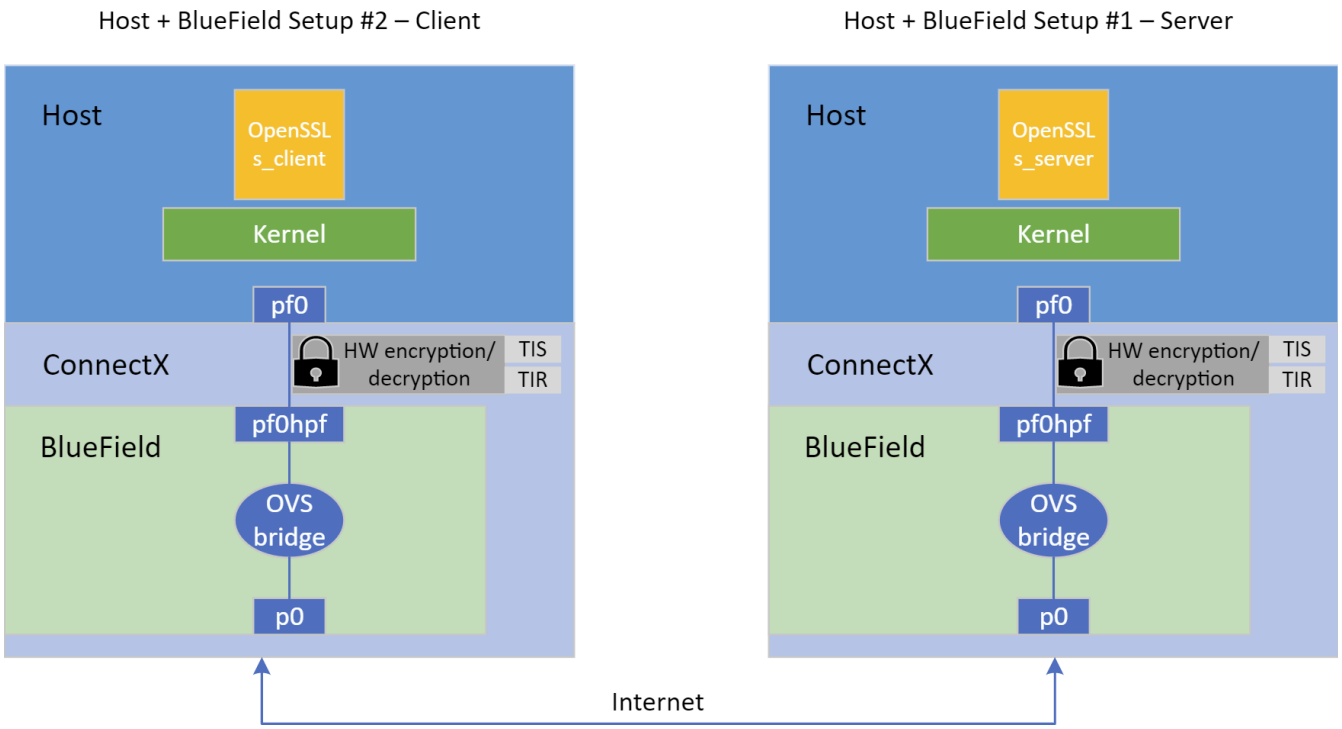
Make sure to refer to section "OpenSSL" before proceeding.

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".



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

Note

The server side should be run before client side so that client's request are answered by server.

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
```

Note

Notice the `-ktls` flag.

Note

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.

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.

> **Note**
>
> Refer to official OpenSSL documentation on `s_client` for more information.

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:

> **Note**
>
> 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
```

Note

More information about the kernel counters can be found in the Statistics section of the Kernel TLS documentation.

Optimizations over kTLS

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.

Note

Even though XLIO is a kernel-bypass library, the kernel must support kTLS for the bypass to work properly.

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:

Note

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:

    1. 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
        ```

    2. 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
        ```

    3. 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
```

4. Update grub:

```
host> sudo update-grub
```

5. 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:

> **Note**
>
> 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.

Note

The script is within MLNX_OFED's sources:

1. You can find it in MLNX_OFED downloads.
2. Under "Download" select the correct version and download the "SOURCES" `.tgz` file.
3. Extract the `.tgz`.
4. 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
```

Additional Reading

- Linux kernel TLS documentation
- Linux kernel TLS offload documentation
- Autonomous NIC offloads research paper

NVIDIA DOCA Troubleshooting Guide

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

DOCA Infrastructure

RShim Troubleshooting and How-Tos

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.

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.

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.

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.

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.

Connectivity Troubleshooting

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
```

> **Note**
>
> 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.

> **Info**
>
> 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.

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.

> **Note**
>
> 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 still 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.

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".

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.

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
  ```

SR-IOV Troubleshooting

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"`.

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"
```

eSwitch Troubleshooting

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>
  ```

Note

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
```

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
  ```

DOCA Applications

This chapter deals with troubleshooting issues related to DOCA applications.

EAL Initialization Failure

EAL initialization failure is a common error that may appear while running various DPDK-related applications.

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

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>
```

Ring Memory Issue

This is a common memory issue when running application on the host.

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).

Solution

Possible solutions:

- Recommended: Increase the amount of allocated huge pages. Instructions for allocating huge pages can be found here.

> **Note**
>
> 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:

```
/opt/mellanox/doca/applications/<app_name>/bin/doca_<app_name>
-a 3b:00.3 -a 3b:00.4 -l 0-64 -- -l 60
```

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.

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.

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:

```
/opt/mellanox/doca/applications/<app_name>/bin/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
```

Compilation of DOCA Apps on CentOS

When compiling gRPC-enabled applications on old (7.6) CentOS machines, there is a conflict between the libstdc++ version available out-of-the-box and the one used by DOCA's SDK when building the gRPC packages.

Error

Compiling the gRPC-enabled application results in the following errors:

```
$ meson /tmp/build -Denable_grpc_support=true ; ninja -C
/tmp/build
...
l_log_severity.a -Wl,--end-group
/opt/mellanox/grpc/lib/libgrpc++.a(server_cc.cc.o): In function
`grpc::Server::RegisterService(std::string const*,
grpc::Service*)':
(.text+0x2467): undefined reference to `std::basic_ios<char,
std::char_traits<char> >::operator bool() const'
/opt/mellanox/grpc/lib/libgrpc++.a(server_cc.cc.o): In function
`grpc::Server::RegisterService(std::string const*,
grpc::Service*)':
(.text+0x249e): undefined reference to `std::basic_ios<char,
std::char_traits<char> >::operator bool() const' collect2: error:
ld returned 1 exit status
```

Solution

Upgrading the devtoolset on the machine to the one used when building the gRPC package resolves the version conflict.

```
$ 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
```

Failure to Set Huge Pages

When trying to configure the huge pages from an unprivileged user account, a permission error is raised.

Error

Compiling the gRPC-enabled application results in the following errors:

```
$ sudo echo 600 > /sys/kernel/mm/hugepages/hugepages-
2048kB/nr_hugepages
-bash: /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages:
Permission denied
```

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
```

DOCA Libraries

This chapter deals with troubleshooting issues related to DOCA libraries.

DOCA Flow Error

When trying to add new entry to the pipe, an error is received.

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.

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
```

DOCA SDK Compilation

This chapter deals with troubleshooting issues related to compiling DOCA-based programs to use the DOCA SDK (e.g., missing dependencies).

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).

Error

There are multiple forms this error may appear in, such as:

- DOCA libraries are missing:

```
Dependency doca found: NO (tried pkgconfig and cmake)
meson.build:13:1: ERROR:  Dependency "doca" not found, tried
pkgconfig and cmake
```

- 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
```

- gRPC definitions are missing (when gRPC support is activated):

```
Dependency protobuf found: NO (tried pkgconfig and cmake)
meson.build:47:1 ERROR:  Dependency "protobuf" not found, tried
pkgconfig and cmake
```

- gRPC compiler definitions are missing (when gRPC support is activated):

```
Dependency protobuf found: YES 3.15.8.0
Dependency grpc++ found: YES 1.39.0
Program protoc found: NO
meson.build:50:1: ERROR:  Program(s) ['protoc'] not found or not
executable
```

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:

Note

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`.

Tip

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, gRPC, FlexIO, etc).

DOCA Libraries & Tools:

- For Ubuntu:

```
export
PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/opt/mellanox/doca/lib/aarc
linux-gnu/pkgconfig
export PATH=${PATH}:/opt/mellanox/doca/tools
```

- For CentOS:

```
export
PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/opt/mellanox/doca/lib64/pk
export PATH=${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/aarc
linux-gnu/pkgconfig
```

- For CentOS:

```
export
PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/opt/mellanox/dpdk/lib64/pk
```

gRPC:

- For Ubuntu:

```
export
PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/opt/mellanox/grpc/lib/pkgc
export PATH=${PATH}:/opt/mellanox/grpc/bin
```

- For CentOS:

```
export
PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/opt/mellanox/grpc/lib/pkgc
export PATH=${PATH}:/opt/mellanox/grpc/bin
```

FlexIO:

- For Ubuntu:

```
export
PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/opt/mellanox/flexio/lib/pk
```

- For CentOS:

```
export
PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/opt/mellanox/flexio/lib/pk
```

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.

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
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
```

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
```

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.

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' ...
```

Solution

Use an updated version of pkg-config or pkgconf instead when building applications (as is recommended in DPDK's compilation instructions).

Cross-compiling DOCA and CUDA

This chapter deals with troubleshooting issues related to DOCA-CUDA cross-compilation.

Application Build Error

When trying to build with meson, an architecture-related error is received.

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.

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.

DOCA Services (Containers)

This section deals with troubleshooting issues related to DOCA-based containers.

YAML Syntax Error #1

When deploying the container using the respective YAML file, the pod fails to start.

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.

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.

YAML Syntax Error #2

When deploying the container using the respective YAML file, the pod fails to start.

Error

The error may happen after modifying a service's YAML file, or after copying an example YAML file from one of the guides.

Note

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.

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.

Missing Huge Pages

When deploying the container using the respective YAML file, the pod fails to start.

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.

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:
    1. Make sure that the huge pages are allocated as required per the desired container.
    2. 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`).

Failed to Reserve Sandbox Name

After rebooting the DPU, the respective pods start. However, the containers repeatedly fail to spawn and their "attempt" counter is not incrementing.

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 was some collision with prior instances of the "doca-hbn-service" container, probably pre-reboot.

> **Note**
>
> 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.

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`).

How to Perform Graceful Shutdown

Before powering off or power cycling the DPU, it is strongly recommended to perform graceful shutdown of the DPU Arm OS.

Graceful shutdown of the Arm OS ensures that data within the eMMC/NVMe cache is properly written to storage, and helps prevent filesystem inconsistencies and file corruption.

There are several ways to gracefully shutdown the DPU Arm OS:

- Log into the DPU Arm OS and perform a shutdown command prior to power cycling the host server. For example:

```
sudo shutdown -h now
```

- Assuming the DPU BMC can issue NC-SI OEM commands to the DPU:

    1. Issue the Shutdown Smart NIC OS NC-SI OEM command.

    2. After DPU Arm OS shutdown, it is recommended to issue DPU Arm OS state query which indicates whether DPU Arm OS shutdown has completed (

`standby` indication). This can be done by issuing the Get Smart NIC OS State NC-SI OEM command.

NVIDIA DOCA Virtual Functions User Guide

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

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.

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

    1. 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.

    Info

    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.

    1. 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"
    ```

    2. 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"
    ```

> **Note**
>
> These commands set trusted mode for all created VFs/SFs after their execution on the DPU.

> **Note**
>
> 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
```

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
   ```

   > **Note**
   >
   > Requires graceful shutdown and power cycle.

```
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)
```

Note

2 new virtual Ethernet devices are created in this example.

Running DOCA Application on Host

Note

Allocate the required amount of VFs as explained previously.

Note

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:

```
./opt/mellanox/doca/applications/<app_name>/bin/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:

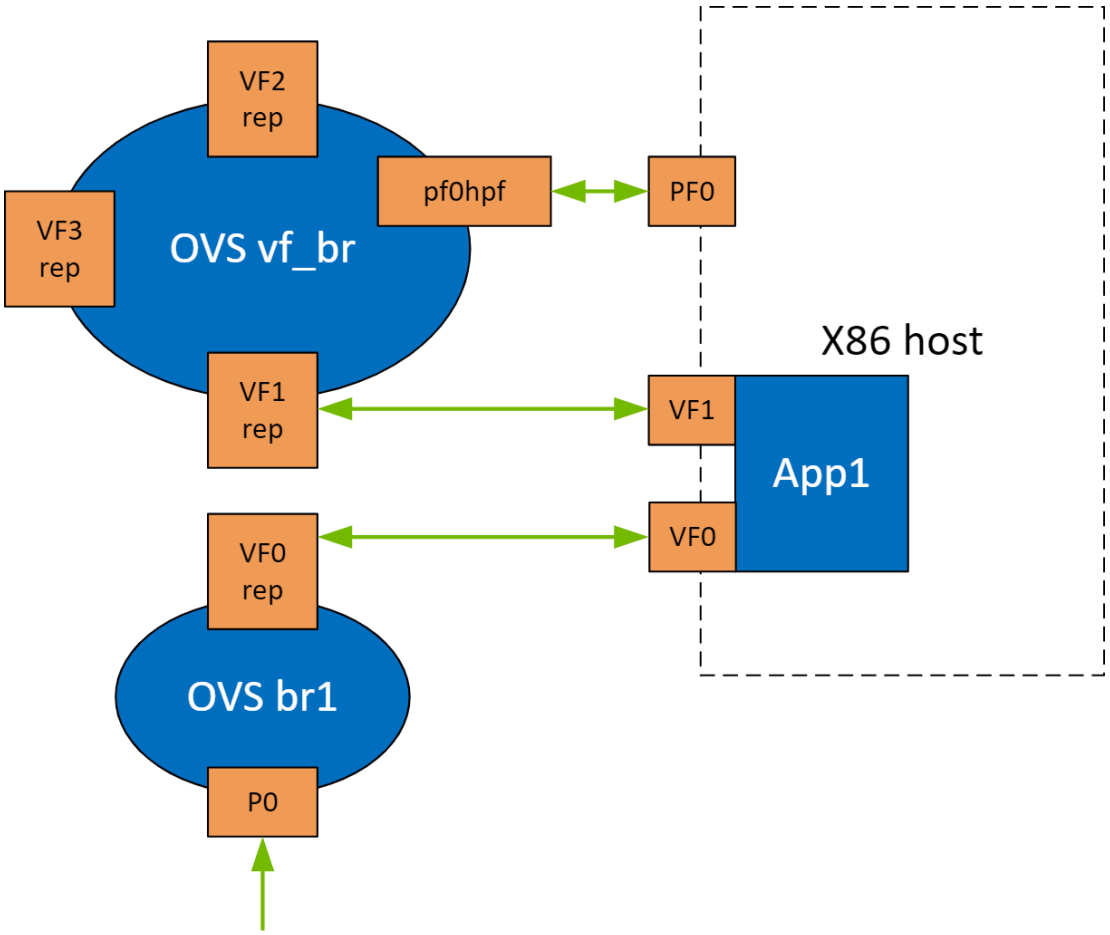
```
./opt/mellanox/doca/applications/<app_name>/bin/doca_<app_name> -a b1:00.3 -a b1:00.4 -c 0xff -- -l 60
```

Note

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.

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` .

VF Creation on Adapter Card

Note

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
```

Archives

This section contains the following pages:

- NVIDIA DOCA LTS Versions
- NVIDIA DOCA Documentation Archives

NVIDIA DOCA LTS Versions

Documentation for DOCA long term support (LTS) releases.

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.

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
  - 1.5.1 LTS update
  - 1.5.2 LTS update

NVIDIA DOCA Documentation Archives

Archived documentation of previous DOCA software releases.

- DOCA v2.5.2 documentation
- DOCA v2.5.1 documentation
- DOCA v2.5.0 documentation
- DOCA v2.2.1 documentation
- DOCA v2.2.0 documentation
- DOCA v2.0.2 documentation
- DOCA v1.5.2 documentation
- DOCA v1.5.1 documentation
- DOCA v1.5.0 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. NVIDIA Corporation (“NVIDIA”) makes no 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

NVIDIA and the NVIDIA logo are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

© Copyright 2024, NVIDIA. PDF Generated on 04/07/2026