



NVIDIA DOCA Flow

Programming Guide

Table of Contents

Chapter 1. Introduction.....	1
Chapter 2. Prerequisites.....	3
Chapter 3. Architecture.....	4
Chapter 4. Steering Domains.....	6
4.1. List of Steering Domains.....	6
4.2. Domains in VNF Mode.....	7
4.3. Domains in Switch Mode.....	8
Chapter 5. API.....	10
5.1. doca_flow_cfg.....	10
5.2. doca_flow_port_cfg.....	11
5.3. doca_flow_pipe_cfg.....	12
5.4. doca_flow_meta.....	14
5.5. doca_flow_header_format.....	16
5.6. doca_flow_tun.....	16
5.7. doca_flow_match.....	18
5.8. doca_flow_actions.....	19
5.8.1. doca_flow_push_action.....	19
5.9. doca_flow_action_desc.....	20
5.10. doca_flow_monitor.....	21
5.11. doca_flow_fwd.....	22
5.12. doca_flow_query.....	23
5.13. doca_flow_aged_query.....	24
5.14. doca_flow_init.....	24
5.15. doca_flow_port_start.....	24
5.16. doca_flow_port_pair.....	25
5.17. doca_flow_pipe_create.....	25
5.18. doca_flow_pipe_add_entry.....	26
5.19. doca_flow_pipe_control_add_entry.....	27
5.20. doca_flow_pipe_lpm_add_entry.....	28
5.21. doca_flow_pipe_acl_add_entry.....	29
5.22. doca_flow_pipe_ordered_list_add_entry.....	29
5.23. doca_flow_pipe_hash_add_entry.....	30
5.24. doca_flow_entries_process.....	31
5.25. doca_flow_entry_status.....	32
5.26. doca_flow_entry_query.....	32

5.27. <code>doca_flow_query_pipe_miss</code>	33
5.28. <code>doca_flow_aging_handle</code>	33
5.29. <code>doca_flow_mpls_label_encode</code>	34
5.30. <code>doca_flow_mpls_label_decode</code>	34
Chapter 6. Shared Counter Resource	35
6.1. On <code>doca_flow_init()</code>	35
6.2. On <code>doca_flow_shared_resource_cfg()</code>	35
6.3. On <code>doca_flow_shared_resource_bind()</code>	35
6.4. On <code>doca_flow_pipe_add_entry()</code> or Pipe Configuration (struct <code>doca_flow_pipe_cfg</code>).....	36
6.5. Querying Bulk of Shared Counter IDs.....	36
6.6. On <code>doca_flow_pipe_destroy()</code> or <code>doca_flow_port_stop()</code>	37
Chapter 7. Shared Meter Resource	38
7.1. On <code>doca_flow_init()</code>	38
7.2. On <code>doca_flow_shared_resource_cfg()</code>	38
7.3. On <code>doca_flow_shared_resource_bind()</code>	39
7.4. On <code>doca_flow_pipe_add_entry()</code> or Pipe Configuration (struct <code>doca_flow_pipe_cfg</code>).....	39
7.5. Querying Bulk of Shared Meter IDs.....	40
7.6. On <code>doca_flow_pipe_destroy()</code> or <code>doca_flow_port_stop()</code>	40
Chapter 8. Shared RSS Resource	41
8.1. On <code>doca_flow_init()</code>	41
8.2. On <code>doca_flow_shared_resource_cfg()</code>	41
8.3. On <code>doca_flow_shared_resource_bind()</code>	41
8.4. On <code>doca_flow_pipe_add_entry()</code>	42
8.5. On <code>doca_flow_port_stop()</code>	42
Chapter 9. Shared Crypto Resource	43
9.1. On <code>doca_flow_init()</code>	43
9.2. On <code>doca_flow_shared_resource_cfg()</code>	43
9.3. On <code>doca_flow_shared_resource_bind()</code>	45
9.4. On <code>doca_flow_pipe_add_entry()</code> or Pipe Configuration (struct <code>doca_flow_pipe_cfg</code>).....	45
9.5. On <code>doca_flow_pipe_destroy()</code> or <code>doca_flow_port_stop()</code>	46
Chapter 10. Shared Mirror Resource	47
10.1. On <code>doca_flow_init()</code>	47
10.2. On <code>doca_flow_shared_resource_cfg()</code>	47
10.3. On <code>doca_flow_shared_resource_bind()</code>	48

10.4. On <code>doca_flow_pipe_add_entry()</code> or Pipe Configuration (struct <code>doca_flow_pipe_cfg</code>).....	49
Chapter 11. Flow Life Cycle.....	50
11.1. Initialization Flow.....	50
11.1.1. Pipe Mode.....	50
11.2. Start Point.....	52
11.3. Create Pipe and Pipe Entry.....	53
11.3.1. Setting Pipe Match.....	53
11.3.1.1. Implicit Match.....	54
11.3.1.2. Explicit Match.....	55
11.3.2. Setting Pipe Actions.....	56
11.3.2.1. Auto-modification.....	56
11.3.2.2. Explicit Modification Type.....	57
11.3.2.3. Copy Field.....	57
11.3.2.4. Multiple Actions List.....	57
11.3.2.5. Summary of Action Types.....	58
11.3.2.6. Summary of Fields.....	58
11.3.3. Setting Pipe Monitoring.....	59
11.3.4. Setting Pipe Forwarding.....	59
11.3.5. Basic Pipe Create.....	60
11.3.6. Pipe Entry (<code>doca_flow_pipe_add_entry()</code>).....	61
11.3.6.1. Pipe Entry Counting.....	62
11.3.6.2. Pipe Entry Aged Query.....	62
11.3.7. Pipe Entry With Multiple Actions.....	62
11.3.8. Miss Pipe and Control Pipe.....	62
11.3.9. <code>doca_flow_pipe_lpm</code>	64
11.3.10. <code>doca_flow_pipe_acl</code>	65
11.3.11. <code>doca_flow_pipe_ordered_list</code>	65
11.3.12. <code>doca_flow_pipe_hash</code>	66
11.3.13. Hardware Steering Mode.....	66
11.3.14. Isolated Mode.....	67
11.4. Teardown.....	67
11.4.1. Pipe Entry Teardown.....	67
11.4.2. Pipe Teardown.....	68
11.4.3. Port Teardown.....	68
11.4.4. Flow Teardown.....	68
Chapter 12. Packet Processing.....	69
Chapter 13. DOCA Flow gRPC.....	71

13.1. Proto-Buff.....	73
13.1.1. Response Message.....	74
13.1.2. DocaFlowCfg.....	74
13.1.3. DocaFlowPortCfg.....	74
13.1.4. DocaFlowPipeCfg.....	74
13.1.5. DocaFlowMeta.....	74
13.1.6. DocaFlowMatch.....	74
13.1.7. DocaFlowActions.....	74
13.1.8. DocaFlowActionDesc.....	75
13.1.9. DocaFlowMonitor.....	75
13.1.10. DocaFlowFwd.....	75
13.1.11. DocaFlowQueryStats.....	75
13.1.12. DocaFlowHandleAgingRes.....	75
13.1.13. DocaFlowInit.....	75
13.1.14. DocaFlowPortStart.....	75
13.1.15. DocaFlowPortPair.....	75
13.1.16. DocaFlowPipeCreate.....	76
13.1.17. DocaFlowPipeAddEntry.....	76
13.1.18. DocaFlowPipeControlAddEntry.....	76
13.1.19. DocaFlowPipeLpmAddEntry.....	77
13.1.20. DocaFlowEntriesProcess.....	77
13.1.21. DocaFlowEntyGetStatus.....	78
13.1.22. DocaFlowQuery.....	78
13.1.23. DocaFlowAgingHandle.....	78
13.1.24. DocaFlowSharedResourceCfg.....	79
13.1.25. DocaFlowSharedResourcesBind.....	79
13.1.26. DocaFlowSharedResourcesQuery.....	79
13.2. DOCA Flow gRPC Client API.....	80
13.2.1. doca_flow_grpc_response.....	80
13.2.2. doca_flow_grpc_pipe_cfg.....	81
13.2.3. doca_flow_grpc_fwd.....	81
13.2.4. doca_flow_grpc_client_create.....	81
13.3. DOCA Flow gRPC Usage.....	81
Chapter 14. DOCA Flow Samples.....	83
14.1. Sample Prerequisites.....	83
14.2. Running the Sample.....	83
14.3. Samples.....	84
14.3.1. Flow ACL.....	84

14.3.2. Flow Aging.....	86
14.3.3. Flow Control Pipe.....	86
14.3.4. Flow Copy to Meta.....	87
14.3.5. Flow Drop.....	88
14.3.6. Flow gRPC Counter.....	88
14.3.7. Flow Hairpin.....	89
14.3.8. Flow Hash Pipe.....	89
14.3.9. Flow LPM.....	90
14.3.10. Flow Modify Header.....	90
14.3.11. Flow Monitor Meter.....	91
14.3.12. Flow Multi-actions.....	91
14.3.13. Flow Multi-fwd.....	92
14.3.14. Flow Ordered List.....	92
14.3.15. Flow RSS Meta.....	93
14.3.16. Flow Set Meta.....	93
14.3.17. Flow Shared Counter.....	94
14.3.18. Flow Shared Meter.....	94
14.3.19. Flow Shared Mirror.....	95
14.3.20. Flow VXLAN Encap.....	96

Chapter 1. Introduction

DOCA Flow is the most fundamental API for building generic packet processing pipes in hardware.

The library provides an API for building a set of pipes, where each pipe consists of match criteria, monitoring, and a set of actions. Pipes can be chained so that after a pipe-defined action is executed, the packet may proceed to another pipe.

Using DOCA Flow API, it is easy to develop hardware-accelerated applications that have a match on up to two layers of packets (tunneled):

- ▶ MAC/VLAN/ETHERTYPE
- ▶ IPv4/IPv6
- ▶ TCP/UDP/ICMP
- ▶ GRE/VXLAN/GTP-U
- ▶ Metadata

The execution pipe may include packet modification actions:

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

The execution pipe may also have monitoring actions:

- ▶ Count
- ▶ Policers

The pipe also has a forwarding target which may be any of the following:

- ▶ Software (RSS to subset of queues)
- ▶ Port
- ▶ Another pipe

- ▶ Drop packets

This document is intended for software developers writing network function applications that focus on packet processing (e.g., gateways). The document assumes familiarity with network stack and DPDK.

Chapter 2. Prerequisites

A DOCA Flow-based application can run either on the host machine or on the NVIDIA® BlueField® DPU target. Since it is based on DPDK, Flow-based programs require an allocation of huge pages:

```
sudo echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
sudo mkdir /mnt/huge
sudo mount -t hugetlbfs nodev /mnt/huge
```

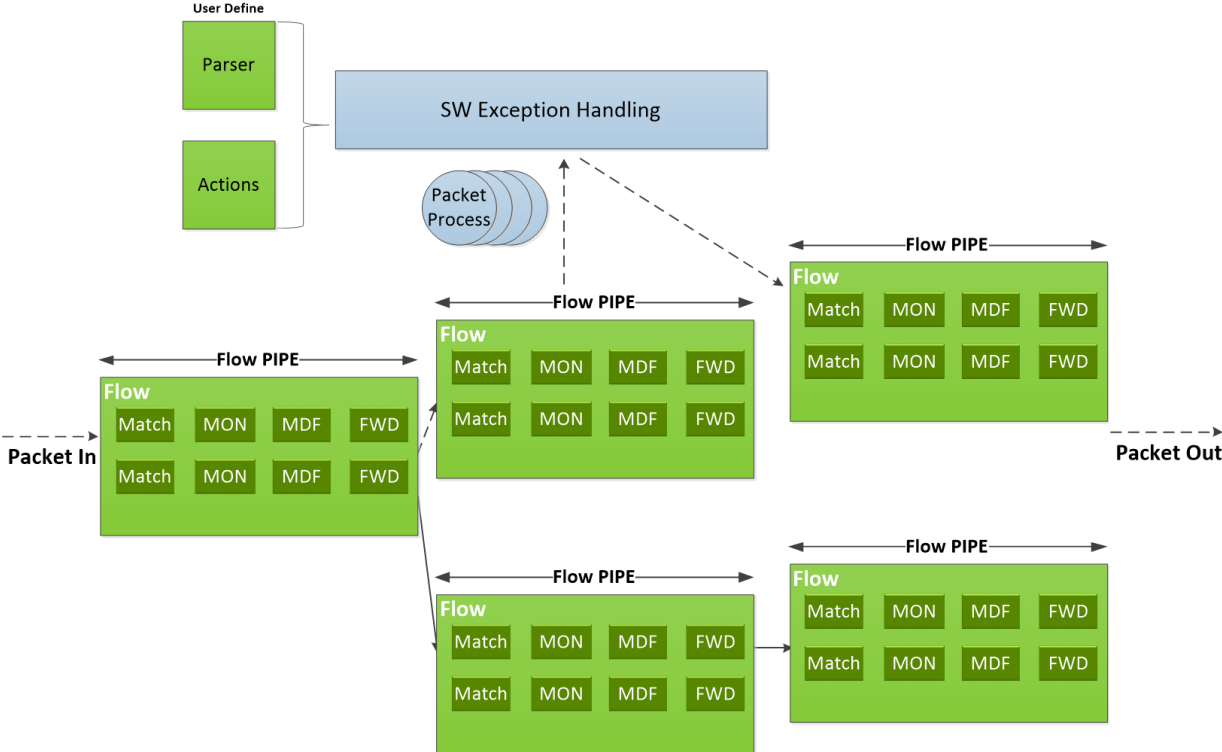
Chapter 3. Architecture

The following diagram shows how the DOCA Flow library defines a pipe template, receives a packet for processing, creates the pipe entry, and offloads the flow rule in NIC hardware.

- ▶ 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:

- ▶ MON: Monitor, can count, meter, or mirror
- ▶ MDF: Modify, can modify a field
- ▶ FWD: Forward to the next stage in packet processing



Chapter 4. Steering Domains

DOCA Flow organizes pipes into high-level containers named domains to address the specific needs of the underlying architecture.

A key element in defining a domain, is the packet direction and a set of allowed actions:

- ▶ A domain is a pipe attribute (also relates to shared objects)
- ▶ A domain restricts the set of allowed actions
- ▶ Transition between domains is well-defined (packets cannot cross domains arbitrarily)
- ▶ A domain may restrict the sharing of objects between packet directions
- ▶ Packet direction can restrict the move between domains

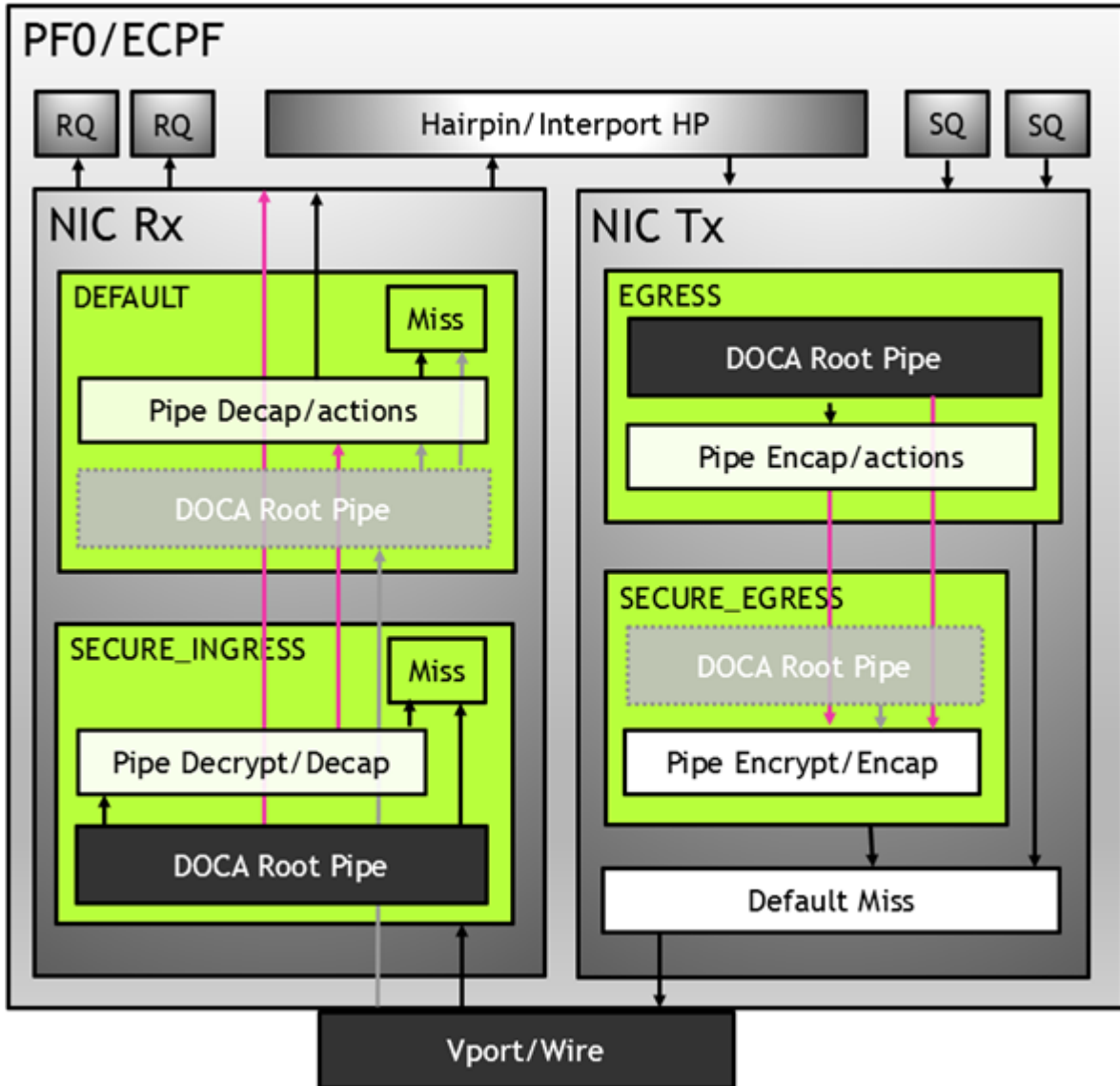
4.1. List of Steering Domains

DOCA Flow provides the following set of predefined steering domains:

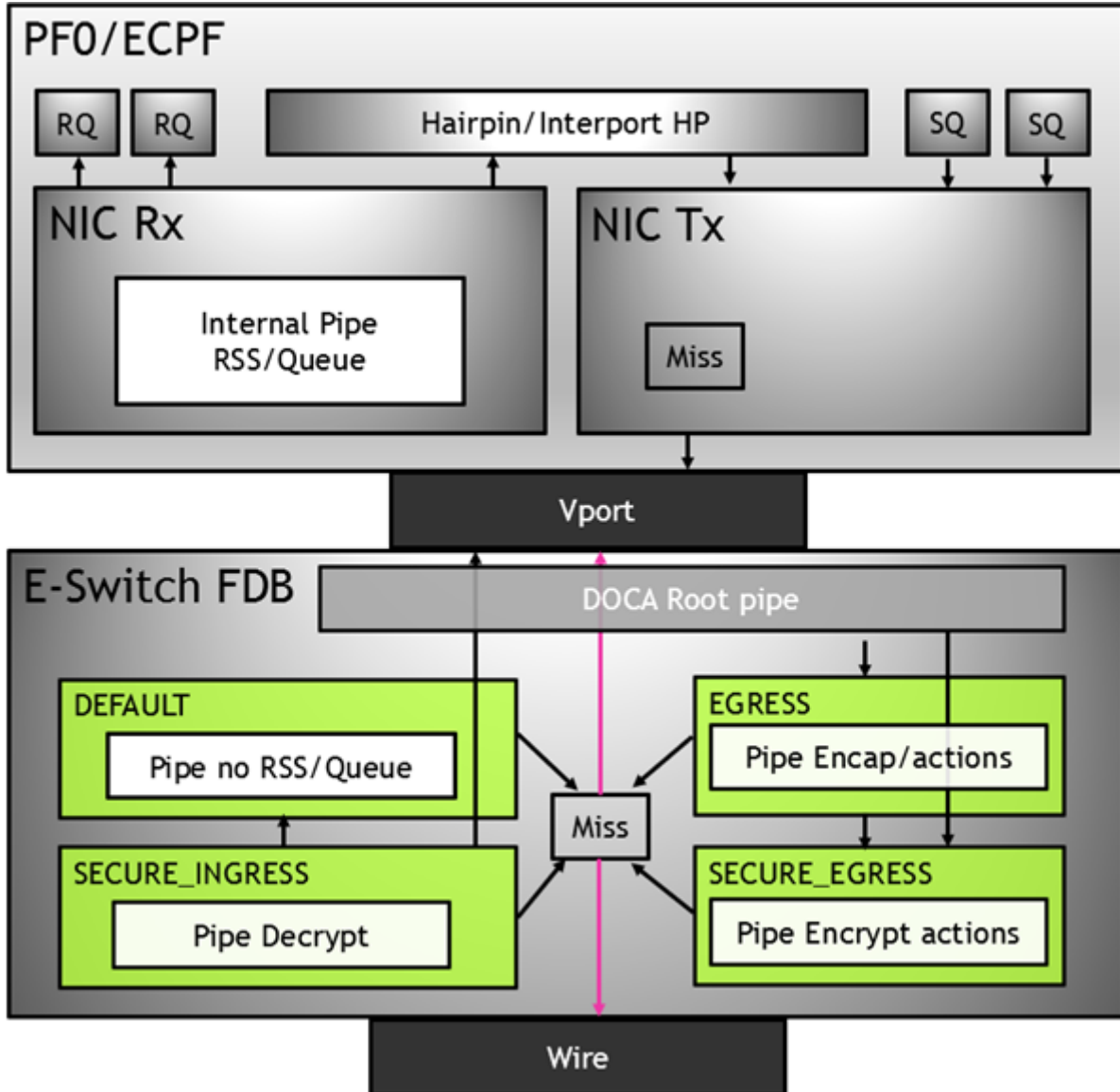
Domain	Description
<code>DOCA_FLOW_PIPE_DOMAIN_DEFAULT</code>	<ul style="list-style-type: none">▶ Default domain for actions on ingress traffic▶ Encapsulated and secure actions are not allowed here▶ The next milestone is queue or pipe in the EGRESS domain▶ Miss action is: Drop
<code>DOCA_FLOW_PIPE_DOMAIN_SECURE_INGRESS</code>	<ul style="list-style-type: none">▶ 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

Domain	Description
DOCA_FLOW_PIPE_DOMAIN_EGRESS	<ul style="list-style-type: none"> ▶ Miss action is: Drop ▶ 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	<ul style="list-style-type: none"> ▶ 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

4.2. Domains in VNF Mode



4.3. Domains in Switch Mode



Chapter 5. API

Refer to [NVIDIA DOCA Libraries API Reference Manual](#), for more detailed information on DOCA Flow API.



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.

5.1. `doca_flow_cfg`

This structure is required input for 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];
    uint32_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 counters to configure
- ▶ `uint32_t nb_meters` – number of traffic meters to configure

nr_acl_collisions

Number of collisions for the ACL module. Default value is 3. Maximum value is 8.

mode_args

Mandatory, set the DOCA Flow architecture [mode](#).

aging

Aging is handled by DOCA Flow while it is set to true. Default is false. See [Setting Pipe Monitoring](#) for information on the aging algorithm.

nr_shared_resources

Total shared resource per type. See section [Shared Counter Resource](#) 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

Number 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 value.

cb

Callback function for entry create/destroy.

unbind_cb

Callback to unbind a shared resource.

5.2. `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;
};
```

port_id

Port ID for the given type of port. For example, the following is a DPDK port ID for type DOCA_FLOW_PORT_DPDK_BY_ID.

type

Determined by the data plane in use.

- ▶ DOCA_FLOW_PORT_DPDK_BY_ID for DPDK dataplane.

devargs

String containing the exact configuration needed according to the type.



Note: For usage information of the type and devargs fields, refer to [Start Port](#).

priv_data_size

Per port, if this field is not set to zero, it means users want to define private data where application-specific information can be stored. See [#unique_18](#) for more information.

5.3. 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;
    uint8_t nb_actions;
    uint8_t nb_ordered_lists;
};
```

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.

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: Mutually exclusive with `nb_ordered_lists`.

nb_ordered_lists

Number of ordered lists in the array. Default is 0.



Note: Mutually exclusive with `nb_actions`.

```
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_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 and only for DOCA_FLOW_PIPE_BASIC, DOCA_FLOW_PIPE_CONTROL, DOCA_FLOW_PIPE_HASH, and DOCA_FLOW_PIPE_ORDERED_LIST.

actions

Actions array for the pipeline and only for DOCA_FLOW_PIPE_BASIC, DOCA_FLOW_PIPE_CONTROL, and DOCA_FLOW_PIPE_HASH.

action_descs

Action descriptions array and only for DOCA_FLOW_PIPE_BASIC, DOCA_FLOW_PIPE_CONTROL, and DOCA_FLOW_PIPE_HASH.

monitor

Monitor for the pipeline and only for DOCA_FLOW_PIPE_BASIC, DOCA_FLOW_PIPE_CONTROL, and DOCA_FLOW_PIPE_HASH.

ordered_lists

Array of ordered list types; only for DOCA_FLOW_PIPE_ORDERED_LIST.

5.4. doca_flow_meta

This is a maximum 20-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 description type:

AUTO

Set metadata value from an action of a specific entry. Pipe action is used as a mask.

CONSTANT

Set metadata value from a pipe action. Which is masked by the description mask.

SET

Set metadata value from an action of a specific entry which is masked by the description as a mask.

ADD

Set metadata scratch value from a pipe action or an action of a specific entry. Width is specified by the description.



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 port_meta; /**< Programmable source vport. */
uint32_t hash; /**< Hash calculation to copy. Only used by modify field action, not
matchable. */
uint32_t mark; /**< Mark id. */
    uint8_t ipsec_syndrome; /**< IPsec decrypt/authentication syndrome. */
uint8_t align[2]; /**< Structure alignment. */
};

```

pkt_meta

Metadata can be received along with 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

hash

Hash calculation to copy. Only used by `modify` field action and only as a source field; not matchable.

mark

Optional 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 taken into account 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]



Note: If `encap` action is used, `pkt_meta` should not be defined by the user as it is defined internally in DOCA to reference the encapsulated tunnel ID.

5.5. 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;
    };
};
```

eth

Ethernet header format including source and destination MAC address and the Ethernet layer type.

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 whether 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 L3.

icmp

ICMP header format.

udp

UDP header format.

tcp

TCP header format.

5.6. 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;
    };
};

```

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, and the next protocol

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.

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

Tunnel Type	Match ^(a)	Encap ^(b)	Decap ^(c)	Modify ^(d)	Copy ^(e)
DOCA_FLOW_TUN_#XLAN	#	#	#	#	#
DOCA_FLOW_TUN_#RE	#	#	#	#	#
DOCA_FLOW_TUN_#TP	#	#	#	#	#
DOCA_FLOW_TUN_#SP	#	#	#	#	#
DOCA_FLOW_TUN_#PLS_O_UDP	#	#	#	#	#
DOCA_FLOW_TUN_#ENEVE	#	#	#	#	#



Note: ^(a) Support for matching on this tunnel header, configured in the `tun` field inside `struct doca_flow_match`.

^(b) Support for encapsulation with this tunnel header, configured in the `tun` field inside `struct doca_flow_action_encap`.

^(c) Support for decapsulation with this tunnel header, configured in the `tun` field inside `struct doca_flow_match`.

^(d) Support for modifying this tunnel header, configured in the `tun` field inside `struct doca_flow_actions`.

^(e) Support for copying fields to/from this tunnel header, configured in the `tunnel` field inside `struct doca_flow_action_descs`.

5.7. 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_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.

outer

Outer packet header format.

tun

Tunnel info.

inner

Inner packet header format.

5.8. 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;
    struct {
        enum doca_flow_crypto_protocol_type proto_type;
        uint32_t crypto_id;
    } security;
};
```

action_idx

Index according to place provided on creation.

flags

Action flags.

decap

Decap while it is set to true.

pop

Pop header while it is set to true.

meta

Mask value if the description type is `AUTO`, specific value if the description type is `CONSTANT`.

outer

Modify outer header.

tun

Modify tunnel header.

has_encap

Encap while it is set to true.

encap

Encap data information.

has_push

Push header while it is set to true.

push

Push header data information.

security

Contains crypto action type and ID.

5.8.1. 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 actions types:

- ▶ `DOCA_FLOW_PUSH_ACTION_VLAN` – push VLAN.

5.9. `doca_flow_action_desc`

This structure is an action description.

```

struct doca_flow_action_desc {
    enum doca_flow_action_type type;
    union {
        union {
            uint32_t u32;
            uint64_t u64;
            uint8_t u8[16];
        } mask;
        struct {
            unit16_t doca_flow_action_field src;
            unit16_t doca_flow_action_field dst;
            unit16_t width;
        } copy;
        struct {
            struct doca_flow_action_field dst; /* destination info. */
            uint32_t width; /* Bit width to add */
        } add;
    };
};

```

type

Action type.

mask

Mask of modification action type `CONSTANT` and `SET`. Big-endian for network fields, host-endian for meta field.

copy

Field copy source and destination description.

add

Field add description. User can use the `dst` field to locate the destination field. Add always applies from field bit 0.

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

- ▶ `DOCA_FLOW_ACTION_AUTO` – modification type derived from pipe action
- ▶ `DOCA_FLOW_ACTION_CONSTANT` – modify action field with the constant value from pipe
- ▶ `DOCA_FLOW_ACTION_SET` – modify action field with the value of pipe entry
- ▶ `DOCA_FLOW_ACTION_ADD` – add field value. Supports meta scratch, `ipv4_ttl`, `ipv6_hop`, `tcp_seq`, and `tcp_ack`.

- ▶ DOCA_FLOW_ACTION_COPY – copy field

Refer to [Setting Pipe Actions](#) for more information.

5.10. doca_flow_monitor

This structure is a monitor configuration.

```
struct doca_flow_monitor {
    uint8_t flags;
    struct {
        uint64_t cir;
        uint64_t cbs;
    };
    uint32_t shared_meter_id;
    uint32_t shared_counter_id;
    uint32_t shared_mirror_id;
    uint32_t aging;
    void *user_data;
};
```

flags

Indicate actions to be included.

cir

Committed information rate in bytes per second. Defines maximum bandwidth.

cbs

Committed burst size in bytes. Defines maximum local burst size.

shared_meter_id

Meter ID that can be shared among multiple pipes.

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

Number of seconds from the last hit after which an entry is aged out.

user_data

Aging user data input.

The `flags` field includes the following monitor types:

- ▶ DOCA_FLOW_ACTION_METER – set monitor with meter action
- ▶ DOCA_FLOW_ACTION_COUNT – set monitor with counter action
- ▶ DOCA_FLOW_ACTION_AGING – set monitor with aging action
- ▶ DOCA_FLOW_ACTION_MIRROR – set monitor with mirror action

```
enum {
    DOCA_FLOW_MONITOR_NONE = 0,
    DOCA_FLOW_MONITOR_METER = (1 << 1),
    DOCA_FLOW_MONITOR_COUNT = (1 << 2),
    DOCA_FLOW_MONITOR_AGING = (1 << 3),
    DOCA_FLOW_MONITOR_MIRROR = (1 << 4),
};
```

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

$T(c)$ tokens are increased according to time, configured CIR, configured CBS, and packet arrival. When a packet is received, prior to anything else, the $t(c)$ tokens are filled. The number of tokens is a relative value that relies on the total time passed since the last update, but it is limited by the CBS value.

CIR is the maximum bandwidth at which packets continue being confirmed. Packets surpassing this bandwidth are dropped. CBS is the maximum 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 bit `DOCA_FLOW_MONITOR_METER` in flags
- ▶ For the shared use case, use a non-zero `shared_meter_id`

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

- ▶ For the individual use case, set bit `DOCA_FLOW_MONITOR_COUNT` in flags
- ▶ 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_meter_id` .



Note: Setting bit `DOCA_FLOW_MONITOR_MIRROR` in flags is not supported.

5.11. `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;
            uint32_t *rss_queues;
            int num_of_queues;
        };
        struct {
            uint16_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.

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

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`).

5.12. `doca_flow_query`

This struct is a flow query result.

```
struct doca_flow_query {
    uint64_t total_bytes;
```

```
uint64_t total_pkts;
};
```

total_bytes

Total bytes hit this flow.

total_pkts

Total packets hit this flow.

5.13. doca_flow_aged_query

This structure is an aged flow callback context.

```
struct doca_flow_aged_query {
    uint64_t user_data;
};
```

user_data

The user input context. Otherwise, the `doca_flow_pipe_entry` pointer is returned.

5.14. 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]

A 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 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.

5.15. 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]

A pointer to flow port config structure.

port [out]

A pointer to 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

5.16. 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: These 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]

A pointer to the DOCA Flow port structure.

pair_port [in]

A 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

5.17. 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]

A pointer to flow pipe config structure.

fwd [in]

A pointer to flow forward config structure.

fwd_miss [in]

A pointer to flow forward miss config structure. NULL for no `fwd_miss`. When creating a pipe, if there is a miss and `fwd_miss` is configured, then packet steering should jump to it.

pipe [out]

A 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

5.18. doca_flow_pipe_add_entry

This function add 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 performs 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]

A pointer to flow pipe.

match [in]

A pointer to flow match. Indicates specific packet match information.

actions [in]

A pointer to modify actions. Indicates specific modify information.

monitor [in]

A pointer to monitor actions.

fwd [in]

A 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 means that this entry waits to be pushed to hardware.

DOCA_FLOW_NO_WAIT means that this entry is pushed to hardware immediately.

usr_ctx [in]

A pointer to user context.

entry [out]

A 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

5.19. 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 performs 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,
                                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_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]

A pointer to flow pipe.

match [in]

A pointer to flow match. Indicates specific packet match information.

match_mask [in]

A pointer to flow match mask information.

actions [in]

A pointer to modify actions. Indicates specific modify information.

action_descs

A pointer to action descriptions.

monitor [in]

A pointer to monitor actions.

fwd [in]

A pointer to flow forward actions.

entry [out]

A 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

5.20. doca_flow_pipe_lpm_add_entry

This function adds a new entry to an LPM 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_lpm_add_entry(uint16_t pipe_queue,
                             uint8_t priority,
                             struct doca_flow_pipe *pipe,
                             const struct doca_flow_match *match,
                             const struct doca_flow_match *match_mask,
                             const struct doca_flow_fwd *fwd,
                             uint32_t flags,
                             void *usr_ctx,
                             struct doca_flow_pipe_entry **entry);

```

pipe_queue [in]

Queue identifier.

priority [in]

Priority value.

pipe [in]

A pointer to flow pipe.

match [in]

A pointer to flow match. Indicates specific packet match information.

match_mask [in]

A pointer to flow match mask information.

fwd [in]

A 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]

A pointer to user context.

entry [out]

A 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

5.21. 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]

A pointer to flow pipe.

match [in]

A pointer to flow match. Indicates specific packet match information.

match_mask [in]

A pointer to flow match mask information.

priority [in]

Priority value.

fwd [in]

A 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 tree, and pushes all entries to hardware immediately

usr_ctx [in]

A pointer to user context.

entry [out]

A 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

5.22. 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

performs 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]

A pointer to flow pipe.

idx [in]

A unique entry index. It is the user's responsibility to ensure uniqueness.

ordered_list [in]

A 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]

A 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]

A pointer to user context.

entry [out]

A pointer to pipe entry handler to fill.

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_DRIVER` – driver error

5.23. `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 performs 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]

A pointer to flow pipe.

entry_index [in]

A unique entry index. If the index is not unique, the function returns error.

action [in]

A pointer to modify actions. Indicates specific modify information.

monitor [in]

A pointer to monitor actions.

fwd [in]

A 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]

A pointer to user context.

entry [out]

A 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

5.24. 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.

```

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.

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

5.25. doca_flow_entry_status

This function get 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

5.26. 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]

A pointer to the flow pipe entry to query.

query_stats [out]

A 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

5.27. `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 `DOCA_FLOW_MONITOR_COUNT` flag 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]

A pointer to the flow pipe to query.

query_stats [out]

A 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

5.28. `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 entries array is filled with aged flows. 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 must 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,
struct doca_flow_aged_query *entries,
int len);
```

queue [in]

Queue identifier.

quota [in]

Max time quota in microseconds for this function to handle aging.

entries [in]

User input entry array for the aged flows.

len [in]

User input length of entries array.

Returns

- >0 – the number of aged flows filled in entries array.
- 0 – no aged entries in current call.
- 1 – full cycle is done.

5.29. 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

5.30. 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

Chapter 6. Shared Counter Resource

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

6.1. 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  
},
```

6.2. On `doca_flow_shared_resource_cfg()`

This call can be skipped for shared counters.

6.3. On `doca_flow_shared_resource_bind()`

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

```
int  
doca_flow_shared_resources_bind(enum doca_flow_shared_resource_type type, uint32_t  
    *res_array,  
                               uint32_t res_array_len, void *bindable_obj,  
                               struct doca_flow_error *error);
```

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_counters-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);
```

6.4. 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 to the pipe object in advance.

```
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.

6.5. 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. Must be allocated prior to calling this API.

The `type` parameter is `DOCA_FLOW_SHARED_RESOURCE_COUNT`.

6.6. On `doca_flow_pipe_destroy()` or `doca_flow_port_stop()`

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

Chapter 7. Shared Meter Resource

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

7.1. 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);
```

7.2. 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);
```

7.3. On `doca_flow_shared_resource_bind()`

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

```
int
doca_flow_shared_resources_bind(enum doca_flow_shared_resource_type type, uint32_t
*res_array,
                                uint32_t res_array_len, void *bindable_obj,
                                struct doca_flow_error *error);
```

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);
```

7.4. 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.

7.5. 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 Meter IDs](#).

7.6. On `doca_flow_pipe_destroy()` or `doca_flow_port_stop()`

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

Chapter 8. Shared RSS Resource

A shared RSS can be used in multiple pipe entries.

8.1. 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 of 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);
```

8.2. 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);
    }
```

8.3. 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, &error);
```

8.4. 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);
```

8.5. On `doca_flow_port_stop()`

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

Chapter 9. Shared Crypto Resource

A shared crypto resource can be used in multiple pipe entries and is intended to perform combinations of crypto offloads and crypto protocol header operations.

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

9.1. 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);
```

9.2. On `doca_flow_shared_resource_cfg()`

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

```
struct doca_flow_resource_crypto_cfg {
    enum doca_flow_crypto_protocol_type proto_type;
    /**< packet reformat action */
    enum doca_flow_crypto_action_type action_type;
    /**< crypto action */
    enum doca_flow_crypto_reformat_type reformat_type;
    /**< packet reformat action */
    enum doca_flow_crypto_net_type net_type;
    /**< packet network mode type */
    enum doca_flow_crypto_header_type header_type;
    /**< packet header type */
    uint16_t reformat_data_sz;
    /**< reformat header length in bytes */
    uint8_t reformat_data[DOCA_FLOW_CRYPTO_REFORMAT_LEN_MAX];
    /**< reformat header buffer */
    union {
        struct {
            uint16_t key_sz;
```

```

    /**< key size in bytes */
    uint8_t key[DOCA_FLOW_CRYPTO_KEY_LEN_MAX];
    /**< Crypto key buffer */
};
void* security_ctx;
/**< crypto object handle */
};
struct doca_flow_fwd fwd;
/**< Crypto action continuation */
};

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);

```

Example Action	Protocol	Reformat	Net	Header	Crypto
None	None	None	None	None	None
ESP tunnel decap only	ESP	Decap	Tunnel	None	None
ESP transport decap only	ESP	Decap	Transport	IPv4/6 + UDP	None
ESP tunnel encap only	ESP	Encap	Tunnel	None	None
ESP transport encap only	ESP	Encap	Transport	IPv4/6 + UDP	None
ESP decrypt only	ESP	None	None	None	Decrypt
ESP encrypt only	ESP	None	None	None	Encrypt
ESP tunnel decrypt + decap	ESP	Decap	Tunnel	None	Decrypt
ESP transport decrypt + decap	ESP	Decap	Transport	IPv4/6 + UDP	Decrypt
ESP tunnel encap + encrypt	ESP	Encap	Tunnel	None	Encrypt
ESP transport encap + encrypt	ESP	Encap	Transport	IPv4/6 + UDP	Encrypt

9.3. On `doca_flow_shared_resource_bind()`

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

```
int
doca_flow_shared_resources_bind(enum doca_flow_shared_resource_type type, uint32_t
*res_array,
                                uint32_t res_array_len, void *bindable_obj,
                                struct doca_flow_error *error);
```

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);
```

9.4. 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 and header reformat operations are performed over the packets matching the pipe entry according on the `crypto_id` configuration. Afterwards, the flow continues from the point specified in the forward part of the crypto action configuration. In pipe

configuration, the `crypto_id` can be changeable (all FFs) and then 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 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.

9.5. On `doca_flow_pipe_destroy()` or `doca_flow_port_stop()`

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

Chapter 10. 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.

10.1. 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);
```

10.2. 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);

```

10.3. On `doca_flow_shared_resource_bind()`

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

```

int
doca_flow_shared_resources_bind(enum doca_flow_shared_resource_type type, uint32_t
                               *res_array,
                               uint32_t res_array_len, void *bindable_obj,
                               struct doca_flow_error *error);

```

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 bind to 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);

```

10.4. 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 bind to a BASIC pipe.

Chapter 11. Flow Life Cycle

11.1. 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.

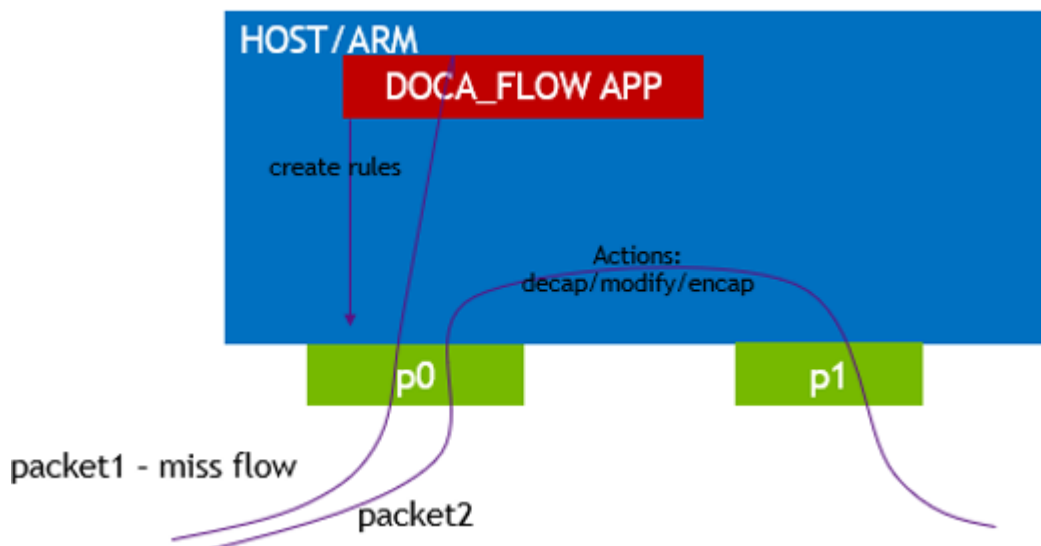
11.1.1. Pipe Mode

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.

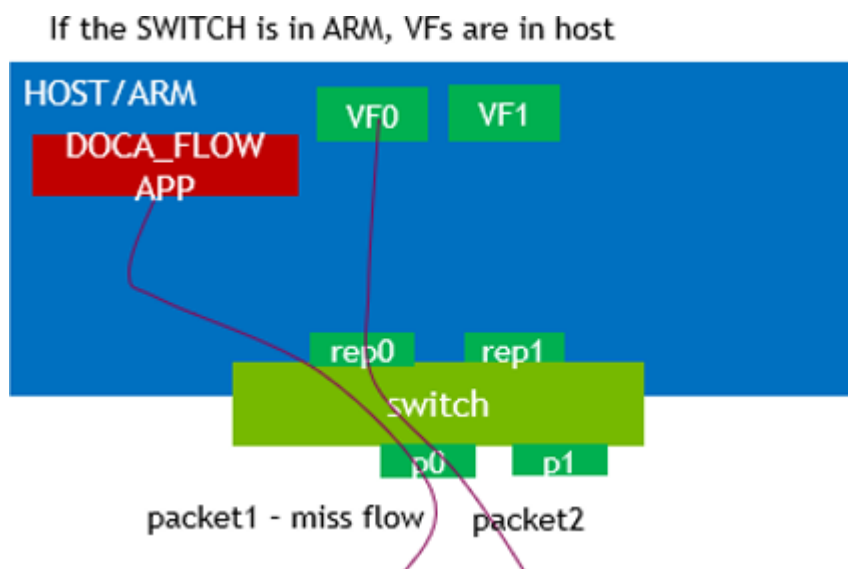
The following diagram shows the basic traffic flow in `vnf` mode. Packet1 firstly misses to host RSS queues. The app captures this packet and decides how to process it and then creates a pipe entry. Packet2 will hit this pipe entry and do the action, for example, for VXLAN, will do decap, modify, and 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.

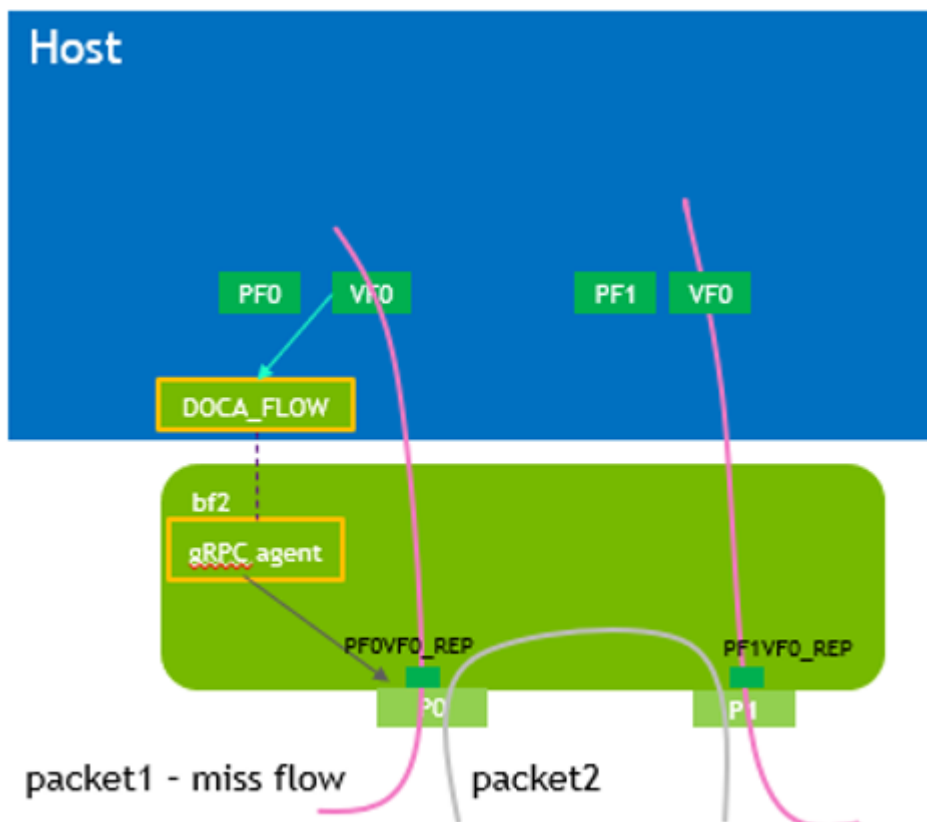
The following diagram shows the basic flow of traffic in `switch` mode. Packet 1 firstly misses to host RSS queues. The app captures this packet and decides which representor goes, and then sets the rule. Packets hit this rule and go to representor0.



► `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. Packet 1, 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.



11.2. Start Point

DOCA Flow API serves as an abstraction layer API for network acceleration. The packet processing in-network function is described from ingress to egress and, therefore, a pipe must be attached to the origin port. Once a packet arrives to the ingress port, it starts the hardware execution as defined by the DOCA API.

`doca_flow_port` is an opaque object since the DOCA Flow API is not bound to a specific packet delivery API, such as DPDK. The first step is to start the DOCA Flow port by calling `doca_flow_port_start()`. The purpose of this step is to attach user application ports to the DOCA Flow ports.

When DPDK is used, the following configuration must be provided:

```
enum doca_flow_port_type type = DOCA_FLOW_PORT_DPDK_BY_ID;
const char *devargs = "1";
```

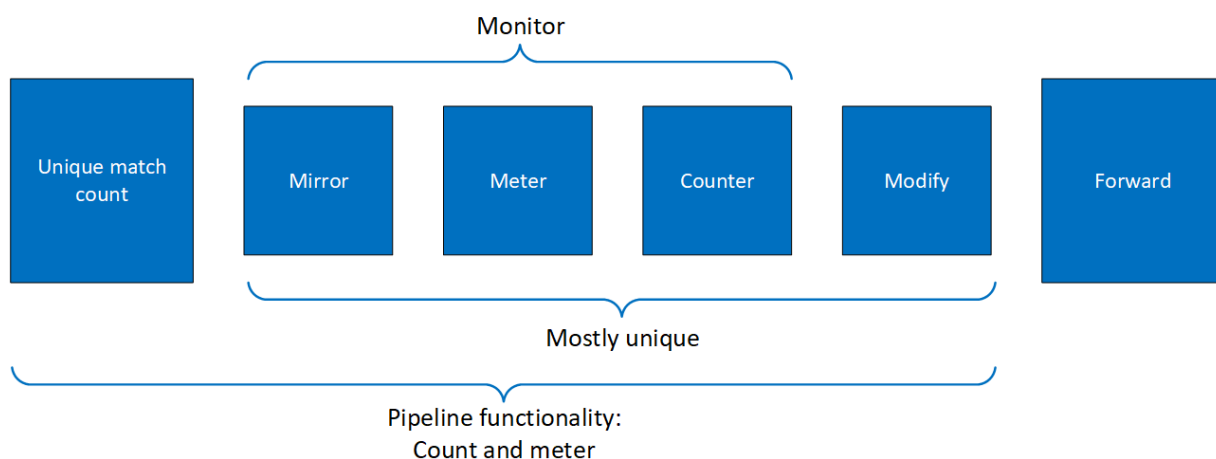
The `devargs` parameter points to a string that has the numeric value of the DPDK `port_id` in decimal format. The port must be configured and started before calling this API. Mapping the DPDK port to the DOCA port is required to synchronize application ports with hardware ports.

11.3. 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).

11.3.1. 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 choose whether the field is:

- ▶ Wildcard (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:

- ▶ P stands for the packet field value
- ▶ V stands for the pipe match field value
- ▶ M stands for the pipe mask field value
- ▶ K stands for the match entry field value

11.3.1.1. Implicit Match

Match Type	Pipe Match Value	Pipe Match Mask Value	Entry Value	Meaning
Wildcard (match any)	0	N/A, NULL pointer	N/A	A match occurs for any value of P
Constant	N, where $0 < N < 0xffffffff$	N/A, NULL pointer	N/A	A match occurs if $P=V$
Variable (per entry)	Full mask (0xffffffff)	N/A, NULL pointer	Per-entry (K)	A match occurs if $P=K$

To match implicitly, the following 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

```
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;
}
```

11.3.1.2. Explicit Match

Match Type	Pipe Match Value	Pipe Match Mask Value	Entry Value	Meaning
Wildcard (match any)	0	0	N/A	A match occurs for any value of P
Constant	V, where $0 \leq V < 0xffffffff$	M, where $0 < M \leq 0xffffffff$	N/A	A match occurs if P and M equal V
Variable (per entry)	Full mask (0xffffffff)	M, where $0 < M \leq 0xffffffff$	Per-entry value (K)	A match occurs if P and M equal K

In this case, there are two `doca_flow_match` items, the following considerations should be taken into account:

- ▶ Ignored fields

- ▶ Field is zeroed
- ▶ Pipeline has no comparison on the field

```
match_mask.inner.ip4.dst_ip = 0;
```

▶ Constant fields

These are fields that have a constant value. For example, as shown in the following, the inner 5-tuple match on IPv4 destination addresses belonging to the 0.0.0.0/24 subnet, and this match is constant among all entries:

```
// BE_IPV4_ADDR converts 4 numbers A,B,C,D to a big endian representation of IP
address A.B.C.D
match.inner.ip4.dst_ip = 0;
match_mask.inner.ip4.dst_ip = BE_IPV4_ADDR(255, 255, 255, 0);
```

For example, as shown in the following, the inner 5-tuple match on IPv4 destination addresses belonging to the 1.2.0.0/16 subnet, and this match is constant among all entries. The last two octets of the `match.inner.ip4.dst_ip` are ignored because the `match_mask` of 255.255.0.0 is applied:

```
// BE_IPV4_ADDR converts 4 numbers A,B,C,D to a big endian representation of IP
address A.B.C.D
match.inner.ip4.dst_ip = BE_IPV4_ADDR(1, 2, 3, 4);
match_mask.inner.ip4.dst_ip = BE_IPV4_ADDR(255, 255, 0, 0);
```

Once a field is defined as constant, the field's value cannot be changed per entry. Users must set constant fields to zero when adding entries to avoid ambiguity.

▶ Changeable fields

These are fields that may change per entry, and the mask is applied to every value.

For example, as shown in the following, match on destination IPv4 address of variable per-entry value (inner 5-tuple) with a 255.255.0.0 mask 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 = 0xffffffff;
match_mask.inner.ip4.dst_ip = BE_IPV4_ADDR(255, 255, 0, 0);
```

Note that for IPs, the prefix mask can be used as well.

11.3.2. Setting Pipe Actions

11.3.2.1. 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

```
match_mask.in_dst_ip.ipv4_addr = 0xffffffff;
```

Metadata is considered as per-packet changeable fields, pipe action is used as a mask.

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

For example:

```
static void
create_decap_inner_modify_actions(struct doca_flow_actions *actions)
{
    actions->decap = true;
    actions->mod_dst_ip.ipv4_addr = 0xffffffff;
}
```

11.3.2.2. Explicit Modification Type

It is possible to force constant modification or per-entry modification with action description type (`CONSTANT` or `SET`) and mask. For example:

```
static void
create_constant_modify_actions(struct doca_flow_actions *actions,
                              struct doca_flow_action_descs *descs)
{
    actions->mod_src_port = 0x1234;
    descs->src_port.type = DOCA_FLOW_ACTION_CONSTANT;
    descs->outer.src_port.mask.u64 = 0xffff;
}
```

11.3.2.3. Copy Field

Action description can be used to copy between packet field and metadata. For example:

```
static void
create_copy_packet_to_meta_actions(struct doca_flow_match *match,
                                  struct doca_flow_action_descs *descs)
{
    descs->src_ip.type = DOCA_FLOW_ACTION_COPY;
    descs->src_ip.copy.dst = &match->meta.u32[1];
}
```

11.3.2.4. 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;
}
```

11.3.2.5. Summary of Action Types

Pipe Creation		Entry Creation	
action_desc		Pipe Actions	Entry Actions
doca_flow_action_type	Configuration		
DOCA_FLOW_ACTION_AUTO	No specific config	0 – field ignored, no modification	N/A
Derived from pipe actions.		val != 0 – apply this val to all entries	N/A
		val = 0xffff – changeable field	Define val per entry
		Specific for Metadata - the meta field in the actions is used as a mask.	Define val per entry
DOCA_FLOW_ACTION_CONSTANT	Define the mask	Define val to apply for all entries	N/A
Pipe action is constant.			
DOCA_FLOW_ACTION_SET	Define the mask	N/A	Define val per entry
Set value from entry action.			
DOCA_FLOW_ACTION_ADD	Define the dst field and width	val != 0 – apply this val to all entries	Define val per entry
Add field value.			
DOCA_FLOW_ACTION_COPY	Define the source and destination fields.	N/A	N/A
Copy field to another field.	<ul style="list-style-type: none"> ▶ Meta field → header field ▶ Header field → meta field ▶ Meta field → meta field 		

11.3.2.6. 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]

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

11.3.3. 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 is to be used, the 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. User data is used to map user usage. If the `user_data` field is set, when the entry ages out, query API returns this `user_data`. If `user_data` is not configured by the application, the aged pipe entry handle is returned.

For example:

```
static void build_entry_monitor(struct doca_flow_monitor *monitor, void *user_ctx)
{
    monitor->flags |= DOCA_FLOW_MONITOR_AGING;
    monitor->aging = 10;
    monitor->user_data = (uint64_t)user_ctx;
}
```

Refer to [Pipe Entry Aged Query](#) for more information.

11.3.4. 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;
```

11.3.5. 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 Libraries API Reference Manual](#) 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

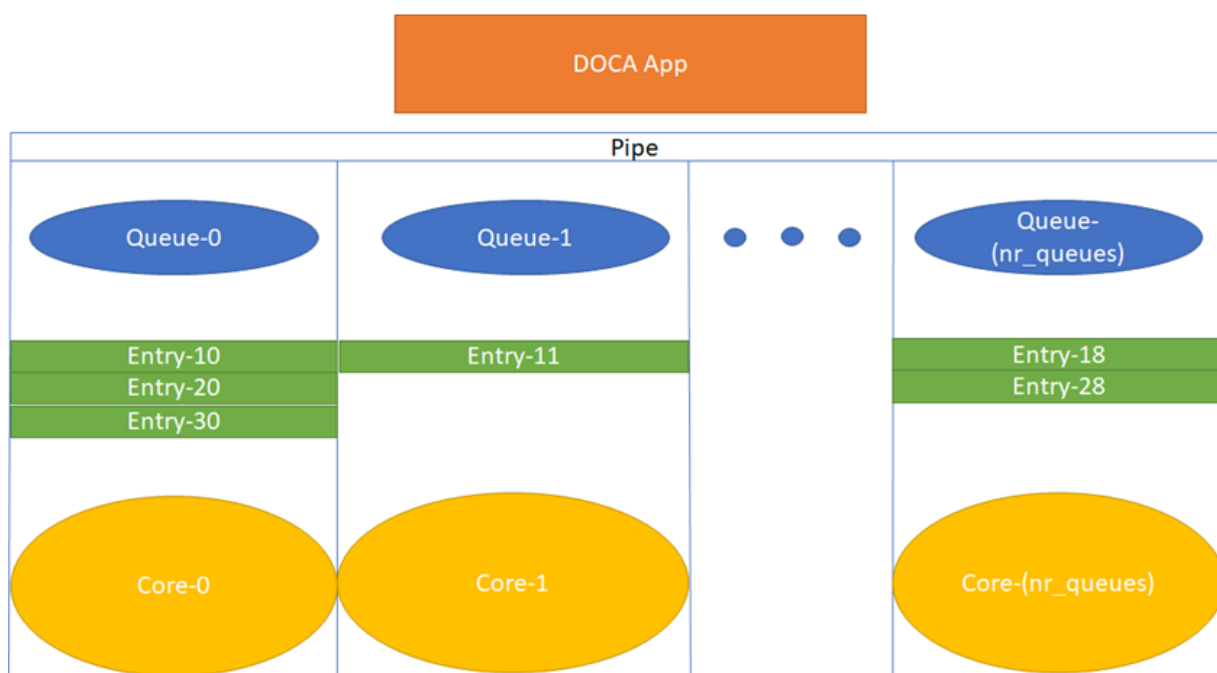
overlap, the priority is set according to the order of entries added. So, if two root pipes have overlapping matching and PIPE1 has higher priority than PIPE2, users should add an entry to PIPE1 after all entries are added to PIPE2.

11.3.6. 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);
```

11.3.6.1. 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.

When a counter is present, it is possible to query the flow and get the counter's data by calling `doca_flow_query`.

The retrieved statistics are stored in struct `doca_flow_query`.

11.3.6.2. Pipe Entry Aged Query

When a user calls `doca_flow_aged_query()`, this query is used to get the aged-out entries by the time quota in microseconds. The entry handle or the `user_data` input is returned by this API.

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.

The struct `doca_flow_aged_query` contains the element `user_data` which contains the aged-out flow contexts.

11.3.7. 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.

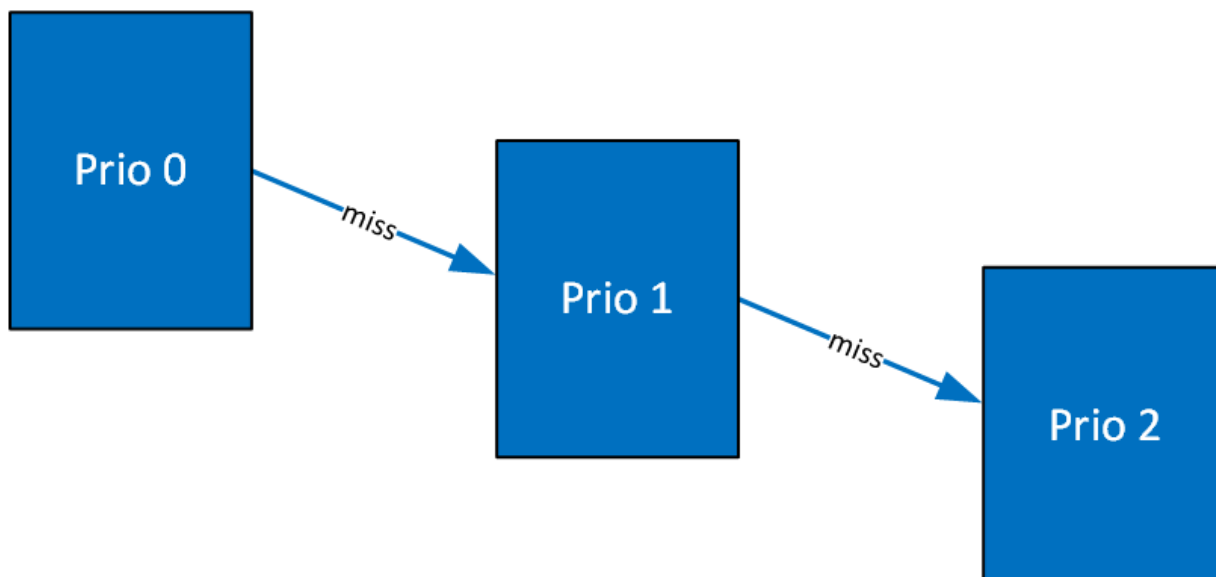
11.3.8. Miss Pipe and Control Pipe



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 VXLAN packets with `ipB` hit the control entry.

11.3.9. `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`.

11.3.10.doca_flow_pipe_acl

`doca_flow_pipe_acl` uses access-control list (ACL) matching. ACL matching is five tuple of the `doca_flow_match`. Each entry consists of a value and a mask (e.g., 10.0.0.0/8, 10.10.0.0/16, etc.) for IP address fields, port range, or specific port in the port fields, protocol, and priority of the entry.

ACL entry port configuration:

- ▶ Mask port is 0 ==> Any port
- ▶ Mask port is equal to match port ==> Exact port. Port with mask 0xffff.
- ▶ Mask port > match port ==> Match port is used as port from and mask port is used as port to

Monitor actions are not supported in ACL. FWD of the DOCA Flow ACL pipe works the same as the basic DOCA Flow pipe.

ACL supports the following types of FWD:

- ▶ DOCA_FLOW_FWD_PORT
- ▶ DOCA_FLOW_FWD_PIPE
- ▶ DOCA_FLOW_FWD_DROP

`doca_flow_pipe_lpm` insertion max latency can be measured in milliseconds in some cases and, therefore, it is better to insert it from the control path. To get the best insertion performance, entries should be added in large batches.



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.

11.3.11.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.

11.3.12. `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.

11.3.13. 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"`, `"remote_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.

11.3.14. Isolated Mode

In non-isolated mode (default) any received packets (e.g., following an RSS forward) 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 useful.

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 pipe 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 = ...
        ...
    },
    ...
};
```

11.4. Teardown

11.4.1. Pipe Entry Teardown

When an entry is terminated by the user application or ages-out, the user should call the entry destroy function, `doca_flow_pipe_rm_entry()`. This frees the pipe entry and cancels hardware offload.

11.4.2. Pipe Teardown

When a pipe is terminated by the user application, the user should call the pipe destroy function, `doca_flow_pipe_destroy()`. This destroys the pipe and the pipe entries that match it.

When all pipes of a port are terminated by the user application, the user should call the pipe flush function, `doca_flow_port_pipes_flush()`. This destroys all pipes and all pipe entries belonging to this port.

11.4.3. Port Teardown

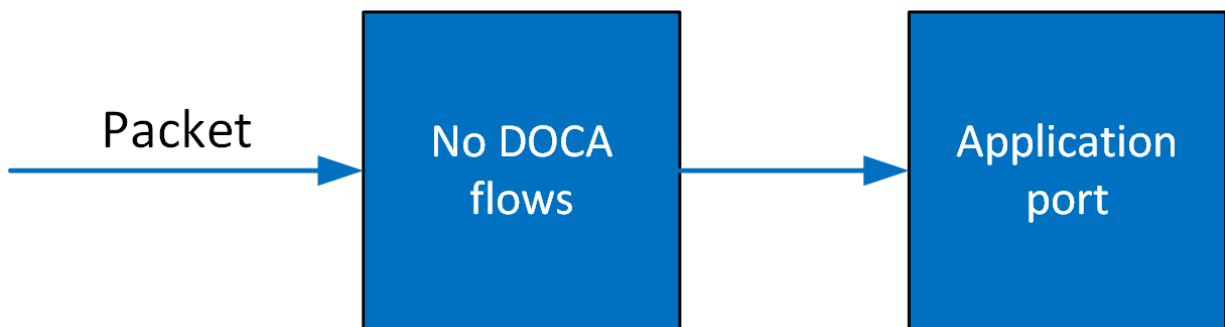
When the port is not used anymore, the user should call the port stop function, `doca_flow_port_stop()`. This stops the DOCA port, disables the traffic, destroys the port and frees all resources of the port.

11.4.4. Flow Teardown

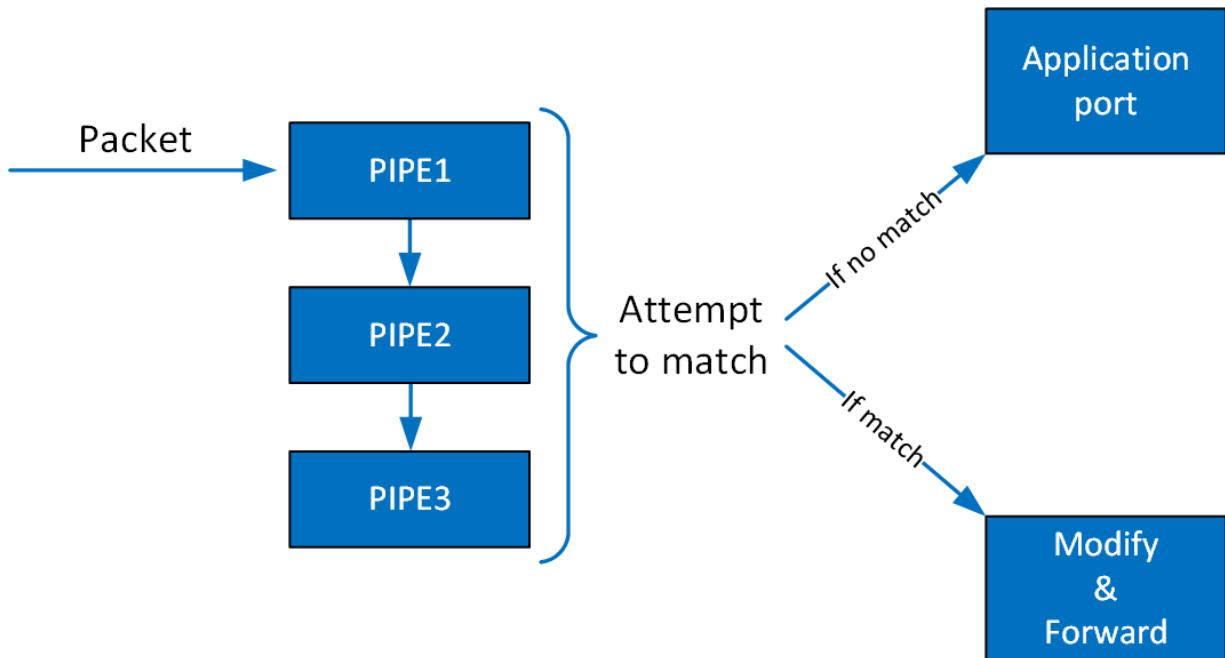
When the DOCA Flow is not used anymore, the user should call the flow destroy function, `doca_flow_destroy()`. This releases all the resources used by DOCA Flow.

Chapter 12. 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.

Chapter 13. 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:

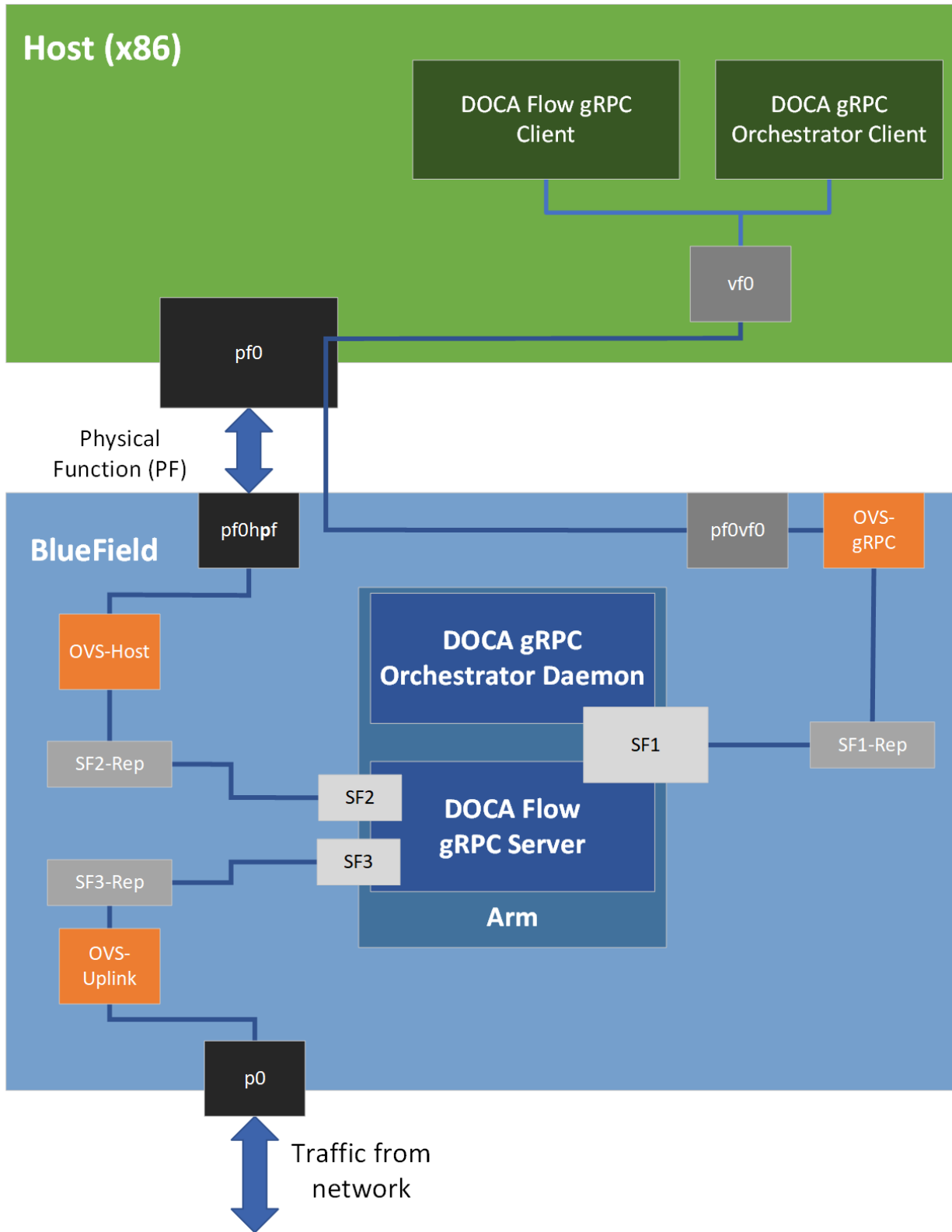
- ▶ proto-buff – this section details the messages defined in the proto-buff
- ▶ Client interface – this section details the API for communicating with the server
- ▶ Usage – this section explains how to use the client interface to develop your own client application based on DOCA Flow gRPC support

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.



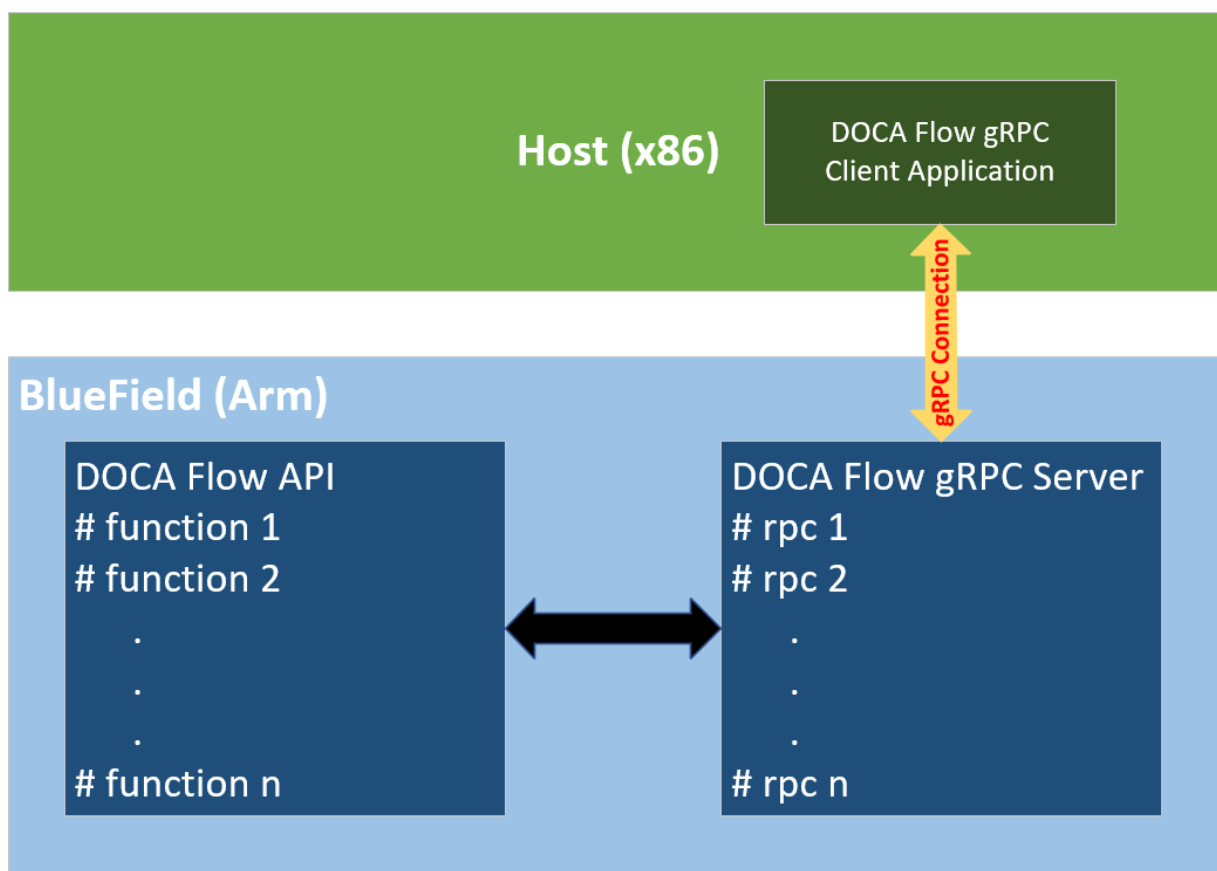
13.1. Proto-Buff

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`.

13.1.1. 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;
}
```

13.1.2. DocaFlowCfg

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

13.1.3. DocaFlowPortCfg

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

13.1.4. DocaFlowPipeCfg

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

13.1.5. 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.

13.1.6. 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 represents `doca_flow_ip_address` and `doca_flow_tun` respectively.

13.1.7. DocaFlowActions

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

13.1.8. DocaFlowActionDesc

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

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

13.1.9. DocaFlowMonitor

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

13.1.10. DocaFlowFwd

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

13.1.11. DocaFlowQueryStats

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

13.1.12. DocaFlowHandleAgingRes

The `DocaFlowHandleAgingRes` message contains all the parameters needed to save the result of an aging handler.

13.1.13. 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.

13.1.14. 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.

13.1.15. DocaFlowPortPair

The `DocaFlowPortPairRequest` message contains all the necessary information for port pairing:

```
message DocaFlowPortPairRequest {
    uint32 port_id = 1;          /* port identifier of doca flow port. */
    uint32 pair_port_id = 2;    /* port identifier 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.

13.1.16. 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.

13.1.17. 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.

13.1.18. DocaFlowPipeControlAddEntry

The `DocaFlowPipeControlAddEntryRequest` message contains the required arguments for adding entries to the control pipe:

```
message DocaFlowPipeControlAddEntryRequest{
    uint32 priority = 2;               /* the 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.

13.1.19. 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.

13.1.20. DocaFlowEntriesProcess

The `DocaFlowEntriesProcessRequest` 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.

13.1.21. DocaFlowEntryGetStatus

The `DocaFlowEntryGetStatusRequest` 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.

13.1.22. DocaFlowQuery

`DocaFlowQueryRequest` 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 is an enum defined in the proto-buff, and represents the `doca_flow_query` struct.

13.1.23. DocaFlowAgingHandle

`DocaFlowAgingHandleRequest` 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. */
    uint64 user_data = 4;        /* the user input context, otherwise the
doca_flow_pipe_entry pointer */
    uint32 len = 5;              /* the user input length of entries array. */
}
```

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` field is populated with the aging handler result. This field's type is `DocaFlowHandleAgingRes`.

13.1.24. DocaFlowSharedResourceCfg

The `DocaFlowSharedResourceCfgRequest` 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`.

13.1.25. DocaFlowSharedResourcesBind

The `DocaFlowSharedResourcesBindRequest` 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`.

13.1.26. DocaFlowSharedResourcesQuery

The `DocaFlowSharedResourcesQueryRequest` 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.

13.2. 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 Libraries API Reference Manual](#) 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.

13.2.1. `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.

13.2.2. 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.

13.2.3. 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.

13.2.4. 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.

13.3. 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.

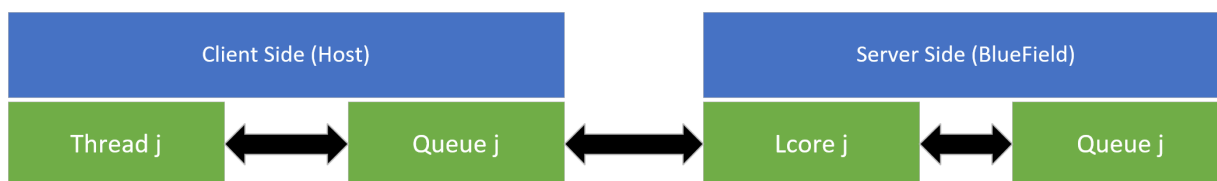
The following phases demonstrate a basic flow of client applications:

- ▶ 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:



Chapter 14. DOCA Flow Samples

This section provides DOCA Flow sample implementation on top of the BlueField DPU.

14.1. 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
```


14.2. Running the Sample

1. Refer to the following documents:

- ▶ [NVIDIA DOCA Installation Guide for Linux](#) for details on how to install BlueField-related software.
- ▶ [NVIDIA DOCA Troubleshooting Guide](#) for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample:

```
cd /opt/mellanox/doca/samples/doca_flow/<sample_name>
meson build
ninja -C build
```

 **Note:** The binary `doca_<sample_name>` will be created under `./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 log level for the program <CRITICAL=20,
ERROR=30, WARNING=40, INFO=50, DEBUG=60>
```

4. For additional information per sample, use the `-h` option after the `--` separator:

```
./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:

```
./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:

```
./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
```



Note: When running on the DPU using the command above, sub-functions must be enabled according to the [Scalable Function Setup Guide](#).



Note: When running on the host, virtual functions must be used according to the instructions in the [NVIDIA DOCA Virtual Functions User Guide](#).

14.3. Samples

14.3.1. Flow ACL

This sample illustrates how to use the access-control list (ACL) pipe.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
 - a). Building an ACL pipe that matches changeable:
 - i. Source IPv4 address
 - ii. Destination IPv4 address
 - iii. Source port
 - iv. Destination port
 - b). Adding four example 5-tuple entries:
 - i. The first entry with:
 - ▶ Full mask on source IPv4 address
 - ▶ Full mask on destination IPv4 address
 - ▶ Null mask on source port (any source port)
 - ▶ Null mask on destination port (any destination port)
 - ▶ TCP protocol
 - ▶ Priority 10
 - ▶ Action "deny" (drop action)
 - ii. The second entry with:
 - ▶ Full mask on source IPv4 address
 - ▶ Full mask on destination IPv4 address

- ▶ Null mask on source port (any source port)
 - ▶ Value set in mask on destination port is used as part of port range:
 - ▶ Destination port in match is used as port from
 - ▶ Destination port in mask is used as port to
 - ▶ UDP protocol
 - ▶ Priority 50
 - ▶ Action "allow" (forward port action)
- iii. The third entry with:
- ▶ Full mask on source IPv4 address
 - ▶ Full mask on destination IPv4 address
 - ▶ Value set in mask on source port is equal to the source port in match. It is the exact port. ACL uses the port with full mask.
 - ▶ Null mask on destination port (any destination port)
 - ▶ TCP protocol
 - ▶ Priority 40
 - ▶ Action "allow" (forward port action)
- iv. The fourth entry with:
- ▶ 24-bit mask on source IPv4 address
 - ▶ 24-bit mask on destination IPv4 address
 - ▶ Value set in mask on source port is used as part of port range : source port in match is used as port from, source port in mask is used as port to.
 - ▶ Value set in mask on destination port is equal to the destination port in match. It is the exact port. ACL uses the port with full mask.
 - ▶ TCP protocol
 - ▶ Priority 20
 - ▶ Action "allow" (forward port action)
- c). The sample shows how to run the ACL pipe on ingress and egress domains. To change the domain use the global parameter `flow_acl_sample.c`.
- i. Ingress domain: ACL is created as root pipe
 - ii. Egress domain:
 - ▶ Building a control pipe with one entry that forwards the IPv4 traffic hairpin port.
 - ▶ ACL is created as a root pipe on the hairpin port.

Reference:

- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_acl/flow_acl_sample.c`

- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_acl/flow_acl_main.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_acl/meson.build`

14.3.2. Flow Aging

This sample illustrates the use of DOCA Flow's aging functionality. It demonstrates how to build a pipe and add different entries with different aging times and user data.

The sample logic includes:

1. Initializing DOCA Flow with `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow port.
3. On each port:
 - a). Building a pipe with changeable 5-tuple match and forward port action.
 - b). Adding 10 entries with different 5-tuple match, a monitor with different aging time (5-60 seconds), and setting user data in the monitor. The user data will contain the port ID, entry number, and entry pointer.
4. Handling aging every 5 seconds and removing each entry after age-out.
5. Running these commands until all entries age out.

Reference:

- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_aging/flow_aging_sample.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_aging/flow_aging_main.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_aging/meson.build`

14.3.3. Flow Control Pipe

This sample shows how to use the DOCA Flow control pipe and decap action.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
 - a). Building VXLAN pipe with match on VNI field, decap action, and forwarding the matched packets to the second port.
 - b). Building GRE pipe with match on GRE key field, decap and build `eth` header actions, and forwarding the matched packets to the second port.
 - c). Building MPLS pipe with match on third MPLS label field, decap and build `eth` header actions, and forwarding the matched packets to the second port.
 - d). 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 either MPLS or GRE tunnels are decapped, the complete outer layer is removed (including L2) and the inner layer (IP layer) is exposed. To keep the packet valid, you must also modify the layer 2 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;
```

The same requirement applies to GTP tunnels or any other tunnel protocol which lacks an L2 layer.

However, for VXLAN it is not necessary to add an L2 since VXLAN itself already includes the L2, so the packet remains valid even after it is decapped.

Reference:

- ▶ /opt/mellanox/doca/samples/doca_flow/flow_control_pipe/flow_control_pipe_sample.c
- ▶ /opt/mellanox/doca/samples/doca_flow/flow_control_pipe/flow_control_pipe_main.c
- ▶ /opt/mellanox/doca/samples/doca_flow/flow_control_pipe/meson.build

14.3.4. Flow Copy to Meta

This sample shows how to use the DOCA Flow copy-to-metadata action to copy the source MAC address and then match on it.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
 - a). Building a pipe with changeable match on `meta_data` and forwarding the matched packets to the second port.
 - b). Adding an entry that matches an example source MAC that has been copied to metadata.
 - c). Building a pipe with changeable 5-tuple match, copying source MAC action, and fwd to the first pipe.
 - d). Adding example 5-tuple entry to the pipe.

Reference:

- ▶ /opt/mellanox/doca/samples/doca_flow/flow_copy_to_meta/flow_copy_to_meta_sample.c

- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_copy_to_meta/flow_copy_to_meta_main.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_copy_to_meta/meson.build`

14.3.5. 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:
 - a). Building a hairpin pipe with an entry that matches all traffic and forwarding traffic to the second port.
 - b). Building a pipe with a changeable 5-tuple match, forwarding action drop and miss forward to the hairpin pipe. This pipe serves as a root pipe.
 - c). Adding 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_sample.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_drop/flow_drop_sample_main.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_drop/meson.build`

14.3.6. 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:
 - a). Building a pipe with changeable 5-tuple match.
 - b). Adding example 5-tuple and monitoring with counter flag.
 - c). Waiting 5 seconds and querying the entries (total bytes and total packets).

References:

- ▶ `/opt/mellanox/doca/samples/doca_flow/grpc_flow_counter/grpc_flow_counter_sample.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/grpc_flow_counter/grpc_flow_counter_main.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/grpc_flow_counter/meson.build`

14.3.7. Flow Hairpin

This sample illustrates how to build a pipe with 5-tuple match and to forward packets to the other port.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
 - a). Building a pipe with changeable 5-tuple match and forwarding port action.
 - b). Adding example 5-tuple entry to the pipe.

Reference:

- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_hairpin/flow_hairpin_sample.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_hairpin/flow_hairpin_main.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_hairpin/meson.build`

14.3.8. Flow Hash Pipe

This sample illustrates how to build a hash pipe in hardware steering (HWS) mode.

The hash pipe contains two entries, each of which forwards "matched" packets to two different SF representors. For each received packet, the hash pipe calculates the entry index to use based on the IPv4 destination address.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct.
2. Starting DOCA Flow ports: Physical port and two SF representors.
3. On switch port:
 - a). Building a hash pipe while indicating which fields to use to calculate the hash in the struct `match_mask`.
 - b). Adding two entries to the created pipe, each of which forwards packets to a different port representor.

Reference:

- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_hairpin/flow_hash_pipe_sample.c`

- ▶ /opt/mellanox/doca/samples/doca_flow/flow_hairpin/flow_hash_pipe_main.c
- ▶ /opt/mellanox/doca/samples/doca_flow/flow_hairpin/meson.build

14.3.9. 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:
 - a). Building an LPM pipe that matches changeable source IPv4 address.
 - b). Adding two example 5-tuple entries:
 - ▶ The first entry with full mask and forward port action
 - ▶ The second entry with 16-bit mask and drop action
 - c). Building a control pipe with one entry that forwards IPv4 traffic to the LPM pipe.

Reference:

- ▶ /opt/mellanox/doca/samples/doca_flow/flow_lpm/flow_lpm_sample.c
- ▶ /opt/mellanox/doca/samples/doca_flow/flow_lpm/flow_lpm_main.c
- ▶ /opt/mellanox/doca/samples/doca_flow/flow_lpm/meson.build

14.3.10. 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:
 - a). Building a pipe with action `dec_ttl=true` and changeable `mod_dst_mac`. The pipe matches IPv4 traffic with a changeable destination IP and forwards the matched packets to the second port.
 - b). Adding an entry with an example destination IP and `mod_dst_mac` value.

Reference:

- ▶ /opt/mellanox/doca/samples/doca_flow/flow_modify_header/flow_modify_header_sample.c
- ▶ /opt/mellanox/doca/samples/doca_flow/flow_modify_header/flow_modify_header_main.c

▶ /opt/mellanox/doca/samples/doca_flow/flow_modify_header/meson.build

14.3.11. Flow Monitor Meter

This sample illustrates how to use DOCA Flow monitor meter.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
 - a). Building a pipe with monitor meter flag and changeable 5-tuple match. The pipe forwards the matched packets to the second port.
 - b). Adding an entry with an example CIR and CBS values.

Reference:

- ▶ /opt/mellanox/doca/samples/doca_flow/flow_monitor_meter/flow_monitor_meter_sample.c
- ▶ /opt/mellanox/doca/samples/doca_flow/flow_monitor_meter/flow_monitor_meter_main.c
- ▶ /opt/mellanox/doca/samples/doca_flow/flow_monitor_meter/meson.build

14.3.12. Flow Multi-actions

This sample shows how to use a DOCA Flow array of actions in a pipe.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
 - a). Building a pipe with changeable source IP match which forwards the matched packets to the second port and sets different actions in the actions array:
 - i. Changeable modify source MAC address
 - ii. Changeable modify source IP address
 - b). Adding two entries to the pipe with different source IP match:
 - i. The first entry with an example modify source MAC address.
 - ii. The second with a modify source IP address.

Reference:

- ▶ /opt/mellanox/doca/samples/doca_flow/flow_multi_actions/flow_multi_actions_sample.c

- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_multi_actions/flow_multi_actions_main.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_multi_actions/meson.build`

14.3.13. Flow Multi-fwd

This sample shows how to use a different forward in pipe entries.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
 - a). Building a pipe with changeable source IP match and sending NULL in the forward.
 - b). Adding two entries to the pipe with different source IP match, and different forward:
 - ▶ The first entry with forward to the second port
 - ▶ The second with drop

Reference:

- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_multi_fwd/flow_multi_fwd_sample.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_multi_fwd/flow_multi_fwd_main.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_multi_fwd/meson.build`

14.3.14. Flow Ordered List

This sample shows how to use a DOCA Flow ordered list pipe.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
 - a). Building a root pipe with changeable 5-tuple match and forwarding to an ordered list pipe with a changeable index.
 - b). Adding two entries to the pipe with an example value sent to a different index in the ordered list pipe.
 - c). Building ordered list pipe with two lists, one for each entry:
 - ▶ First list uses meter and then shared counter
 - ▶ Second list uses shared counter and then meter

4. Waiting 5 seconds and querying the entries (total bytes and total packets).

Reference:

- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_ordered_list/flow_ordered_list_sample.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_ordered_list/flow_ordered_list_main.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_ordered_list/meson.build`

14.3.15. Flow RSS Meta

This sample shows how to use DOCA Flow forward RSS, set meta action, and then retrieve the matched packets in the sample.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
 - a). Building a pipe with a changeable 5-tuple match, forwarding to RSS queue with index 0, and setting changeable packet meta data.
 - b). Adding an entry with an example 5-tuple and metadata value to the pipe.
 - c).
4. Retrieving the packets on both ports from a receive queue, and printing the packet metadata value.

Reference:

- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_rss_meta/flow_rss_meta_sample.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_rss_meta/flow_rss_meta_main.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_rss_meta/meson.build`

14.3.16. 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:
 - a). Building a pipe with a changeable match on metadata and forwarding the matched packets to the second port.
 - b). Adding an entry that matches an example metadata value.

- c). Building a pipe with changeable 5-tuple match, changeable metadata action, and fwd to the first pipe.
- d). Adding entry with an example 5-tuple and metadata value to the pipe.

Reference:

- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_set_meta/flow_set_meta_sample.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_set_meta/flow_set_meta_main.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_set_meta/meson.build`

14.3.17. Flow Shared Counter

This sample shows how to use the DOCA Flow shared counter and query it to get the counter statistics.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
 - a). Binding the shared counter to the port.
 - b). Building a pipe with changeable 5-tuple match with UDP protocol, changeable shared counter ID and forwarding the matched packets to the second port.
 - c). Adding an entry with an example 5-tuple match and shared counter with `ID=port_id`.
 - d). Building a pipe with changeable 5-tuple match with TCP protocol, changeable shared counter ID and forwarding the matched packets to the second port.
 - e). Adding an entry with an example 5-tuple match and shared counter with `ID=port_id`.
 - f). Building a control pipe with the following entries:
 - ▶ If L4 type is UDP, forwards the packets to the UDP pipe
 - ▶ If L4 type is TCP, forwards the packets to the TCP pipe
4. Waiting 5 seconds and querying the shared counters (total bytes and total packets).

Reference:

- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_shared_counter/flow_shared_counter_sample.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_shared_counter/flow_shared_counter_main.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_shared_counter/meson.build`

14.3.18. Flow Shared Meter

This sample shows how to use the DOCA Flow shared meter.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
 - a). Config a shared meter with specific cir and cbs values.
 - b). Binding the shared meter to the port.
 - c). Building a pipe with a changeable 5-tuple match with UDP protocol, changeable shared meter ID and forwarding the matched packets to the second port.
 - d). Adding an entry with an example 5-tuple match and shared meter with `ID=port_id`.
 - e). Building a pipe with a changeable 5-tuple match with TCP protocol, changeable shared meter ID and forwarding the matched packets to the second port.
 - f). Adding an entry with an example 5-tuple match and shared meter with `ID=port_id`.
 - g). Building a control pipe with the following entries:
 - ▶ If L4 type is UDP, forwards the packets to the UDP pipe
 - ▶ If L4 type is TCP, forwards the packets to the TCP pipe

Reference:

- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_shared_meter/flow_shared_meter_sample.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_shared_meter/flow_shared_meter_main.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_shared_meter/meson.build`

14.3.19. 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:
 - a). Configuring a shared mirror with a clone destination hairpin to the second port.
 - b). Binding the shared mirror to the port.
 - c). Building a pipe with a changeable 5-tuple match with UDP protocol, changeable shared mirror ID, and forwarding the matched packets to the second port.
 - d). Adding an entry with an example 5-tuple match and shared mirror with `ID=port_id+1`.

- e). Building a pipe with a changeable 5-tuple match with TCP protocol, changeable shared mirror ID, and forwarding the matched packets to the second port.
- f). Adding an entry with an example 5-tuple match and shared mirror with `ID=port_id+1`.
- g). Building a control pipe with the following entries:
 - ▶ If L4 type is UDP, forwards the packets to the UDP pipe
 - ▶ If L4 type is TCP, forwards the packets to the TCP pipe
- h). Waiting 15 seconds to clone any incoming traffic. Should see the same two packets received on the second port (one from the clone and another from the original).

Reference:

- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_shared_mirror/flow_shared_mirror_sample.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_shared_mirror/flow_shared_mirror_main.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_shared_mirror/meson.build`

14.3.20. Flow VXLAN Encap

This sample shows how to use DOCA Flow actions to create a VXLAN tunnel.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
 - a). Building a pipe with changeable 5-tuple match, encap action, and forward port action.
 - b). Adding example 5-tuple and encapsulation values entry to the pipe.

Reference:

- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_vxlan_encap/flow_vxlan_encap_sample.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_vxlan_encap/flow_vxlan_encap_main.c`
- ▶ `/opt/mellanox/doca/samples/doca_flow/flow_vxlan_encap/meson.build`

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 nor any of its direct or indirect subsidiaries and affiliates (collectively: "NVIDIA") make no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assume 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 NON-INFRINGEMENT, 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, the NVIDIA logo, and Mellanox are trademarks and/or registered trademarks of Mellanox Technologies Ltd. and/or NVIDIA Corporation in the U.S. and in other countries. The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2023 NVIDIA Corporation & affiliates. All rights reserved.