DOCA Flow

Table of contents

DOCA Flow Connection Tracking 140

DOCA Flow Tune Server 164

This guide describes how to deploy the DOCA Flow library, the philosophy of the DOCA Flow API, and how to use it. The guide is intended for developers writing network function applications that focus on packet processing (such as gateways). It assumes familiarity with the network stack and DPDK.

Introduction

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

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

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

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

- Modify MAC address
- Modify IP address
- Modify L4 (ports)
- Strip tunnel
- Add tunnel
- Set metadata
- Encrypt/Decrypt

The execution pipe can also have monitoring actions such as the following:

- Count
- Policers

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

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

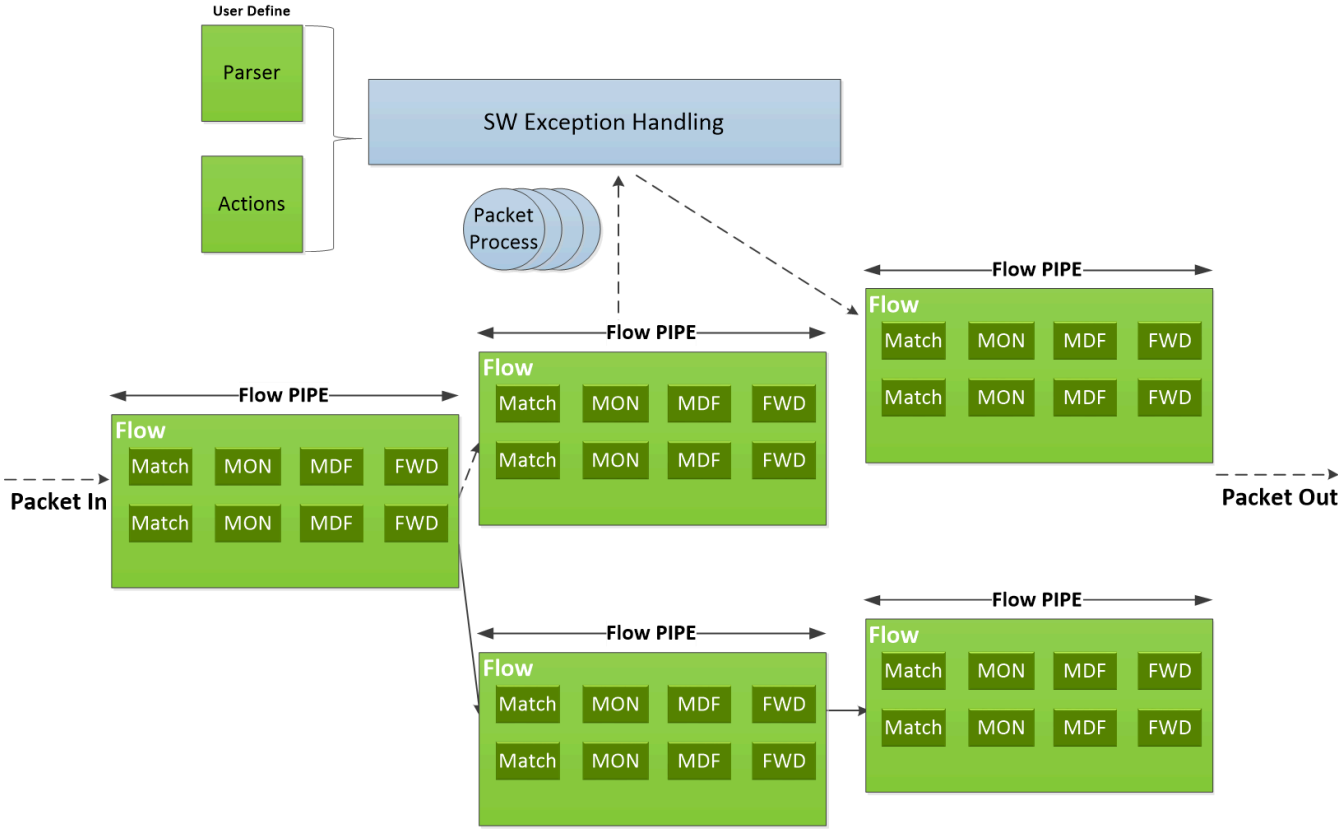
Prerequisites

A DOCA Flow-based application can run either on the host machine or on an NVIDIA® BlueField® DPU target. Flow-based programs require an allocation of huge pages, hence the following commands are required:

```
$ echo '1024' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-
2048kB/nr_hugepages
$ sudo mkdir /mnt/huge
$ sudo mount -t hugetlbfs -o pagesize=2M nodev /mnt/huge
```

Architecture

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



Features of DOCA Flow:

- User-defined set of matches parser and actions
- DOCA Flow pipes can be created or destroyed dynamically
- Packet processing is fully accelerated by hardware with a specific entry in a flow pipe
- Packets that do not match any of the pipe entries in hardware can be sent to Arm cores for exception handling and then reinjected back to hardware

The DOCA Flow pipe consists of the following components:

- Monitor (MON in the diagram) - counts, meters, or mirrors
- Modify (MDF in the diagram) - modifies a field
- Forward (FWD in the diagram) - forwards to the next stage in packet processing

Steering Domains

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

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

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

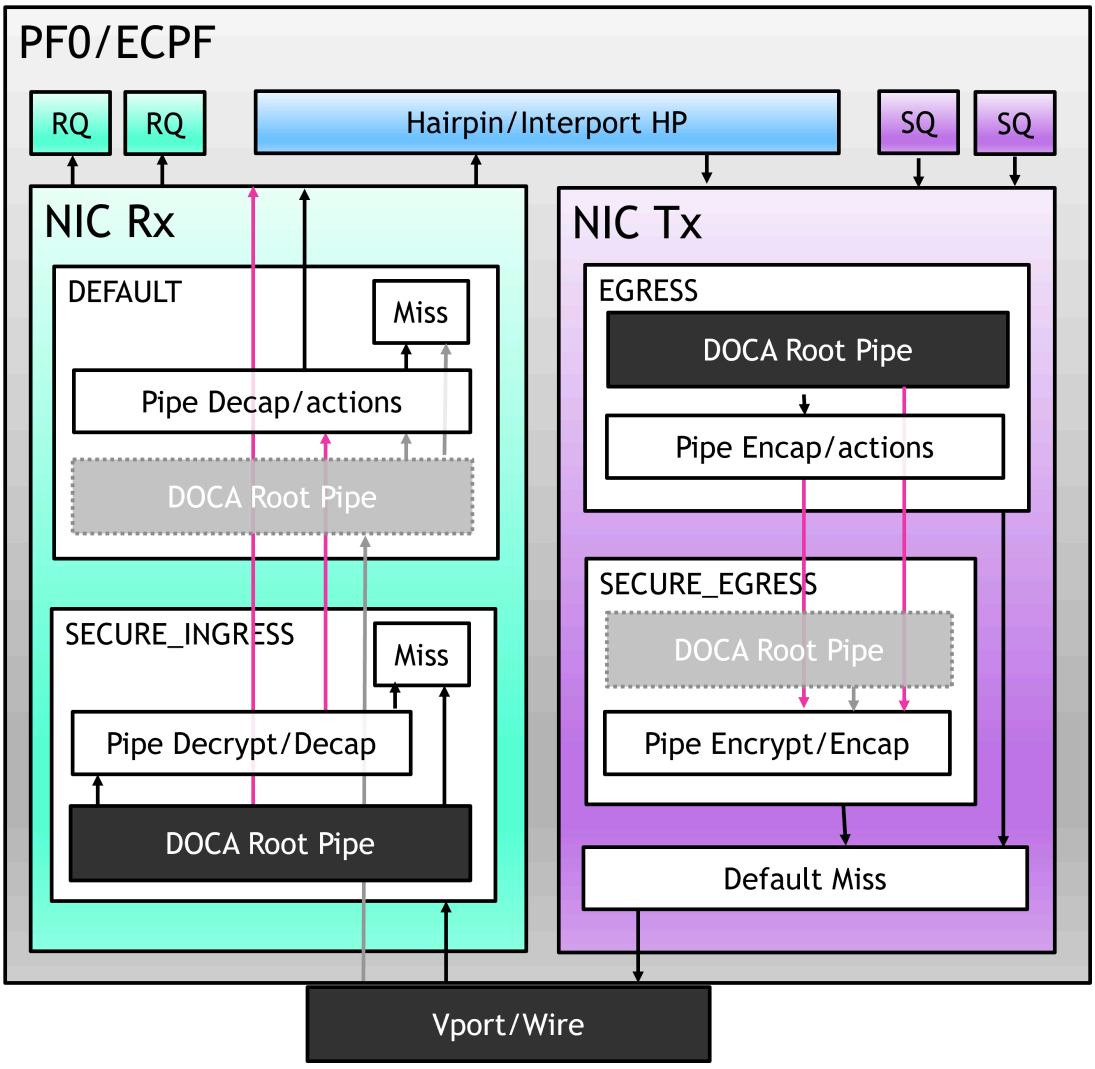
List of Steering Domains

DOCA Flow provides the following set of predefined steering domains:

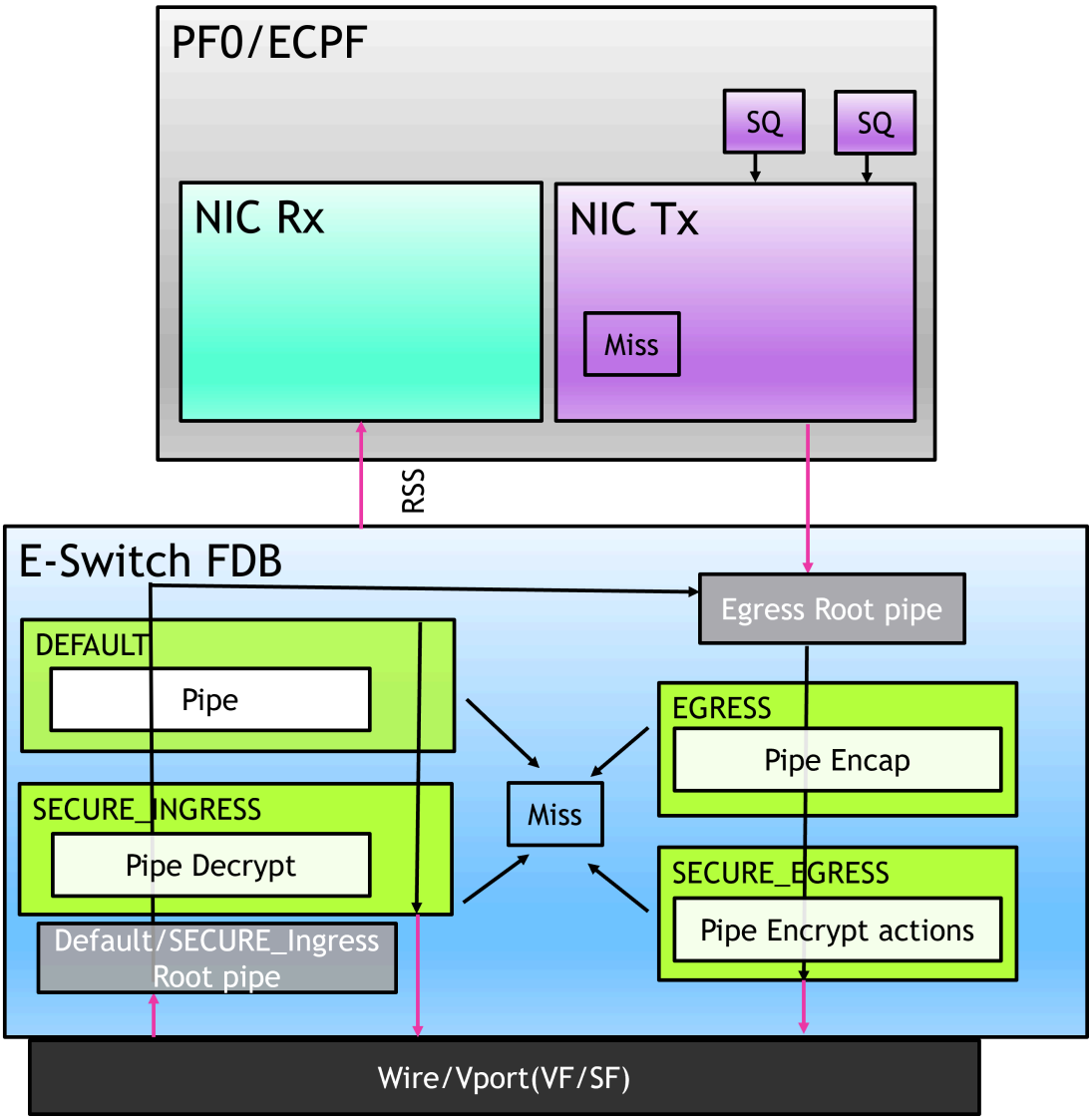
Domain	Description
`DOCA_FLOW_PIPE_DOMAIN_DEFAULT`	• Default domain for actions on ingress traffic • Encapsulated and secure actions are not allowed here • The next milestone is queue or pipe in the EGRESS domain • Miss action is: Drop
`DOCA_FLOW_PIPE_DOMAIN_SECURE_INGRESS`	• For secure actions on ingress traffic • Encapsulation and encrypting actions not allowed here • The only allowed domain for decrypting secure actions • The next milestone is queue or pipe in the DEFAULT or EGRESS domain • Only meta register is preserved • Miss action is: Drop • Memory may be optimized if set with `DOCA_FLOW_DIRECTION_NETWORK_TO_HOST` direction information
`DOCA_FLOW_PIPE_DOMAIN_EGRESS`	• Domain for actions on egress traffic • Decapsulation and secure actions are not allowed here • The next milestone is wire/representor or pipe in SECURE_EGRESS domain • Miss action is: Send to wire/representor

Domain	Description
`DOCA_FLOW_PIPE_DOMAIN_SECURE_EGRESS`	• Domain for secure actions on egress traffic • Decapsulation actions are not allowed here • The only allowed domain for encrypting secure action • The next milestone is wire/representor • Miss action is: Send to wire/representor • Memory may be optimized if set with `DOCA_FLOW_DIRECTION_HOST_TO_NETWORK` direction information

Domains in VNF Mode



Domains in Switch Mode



Note

In switch mode, forwarding from a pipe with the default domain to the egress domain root pipe is allowed, while forwarding from the egress domain to the default domain is not allowed.

Note

Traffic from software Tx forwards to the egress root pipe.

Note

A pipe with RSS forward follows the above rules.

Note

Encap is suggested to be done from egress.

API

DOCA API is available through the NVIDIA DOCA Library APIs page.

Info

The pkg-config (`*.pc` file) for the DOCA Flow library is `doca-flow`.

Flow Life Cycle

Initialization Flow

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

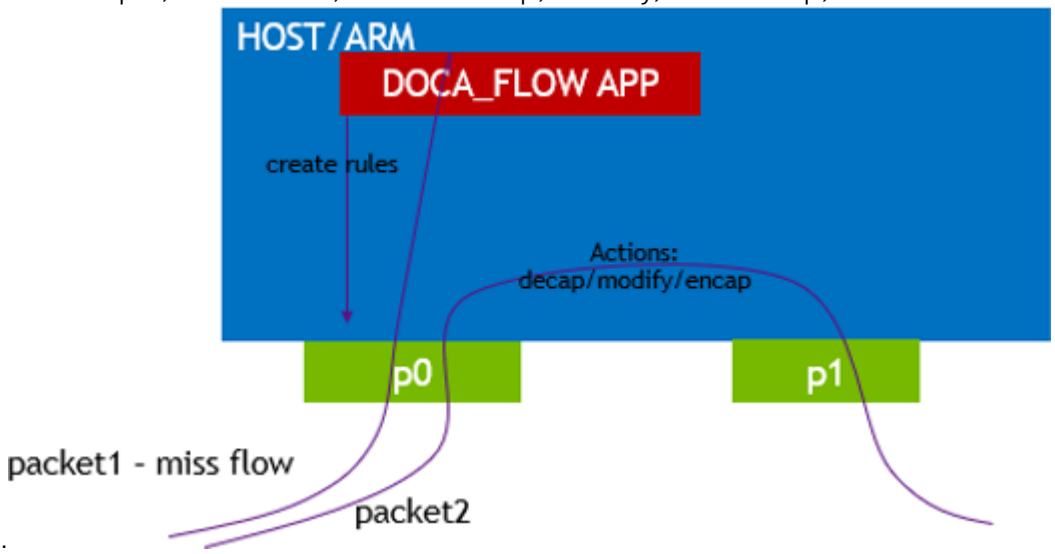
Pipe Mode

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

- `vnf`

  A packet arriving from one of the device's ports is processed, and can be sent to another port. By default, missed packets go to RSS.

  The following diagram shows the basic traffic flow in `vnf` mode. Packet1 firstly misses and is forwarded to host RSS. The app captures this packet and decides how to process it and then creates a pipe entry. Packet2 will hit this pipe entry and do the action, for example, for VXLAN, will do decap, modify, and encap, then is sent out from P1.

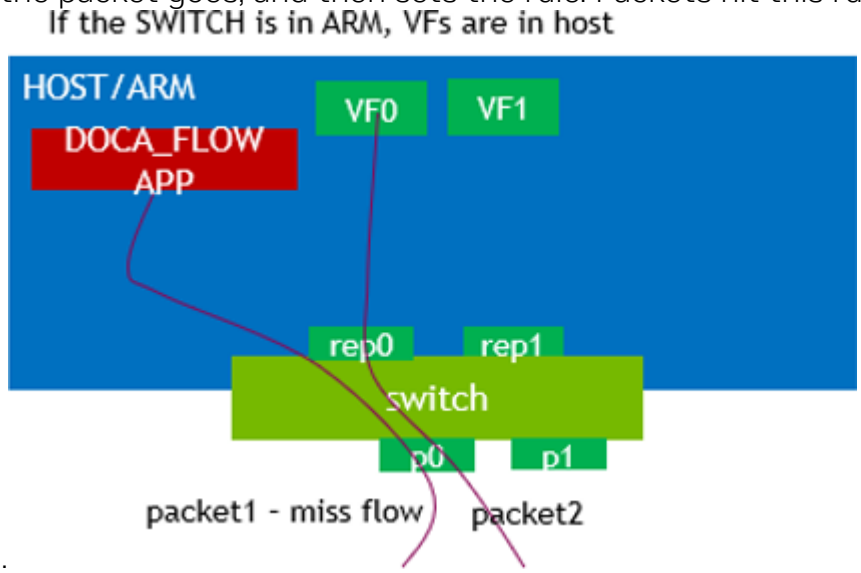


- `switch`

  Used for internal switching, only representor ports are allowed, for example, uplink representors and SF/VF representors. Packet is forwarded from one port to another. If a packet arrives from an uplink and does not hit the rules defined by the user's pipe, then the packet is received on all RSS queues of the representor of the uplink.

  The following diagram shows the basic flow of traffic in `switch` mode. Packet1 firstly misses to host RSS queues. The app captures this packet and decides to

which representor the packet goes, and then sets the rule. Packets hit this rule and



go to representor0.

`doca_dev` field is mandatory in `doca_flow_port_cfg` (using `doca_flow_port_cfg_set_dev()`) and isolated mode should be specified.

> **Note**
>
> The application must avoid initialization of the VF/SF representor ports in DPDK API (i.e., the following functions `rte_eth_dev_configure()`, `rte_eth_rx_queue_setup()`, `rte_eth_dev_start()` must not be called for VF/SF representor ports).

DOCA Flow switch mode unifies all the ports to the switch manager port for traffic management. This means that all the traffic is handled by switch manager port. Users only must create an RSS pipe on the switch manager port to get the missed traffic, and they should only manage the pipes on the switch manager port. Switch mode can work with two different `mode_args` configurations: With or without `expert`. The way to retrieve the miss traffic source's `port_id` depends on this configuration:

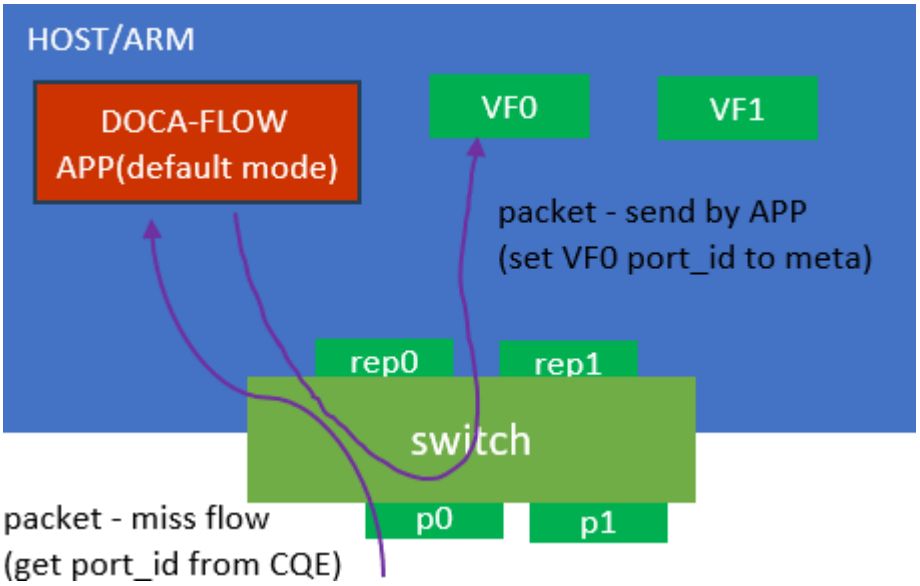
> **Note**

In switch mode, the default `fwd_miss` of the pipe (once `fwd_miss` is not configured by the user) is as follows:

- Forward to kernel in `isolated` mode
- Forward to the port RSS in `non-isolated` mode

- If `expert` is not set, the traffic misses to software would be tagged with `port_id` information in the mbuf CQE field to allow users to deduce the source `port_id`. Meanwhile, users can set the destination `port_id` to mbuf meta and the packet is sent out directly to the destination port based on the meta information.

> **Info**
>
> Please refer to the "Flow Switch to Wire" sample to get more information regarding the `port_id` management with missed traffic mbuf.



- If `expert` is set, the `port_id` is not added to the packet. Users can configure the pipes freely to implement their own solution.

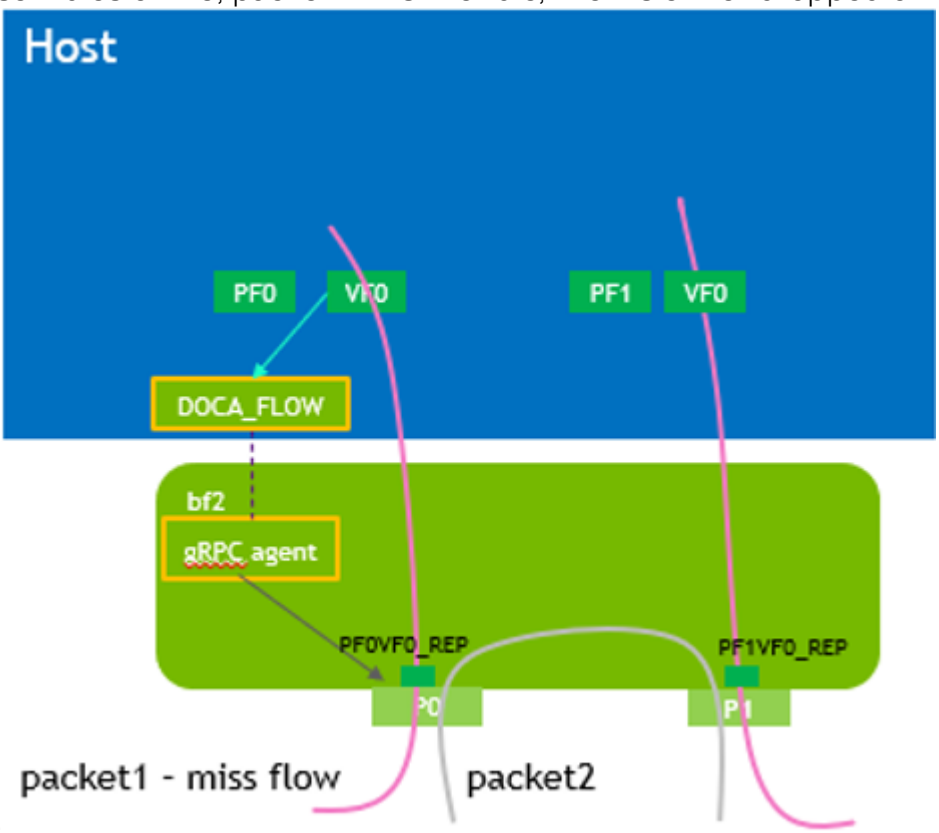
  > **Note**
  >
  > Traffic cloned from the VF to the RSS pipe misses its `port_id` information due to firmware limitation.

- `remote-vnf`

  Remote mode is a BlueField mode only, with two physical ports (uplinks). Users must use `doca_flow_port_pair` to pair one physical port and one of its representors. A packet from this uplink, if it does not hit any rules from the users, is firstly received on this representor. Users must also use `doca_flow_port_pair` to pair two physical uplinks. If a packet is received from one uplink and hits the rule whose FWD action is to another uplink, then the packets are sent out from it.

  The following diagram shows the basic traffic flow in remote-vnf mode. Packet1, from BlueField uplink P0, firstly misses to host VF0. The app captures this packet and decides whether to drop it or forward it to another uplink (P1). Then, using gRPC

to set rules on P0, packet2 hits the rule, then is either dropped or is sent out from P1.



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.

Port Operation State

DOCA Flow ports can be initialized multiple times from different instances. Each instance prepares its pipeline, but only one actively receives port traffic at a time. The instance actively handling the port traffic depends on the operation state set by the `doca_flow_port_cfg_set_operation_state()` function:

- `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` – The instance actively handles incoming and outgoing traffic
- `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE_READY_TO_SWAP` – The instance handles traffic actively when no other active instance is available
- `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY` – The instance handles traffic only when no active or `active_ready_to_swap` instance is available
- `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED` – The instance does not handle traffic, regardless of the state of other instances

If the `doca_flow_port_cfg_set_operation_state()` function is not called, the default state `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` is applied.

> **Note**
>
> When a port is configured with a state that expects to handle traffic, it takes effect only after root pipes are created for this port.

When the active port is closed, either gracefully or due to a crash, the standby instance automatically becomes active without any action required.

The port operation state can be modified after the port is started using the `doca_flow_port_operation_state_modify()` function.

Use Case Examples

Hot Upgrade

This operation state mechanism allows upgrading the DOCA Flow program without losing any traffic.

To upgrade an existing DOCA Flow program with ports started in `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` state (Instance A):

1. Open a new Instance B and start its ports in `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY` state.
2. Modify Instance A's ports from `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` to `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED` state. At this point, Instance B starts receiving traffic.
3. Close Instance A.
4. Open a new Instance C with `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED` state. Instance C is the upgraded version of Instance A.
5. Create the entire pipeline for Instance C.
6. Change Instance C's state from `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED` to `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE`. At this point, Instance B stops receiving traffic and Instance C starts.
7. Instance B can either be closed or kept as a backup should Instance C crash.

Swap Existing Instances

This mechanism also facilitates swapping two different DOCA Flow programs without losing any traffic.

To swap between two existing DOCA Flow programs with ports started in `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` and `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY` states (Instance A and Instance B, respectively):

1. Modify Instance A's ports from `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` to `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE_READY_TO_SWAP`.
2. Modify Instance B's ports from `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY` to `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE`. At this point, Instance B starts receiving traffic.
3. Modify Instance A's ports from `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE_READY_TO_SWAP` to `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY`.

Limitations

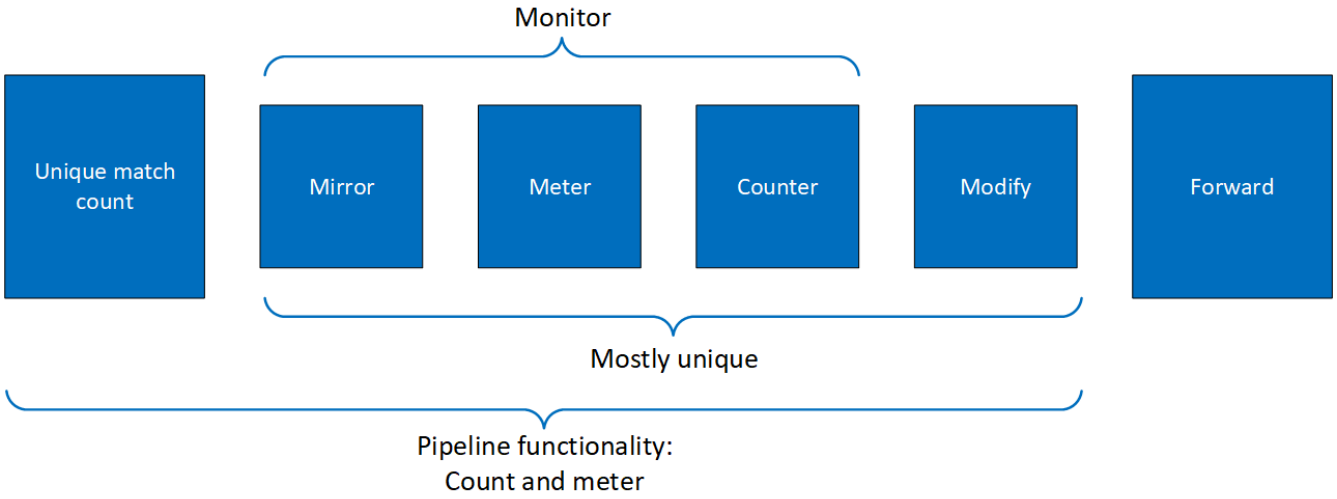
- Supported only in switch mode – the `mode_args` string must include `"switch"`.
- Only the switch port supports states; its representors are affected by its state. Starting a representor port or calling the modify function with a non-active operation state should fail.
- Two instances cannot be in the same operation state simultaneously, except for `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED`. If a new instance initializes a port while another instance with the same operation state exists, a `DOCA_ERROR_ALREADY_EXIST` error is returned either during port startup or root pipe creation.

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).

Pipe Matching or Action Applying

DOCA Flow allows defining criteria for matching on a packet or for taking actions on a matched packet by modifying it. The information defining these criteria is provided through the following pointers:

- Match or action pointer – given at pipe or entry creation
- Mask pointer – optionally given at pipe creation

Defining criteria for matching or actions on a packet can be done at the pipe level, where it applies to all packets of a pipe, or specified on a per entry basis, where each entry defines the operation on either the match, actions, or both.

In DOCA Flow terminology, when a field is identified as CHANGEABLE at pipe creation, this means that the actual criterion of the field is deferred to entry creation. Different entries can provide different criteria for a CHANGEABLE field.

A match or action field can be categorized, during pipe creation, as one of the following:

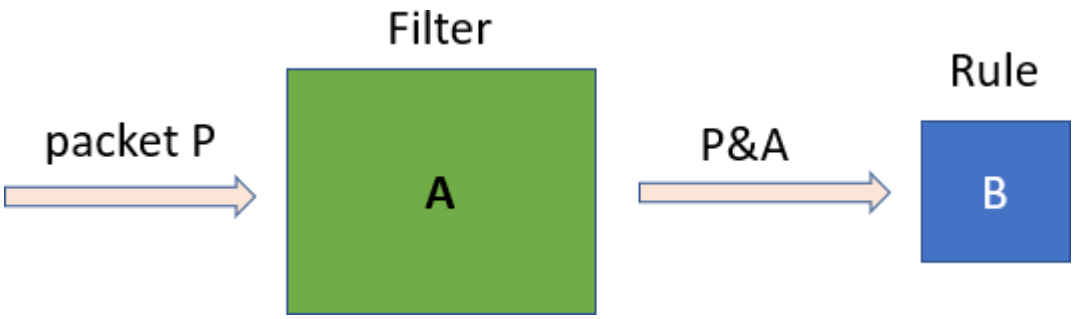
- IGNORED – Ignored in either the match or action taking process
- CHANGEABLE – When the actual behavior is deferred to the entry creation stage
- SPECIFIC – Value is used as is in either match or action process

A mask field can either be provided, in which case it is called explicit matching, or action applying. If the mask pointer is NULL, we call it implicit matching or action applying. The following subsections provide the logic governing matching and action applying.

When a field value is specified as `0xffff` it means that all the field's bits are set (e.g., for TTL it means `0xff` and for IPv4 address it means `0xffffffff`).

Matching

Matching is the process of selecting packets based on their fields' values and steering them for further processing. Processing can either be further matching or actions applying.



The packet enters the green filter which modifies it by masking it with the value A. The output value, P&A, is then compared to the value B, and if they are equal, then that is a match.

The values of A and B are evaluated according to the values of the pipe configuration and entry configuration fields, according to the tables in sections "Implicit matching" and "Explicit matching".

Implicit Matching

Match Type	Pipe Match Value (V)	Pipe Match Mask (M)	Entry Match Value (E)	Filter (A)	Rule (B)
Ignore	0	NULL	N/A	0	0

Match Type	Pipe Match Value (V)	Pipe Match Mask (M)	Entry Match Value (E)	Filter (A)	Rule (B)
Constant	0<V<0xffff	NULL	N/A	0xffff	V
Changeable (per entry)	0xffff	NULL	0⩽E⩽0xffff	0xffff	E

Explicit Matching

Match Type	Pipe Match Value (V)	Pipe Match Mask (M)	Entry Match Value (E)	Filter (A)	Rule (B)
Constant	V!=0xffff	0<M⩽0xffff	0⩽E⩽0xffff	M	M&V
Changeable	V==0xffff	0<M⩽0xffff	0⩽E⩽0xffff	M	M&E
Ignored	0⩽V<0xffff	M==0	0⩽E⩽0xffff	0	0

Action Applying

Implicit Action Applying

Action Type	Pipe Action value (V)	Pipe Action Mask (M)	Entry Action value (E)	Action on the field
Ignore	0	NULL	N/A	none
Constant	0 < V < 0xffff	NULL	N/A	set to V
Changeable	0xffff	NULL	E	set to E

Implicit action applying example:

- Destination IPv4 address is 255.255.255.255
- No mask provided
- Entry value is 192.168.0.1

- Result – The action field is changeable. Therefore, the value is provided by the entry. If a match on the packet occurs, the packet destination IPv4 address is changed to 192.168.0.1.

Explicit Action Applying

> **Info**
>
> Assume P is packet's field value.

Action Type	Pipe Action value (V)	Pipe Action Mask (M)	Entry Action value (E)	Action on the field
constant	V!=0xffff	0⩽M⩽0xffff	0⩽E⩽0xffff	set to (~M & P) | (M & V) In words: modify only bits that are set on the mask to the values in V
Changeable	V==0xffff	0<M⩽0xffff	0⩽E⩽0xffff	set to (~M & P) | (M & E)
Ignored	0⩽V<0xffff	M==0	0⩽E⩽0xffff	none

Explicit action applying example:

- Destination IPv4 address is 192.168.10.1
- Mask is provided and equals 255.255.0.0
- Entry value is ignored
- Result – If a match on the packet occurs, the packet destination IPv4 value changes to 192.168.0.0.

Setting Pipe Match or Action

Match is a mandatory parameter when creating a pipe. Using the `doca_flow_match` struct, users must define the packet fields to be matched by the pipe.

For each `doca_flow_match` field, users select whether the field type is:

- Ignore (match any) – the value of the field is ignored in a packet. In other words, match on any value of the field.
- Constant (specific) – all entries in the pipe have the same value for this field. Users should not put a value for each entry.
- Changeable – the value of the field is defined per entry. Users must provide it upon adding an entry.

> **Note**
>
> L4 type, L3 type, and tunnel type cannot be changeable.

> **Note**
>
> `gtp_next_ext_hdr_type` supports only `psc` type (0x85).

The match field type can be defined either implicitly or explicitly using the `doca_flow_pipe_cfg_set_match(struct doca_flow_pipe_cfg *cfg, const doca_flow_match *match, const doca_flow_match *match_mask)` function. If `match_mask == NULL`, then it is done implicitly. Otherwise, it is explicit.

In the tables in the following subsections, an example is used of a 16-bit field (such as layer-4 destination port) where:

> **Note**

The same concept would apply to any other field (such as an IP address occupying 32 bits).

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

Implicit Match

Match Type	Pipe Match Value (V)	Pipe Match Mask (M)	Entry Match Value (E)	Filter (A)	Rule (B)
Ignore	0	NULL	N/A	0	0
Constant	0<V<0xffff	NULL	N/A	0xffff	V
Changeable (per entry)	0xffff	NULL	0⩽E⩽0xffff	0xffff	E

To match implicitly, the following considerations should be taken into account.

- Ignored fields:
  - Field is zeroed
  - Pipeline has no comparison on the field
- Constant fields – These are fields that have a constant value among all entries. For example, as shown in the following, the tunnel type is VXLAN:

  ```
  match.tun.type = DOCA_FLOW_TUN_VXLAN;
  ```

  These fields must only be configured once at pipe build stage, not once per new pipeline entry.

- Changeable fields – These are fields whose value may change per entry. For example, the following shows match on a destination IPv4 address of variable per-entry value (outer 5-tuple):

```
match.outer.ip4.dst_ip = 0xffffffff;
```

- The following is an example of a match, where:
  - Outer 5-tuple
    - L3 type is IPv4 – constant among entries by design
    - L4 type is UDP – constant among entries by design
    - Tunnel type is `DOCA_FLOW_TUN_VXLAN` – constant among entries by design
    - IPv4 destination address varies per entry
    - UDP destination port is always `DOCA_VXLAN_DEFAULT_PORT`
    - VXLAN tunnel ID varies per entry
    - The rest of the packet fields are ignored
  - Inner 5-tuple
    - L3 type is IPv4 – constant among entries by design
    - L4 type is TCP – constant among entries by design
    - IPv4 source and destination addresses vary per entry
    - TCP source and destination ports vary per entry
    - The rest of the packet fields are ignored

```
// filter creation
```

```
static void build_underlay_overlay_match(struct doca_flow_match
*match)
{
    //outer
    match->outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
    match->outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;
    match->tun.type = DOCA_FLOW_TUN_VXLAN;
    match->outer.ip4.dst_ip = 0xffffffff;
    match->outer.udp.l4_port.dst_port = DOCA_VXLAN_DEFAULT_PORT;
    match->tun.vxlan_tun_id = 0xffffffff;

    //inner
    match->inner.l3_type = DOCA_FLOW_L3_TYPE_IP4;
    match->inner.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_TCP;
    match->inner.ip4.dst_ip = 0xffffffff;
    match->inner.ip4.src_ip = 0xffffffff;
    match->inner.tcp.l4_port.src_port = 0xffff;
    match->inner.tcp.l4_port.dst_port = 0xffff;
}

// create entry specifying specific values to match upon
doca_error_t add_entry(struct doca_flow_pipe *pipe, struct
doca_flow_port *port,
                       struct doca_flow_pipe_entry **entry)
{
    struct doca_flow_match match = {};
    struct entries_status status = {};
    doca_error_t result;

    match.outer.ip4.dst_ip = BE_IPV4_ADDR(7, 7, 7, 1);
    match.tun.vxlan_tun_id = RTE_BE32(9876);
    match.inner.ip4.src_ip = BE_IPV4_ADDR(8, 8, 8, 1);
    match.inner.ip4.dst_ip = BE_IPV4_ADDR(9, 9, 9, 1);
    match.inner.tcp.l4_port.src_port = rte_cpu_to_be_16(5678);
    match.inner.tcp.l4_port.dst_port = rte_cpu_to_be_16(1234);
```

```
    result = doca_flow_pipe_add_entry(0, pipe, &match, &actions,
NULL, NULL, 0, &status, entry);
}
```

> **Note**
>
> The fields of the `doca_flow_meta` struct inside the match are not subject to implicit match rules and must be paired with explicit mask values.

Explicit Match

Match Type	Pipe Match Value (V)	Pipe Match Mask (M)	Entry Match Value (E)	Filter (A)	Rule (B)
Constant	V!=0xffff	0<M⩽0xffff	0⩽E⩽0xffff	M	M&V
Changeable	V==0xffff	0<M⩽0xffff	0⩽E⩽0xffff	M	M&E
Ignored	0⩽V<0xffff	M==0	0⩽E⩽0xffff	0	0

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

- Ignored fields
  - M equals zero. This can be seen from the table where the rule equals 0. Since mask is also 0, the resulting packet after the filter is0. Thus, the comparison always succeeds.

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

- Constant fields

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

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

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

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

  Once a field is defined as constant, the field's value cannot be changed per entry.

  > **Tip**
  >
  > Users should set constant fields to zero when adding entries for better code readability.

  A more complex example of constant matches may be achieved as follows:

```
match_mask.outer.tcp.l4_port.dst_port =
rte_cpu_to_be_16(0xf0f0);
match.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(0x5020)
```

The following ports would be matched:

- 0x5020 - 0x502f
- 0x5120 - 0x512f
- ...
- 0x5f20 - 0x5f2f

Changeable fields

The following example matches on either FTP or TELNET well known port numbers and forwards packets to a server after modifying the destination IP address and destination port numbers. In the example, either FTP or TELNET are forwarded to the same server. FTP is forwarded to port 8000 and TELNET is forwarded to port 9000.

```
// at Pipe creation
doca_flow_pipe_cfg_set_name(pipe_cfg, "PORT_MAPPER");
doca_flow_pipe_cfg_set_type(pipe_cfg, DOCA_FLOW_PIPE_BASIC);
match.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(0xffff); // v
match_mask.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(0xffff);
// M
doca_flow_pipe_cfg_set_match(pipe_cfg, &match, &match_mask);
actions_arr[0] = &actions;
doca_flow_pipe_cfg_set_actions(pipe_cfg, action_arr, NULL, NULL,
1);
doca_flow_pipe_cfg_set_is_root(pipe_cfg, true);

// Adding entries
// FTP
match.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(20); // E
```

```
actions.outer.ip4.src_ip = server_addr;
actions.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(8000);
result = doca_flow_pipe_add_entry(0, pipe, &match, &actions,
NULL, NULL, 0, &status, entry);

// TELNET
match.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(23); // E
actions.outer.ip4.src_ip = server_addr;
actions.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(9000);
result = doca_flow_pipe_add_entry(0, pipe, &match, &actions,
NULL, NULL, 0, &status, entry);
```

Relaxed Match

Relaxed matching is the default working mode in DOCA Flow. Relaxed mode grants users full control on matching fields and guarantees that no fields are implicitly added by DOCA Flow.

> **Note**
>
> Although relaxed matching can be disabled per pipe using the `enable_strict_matching` pipe attribute, be aware that this attribute will be deprecated at some point in the future.

Relaxed Matching and Pipeline Design Considerations

Relaxed matching mode provides full control to the DOCA application developer over the match design, without adding implicit match logic by the DOCA Flow library. This approach increases user responsibility to prevent unintended side effects caused by packet layout similarities (e.g., between UDP and TCP source/destination ports) or by skipping header type validation before matching on header fields. For instance, matching solely on a UDP destination port does not verify the presence of an L4 header or confirm that the L4 header is of UDP type.

To ensure effective design, early-stage pipes should classify packet types to filter out undesired packets. As the pipeline progresses, later stages can focus on more specific packet types based on prior matches. Examples include:

- Match on VXLAN VNI:
  - For Early-stage pipes, ensure the packet contains a VXLAN header (this can be achieved in a single pipe):
    - The first pipe verifies the packet has a UDP header by matching the L4 packet type to UDP or the L3 `next_proto` field to UDP
    - The second pipe matches the UDP destination port to the commonly used VXLAN value (4789)
  - For later-stage pipes, match on the VXLAN VNI field
- Match on UDP destination port:
  - For early-stage pipes, verify the packet contains a UDP header by matching the L4 packet type to UDP or the L3 `next_proto` field to UDP
  - For later-stage pipes, match on the UDP destination port field

Relaxed Matching Memory Footprint and Performance Impact

Consider the following strict matching mode example. There are three pipes:

- Basic pipe `A` with `match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_TCP` and `match.outer.tcp.flags = 1`
- Basic pipe `B` with `match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP` and `match.outer.udp.l4_port.src_port = 8080`
- Control pipe `C` with two entries to direct TCP traffic to pipe `A` and UDP traffic to pipe `B`. The first entry has `match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_TCP` while the second has `match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP`.

As a result, the hardware matches on the L4 header type twice:

- When the packet enters the filter in control pipe `C` to decide the next pipe
- When the packet enters the filter of pipe `A` or `B` to match on the L4 header fields

With particularly large pipelines, such double matches decrease performance and increase the memory footprint in hardware. Relaxed matching mode gives the user greater control of the match logic to eliminate the implicitly added matches, consequently reducing hardware memory footprint and improving performance as well.

Parser Meta Usage with Relaxed Match

Parser meta matching is particularly useful when it comes to matching on a specific packet type. In relaxed mode, type selectors in the `outer`, `inner`, and `tun` parts of the `doca_flow_match` structs are used only for the type cast of the underlying unions. Header-type (packet type) matches are available using the `parser_meta` API.

For example, the scenario from the previous section may be overwritten by changing the match of control pipe `C` (with the same `A` and `B` pipes):

- Basic pipe `A` with `match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_TCP` and `match.outer.tcp.flags = 1`
- Basic pipe `B` with `match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP` and `match.outer.udp.l4_port.src_port = 8080`
- Control pipe `C` with two entries to direct TCP traffic to pipe `A` and UDP traffic to pipe `B`. The first entry has `match.parser_meta.outer_l4_type = DOCA_FLOW_L4_META_TCP` while the second has `match.parser_meta.outer_l4_type = DOCA_FLOW_L4_META_UDP` .

As a result, the hardware performs the L4 header-type match only once, when the packet enters the filter of control pipe. Basic pipes' `match.outer.l4_type_ext` are used only

for the selection of the `match.outer.tcp` or `match.outer.udp` structures during the inspection of match struct.

Examples

The following code snippets are used to demonstrate the redesign of a pipeline with relaxed matching for non-tunnel match cases.

The following is the code before the redesign:

```
static void pipe_match_build(struct doca_flow_match *match)
{
        match->outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
        match->outer.ip4.dst_ip = 0xffffffff;
        match->outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;
        match->outer.udp.l4_port.src_port = 22;
}
```

With relaxed matching disabled (i.e., enable_strict_matching attribute explicitly set to `true`), the following hardware matches are performed for the code snippet above:

- L3 header type is IPv4 – constant among entries by design
- L4 header type is UDP – constant among entries by design
- IPv4 destination address varies per entry
- UDP source port is constant among entries
- The rest of the packet fields are ignored

With relaxed matching enabled (i.e., default mode), the following pipeline stages, where `pipe1` forwards packets to `pipe2`, should be considered to achieve a similar match as above:

```
static void pipe1_match_build(struct doca_flow_match *match)
```

```
{
        // Classifier logic. Only IPv4, UDP packets are to be forwarded to pipe2
            match.parser_meta.outer_l3_type = DOCA_FLOW_L3_META_IPV4;
            match.parser_meta.outer_l4_type = DOCA_FLOW_L4_META_UDP;
}

static void pipe2_match_build(struct doca_flow_match *match)
{
        // Main logic. Match on the specific packet fields
            match->outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
            match->outer.ip4.dst_ip = 0xffffffff;
            match->outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;
            match->outer.udp.l4_port.src_port = 22;
}
```

The following code snippet demonstrates the redesign of a pipeline with relaxed matching for tunnel match cases:

```
static void pipe1_match_build(struct doca_flow_match *match)
{
    match->outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
            match->outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;
            match->tun.type = DOCA_FLOW_TUN_VXLAN;
            match->outer.ip4.dst_ip = 0xffffffff;
            match->outer.udp.l4_port.src_port = 0x22;
            match->tun.vxlan_tun_id = 0xffffffff;
}
```

With relaxed matching disabled (i.e., enable_strict_matching attribute set to `true`), the following hardware matches are performed for the code snippet above:

- L3 type is IPv4 – constant among entries by design
- L4 type is UDP – constant among entries by design

- Tunnel type is `DOCA_FLOW_TUN_VXLAN` – constant among entries by design
- IPv4 destination address varies per entry
- UDP source port is always 22
- VXLAN tunnel ID varies per entry
- The rest of the packet fields are ignored

With relaxed matching enabled (i.e., default mode), the following pipeline stages, where `pipe1` forwards packets to `pipe2`, should be considered to achieve a similar match as above:

```
static void pipe1_match_build(struct doca_flow_match *match)
{
        // Classifier logic. Only IPv4, UDP packets are to be forwarded to pipe2
        match.parser_meta.outer_l3_type = DOCA_FLOW_L3_META_IPV4;
        match.parser_meta.outer_l4_type = DOCA_FLOW_L4_META_UDP;
}

static void pipe2_match_build(struct doca_flow_match *match)
{
        // Main logic. Match on the specific packet fields
        match->outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;
        match->outer.udp.l4_port.src_port = 22;
    match->outer.udp.l4_port.dst_port = DOCA_VXLAN_DEFAULT_PORT;
    match->outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
        match->outer.ip4.dst_ip = 0xffffffff;
        match->tun.type = DOCA_FLOW_TUN_VXLAN;
        match->tun.vxlan_tun_id = 0xffffffff;
}
```

Relaxed Matching Considerations

With relaxed matching, header type fields within `outer`, `inner`, or `tun` structs no longer indicate a match on the specific protocol. They are use d solely as a selector for the relevant header fields. For example, to match on `outer.ip4.dst_ip`, users must set `outer.l3_type = DOCA_FLOW_L3_TYPE_IP4`. With this match, the L3 header is checked for the IPv4 destination address, however there is no check that the L3 header is of IPv4 type. It is user responsibility to make sure that packets arriving to such a filter indeed have an L3 header of type IPv4.

To match on a specific protocol/tunnel type, consider the following:

- To match on an inner/outer L3/L4 protocol type, users can use relevant `doca_flow_parser_meta` fields as explained above. For example, for outer protocols, `parser_meta.outer_l[3,4]_type` fields can be used.

- To match on a specific tunnel type, users should match on a tunnel according to its specification. For example, for a VXLAN tunnel, a match on UDP destination port 4789 can be used. Another option is to use the L3 next protocol field. For example, for IPv4 with next header GRE, one can match on the IPv4 headers' next protocol field value to match GRE IP protocol number 47.

More relaxed matching design best practices can be found in the samples Flow Drop, Flow VXLAN Encap, and LPM with Exact Match Logic.

Note

With relaxed matching, to achieve a match-all functionality, either one of the following methods can be used during pipe creation:

- Set the `match_mask` structure to NULL and set the match structure to all zeroes

- Set the `match_mask` structure to all zeroes while the match structure have any setting

Note

With relaxed matching, if any of the selectors is used without setting a relevant field, the pipe/entry creation would fail with the following error message:

```
failed building active opcode - active opcode
<opcode number> is protocol only
```

Setting Pipe Actions

Pipe Execution Order

When setting actions, they are executed in the following order:

1. Crypto (decryption)
2. Decapsulation
3. Pop
4. Meta
5. Outer
6. Tun
7. Push
8. Encapsulation
9. Crypto (encryption)

Note

Modifying a field while simultaneously using it as a source for other modifications should be avoided, as the sequence of modification

actions cannot be guaranteed.

Auto-modification

Similarly to setting pipe match, actions also have a template definition.

Similarly to `doca_flow_match` in the creation phase, only the subset of actions that should be executed per packet are defined. This is done in a similar way to match, namely by classifying a field of `doca_flow_match` to one of the following:

- Ignored field – field is zeroed, modify is not used.
- Constant fields – when a field must be modified per packet, but the value is the same for all packets, a one-time value on action definitions can be used
- Changeable fields – fields that may have more than one possible value, and the exact values are set by the user per entry

```
actions.outer.ip4.dst_ip = 0xffffffff
```

> **Note**
>
> The `action_mask` should be set as `0xffffffff` and action as 0 if the user wants to configure 0 to this field.

Explicit Modification Type

It is possible to force constant modification or per-entry modification with action mask. For example:

```
static void
create_constant_modify_actions(struct doca_flow_actions *actions
                               struct doca_flow_actions
*actions_mask,
                               struct doca_flow_action_descs
*descs)
{
        actions->outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;
        actions->outer.udp.src_port = 0x1234;
        actions_mask->outer.udp.src_port = 0xffff;
}
```

Copy Field

The action descriptor can be used to copy between the packet field and metadata. For example:

```
#define META_U32_BIT_OFFSET(idx) (offsetof(struct doca_flow_meta,
u32[(idx)]) << 3)

static void
create_copy_packet_to_meta_actions(struct doca_flow_match *match
                                   struct doca_flow_action_desc
*desc)
{
        desc->type = DOCA_FLOW_ACTION_COPY;
        desc->field_op.src.field_string = "outer.ipv4.src_ip";
        desc->field_op.src.bit_offset = 0;
        desc->field_op.dst.field_string = "meta.data";
        desc->field_op.dst.bit_offset = META_U32_BIT_OFFSET(1); /*
Bit offset of meta.u32[1] */;
}
```

Multiple Actions List

Creating a pipe is possible using a list of multiple actions. For example:

```
static void
create_multi_actions_for_pipe_cfg()
{
    struct doca_flow_actions *actions_arr[2];
    struct doca_flow_actions actions_0 = {0}, actions_1 = {0};
    struct doca_flow_pipe_cfg *pipe_cfg;
    /* input configurations for actions_0 and actions_1 */
    actions_arr[0] = &actions_0;
    actions_arr[1] = &actions_1;
        doca_flow_pipe_cfg_set_actions(pipe_cfg, actions_arr,
NULL, NULL, 2);
}
```

Summary of Action Types

Pipe Creation				Entry Creation	Behavior
action_desc		Pipe Actions	Pipe Actions Mask	Entry Actions	
doca_flow_action_type	Configuration				
`DOCA_FLOW_ACTION_AUTO` / `action_desc = NULL`	No specific config	`0`	`0`	N/A	Field ignored, no modification
		`0`	`mask` != 0	N/A	Apply `0` and `mask` to all entries
		`val` != 0 && `val` != 0xFF	`mask` != 0	N/A	Apply `val` and `mask` to all

Pipe Creation				Entry Creation	Behavior
					entries
		`val` = 0xFF	`mask` = 0	N/A	Apply `0xFF` to all entries
		`val` = 0xFF	`mask` != 0	Define `val` per entry	Apply entry's `val` and `mask`
`DOCA_FLOW_ACTION_ADD` Add field value or from `src`	Define only the `dst` field and width	`val` != 0	N/A	N/A	Apply this `val` to all entries
		`val` == 0	N/A	Define `val` per entry	Apply entry's `val`
	Define the `src` and `dst` fields and width	Define the source and destination fields. • Meta field → header field • Header field → meta field • Meta field → meta field	N/A	N/A	Add data from `src` fields to `dst` for all entries
`DOCA_FLOW_ACTION_COPY` Copy field to another field	N/A	Define the source and destination fields. • Meta field → header field • Header field → meta field • Meta field → meta field	N/A	N/A	Copy data between fields for all entries

Setting Pipe Monitoring

If a meter policer should be used, then it is possible to have the same configuration for all policers on the pipe or to have a specific configuration per entry. The meter policer is determined by the FWD action. If an entry has NULL FWD action, the policer FWD action is taken from the pipe.

If a mirror should be used, mirror can be shared on the pipe or configured to have a specific value per entry.

The monitor also includes the aging configuration, if the aging time is set, this entry ages out if timeout passes without any matching on the entry.

For example:

```
static void build_entry_monitor(struct doca_flow_monitor
*monitor, void *user_ctx)
{
        monitor->aging_sec = 10;
}
```

Refer to Pipe Entry Aged Query for more information.

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.

Pipe forwarding can be s et either at creation time or be deferred to entry addition:

- If the `fwd.type` type is not `DOCA_FLOW_FWD_CHANGEABLE`, any match on the pipe, on any entry, is forwarded to the specified target
- If the `fwd.type` type is `DOCA_FLOW_FWD_CHANGEABLE`, the target would match on whatever is defined in the `fwd.type` field of the specific entry

Putting this logic in a table look like this:

Pipe Fwd.type	Entry Fwd.type	Actual Forward	Comment
Equals `DOCA_FLOW_FWD_CHANGEABLE`	X	X	X must not equal `DOCA_FLOW_FWD_CHANGEABLE`
X != `DOCA_FLOW_FWD_CHANGEABLE`	Does not care	X	

When a pipe includes meter monitor `<cir, cbs>`, it must h ave `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_type = DOCA_FLOW_RESOURCE_TYPE_NON_SHARED;
fwd.rss.queues_array = queues;
fwd.rss.outer_flags = DOCA_FLOW_RSS_IPV4 | DOCA_FLOW_RSS_UDP;
fwd.rss.nr_queues = 4;
```

Queues point to the `uint16_t` array that contains the queue numbers. When a port is started, the number of queues is defined, starting from zero up to the number of queues

minus 1. RSS queue numbers may contain any subset of those predefined queue numbers. For a specific match, a packet may be directed to a single queue by having RSS forwarding with a single queue.

Changeable RSS forwarding is supported. When creating the pipe, the `num_of_queues` must be set to `0xffffffff`, then different forwarding RSS information can be set when adding each entry.

```
fwd->num_of_queues = 0xffffffff;
```

The packet is directed to the port. In many instances the complete pipe is executed in the hardware, including the forwarding of the packet back to the wire. The packet never arrives to the software.

Example code for forwarding to port:

```
struct doca_flow_fwd *fwd = malloc(sizeof(struct doca_flow_fwd));
memset(fwd, 0, sizeof(struct doca_flow_fwd));
fwd->type = DOCA_FLOW_FWD_PORT;
fwd->port_id = port_id; // this should the same port_id that was
set in doca_flow_port_cfg_set_devargs()
```

The type of forwarding is `DOCA_FLOW_FWD_PORT` and the only data required is the `port_id` as defined in `DOCA_FLOW_PORT`.

Changeable port forwarding is also supported. When creating the pipe, the `port_id` must be set to `0xffff`, then different forwarding `port_id` values can be set when adding each entry.

```
fwd->port_id = 0xffff;
```

Shared Resources

DOCA Flow supports several types of resources that can be shared. The supported types of resources can be:

- Meters
- Counters
- RSS queues
- Mirrors
- PSPs
- Encap
- Decap
- IPsec SA

Shared resources can be used by several pipes and can save device and memory resources while promoting better performance.

To create and configure shared resource, the user should go through the steps detailed in the following subsections.

Creating Shared Resource Configuration Object

Call `doca_flow_cfg_create(&flow_cfg)`, passing a pointer to `struct doca_flow_cfg` to be used to fill the required parameters for the shared resource.

> **Note**
>
> The `struct doca_flow_cfg` object is used for configuring other resources besides the aforementioned shared resources, but this section only refers to the configuration of shared resources.

Setting Number of Shared Resources per Shared Resource Type

This can be done by calling `doca_flow_cfg_set_nr_shared_resource()`. Refer to the API documentation for details on the configuration process.

Conclude the configuration by calling `doca_flow_init()`.

Configuring Shared Resource

When shared resources are allocated, they are assigned identifiers ranging from 0 and increasing incrementally. For example, if the user configures two shared counters, they would bear the identifiers 0 and 1.

> **Note**
>
> Note that each resource has its own identifier space. So, if users have two shared counters and three meters, they would bear identifiers 0..1 and 0..2 respectively.

Configuring the shared resources requires the user to call `doca_flow_shared_resource_set_cfg()`.

Binding Shared Resource

A shared resource must be bound by calling `doca_flow_shared_resources_bind()` which binds the resource to a pointer. The object to which the resource is bound is usually a `struct doca_flow_port` pointer.

Using Shared Resources

After a resource has been configured, it can be used by referring to its ID.

In the case of meters, counters, and mirrors, they are referenced through `struct doca_flow_monitor` during pipe creation or entry addition.

Querying Shared Resource

Querying shared resources can be done by calling `doca_flow_shared_resources_query()`. The function accepts the resource type and an array of resource numbers, and returns an array of `struct doca_flow_shared_resource_result` with the results.

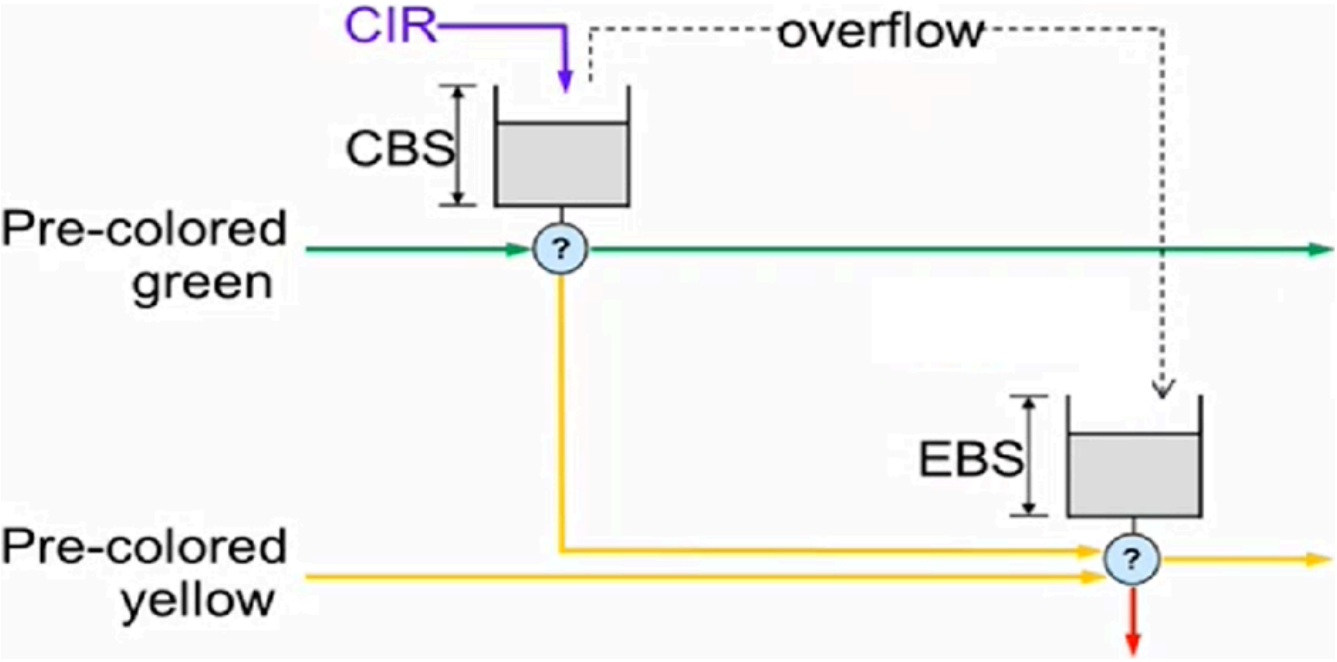
Shared Meter Resource

A shared meter can be used in multiple pipe entries (hardware steering mode support only).

The shared meter action marks a packet with one of three colors: Green, Yellow, and Red. The packet color can then be matched in the next pipe, and an appropriate action may be taken. For example, packets marked in red color are usually dropped. So, the next pipe to meter action may have an entry which matches on red and has fwd type `DOCA_FLOW_FWD_DROP`.

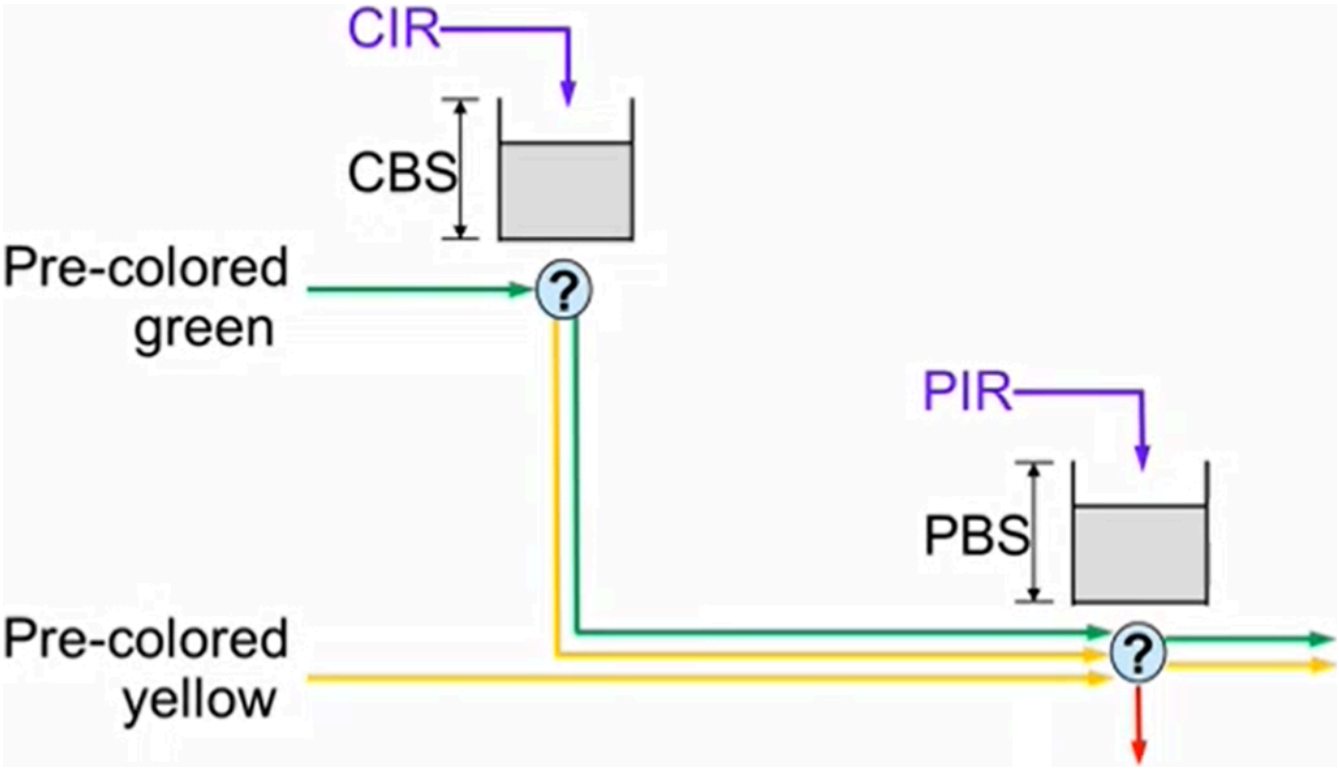
DOCA Flow supports three marking algorithms based on RFCs: 2697, 2698, and 4115.

RFC 2697 – Single-rate Three Color Marker (srTCM)



CBS (committed burst size) is the bucket size which is granted credentials at a CIR (committed information rate). If CBS overflow occurs, credentials are passed to the EBS (excess burst size) bucket. Packets passing through the meter consume credentials. A packet is marked green if it does not exceed the CBS, yellow if it exceeds the CBS but not the EBS, and red otherwise. A packet can have an initial color upon entering the meter. A pre-colored yellow packet will start consuming credentials from the EBS.

RFC 2698 – Two-rate Three Color Marker (trTCM)



CBS and CIR are defined as in RFC 2697. PBS (peak burst size) is a second bucket which is granted credentials at a PIR (peak information rate). There is no overflow of credentials from the CBS bucket to the PBS bucket. The PIR must be equal to or greater than the CIR. Packets consuming CBS credentials consume PBS credentials as well. A packet is marked red if it exceeds the PIR. Otherwise, it is marked either yellow or green depending on whether it exceeds the CIR or not. A packet can have an initial color upon entering the meter. A pre-colored yellow packet starts consuming credentials from the PBS.

RFC 4115 – trTCM without Peak-rate Dependency

EBS is a second bucket which is granted credentials at a EIR (excess information rate) and gets overflowed credentials from the CBS. For the packet marking algorithm, refer to RFC 4115.

The following sections present the steps for configuring and using shared meters to mark packets.

Shared IPsec SA Resource

The IPsec Security Association (SA) shared resource is used for IPsec ESP encryption protocol. The resource should be pointed from the `doca_flow_crypto_actions` struct that inside `doca_flow_actions`.

By default, the resource manages the state of the sequence number (SN), incrementing each packet on the encryption side, and performing anti-replay protection on the decryption side. The anti-replay syndrome is stored in `meta.u32[0]`.

To control the SN in software, `sn_offload` should be disabled per port in the configuration for `doca_flow_port_start` (see DOCA API documentation for details). Once `sn_offload` is disabled, the following fields are ignored: `sn_offload_type`, `win_size`, `sn_initial`, and `lifetime_threshold`.

When shared resource query is called for an IPsec SA resource, the current SN is retrieved for the encryption resource and the lower bound of anti-replay window is retrieved for the decryption resource. Querying IPsec SA can only be called when `sn_offload` is enabled.

To maintain a valid state of the resource during its usage, `doca_flow_crypto_ipsec_resource_handle` should be called periodically.

Shared Mirror Resource

The mirror shared resource is used to clone packets to other pipes, vports (switch mode only), RSS queues (VNF mode only), or drop.

> **Info**
>
> The maximum supported mirror number is 4K.

> **Info**
>
> The maximum supported mirror clone destination is 254.

Mirror clone destination as `next_pipe` cannot be intermixed with `port` or `rss` types. Only clone destination and origin destination both as `next_pipe` is supported.

The register copy for packet after mirroring is not saved.

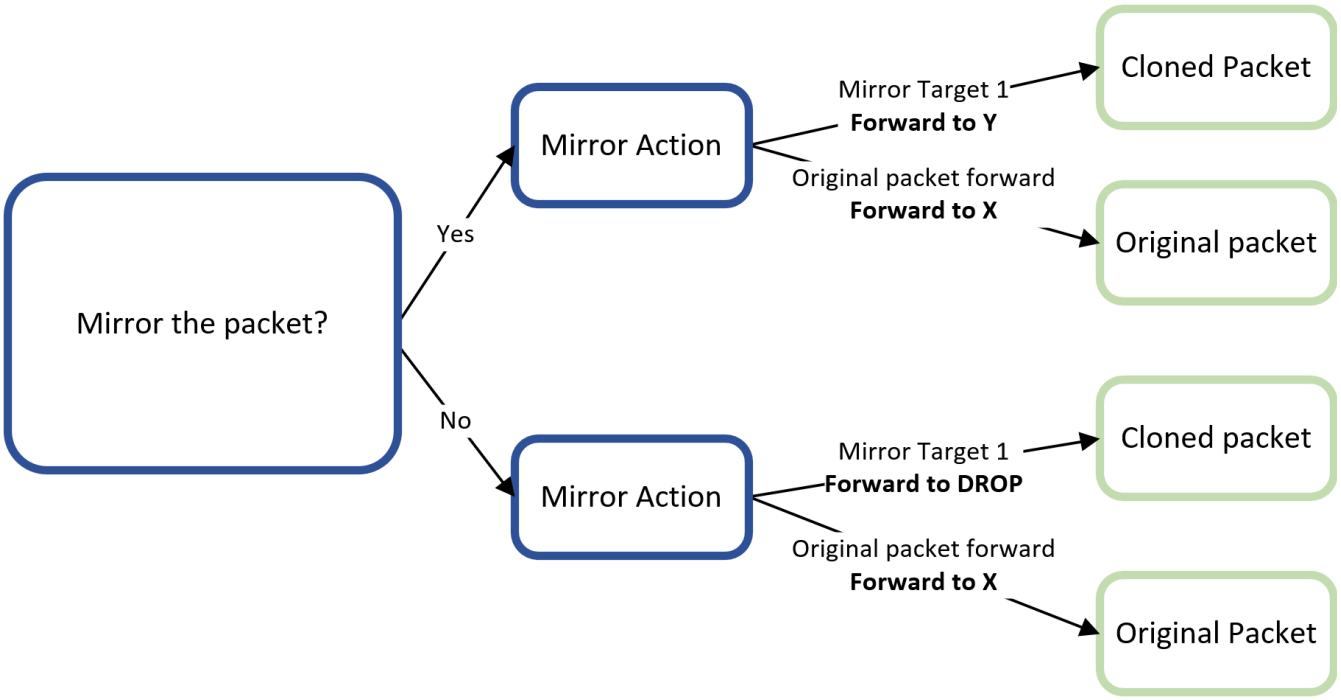
> **Note**
>
> For switch mode, there are several mirror limitations which should be noted:
>
> - Mirror should be cloned to `DOCA_FLOW_DIRECTION_BIDIRECTIONAL` pipe
> - The register copy for pkt after mirroring is not saved
> - Mirror should not be cloned to RSS pipe directly
> - Encap is supported while cloning a packet to a wire port only
> - Mirror must not be configured on a resizable pipe

If mirror creation fails, users should check the resulting syndrome for failure details.

Mirroring and Packet Order

To maintain the order of the mirrored packets in relation to the non-mirrored ones, set a first mirror target forward destination equivalent to the non-mirrored packets as illustrated in the following diagram:

In NVIDIA® BlueField®-3, NVIDIA® ConnectX®-7, and lower, when using the mirror action in the egress domain, mirrored packets cannot preserve the order with the non-mirrored packets due to the high latency of the mirror operation. To maintain the order, use `DOCA_FLOW_FWD_DROP` as the target forward as illustrated in the following diagram:



Shared Encap Resource

The encap shared resource is used for encapsulation. A shared encap ID represents one kind of encap configuration and can be used in multiple pipes and entries (hardware steering mode support only).

The shared encap action encapsulates the packet with the configured tunnel information.

Shared Decap Resource

The decap shared resource is used for decapsulation. A shared decap ID represents one kind of decap configuration and can be used in multiple pipes and entries (hardware steering mode support only).

The shared decap action decapsulates the packet. Ethernet information should be provided when is_l2 is false.

Shared PSP Resource

The PSP shared resource is used for PSP encryption. The resource should be pointed to from the `doca_flow_crypto_actions` struct in `doca_flow_actions`.

The resource should be configured with a key to encrypt the packets. See NVIDIA DOCA Library API documentation for PSP key generation for a reference about key handling on decrypt side.

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 DOCA Library APIs to see which fields are optional and may be skipped. It is typically recommended to set optional fields to 0 when not in use. See Miss Pipe and Control Pipe for more information.

Once a pipe is created, a new entry can be added to it. These entries are bound to a pipe, so when a pipe is destroyed, all the entries in the pipe are removed. Please refer to section Pipe Entry for more information.

There is no priority between pipes or entries. The way that priority can be implemented is to match the highest priority first, and if a miss occurs, to jump to the next PIPE. There can be more than one PIPE on a root as long the pipes are not overlapping. If entries overlap, the priority is set according to the order of entries added. So, if two pipes have overlapping matching and PIPE1 has higher priority than PIPE2, users should add an entry to PIPE1 after all entries are added to PIPE2.

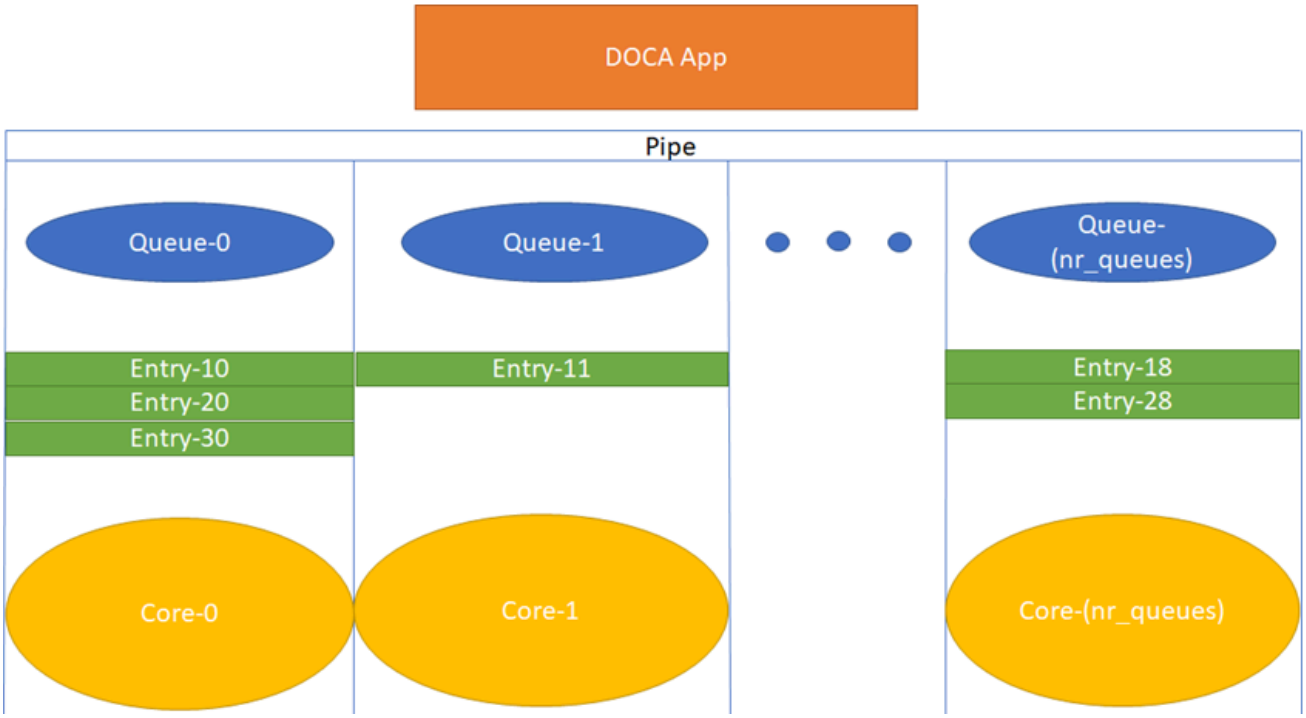
Pipe Entry (doca_flow_pipe_add_entry)

An entry is a specific instance inside of a pipe. When defining a pipe, users define match criteria (subset of fields to be matched), the type of actions to be done on matched packets, monitor, and, optionally, the FWD action.

When a user calls `doca_flow_pipe_add_entry()` to add an entry, they should define the values that are not constant among all entries in the pipe. And if FWD is not defined

then that is also mandatory.

DOCA Flow is designed to support concurrency in an efficient way. Since the expected rate is going to be in millions of new entries per second, it is mandatory to use a similar architecture as the data path. Having a unique queue ID per core saves the DOCA engine from having to lock the data structure and enables the usage of multiple queues when interacting with hardware.



Each core is expected to use its own dedicated `pipe_queue` number when calling `doca_flow_pipe_entry`. Using the same `pipe_queue` from different cores causes a race condition and has unexpected results.

> **Note**
>
> Applications are expected to avoid adding, removing, or updating pipe entries from within a `doca_flow_entry_process_cb`.

Failure Path

Entry insertion can fail in two places, `add_entry` and `add_entry_cb`.

- When `add_entry` fails, no cleanup is required.
- When `add_entry` succeeds, a handle is returned to the user. If the subsequent `add_entry_cb` fails, the user is responsible for releasing the handle through a `rm_entry` call. This `rm_entry` call is expected to return `DOCA_SUCCESS` and is expected to invoke `doca_rm_entry_cb` with a successful return code.

Pipe Entry Counting

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

> **Note**
>
> Having a counter per entry affects performance and should be avoided if it is not required by the application.

The retrieved statistics are stored in struct `doca_flow_query`.

> **Note**
>
> Counters have a granularity of 1 second.

Pipe Entry Aged Query

When a user calls `doca_flow_aging_handle()`, this query is used to get the aged-out entries by the time quota in microseconds. The user callback is invoked by this API with the aged entries.

Since the number of flows can be very large, the query of aged flows is limited by a quota in microseconds. This means that it may return without all flows and requires the user to

call it again. When the query has gone over all flows, a full cycle is done.

Pipes with Multiple Actions

Users can define multiple actions per pipe. This gives the user the option to specify a different action per entry in the pipe by providing the `action_idx` in `struct doca_flow_actions`. Note that even with multiple actions defined for a pipe, any packet processed will still result in at most one action being executed.

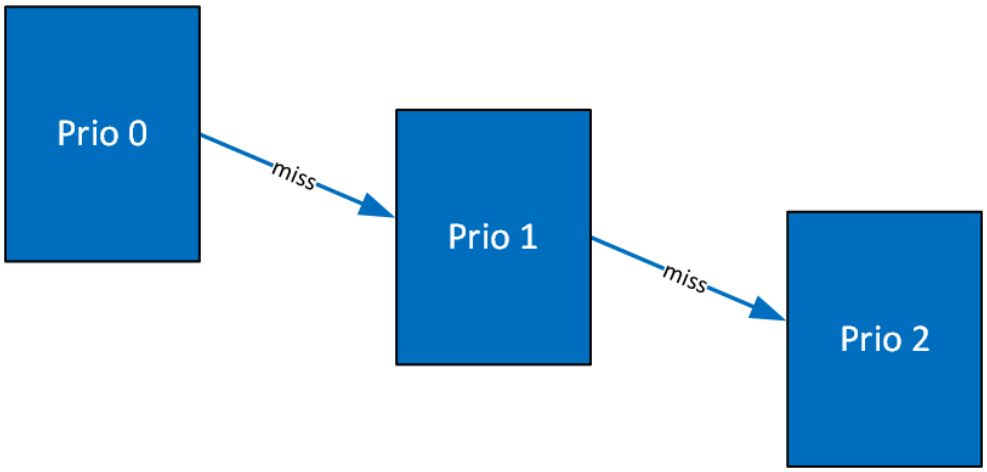
For example, to create multiple flows with identical match fields but different actions, users can define two actions during pipe creation, `Action_0` and `Action_1`. These actions are respectively assigned indices 0 and 1 in the pipe configuration's actions array. `Action_0` includes `modify_mac`, while `Action_1` includes `modify_ip`. Users can then add two types of entries to the pipe: the first entry uses `Action_0` by setting the `action_idx` field in `struct doca_flow_actions` to 0, and the second entry uses `Action_1` by setting `action_idx` to 1.

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. After creating a control pipe, the user can add control entries with different matches, forwarding, and priorities when there are conflicts.

The user can add a control entry by calling `doca_flow_control_pipe_add_entry()`.

`priority` must be defined as higher than the lowest priority (3) and lower than the highest one (0).

The other parameters represent the same meaning of the parameters in `doca_flow_pipe_create`. In the example above, a control entry for VXLAN is created. The VLXAN packets with `ipB` hit the control entry.

doca_flow_pipe_lpm

> **Note**

This feature is not supported in this DOCA release. It will be re-enabled in DOCA version 3.0.

`doca_flow_pipe_lpm` uses longest prefix match (LPM) matching. LPM matching is limited to a single field of the `match` provided by the user at pipe creation (e.g., the outer destination IP). Each entry is consisted of a value and a mask (e.g., 10.0.0.0/8, 10.10.0.0/16, etc). The LPM match is defined as the entry that has the maximum matching bits. For example, using the two entries 10.7.0.0/16 and 10.0.0.0/8, the IP 10.1.9.2 matches on 10.0.0.0/8 and IP 10.7.9.2 matches on 10.7.0.0/16 because 16 bits are the longest prefix matched.

In addition to the longest prefix match logic, LPM supports exact match (EM) logic on the `meta.u32`, inner destination MAC and VNI. Only index `1` is supported for `meta.u32`. Any combination of these three fields can be chosen for EM. However, if inner destination MAC is chosen for LPM, then it should not be chosen for EM as well. If more than one field is chosen for EM, a logical AND is applied. Support for EM on meta allows working with any single field by copying its value to the `meta.u32[1]` on pipes before LPM. EM is performed at the same time as LPM matching (i.e., a logical AND is applied for both logics). For example, if there is a match on LPM logic, but the value in the fields chosen for EM is not exactly matched, this constitutes an LPM pipe miss.

To enable EM logic in an LPM pipe, two steps are required:

1. Provide `match_mask` to the LPM pipe creation with `meta.u32[1]` being fully masked and/or `inner.eth.dst_mac` and/or `tun.vxlan_tun_id`, while setting `match_mask.tun.type` to `DOCA_FLOW_TUN_VXLAN`. Thus, the `match` parameter is responsible for the choice of field for LPM logic, while the `match_mask` parameter is responsible for the enablement of EM logic. Separation into two parameters is done to distinguish which field is for LPM logic and which is for EM logic, when both fields can be used for LPM (e.g., destination IP address and source MAC address).

2. Per entry, provide values to do exact match using the `match` structure. `match_mask` is used only for LPM-related masks and is not involved into EM logic.

EM logic allows inserting many entries with different meta values for the same pair of LPM-related data. Regarding IPv4-based LPM logic with exact match enabled: LPM pipe can have 1.1.1.1/32 with `meta` 42, 555, and 1020. If a packet with 1.1.1.1/32 goes through such an LPM pipe, its `meta` value is compared against 42, 555, and 1020.

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

> **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**
>
> An LPM pipe can only do LPM matching on inner and outer IP and MAC addresses.

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

doca_flow_pipe_acl

> **Note**
>
> This feature is not supported in this DOCA release. It will be re-enabled in DOCA version 3.0.

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

ACL entry port configuration:

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

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

ACL supports the following types of FWD:

- `DOCA_FLOW_FWD_PORT`
- `DOCA_FLOW_FWD_PIPE`
- `DOCA_FLOW_FWD_DROP`

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

> **Note**

An ACL pipe can be a root pipe.

Note

An ACL pipe can be in ingress and egress domain.

Note

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

Note

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

Note

Removing an entry from an ACL pipe is not supported.

doca_flow_pipe_ordered_list

> **Note**
>
> This feature is not supported in this DOCA release. It will be re-enabled in DOCA version 3.0."

`doca_flow_pipe_ordered_list` allows the user to define a specific order of actions and multiply the same type of actions (i.e., specific ordering between counter/meter and encap/decap).

An ordered list pipe is defined by an array of actions (i.e., sequences of actions). Each entry can be an instance one of these sequences. An ordered list pipe may consist of up to an array of 8 different actions. The maximum size of each action array is 4 elements. Resource allocation may be optimized when combining multiple action arrays in one ordered list pipe.

doca_flow_pipe_hash

`doca_flow_pipe_hash` allows the user to insert entries by index. The index represents the packet hash calculation.

An hash pipe gets `doca_flow_match` only on pipe creation and only mask. The mask provides all fields to be used for hash calculation.

The `monitor`, `actions`, `actions_descs`, and `FWD` of the DOCA Flow hash pipe works the same as the basic DOCA Flow pipe.

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

Hardware Steering Mode

Users can enable hardware steering mode by setting devarg `dv_flow_en` to `2`.

The following is an example of running DOCA with hardware steering mode:

```
.... –a 03:00.0, dv_flow_en=2 –a 03:00.1, dv_flow_en=2....
```

The following is an example of running DOCA with software steering mode:

```
.... –a 03:00.0 –a 03:00.1 ....
```

The `dv_flow_en=2` means that hardware steering mode is enabled.

In the struct `doca_flow_cfg`, setting `mode_args` using (`doca_flow_cfg_set_mode_args()`) represents DOCA applications. If it is set with `hws` (e.g., `"vnf,hws"`, `"switch,hws"`, `"remmote_vnf,hws"`) then hardware steering mode is enabled.

In switch mode, `fdb_def_rule_en=0,vport_match=1,repr_matching_en=0,dv_xmeta_en=4` should be added to DPDK PMD devargs, which makes DOCA Flow switch module take over all the traffic.

To create an entry by calling `doca_flow_pipe_add_entry`, the parameter flags can be set as `DOCA_FLOW_WAIT_FOR_BATCH` or `DOCA_FLOW_NO_WAIT`:

- `DOCA_FLOW_WAIT_FOR_BATCH` means that this flow entry waits to be pushed to hardware. Batch flows then can be pushed only at once. This reduces the push times and enhances the insertion rate.
- `DOCA_FLOW_NO_WAIT` means that the flow entry is pushed to hardware immediately.

The parameter `usr_ctx` is handled in the callback set in struct `doca_flow_cfg`.

`doca_flow_entries_process` processes all the flows in this queue. After the flow is handled and the status is returned, the callback is executed with the status and

`usr_ctx`.

If the user does not set the callback in `doca_flow_cfg`, the user can get the status using `doca_flow_entry_get_status` to check if the flow has completed offloading or not.

Isolated Mode

In non-isolated mode (default) any received packets (following an RSS forward, for example) can be processed by the DOCA application, bypassing the kernel. In the same way, the DOCA application can send packets to the NIC without kernel knowledge. This is why, by default, no replies are received when pinging a host with a running DOCA application. If only specific packet types (e.g., DNS packets) should be processed by the DOCA application, while other packets (e.g., ICMP ping) should be handled directly the kernel, then isolated mode becomes relevant.

In isolated mode, packets that match root pipe entries are steered to the DOCA application (as usual) while other packets are received/sent directly by the kernel.

If you plan to create a pipe with matches followed by action/monitor/forward operations, due to functional/performance considerations, it is advised that root pipes entries include the matches followed by a next pipe forward operation. In the next pipe, all the planned matches actions/monitor/forward operations could be specified. Unmatched packets are received and sent by the kernel.

> **Info**
>
> In switch mode, DPDK must be in `isolated` mode. DOCA Flow may be in `isolated` or `non-isolated`.

To activate isolated mode, two configurations are required:

1. DOCA configuration: Update the string member `mode_args` (`struct doca_flow_cfg`) using `doca_flow_cfg_set_mode_args()` which represents the DOCA application mode and add "isolated" (separated by comma) to the other mode arguments. For example:

`doca_flow_cfg_set_mode_args(cfg, "vnf,hws,isolated")`
`doca_flow_cfg_set_mode_args(cfg, "switch,isolated")`

2. DPDK configuration: Set `isolated_mode` to 1 (`struct application_port_config`). For example, if DPDK is initialized by the API:
`dpdk_queues_and_ports_init(struct application_dpdk_config *app_dpdk_config)`
.

```
struct application_dpdk_config app_dpdk_config = {
    .port_config = {
        .isolated_mode = 1,
        .nb_ports = ...
        ...
    },
    ...
};
```

Pipe Resize

> **Note**
>
> This feature is not supported in this DOCA release. It will be re-enabled in DOCA version 3.0.

The move to HWS improves performance because rule insertion is implemented in hardware rather than software. However, this move imposes additional limitations, such as the need to commit in advance on the size of the pipes (the number of rule entries). For applications that require pipe sizes to grow over time, a static size can be challenging: Committing to a pipe size too small can cause the the application to fail once the number of rule entries exceeds the committed number, and pre-committing to an excessively high number of rules can result in memory over-allocation.

This is where pipe resizing comes in handy. This feature allows the pipe size to increase during runtime with support for all entries in a new resized pipe.

> **Info**
>
> Pipe resizing is supported in a basic pipe and a control pipe.

Increasing Pipe Size

It is possible to set a congestion level by percentage (`CONGESTION_PERCENTAGE`). Once the number of entries in the pipe exceeds this value, a callback is invoked. For example, for a pipe with 1000 entries and a `CONGESTION_PERCENTAGE` of 80%, the `CONGESTION_REACHED` callback is invoked after the 800th entry is added.

Following the `CONGESTION_REACHED` callback, the application should call the pipe resize API (`resize()`). The following are optional callbacks during the resize callback:

- A callback on the new number of entries allocated to the pipe
- A callback on each entry that existed in the smaller pipe and is now allocated to the resized pipe

> **Info**
>
> The pipe pointer remains the same for the application to use even after being resized.

Upon completion of the internal transfer of all entries from the small pipe to the resized pipe, a `RESIZED` callback is invoked.

A `CONGESTION_REACHED` callback is received exactly once before the `RESIZED` callback. Receiving another `CONGESTION_REACHED` only happens after calling `resize()` and receiving its completion with a `RESIZED` callback.

List of Callbacks

- `CONGESTION_REACHED` – on the updated number of entries in the pipe (if pipe is resizable)

> **Info**
>
> Receiving a `CONGESTION_REACHED` callback can occur after adding a small number of entries and for moving entries from a small to resized pipe. The application must always call pipe resize after receiving the `CONGESTION_REACHED` callback to handle such cases.

- `RESIZED` – upon completion of the resize operation

> **Note**
>
> Calling pipe resize returns immediately. It starts an internal process that ends later with the `RESIZED` callback.

- `NR_ENTRIES_CHANGED` (optional) – on the new max number of entries in the pipe
- `ENTRY_RELOCATE` (optional) – on each entry moved from the small pipe to the resized pipe

Order of Operations for Pipe Resizing

1. Set a process callback on flow configuration:

```
struct doca_flow_cfg *flow_cfg;
doca_flow_cfg_create(&flow_cfg);
doca_flow_cfg_set_cb_pipe_process(flow_cfg, <pipe-process-
callback>);
```

Info

This informs on `OP_CONGESTION_REACHED` and `OP_RESIZED` operations when applicable.

2. Set the following pipe attribute configurations:

```
struct doca_flow_pipe_cfg *pipe_cfg;
doca_flow_pipe_cfg_create(&pipe_cfg, port);
doca_flow_pipe_cfg_set_nr_entries(pipe_cfg, <initial-number-
of-entries>);
doca_flow_pipe_cfg_set_is_resizable(pipe_cfg, true);
doca_flow_pipe_cfg_set_congestion_level_threshold(pipe_cfg,
<CONGESTION_PERCENTAGE>);
doca_flow_pipe_cfg_set_user_ctx(pipe_cfg, <pipe-user-
context>);
```

3. Start adding entries:

```
/* Basic pipe */
doca_flow_pipe_add_entry()
/* Contorl pipe */
doca_flow_pipe_control_add_entry()
```

4. Once the number of entries in the pipe crosses the congestion threshold, an `OP_CONGESTION_REACHED` operation callback is received.

5. Mark the pipe's congestion threshold event and, upon return, call `doca_flow_pipe_resize()`. For this call, add the following parameters:

    - The new threshold percentage for calculating the new size.
    - A callback on the new pipe size (optional):

      ```
      doca_flow_pipe_resize_nr_entries_changed_cb
      nr_entries_changed_cb
      ```

    - A callback on the entries to be transferred to the resized pipe:

      ```
      doca_flow_pipe_resize_entry_relocate_cb
      entry_relocation_cb
      ```

6. Call `doca_flow_entries_process()` to trigger the transfer of entries. It is relevant for both a basic pipe and a control pipe.

7. At this phase, adding new entries to the pipe is permitted. The entries are added directly to the resized pipe and therefore do not need to be transferred.

8. Once all entries are transferred, an `OP_RESIZED` operation callback is received. Also, at this point a new `OP_CONGESTION_REACHED` operation callback can be received again.

9. At this point calling `doca_flow_entries_process()` can be stopped for a control pipe. For a basic pipe an additional call is required to complete the call to `doca_flow_pipe_add_entry()`.

Info

`doca_flow_entries_process()` has the following roles:

- Triggering entry transfer from the smaller to the bigger pipe (until an `OP_RESIZED` callback is received)
- Follow up API on previous `add_entries` API (basic pipe relevance only)

Hairpin Configuration

In switch mode, if `dev` is set in struct `doca_flow_port_cfg` (using `doca_flow_port_cfg_set_dev()`), then an internal hairpin is created for direct wire-to-wire fwd. Users may specify the hairpin configuration using `mode_args`. The supported options as follows:

- `hairpinq_num=[n]` – the hairpin queue number
- `use_huge_mem` – determines whether the Tx buffer uses hugepage memory
- `lock_rx_mem` – locks Rx queue memory

Teardown

Pipe Entry Teardown

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

Pipe Teardown

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

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

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

Port Teardown

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

Flow Teardown

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

Metadata

> **Info**
>
> A scratch area exists throughout the pipeline whose maximum size is `DOCA_FLOW_META_MAX` bytes.

The user can set a value to metadata, copy from a packet field, then match in later pipes. Mask is supported in both match and modification actions.

The user can modify the metadata in different ways based on its actions' masks or descriptors:

- `ADD` – set metadata scratch value from a pipe action or an action of a specific entry. Width is specified by the descriptor.
- `COPY` – copy metadata scratch value from a packet field (including the metadata scratch itself). Width is specified by the descriptor.

> **Info**
>
> Refer to DOCA API documentation for details on `struct doca_flow_meta`.

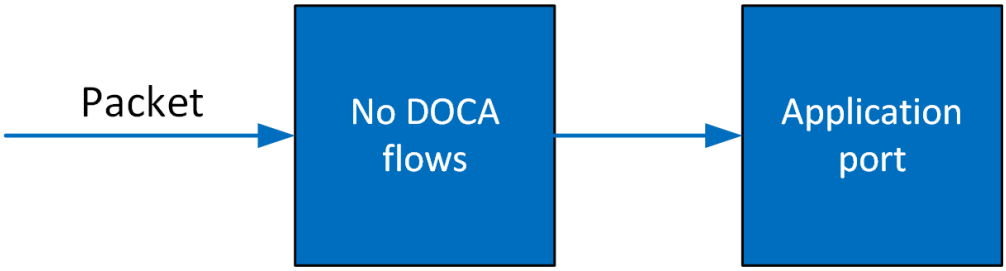
Some DOCA pipe types (or actions) use several bytes in the scratch area for internal usage. So, if the user has set these bytes in PIPE-1 and read them in PIPE-2, and between PIPE-1 and PIPE-2 there is PIPE-A which also uses these bytes for internal purpose, then these bytes are overwritten by the PIPE-A. This must be considered when designing the pipe tree.

The bytes used in the scratch area are presented by pipe type in the following table:

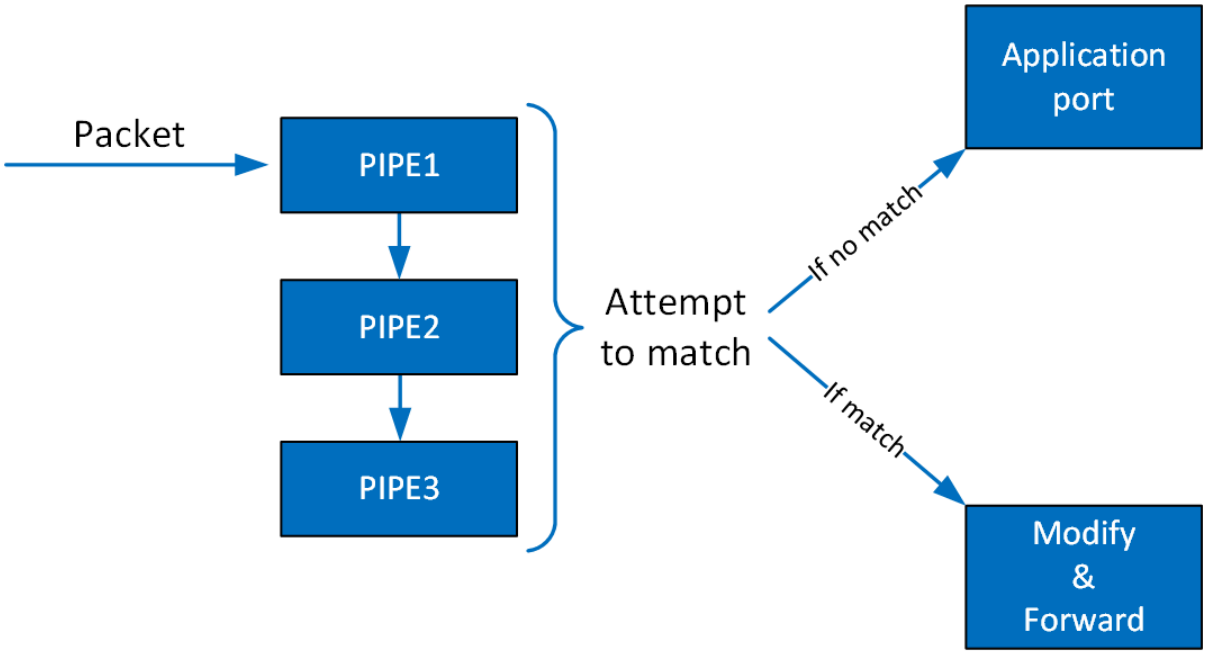
Pipe Type/Action	Bytes Used in Scratch
ordered_list	[0, 1, 2, 3]
LPM	[0, 1, 2, 3]
LPM EM	[0, 1, 2, 3, 4, 5, 6, 7]
Mirror	[0, 1, 2, 3]
ACL	[0, 1, 2, 3, 4, 5, 6, 7, 8 ,9, 10, 11, 12, 13, 14, 15]
Fwd from ingress to egress	[0, 1, 2, 3]
IPsec	[0, 1, 2, 3, 4, 5, 6, 7]
METER COLOR	[24]

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.

Debug and Trace Features

DOCA Flow supports trace and debugging of DOCA Flow applications which enable collecting predefined internal key performance indicators (KPIs) and pipeline visualization.

Installation

The set of DOCA's SDK development packages include also a developer-oriented package that includes additional trace and debug features which are not included in the production libraries:

- `.deb` based systems – `libdoca-sdk-flow-trace`
- `.rpm` based systems – `doca-sdk-flow-trace`

These packages install the trace-version of the libraries under the following directories:

- `.deb` based systems – `/opt/mellanox/doca/lib/<arch>/trace`
- `.rpm` based systems – `/opt/mellanox/doca/lib64/trace`

Using Trace Libraries

Runtime Linking

The trace libraries are designed to allow a user to link their existing (production) program to the trace library without needing to recompile the program. To do so, one should simply update the matching environment variable so that the OS will prioritize loading libraries from the above trace directory.

The following is an example for such an update for the Ubuntu 22.04 BlueField image:

```
LD_LIBRARY_PATH=/opt/mellanox/doca/lib/aarch64-linux-gnu/trace:${LD_LIBRARY_PATH} doca_ipsec_security_gw <program parameters>
```

Compilation

The trace-level development packages in the previous section provide additional compilation definitions (`doca-flow-trace.pc`) to be used in addition to the regular compilation definitions for the DOCA Flow SDK library (`doca-flow.pc`). It is recommended to use these compilation definitions for the following scenarios:

- Static linking of the trace-level DOCA Flow library into your program
- Regular (dynamic) linking of the trace-level DOCA Flow library into your program during development and testing

Although the latter could also be determined at runtime as explained in the previous section, many developers find it handy to compile directly against the trace version during initial development phases.

Trace Features

DOCA Log – Trace Level

DOCA's trace logging level (`DOCA_LOG_LEVEL_TRACE`) is compiled as part of this trace version of the library. That is, any program compiled against the library can activate this additional logging level through DOCA's API or even through DOCA's built-in argument parsing (ARGP) library:

```
LD_LIBRARY_PATH=/opt/mellanox/doca/lib/aarch64-linux-
gnu/trace:${LD_LIBRARY_PATH} doca_ipsec_security_gw <program
parameters> --sdk-log-level 70
```

DOCA Flow – Additional Sanity Checks

When using the trace version of the library, additional input sanitation checks are added, at the cost of introducing minor performance implications. These checks are meant to assist developers in their early steps of using the library, as they provide early detection and improved logging for common coding mistakes.

DOCA Flow Samples

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

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

Sample Prerequisites

A DOCA Flow-based program can either run on the host machine or on the BlueField.

Flow-based programs require an allocation of huge pages, hence the following commands are required:

```
$ echo '1024' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-
2048kB/nr_hugepages
$ sudo mkdir /mnt/huge
$ sudo mount -t hugetlbfs -o pagesize=2M nodev /mnt/huge
```

Running the Sample

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

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

> **Note**
>
> The binary `doca_<sample_name>` will be created under `/tmp/build/`.

3. Sample (e.g., `flow_aging`) usage:

```
Usage: doca_flow_aging [DPDK Flags] -- [DOCA Flags]

DOCA Flags:
  -h, --help                                Print a help synopsis
  -v, --version                             Print program version information
  -l, --log-level                           Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                           Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                         Parse all command flags from an input json file
```

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

```
/tmp/build/doca_<sample_name> -- -h
```

5. DOCA Flow samples are based on DPDK libraries. Therefore, the user is required to provide DPDK flags. The following is an example from an execution on the DPU:

   - CLI example for running the samples with "vnf" mode:

     ```
     /tmp/build/doca_<sample_name> -a
     auxiliary:mlx5_core.sf.2 -a auxiliary:mlx5_core.sf.3 --
     -l 60
     ```

   - CLI example for running the VNF samples with `vnf,hws` mode:

     ```
     /tmp/build/doca_<sample_name> -a
     auxiliary:mlx5_core.sf.2,dv_flow_en=2 -a
     auxiliary:mlx5_core.sf.3,dv_flow_en=2 -- -l 60
     ```

   - CLI example for running the switch samples with `switch,hws` mode:

     ```
     /tmp/build/doca_<sample_name> -- -p 03:00.0 -r sf[2-3] -
     l 60
     ```

     Note

     When running on the BlueField with `switch,hws` mode ,
     it is not necessary to configure the OVS.

     DOCA switch sample hides the extra
     `fdb_def_rule_en=0,vport_match=1,repr_matching_en=0,dv_x`

DPDK devargs with a simple `-p` and `-r` to specify the PCIe ID and representor information.

Note

When running on the DPU using the command above, sub-functions must be enabled according to the BlueField Scalable Function User Guide.

Note

When running on the host, virtual functions must be used according to the instructions in the DOCA Virtual Functions User Guide.

Samples

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`.
2. Starting two DOCA Flow port.
3. On each port:
    1. Building a pipe with changeable 5-tuple match and forward port action.

2. Adding 10 entries with different 5-tuple match, a monitor with different aging time (5-60 seconds), and setting user data in the monitor. The user data will contain the port ID, entry number, and entry pointer.

4. Handling aging every 5 seconds and removing each entry after age-out.

5. Running these commands until all entries age out.

Reference:

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

Flow ACL

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

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    1. Building an ACL pipe that matches changeable:
        1. Source IPv4 address
        2. Destination IPv4 address
        3. Source port
        4. Destination port
    2. Adding four example 5-tuple entries:
        1. The first entry with:

- Full mask on source IPv4 address
- Full mask on destination IPv4 address
- Null mask on source port (any source port)
- Null mask on destination port (any destination port)
- TCP protocol
- Priority 10
- Action "deny" (drop action)

2. The second entry with:
    - Full mask on source IPv4 address
    - Full mask on destination IPv4 address
    - Null mask on source port (any source port)
    - Value set in mask on destination port is used as part of port range:
        - Destination port in match is used as port from
        - Destination port in mask is used as port to
    - UDP protocol
    - Priority 50
    - Action "allow" (forward port action)
3. The third entry with:
    - Full mask on source IPv4 address
    - Full mask on destination IPv4 address
    - Value set in mask on source port is equal to the source port in match. It is the exact port. ACL uses the port with full mask.
    - Null mask on destination port (any destination port)

- TCP protocol
- Priority 40
- Action "allow" (forward port action)

4. The fourth entry with:
   - 24-bit mask on source IPv4 address
   - 24-bit mask on destination IPv4 address
   - Value set in mask on source port is used as part of port range : source port in match is used as port from, source port in mask is used as port to.
   - Value set in mask on destination port is equal to the destination port in match. It is the exact port. ACL uses the port with full mask.
   - TCP protocol
   - Priority 20
   - Action "allow" (forward port action)

3. The sample shows how to run the ACL pipe on ingress and egress domains. To change the domain, use the global parameter `flow_acl_sample.c`.
   1. Ingress domain: ACL is created as root pipe
   2. Egress domain:
      - Building a control pipe with one entry that forwards the IPv4 traffic hairpin port.
      - ACL is created as a root pipe on the hairpin port.

Reference:

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

Flow Aging

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

The sample logic includes:

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

Reference:

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

Flow Control Pipe

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

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:

    1. Building VXLAN pipe with match on VNI field, decap action, action descriptor for decap, and forwarding the matched packets to the second port.

    2. Building VXLAN-GPE pipe with match on VNI, flags and next protocol fields, and forwarding the matched packets to the second port.

    3. Building GRE pipe with match on GRE key field, decap and build eth header actions, action descriptor for decap, and forwarding the matched packets to the second port.

    4. Building NVGRE pipe with match on protocol is 0x6558, `vs_id`, `flow_id`, and inner UDP source port fields, and forwarding the matched packets to the second port. This pipe has a higher priority than the GRE pipe. The NVGRE packets are matched first.

    5. Building MPLS pipe with match on third MPLS label field, decap and build eth header actions, action descriptor for decap, and forwarding the matched packets to the second port.

    6. Building a control pipe with the following entries:

        - If L4 type is UDP and destination port is 4789, forward to VXLAN pipe
        - If L4 type is UDP and destination port is 4790, forward to VXLAN-GPE pipe
        - If L4 type is UDP and destination port is 6635, forward to MPLS pipe
        - If tunnel type and L4 type is GRE, forward to GRE pipe

Note

When any tunnel is decapped, it is user responsibility to identify if it is an L2 or L3 tunnel within the action. If the tunnel is L3, the complete outer layer, tunnel, and inner L2 are removed and the inner L3 layer is

exposed. To keep the packet valid, the user should provide the ETH header to encap the inner packet. For example:

```
actions.decap_type =
DOCA_FLOW_RESOURCE_TYPE_NON_SHARED;
actions.decap_cfg.is_l2 = false;
/* append eth header after decap GRE tunnel */
SET_MAC_ADDR(actions.decap_cfg.eth.src_mac,
src_mac[0], src_mac[1], src_mac[2], src_mac[3],
src_mac[4], src_mac[5]);
SET_MAC_ADDR(actions.decap_cfg.eth.dst_mac,
dst_mac[0], dst_mac[1], dst_mac[2], dst_mac[3],
dst_mac[4], dst_mac[5]);
actions.decap_cfg.eth.type = DOCA_FLOW_L3_TYPE_IP4;
```

For a VXLAN tunnel, since VXLAN is a L2 tunnel, the user must indicate it within the action:

```
actions.decap_type =
DOCA_FLOW_RESOURCE_TYPE_NON_SHARED;
actions.decap_cfg.is_l2 = true;
```

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_control_pipe/flow_contr`
- `/opt/mellanox/doca/samples/doca_flow/flow_control_pipe/flow_contr`
- `/opt/mellanox/doca/samples/doca_flow/flow_control_pipe/meson.buil`

Flow Copy to Meta

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

The sample logic includes:

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

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_copy_to_meta/flow_copy_`
- `/opt/mellanox/doca/samples/doca_flow/flow_copy_to_meta/flow_copy_`
- `/opt/mellanox/doca/samples/doca_flow/flow_copy_to_meta/meson.buil`

Flow Add to Metadata

This sample shows how to use the DOCA Flow add-to-metadata action to accumulate the source IPv4 address for double to meta and then match on the meta.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:

1. Building a pipe with changeable match on `meta_data` and forwarding the matched packets to the second port.

2. Adding an entry that matches an example double of source IPv4 address that has been added to metadata.

3. Building a pipe with changeable 5-tuple match, copying the source IPv4, and adding the value again to the meta action, and forwarding to the first pipe.

4. Adding an example 5-tuple entry to the pipe.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_add_to_meta/flow_add_tc`
- `/opt/mellanox/doca/samples/doca_flow/flow_add_to_meta/flow_add_tc`
- `/opt/mellanox/doca/samples/doca_flow/flow_add_to_meta/meson.build`

Flow Drop

This sample illustrates how to build a pipe with 5-tuple match, forward action drop, and forward miss action to the hairpin pipe. The sample also demonstrates how to dump pipe information to a file and query entry.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:

    1. Building a hairpin pipe with an entry that matches all traffic and forwarding traffic to the second port.

2. Building a pipe with a changeable 5-tuple match, forwarding action drop, and miss forward to the hairpin pipe. This pipe serves as a root pipe.

3. Adding an example 5-tuple entry to the drop pipe with a counter as monitor to query the entry later.

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

5. Dumping the pipe information to a file.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_drop/flow_drop_sample_s`
- `/opt/mellanox/doca/samples/doca_flow/flow_drop/flow_drop_sample_m`
- `/opt/mellanox/doca/samples/doca_flow/flow_drop/meson.build`

Flow ECMP

This sample illustrates ECMP feature using a hash pipe.

The sample enables users to determine how many port are included in ECMP distribution:

- The number of ports, `n`, is determined by DPDK device argument `representor=sf[0-m]` where `m=n-1`.
- CLI example for running this samples with `n=4` ports:

```
/tmp/build/doca_flow_ecmp -- -p 03:00.0 -r sf[0-3] -l 60 --
sdk-log-level 60
```

- `n` should be power of 2. Max supported value is `n=8`.

The sample logic includes:

1. Calculate the number of SF representors (`n`) created by DPDK according to user input.

2. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` structure.
3. Starting DOCA Flow ports: Physical port and `n` SF representors.
4. On switch port:
    1. Constructing a hash pipe that signifies the `match_mask` structure to compute the hash based on the outer IPv6 flow label field.
    2. Adding `n` entries to the created pipe, each of which forwards packets to a different port representor.
5. Waiting 15 seconds and querying the entries.
6. Print the ECMP results per port (number packets in each port related to total packets).

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ecmp/flow_ecmp_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ecmp/flow_ecmp_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ecmp/meson.build`

Flow ESP

This sample illustrates how to match match ESP fields in two ways:

- Exact match for both `esp_spi` and `esp_en` fields using the `doca_flow_match` structure.
- Comparison match for `esp_en` field using the `doca_flow_match_condition` structure.

> **Note**

This sample is supported for ConnectX-7, BlueField-3, and above.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    1. Building a control pipe with entry that match `esp_en > 3` (GT pipe).
    2. Building a control pipe with entry that match `esp_en < 3` (LT pipe).
    3. Building a root pipe with changeable `next_pipe` FWD and `esp_spi` match along with specific `esp_sn` match + IPv4 and ESP exitance (matching `parser_meta`).
    4. Adding example `esp_spi = 8` entry to the root pipe which forwards to GT pipe (and miss condition).
    5. Adding example `esp_spi = 5` entry to the root pipe which forwards to LT pipe (and hit condition).

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_esp/flow_esp_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_esp/flow_esp_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_esp/meson.build`

Flow Forward Miss

The sample illustrates how to use FWD miss query and update with or without miss counter.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    1. Building a copy pipe with a changeable outer L3 type match and forwarding traffic to the second port.
    2. Add entries doing different copy action depending on the outer L3 type:
        1. `IPv4` – copy IHL field into Type Of Service field.
        2. `IPv6` – copy Payload Length field into Traffic Class field.
    3. Building a pipe with a IPv4 addresses match, forwarding traffic to the second port, and miss forward to the copy pipe.
    4. Building an IP selector pipe with outer L3 type match, forwarding IPv4 traffic to IPv4 pipe, and miss forward to the copy pipe with miss counter.
    5. Building a root pipe with outer L3 type match, forwarding IPv4 and IPv6 traffic to IP selector pipe, and dropping all other traffic by miss forward with miss counter.
4. Waiting 5 seconds for first batch of traffic.
5. On each port:
    1. Querying the miss counters using `doca_flow_query_pipe_miss` API.
    2. Printing the miss results.
6. On each port:
    1. Building a push pipe that pushes VLAN header and forwarding traffic to the second port.
    2. Updating both IP selector and IPv4 pipes miss FWD pipe target to push pipe using `doca_flow_pipe_update_miss` API.

7. Waiting 5 seconds for second batch of traffic, same flow as before.
8. On each port:
    1. Querying again the miss counters using `doca_flow_query_pipe_miss` API.
    2. Printing the miss results again, the results should include miss packets coming either before or after miss action updating.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_fwd_miss/flow_fwd_miss_`
- `/opt/mellanox/doca/samples/doca_flow/flow_fwd_miss/flow_fwd_miss_`
- `/opt/mellanox/doca/samples/doca_flow/flow_fwd_miss/meson.build`

Flow Forward Target (DOCA_FLOW_TARGET_KERNEL)

The sample illustrates how to use `DOCA_FLOW_FWD_TARGET` type of forward, as well as the `doca_flow_get_target` API to obtain an instance of `struct doca_flow_target`.

The sample logic includes:

1. Initializing DOCA Flow with `"vnf,isolated,hws"`.
2. Initializing two ports.
3. Obtaining an instance of `doca_flow_target` by calling `doca_flow_get_target(DOCA_FLOW_TARGET_KERNEL, &kernel_target);`.
4. On each port, creating:
    1. Non-root basic pipe with 5 tuple match.
        1. If hit – forward the packet to another port.
        2. If miss – forward the packet to the kernel for processing by using the instance of `doca_flow_target` obtained in previous steps.

3. Then add a single entry with a specific 5-tuple which is hit, and the rest is forwarded to the kernel.

2. Root control pipe with a match on outer L3 type being IPv4.

    1. If hit – forward the packet to the non-root pipe.

    2. If miss – drop the packet.

    3. Add a single entry that implements the logic described.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_fwd_target/flow_fwd_tar`
- `/opt/mellanox/doca/samples/doca_flow/flow_fwd_target/flow_fwd_tar`
- `/opt/mellanox/doca/samples/doca_flow/flow_fwd_target/meson.build`

Flow GENEVE Encap

This sample illustrates how to use DOCA Flow actions to create a GENEVE tunnel.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:

    1. Building ingress pipe with changeable 5-tuple match, copying to `pkt_meta` action, and forwarding port action.

    2. Building egress pipe with `pkt_meta` match and 4 different encapsulation actions:

        - L2 encap without options

        - L2 encap with options

        - L3 encap without options
        - L3 encap with options
    3. Adding example 5-tuple and encapsulation values entries to the pipes.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_geneve_encap/flow_genev`
- `/opt/mellanox/doca/samples/doca_flow/flow_geneve_encap/flow_genev`
- `/opt/mellanox/doca/samples/doca_flow/flow_geneve_encap/meson.buil`

Flow GENEVE Options

This sample illustrates how to prepare a GENEVE options parser, match on configured options, and decap GENEVE tunnel.

> **Note**
>
> This sample works only with PF. VFs and SFs are not supported.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    1. Building GENEVE options parser, same input for all ports.
    2. Building match pipe with GENEVE VNI and options match and forwards decap pipe.

3. Building decap pipe with more GENEVE options match, and 2 different decapsulation actions:
    - L2 decap
    - L3 decap with changeable mac addresses
4. Adding example GENEVE options and MAC address values entries to the pipes.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_geneve_opt/flow_geneve_`
- `/opt/mellanox/doca/samples/doca_flow/flow_geneve_opt/flow_geneve_`
- `/opt/mellanox/doca/samples/doca_flow/flow_geneve_opt/meson.build`

Flow GTP

This sample demonstrates how to use DOCA Flow to process and modify GTP PSC packets with specific QFI values.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. Creating two pipes on each port:
    1. The first pipe:
        1. Matches GTP PSC packets.
        2. Forwards only GTP PSC packets to the second pipe.
        3. Drops all other packets.
    2. The second pipe:
        1. Matches packets with QFI equal to `0x1`.

2. Modifies these packets, changing the QFI value from `0x1` to `0x3a`.
3. Hairpins the modified packets to the other port.
4. Drops all other packets.

Sample result: Only GTP PSC packets with an initial QFI value of `0x1` are transmitted to the other port, where their QFI value has been updated to `0x3a`. All other packets are dropped.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_gtp/flow_gtp_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_gtp/flow_gtp_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_gtp/meson.build`

Flow GTP Encap

This sample illustrates how to use DOCA Flow actions to create a GTP tunnel.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    1. Building ingress pipe with IPv4 match, and forwarding port action.
    2. Building egress pipe with GTP encapsulation action. `gtp_teid`, `gtp_ext_psc_qfi`, eth and IPv4 fields are changeable.
    3. Adding example encapsulation values entries to the pipes.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_gtp_encap/flow_gtp_enca`
- `/opt/mellanox/doca/samples/doca_flow/flow_gtp_encap/flow_gtp_enca`
- `/opt/mellanox/doca/samples/doca_flow/flow_gtp_encap/meson.build`

Flow Hairpin VNF

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

The sample logic includes:

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

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_hairpin_vnf/flow_hairpi`
- `/opt/mellanox/doca/samples/doca_flow/flow_hairpin_vnf/flow_hairpi`
- `/opt/mellanox/doca/samples/doca_flow/flow_hairpin_vnf/meson.build`

Flow Switch to Wire

This sample illustrates how to build a pipe with 5-tuple match and forward packets from the wire back to the wire.

The sample shows how to build a basic pipe in a switch and hardware steering (HWS) mode. Each pipe contains two entries, each of which forwards matched packets to two

different representors.

The sample also demonstrates how to obtain the switch port of a given port using `doca_flow_port_switch_get()`.

> **Note**
>
> The test requires one PF with three representors (either VFs or SFs).

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct.
2. Starting DOCA Flow ports with `doca_dev` in `struct doca_flow_port_cfg`.
3. On the switch's PF port:
    1. Building ingress, egress, vport, and RSS pipes with changeable 5-tuple match and forwarding port action.
    2. Adding example 5-tuple entry to the pipe.
    3. The matched traffic goes to its destination port, the missed traffic is handled by the `rx_tx` function and is sent to a dedicate port based on the protocol.
        - Ingress pipe:

```
Entry 0: IP src 1.2.3.4 / TCP src 1234 dst 80 -> egress
pipe
Entry 1: IP src 1.2.3.5 / TCP src 1234 dst 80 -> vport
pipe
```

        - Egress pipe (test ingress to egress cross domain):

```
Entry 0: IP dst 8.8.8.8 / TCP src 1234 dst 80 -> port 0
Entry 1: IP dst 8.8.8.9 / TCP src 1234 dst 80 -> port 1
Entry 2: IP dst 8.8.8.10 / TCP src 1234 dst 80 -> port 2
Entry 3: IP dst 8.8.8.11 / TCP src 1234 dst 80 -> port 3
```

- Vport pipe (test ingress direct to vport):

```
Entry 0: IP dst 8.8.8.8 / TCP src 1234 -> port 0
Entry 1: IP dst 8.8.8.9 / TCP src 1234 -> port 1
Entry 2: IP dst 8.8.8.10 / TCP src 1234-> port 2
Entry 3: IP dst 8.8.8.11 / TCP src 1234-> port 3
```

- RSS pipe (test miss traffic `port_id` get and destination `port_id` set):

```
Entry 0: IPv4 / TCP -> port 0
Entry 0: IPv4 / UDP -> port 1
Entry 0: IPv4 / ICMP -> port 2
```

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_switch_to_wire/flow_swi`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_to_wire/flow_swi`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_to_wire/meson.bu`

Flow Hash Pipe

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

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

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct.
2. Starting DOCA Flow ports: Physical port and two SF representors.
3. On switch port:
    1. Building a hash pipe while indicating which fields to use to calculate the hash in the `struct match_mask`.
    2. Adding two entries to the created pipe, each of which forwards packets to a different port representor.
4. Printing the hash result calculated by the software with the following message: `"hash value for" for dest ip = 192.168.1.1`.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_hash_pipe/flow_hash_pip`
- `/opt/mellanox/doca/samples/doca_flow/flow_hash_pipe/flow_hash_pip`
- `/opt/mellanox/doca/samples/doca_flow/flow_hash_pipe/meson.build`

Flow IPv6 Flow Label

This sample shows how to use DOCA Flow actions to update IPv6 flow label field after encapsulation.

As a side effect, it shows also example for IPv6 + MPLS encapsulation.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:

    1. Building an ingress pipe with changeable L4 type and ports matching, which updates metadata and goes to the peer port.

    2. Adding example UDP/TCP type and ports and metadata values entries to the pipe. This pipe is L3 type agnostic.

    3. Building an egress pipe on the peer port with changeable metadata matching, which encapsulates packets with IPv6 + MPLS headers, and goes to the next pipe.

    4. Adding entries to the pipe, with different encapsulation values for different metadata values.

    5. Building another egress pipe on the peer port with changeable L3 inner type matching, which copies value into outer IPv6 flow label field.

    6. Adding two entries to the pipe:

        1. L3 inner type is IPv6 - copy IPv6 flow label from inner to outer.

        2. L3 inner type is IPv6 - copy outer IPv6 flow label from metadata.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ipv6_flow_label/flow_ip`
- `/opt/mellanox/doca/samples/doca_flow/flow_ipv6_flow_label/flow_ip`
- `/opt/mellanox/doca/samples/doca_flow/flow_ipv6_flow_label/meson.b`

Flow Loopback

This sample illustrates how to implement packet re-injection, or loopback, in VNF mode.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:

    1. Building a UDP pipe that matches a changeable source and destination IPv4 address, while the forwarding component is RSS to queues. Upon match, setting the packet meta on this UDP pipe which is referred to as an `RSS_UDP_IP` pipe.

    2. Adding one entry to the `RSS_UDP_IP` pipe that matches a packet with a specific source and destination IPv4 address and setting the meta to 10.

    3. Building a TCP pipe that matches changeable 4-tuple source and destination IPv4 and port addresses, while the forwarding component is RSS to queues (this pipe is called `RSS_TCP_IP` and it is the root pipe on ingress domain).

    4. Adding one entry to the `RSS_TCP_IP` pipe, that matches a packet with a specific source and destination port and IPv4 addresses.

    5. On the egress domain, creating the loopback pipe, which is root, and matching TCP over IPv4 with changeable 4-tuple source and destination port and IPv4 addresses, while encapsulating the matched packets with VXLAN tunneling and setting the destination and source MAC addresses to be changeable per entry.

    6. Adding one entry to the loopback pipe with specific values for the match and actions part while setting the destination MAC address to the port to which to inject the packet (in this case, it is the ingress port where the packet arrived).

    7. Starting to receive packets loop and printing the metadata

        - For packets that were re-injected, metadata equaling 10 is printed

        - Otherwise, 0 is be printed as metadata (indicating that it is the first time the packet has been encountered)

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_loopback/flow_loopback_`
- `/opt/mellanox/doca/samples/doca_flow/flow_loopback/flow_loopback_`
- `/opt/mellanox/doca/samples/doca_flow/flow_loopback/meson.build`

Flow Modify Header

This sample illustrates how to use DOCA Flow actions to modify the specific packet fields.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port, creating serial pipes and jumping to the next pipe if traffic is unmatched:
    1. Building a pipe with action `dec_ttl=true` and changeable `mod_dst_mac`. The pipe matches IPv4 traffic with a changeable destination IP and forwards the matched packets to the second port.
        - Adding an entry with an example destination IP (8.8.8.8) and `mod_dst_mac` value.
    2. Building a pipe with action-changeable `mod_vxlan_tun_rsvd1`. The pipe matches IPv4 traffic with a changeable UDP destination port and VXLAN-GPE tunnel ID then forwards the matched packets to the second port.
        - Adding an entry with an example VXLAN-GPE tunnel ID (100) and UDP destination port (4790), then `mod_vxlan_tun_rsvd1` value.
    3. Building a pipe with action-changeable `mod_vxlan_tun_rsvd1`. The pipe matches IPv4 traffic with a changeable UDP destination port and VXLAN tunnel ID then forwards the matched packets to the second port.
        - Adding an entry with an example VXLAN tunnel ID (100) and UDP destination port (4789), then `mod_vxlan_tun_rsvd1` value.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_modify_header/flow_modi`
- `/opt/mellanox/doca/samples/doca_flow/flow_modify_header/flow_modi`

- `/opt/mellanox/doca/samples/doca_flow/flow_modify_header/meson.bui`

Flow Monitor Meter

This sample illustrates how to use DOCA Flow monitor meter.

The sample logic includes:

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

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_monitor_meter/flow_moni`
- `/opt/mellanox/doca/samples/doca_flow/flow_monitor_meter/flow_moni`
- `/opt/mellanox/doca/samples/doca_flow/flow_monitor_meter/meson.bui`

Flow Multi-actions

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

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.

3. On each port:
    1. Building a pipe with changeable source IP match which forwards the matched packets to the second port and sets different actions in the actions array:
        - Changeable modify source MAC address
        - Changeable modify source IP address
    2. Adding two entries to the pipe with different source IP match:
        1. The first entry with an example modify source MAC address.
        2. The second with a modify source IP address.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_multi_actions/flow_mult`
- `/opt/mellanox/doca/samples/doca_flow/flow_multi_actions/flow_mult`
- `/opt/mellanox/doca/samples/doca_flow/flow_multi_actions/meson.bui`

Flow Multi-fwd

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

The sample logic includes:

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

        - The first entry with forward to the second port
        - The second with drop

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_multi_fwd/flow_multi_fw`
- `/opt/mellanox/doca/samples/doca_flow/flow_multi_fwd/flow_multi_fw`
- `/opt/mellanox/doca/samples/doca_flow/flow_multi_fwd/meson.build`

Flow Parser Meta

This sample shows how to use some of `match.parser_meta` fields from 3 families:

- IP fragmentation – matching on whether a packet is IP fragmented
- Integrity bits – matching on whether a specific protocol is OK (length, checksum etc.)
- Packet types – matching on a specific layer packet type

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    1. Building a root pipe with outer IP fragmentation match:
        - If a packet is IP fragmented – forward it to the second port regardless of next pipes in the pipeline
        - If a packet is not IP fragmented – proceed with the the pipeline by forwarding it to integrity pipe
    2. Building an "integrity" pipe with a single entry which continues to the next pipe when:

        - The outer IPv4 checksum is OK
        - The inner L3 is OK (incorrect length should be dropped)
    3. Building a "packet type" pipe which forwards packets to the second port when:
        - The outer L3 type is IPv4
        - The inner L4 type is either TCP or UDP
4. Waiting 5 seconds for traffic to arrive.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_parser_meta/flow_parser`
- `/opt/mellanox/doca/samples/doca_flow/flow_parser_meta/flow_parser`
- `/opt/mellanox/doca/samples/doca_flow/flow_parser_meta/meson.build`

Flow Random

This sample shows how to use `match.parser_meta.random` field for 2 different use-cases:

- Sampling – sampling certain percentage of traffic regardless of flow content
- Distribution – distributing traffic in 8 different queues

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    1. Building a root pipe with changeable 5-tuple match and forwarding to specific use-case pipe according to changeable source IP address.

2. Adding two entries to the pipe with different source IP match, and different forward:
    - The first entry with forward to the sampling pipe.
    - The second entry with forward to the distribution pipe.
3. Building a "sampling" pipe with a single entry and preparing the entry to sample 12.5% of traffic.
4. Building a "distribution" hash pipe with 8 entries and preparing the entries to get 12.5% of traffic for each queue.

4. Waiting 15 seconds and querying the entries (total packets after sampling/distribution related to total packets before).

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_random/flow_random_samp`
- `/opt/mellanox/doca/samples/doca_flow/flow_random/flow_random_main`
- `/opt/mellanox/doca/samples/doca_flow/flow_random/meson.build`

Flow RSS ESP

This sample shows how to use DOCA Flow forward RSS according to ESP SPI field, and distribute the traffic between queues.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    1. Building a pipe with both L3 and L4 types match, copy the `SPI` field into packet meta data, and forwarding to RSS with 7 queues.

2. Adding an entry with both IPv4 and ESP existence matching.

4. Waiting 15 seconds for traffic to arrived.

5. On each port:

    1. Calculates the traffic percentage distributed into each port and prints the result.

    2. Printing for each packet its `SPI` value. (only in debug mode, `-l ≥ 60`)

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_rss_esp/flow_rss_esp_sa`
- `/opt/mellanox/doca/samples/doca_flow/flow_rss_esp/flow_rss_esp_ma`
- `/opt/mellanox/doca/samples/doca_flow/flow_rss_esp/meson.build`

Flow RSS Meta

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

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:

    1. Building a pipe with a changeable 5-tuple match, forwarding to RSS queue with index 0, and setting changeable packet meta data.

    2. Adding an entry with an example 5-tuple and metadata value to the pipe.

4. Retrieving the packets on both ports from a receive queue , and printing the packet metadata value.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_rss_meta/flow_rss_meta_`
- `/opt/mellanox/doca/samples/doca_flow/flow_rss_meta/flow_rss_meta_`
- `/opt/mellanox/doca/samples/doca_flow/flow_rss_meta/meson.build`

Flow Sampling

This sample shows how to sample certain percentage of traffic regardless of flow content using `doca_flow_match_condition` structure with `parser_meta.random.value` field string.

> **Note**
>
> This sample is supported for ConnectX-7/BlueField-3 and above.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct.
2. Starting DOCA Flow ports: Physical port and two SF representors.
3. On switch port:
    1. Building a root pipe with changeable 5-tuple match and forwarding to sampling pipe.
    2. Adding entry with an example 5-tuple to the pipe.
    3. Building a "sampling" control pipe with a single entry.
    4. calculating the requested random value for getting 35% of traffic.
    5. Adding entry with an example condition random value to the pipe.

4. Waiting 15 seconds and querying the entries (total packets after sampling related to total packets before).

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_sampling/flow_sampling_`
- `/opt/mellanox/doca/samples/doca_flow/flow_sampling/flow_sampling_`
- `/opt/mellanox/doca/samples/doca_flow/flow_sampling/meson.build`

Flow Set Meta

This sample shows how to use the DOCA Flow set metadata action and then match on it.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:

1. Building a pipe with a changeable match on metadata and forwarding the matched packets to the second port.
2. Adding an entry that matches an example metadata value.
3. Building a pipe with changeable 5-tuple match, changeable metadata action, and fwd to the first pipe.
4. Adding entry with an example 5-tuple and metadata value to the pipe.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_set_meta/flow_set_meta_`
- `/opt/mellanox/doca/samples/doca_flow/flow_set_meta/flow_set_meta_`
- `/opt/mellanox/doca/samples/doca_flow/flow_set_meta/meson.build`

Flow Shared Counter

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

The sample logic includes:

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

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_shared_counter/flow_sha`
- `/opt/mellanox/doca/samples/doca_flow/flow_shared_counter/flow_sha`

- `/opt/mellanox/doca/samples/doca_flow/flow_shared_counter/meson.bu`

Flow Shared Meter

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

The sample logic includes:

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

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_shared_meter/flow_share`

- `/opt/mellanox/doca/samples/doca_flow/flow_shared_meter/flow_share`
- `/opt/mellanox/doca/samples/doca_flow/flow_shared_meter/meson.buil`

Flow Switch Control Pipe

This sample shows how to use the DOCA Flow control pipe in switch mode.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    1. Building control pipe with match on VNI field.
    2. Adding two entries to the control pipe, both matching TRANSPORT (UDP or TCP proto) over IPv4 with source port 80 and forwarding to the other port, where the first entry matches destination port 1234 and the second 12345.
    3. Both entries have counters, so that after the successful insertions of both entries, the sample queries those counters to check the number of matched packets per entry.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_switch_control_pipe/flo`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_control_pipe/flo`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_control_pipe/mes`

Flow Switch – Multiple Switches

This sample illustrates how to use two switches working concurrently on two different physical functions.

It shows how to build a basic pipe in a switch and hardware steering (HWS) mode. Each pipe contains two entries, each of which forwards matched packets to two different representors.

The sample also demonstrates how to obtain the switch port of a given port using `doca_flow_port_switch_get()`.

> **Note**
>
> The test requires two PFs with two (either VF or SF) representors on each.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct.
2. Starting DOCA Flow ports: Two physical ports and two representors each (totaling six ports).
3. On the switch port:
    1. Building a basic pipe while indicating which fields to match on using `struct doca_flow_match match`.
    2. Adding two entries to the created pipe, each of which forwards packets to a different port representor.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_switch/flow_switch_samp`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch/flow_switch_main`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch/meson.build`

Flow Switch – Single Switch

This sample is identical to the previous sample, before the flow switch sample was extended to take advantage of the capabilities of DOCA to support multiple switches concurrently, each based on a different physical device.

The reason we add this original version is that it removes the constraints imposed by the modified flow switch version, allowing to use arbitrary number of representors in the switch configuration.

The logic of this sample is identical to that of the previous sample with 2 new pipes.

- A user RSS pipe which receives the packets which missed TC rules (in the kernel domain in this case)
- A simple pipe forwarding packets to kernel domain by using `DOCA_FLOW_FWD_TARGET`

In the `to_kernel_pipe`, all the IPv4 packets are forwarded to the kernel (i.e., entry 0 in `to_kernel_pipe`). In the kernel domain, all the IPv4 packets are missed to the NIC domain if there is no TC rule. In the NIC domain, the IPv4 packets missed from the NIC domain are forwarded to slow path (i.e., the representor of the PF/VF).

- Root pipe:

```
Entry 0: IP src 1.2.3.4 / dst 8.8.8.8 / TCP src 1234 dst 80 -> port 0
Entry 1: IP src 1.2.3.5 / dst 8.8.8.9 / TCP src 1234 dst 80 -> port 1
Miss: -> To kernel pipe
```

- To kernel pipe:

```
Entry 0: IPv4 -> send to kernel
IPv6 traffic would be dropped
```

- RSS pipe:

```
Entry 0: IPv4 -> port 0 rss queue 0
```

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_switch_single/flow_swit`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_single/flow_swit`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_single/meson.bui`

Flow Switch (Direction Info)

This sample illustrates how to give a hint to the driver for potential optimizations based on the direction information.

> **Info**
>
> This sample requires a single PF with two representors (either VF or SF).

The sample also demonstrates usage of the `match.parser_meta.port_meta` to detect by the switch pipe the source from where the packet has arrived.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct.
2. Starting 3 DOCA Flow ports, 1 physical port and 2 representors.
3. On the switch port:
    1. Network-to-host pipe:

1. Building basic pipe with a changeable `ipv4.next_proto` field and configuring the pipe with the hint of direction by setting `attr.dir_info = DOCA_FLOW_DIRECTION_NETWORK_TO_HOST`.

2. Adding two entries:

   - If `ipv4.next_proto` is TCP, the packet is forwarded to the first representor, to the host.

   - If `ipv4.next_proto` is UDP, the packet is forwarder to the second representor, to the host.

2. Host-to-network pipe:

   1. Building a basic pipe with a match on `aa:aa:aa:aa:aa:aa` as a source MAC address and configuring a pipe with the hint of direction by setting `attr.dir_info = DOCA_FLOW_DIRECTION_HOST_TO_NETWORK`.

   2. Adding an entry. If the source MAC is matched, forward the packet to the physical port (i.e., to the network).

3. Switch pipe:

   1. Building a basic pipe with a changeable `parser_meta.port_meta` to detect where the packet has arrived from.

   2. Adding 3 entries:

      - If the packet arrived from port 0 (i.e., the network), forward it to the network-to-host pipe to decide for further logic

      - If the packet arrived from port 1 (i.e., the host's first representor), forward it to the host-to-network pipe to decide for further logic

      - If the packet arrived from port 2, (i.e., the host's second representor), forward it to the host-to-network pipe to decide for further logic

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_switch_direction_info/f`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_direction_info/f`

- `/opt/mellanox/doca/samples/doca_flow/flow_switch_direction_info/m`

Flow Switch Hot Upgrade

This sample demonstrates how to use the port operation state mechanism for a hot upgrade use case. It shows how to configure the state of a port during initialization and how to modify the state after the port has already been started.

Prerequisites

The test requires two physical functions (PFs) with two (either VFs or SFs) representors on each.

Command-line Arguments

The sample allows users to specify the operation state of the instance using the `--state <value>` argument. The relevant values are:

- `0` for `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE`
- `1` for `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE_READY_TO_SWAP`
- `2` for `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY`

Sample Logic

1. Initialize DOCA Flow:
    - Indicate `mode_args="switch"` in the `doca_flow_cfg` structure.
2. Start DOCA Flow ports:
    - Two physical ports and two representors each (totaling six ports) are started.
    - Both switch ports are configured with `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED` state.

3. Configure each switch port:

    1. Build a basic pipe with a miss counter matching on outer L3 type (specific IPv4) and outer L4 type (changeable).

    2. Add two entries to the created pipe with counters, each forwarding packets to a different port representor.

    3. Modify the port operation state from `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED` to the required state.

4. Traffic handling:

    - Wait for traffic until a SIGQUIT signal (Ctrl+) is received.

    - While traffic is being received, traffic statistics are printed to stdout.

Hot Upgrade Use Case

To illustrate the hot upgrade use case, follow these steps:

1. Create two different instances in separate windows with different states.

> **Note**
>
> DPDK prevents users from creating two primary instances. To avoid this limitation, use the `--file-prefix` EAL argument.
>
> - Example for the "active" instance:
>
> ```
> /tmp/build/samples/doca_flow_switch_hot_upgrade
> -- -p 08:00.0 -p 08:00.1 -r vf[0-1] -r
> vf[0-1] -l 70
> ```
>
> - Example for the "stand-by" instance:

```
/tmp/build/samples/doca_flow_switch_hot_upgrade
--file-prefix standby -- -p 08:00.0 -p
08:00.1 -r vf[0-1] -r vf[0-1] -l 70 --
state 2
```

2. Close the active process by typing Ctrl+\ while traffic is being received. The traffic statistics will start printing in the standby instance.

3. Restart the first instance. The traffic statistics will stop printing in the standby instance and start printing in the active instance again.

Swap Use Case

When both instances are running, the swap use case can be demonstrated by typing Ctrl+C:

- Typing Ctrl+C in the active instance changes its state to `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE_READY_TO_SWAP`
- Typing Ctrl+C in the standby instance changes its state to `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE`
- Typing Ctrl+C in the active instance again changes its state to `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY`

References

- `/opt/mellanox/doca/samples/doca_flow/flow_switch_hot_upgrade/flow`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_hot_upgrade/flow`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_hot_upgrade/mesc`

Flow VXLAN Encap

This sample shows how to use DOCA Flow actions to create a VXLAN/VXLANGPE/VXLANGBP tunnel as well as illustrating the usage of matching TCP and UDP packets in the same pipe.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    1. Building a pipe with changeable 5-tuple match, encap action, and forward port action.
    2. Adding example 5-tuple and encapsulation values entry to the pipe. Every TCP or UDP over IPv4 packet with the same 5-tuple is matched and encapsulated.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_vxlan_encap/flow_vxlan_`
- `/opt/mellanox/doca/samples/doca_flow/flow_vxlan_encap/flow_vxlan_`
- `/opt/mellanox/doca/samples/doca_flow/flow_vxlan_encap/meson.build`

Flow Shared Mirror

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

> **Note**

A current limitation does not allow using shared mirror IDs bearing the value zero.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    1. Configuring a shared mirror with a clone destination hairpin to the second port.
    2. Binding the shared mirror to the port.
    3. Building a pipe with a changeable 5-tuple match with UDP protocol, changeable shared mirror ID, and forwarding the matched packets to the second port.
    4. Adding an entry with an example 5-tuple match and shared mirror with ID= `port_id+1`.
    5. Building a pipe with a changeable 5-tuple match with TCP protocol, changeable shared mirror ID, and forwarding the matched packets to the second port.
    6. Adding an entry with an example 5-tuple match and shared mirror with ID= `port_id+1`.
    7. Building a control pipe with the following entries:
        - If L4 type is UDP, forwards the packets to the UDP pipe
        - If L4 type is TCP, forwards the packets to the TCP pipe
    8. Waiting 15 seconds to clone any incoming traffic. Should see the same two packets received on the second port (one from the clone and another from the original).

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_shared_mirror/flow_shar`

- `/opt/mellanox/doca/samples/doca_flow/flow_shared_mirror/flow_shar`
- `/opt/mellanox/doca/samples/doca_flow/flow_shared_mirror/meson.bui`

Flow Match Comparison

This sample shows how to use the DOCA Flow match with a comparison result.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    1. Building a pipe with a changeable match on `meta_data[0]` and forwarding the matched packets to the second port.
    2. Adding an entry that matches on `meta_data[0]` equal with TCP header length.
    3. Building a control pipe for comparison purpose.
    4. Adding an entry to the control pipe match with comparison result the `meta_data[0]` value greater than `meta_data[1]` and forwarding the matched packets to match with the meta pipe.
    5. Building a pipe with a changeable 5-tuple match, copying `ipv4.total_len` to `meta_data[1]`, and accumulating `ipv4.version_ihl << 2` `tcp.data_offset << 2` to `meta_data[1]`, then forwarding to the second pipe.
    6. Adding an example 5-tuple entry to the pipe.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_match_comparison/flow_m`

- `/opt/mellanox/doca/samples/doca_flow/flow_match_comparison/flow_m`
- `/opt/mellanox/doca/samples/doca_flow/flow_match_comparison/meson.`

Flow Entropy

This sample shows how to use the DOCA Flow entropy calculation.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct.
2. Starting one DOCA Flow port.
3. Configuring the `doca_flow_entropy_format` structure with 5-tuple values.
4. Calling to `doca_flow_port_calc_entropy` to get the calculated entropy.
5. Logging the calculated entropy.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_entropy/flow_entropy_sa`
- `/opt/mellanox/doca/samples/doca_flow/flow_entropy/flow_entropy_ma`
- `/opt/mellanox/doca/samples/doca_flow/flow_entropy/meson.build`

Flow VXLAN Shared Encap

This sample shows how to use DOCA Flow actions to create a VXLAN tunnel as well as illustrating the usage of matching TCP and UDP packets in the same pipe.

The VXLAN tunnel is created by `shared_resource_encap`.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
    1. Configure and bind shared encap resources. The encap resources are for VXLAN encap.
    2. Building a pipe with changeable 5-tuple match, `shared_encap_id`, and forward port action.
    3. Adding example 5-tuple and encapsulation values entry to the pipe. Every TCP or UDP over IPv4 packet with the same 5-tuple is matched and encapsulated.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_vxlan_shared_encap/flow`
- `/opt/mellanox/doca/samples/doca_flow/flow_vxlan_shared_encap/flow`
- `/opt/mellanox/doca/samples/doca_flow/flow_vxlan_shared_encap/mesc`

Flow Switch RSS

This sample shows DOCA Flow switch RSS creation, and verifies switch RSS feature with shared/immediate/internal hairpin RSS and packet handling.

> **Note**
>
> Only 512 different immediate RSS actions are supported in DOCA Flow.

> **Info**

The shared RSS is created by `shared_resource_rss`.

Info

To cover MPESW mode P0/P1/VF, 3 ports are designed to be used.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. Configuring and binding shared resources.
4. Creating shared/immediate RSS pipes both in default and egress domains and adding entries to check both domain switch RSS creation.
5. Creating a forward-to-port pipe and adding the entries to check internal wire-to-wire hairpin.
6. Sending the traffic to make sure all the queues can receive the traffic.

The matched traffic goes to its destination queue or port, the missed traffic is handled by the `rx_tx` function and is sent to a dedicated queue/port based on the IP address:

- Default pipe network to host:

```
Entry: IP src 1.2.3.4 dst 8.8.8.8 -> basic pipe constant RSS
Entry: IP src 1.2.3.5 dst 8.8.8.8 -> basic pipe changeable
immediate RSS
Entry: IP src 1.2.3.6 dst 8.8.8.8 -> basic pipe changeable shared
RSS
Entry: IP src 1.2.3.7 dst 8.8.8.8 -> control pipe immediate RSS
```

```
Entry: IP src 1.2.3.8 dst 8.8.8.9 -> control pipe shared RSS
```

- Egress pipe:

```
Entry: IP src 1.2.3.9 dst 8.8.8.8 -> basic pipe constant RSS
Entry: IP src 1.2.3.10 dst 8.8.8.8 -> basic pipe changeable
immediate RSS
Entry: IP src 1.2.3.11 dst 8.8.8.8 -> basic pipe changeable
shared RSS
Entry: IP src 1.2.3.12 dst 8.8.8.8 -> control pipe immediate RSS
Entry: IP src 1.2.3.13 dst 8.8.8.9 -> control pipe shared RSS
```

- Vport pipe:

```
Entry: IP src 1.2.3.14 dst 8.8.8.8 -> port 0
Entry: IP src 1.2.3.14 dst 8.8.8.9 -> port 1
Entry: IP src 1.2.3.14 dst 8.8.8.10 -> port 2
```

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_switch_rss/flow_switch_`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_rss/flow_switch_`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch_rss/meson.build`

Field String Mapping

Field String Supported Actions

The following is a list of all the API fields available for matching criteria and action execution.

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
`meta.data` (bit_offset < 32)	`meta.pkt_meta`	`meta.pkt_meta`	✔	✔	✔	✔	✔	✔	✔
`meta.data` (bit_offset ⩾ 32)	`meta.u32[i]`	`meta.u32[i]`	✔	✔	✔	✔	✔	✔	✔
`meta.mark`	`meta.mark`	`meta.mark`	☐	☐	☐	☐	☐	☐	☐
`parser_meta.hash.result`	None. See section "Copy Hash Result" for details.		N/A	N/A	✔	N/A	✔	☐	☐
`parser_meta.port.id`	`parser_meta.port_meta`		N/A	N/A	☐	N/A	☐	☐	☐
`parser_meta.ipsec.syndrome`	`parser_meta.ipsec_syndrome`		N/A	N/A	☐	N/A	☐	☐	☐
`parser_meta.psp.syndrome`	`parser_meta.psp_syndrome`		N/A	N/A	☐	N/A	☐	☐	☐
`parser_meta.random.value`	`parser_meta.random`		N/A	N/A	☐	N/A	☐	✔	☐
`parser_meta.meter.color`	`parser_meta.meter_color`		N/A	N/A	☐	N/A	☐	☐	☐
`parser_meta.packet_type.l2_outer`	`parser_meta.outer_l2_type`		N/A	N/A	N/A	N/A	N/A	N/A	N/A
`parser_meta.packet_type.l3_outer`	`parser_meta.outer_l3_type`		N/A	N/A	N/A	N/A	N/A	N/A	N/A

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
parser_meta.packet_type.l4_outer	parser_meta.outer_l4_type		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.packet_type.l2_inner	parser_meta.inner_l2_type		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.packet_type.l3_inner	parser_meta.inner_l3_type		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.packet_type.l4_inner	parser_meta.inner_l4_type		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.outer_ip_fragmented.flag	parser_meta.outer_ip_fragmented		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.inner_ip_fragmented.flag	parser_meta.inner_ip_fragmented		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.outer_integrity.l3_ok	parser_meta.outer_l3_ok		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.outer_integrity.ipv4_checksum_ok	parser_meta.outer_ip4_checksum_ok		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.outer_integrity.l4_ok	parser_meta.outer_l4_ok		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.outer_integrity.l4_checksum_ok	parser_meta.outer_l4_checksum_ok		N/A	N/A	N/A	N/A	N/A	N/A	N/A

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
parser_meta.inner_integrity.l3_ok	parser_meta.inner_l3_ok		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.inner_integrity.ipv4_checksum_ok	parser_meta.inner_ip4_checksum_ok		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.inner_integrity.l4_ok	parser_meta.inner_l4_ok		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.inner_integrity.l4_checksum_ok	parser_meta.inner_l4_checksum_ok		N/A	N/A	N/A	N/A	N/A	N/A	N/A
outer.eth.dst_mac	outer.eth.dst_mac	outer.eth.dst_mac	✔	☐	✔	✔	✔	☐	☐
outer.eth.src_mac	outer.eth.src_mac	outer.eth.src_mac	✔	☐	✔	✔	✔	☐	☐
outer.eth.type	outer.eth.type	outer.eth.type	✔	☐	✔	✔	✔	☐	☐
outer.eth_vlan0.tci	outer.eth_vlan[0].tci	outer.eth_vlan[0].tci	✔	☐	✔	✔	✔	☐	☐
outer.eth_vlan1.tci	outer.eth_vlan[1].tci	outer.eth_vlan[1].tci	☐	☐	☐	☐	☐	☐	☐
outer.ipv4.src_ip	outer.ip4.src_ip	outer.ip4.src_ip	✔	☐	✔	✔	✔	☐	☐
outer.ipv4.dst_ip	outer.ip4.dst_ip	outer.ip4.dst_ip	✔	☐	✔	✔	✔	☐	☐
outer.ipv4.dscp_ecn	outer.ip4.dscp_ecn	outer.ip4.dscp_ecn	✔	☐	✔	✔	✔	☐	☐

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
outer.ipv4.next_proto	outer.ip4.next_proto	outer.ip4.next_proto	✔	☐	✔	✔	✔	☐	☐
outer.ipv4.ttl	outer.ip4.ttl	outer.ip4.ttl	✔	✔	✔	✔	✔	☐	☐
outer.ipv4.version_ihl	outer.ip4.version_ihl	outer.ip4.version_ihl	✔	✔	✔	✔	✔	☐	☐
outer.ipv4.total_len	outer.ip4.total_len	outer.ip4.total_len	✔	✔	✔	✔	✔	☐	☐
outer.ipv4.identification	outer.ip4.identification	outer.ip4.identification	☐	☐	☐	☐	☐	☐	☐
outer.ipv4.flags_fragment_offset	outer.ip4.flags_fragment_offset	outer.ip4.flags_fragment_offset	☐	☐	☐	☐	☐	☐	☐
outer.ipv6.src_ip	outer.ip6.src_ip	outer.ip6.src_ip	✔	☐	✔	✔	✔	☐	☐
outer.ipv6.dst_ip	outer.ip6.dst_ip	outer.ip6.dst_ip	✔	☐	✔	✔	✔	☐	☐
outer.ipv6.traffic_class	outer.ip6.traffic_class	outer.ip6.traffic_class	✔	☐	✔	✔	✔	☐	☐
outer.ipv6.flow_label	outer.ip6.flow_label	outer.ip6.flow_label	✔	☐	✔	✔	✔	☐	☐
outer.ipv6.next_proto	outer.ip6.next_proto	outer.ip6.next_proto	✔	☐	✔	✔	✔	☐	☐
outer.ipv6.hop_limit	outer.ip6.hop_limit	outer.ip6.hop_limit	✔	✔	✔	✔	✔	☐	☐
outer.ipv6.payload_len	outer.ip6.payload_len	outer.ip6.payload_len	✔	✔	✔	✔	✔	☐	☐

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
outer.udp.src_port	outer.udp.l4_port.src_port	outer.udp.l4_port.src_port	✔	☐	✔	✔	✔	☐	☐
outer.udp.dst_port	outer.udp.l4_port.dst_port	outer.udp.l4_port.dst_port	✔	☐	✔	✔	✔	☐	☐
outer.transport.src_port	outer.transport.src_port	outer.transport.src_port	✔	☐	✔	✔	✔	☐	☐
outer.transport.dst_port	outer.transport.dst_port	outer.transport.dst_port	✔	☐	✔	✔	✔	☐	☐
outer.tcp.src_port	outer.tcp.l4_port.src_port	outer.tcp.l4_port.src_port	✔	☐	✔	✔	✔	☐	☐
outer.tcp.dst_port	outer.tcp.l4_port.dst_port	outer.tcp.l4_port.dst_port	✔	☐	✔	✔	✔	☐	☐
outer.tcp.flags	outer.tcp.flags	outer.tcp.flags	☐	☐	☐	☐	☐	☐	☐
outer.tcp.data_offset	outer.tcp.data_offset	outer.tcp.data_offset	☐	✔	✔	✔	✔	☐	☐
outer.icmp4.type	outer.icmp.type	outer.icmp.type	☐	☐	☐	☐	☐	☐	☐
outer.icmp4.code	outer.icmp.code	outer.icmp.code	☐	☐	☐	☐	☐	☐	☐
outer.icmp4.ident	outer.icmp.ident	outer.icmp.ident	☐	☐	☐	☐	☐	☐	☐

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
outer.icmp6.type	outer.icmp.type	outer.icmp.type	☐	☐	☐	☐	☐	☐	☐
outer.icmp6.code	outer.icmp.code	outer.icmp.code	☐	☐	☐	☐	☐	☐	☐
tunnel.gre.protocol	tun.protocol	tun.protocol	☐	☐	☐	☐	☐	☐	☐
tunnel.gre_key.value	tun.gre_key	tun.gre_key	☐	☐	☐	☐	☐	☐	☐
tunnel.nvgre.protocol	tun.protocol	tun.protocol	☐	☐	☐	☐	☐	☐	☐
tunnel.nvgre.nvgre_vs_id	tun.nvgre_vs_id	tun.nvgre_vs_id	☐	☐	☐	☐	☐	☐	☐
tunnel.nvgre.nvgre_flow_id	tun.nvgre_flow_id	tun.nvgre_flow_id	☐	☐	☐	☐	☐	☐	☐
tunnel.vxlan.vni	tun.vxlan_tun_id	tun.vxlan_tun_id	✔	☐	✔	✔	✔	☐	☐
tunnel.vxlan_gpe.vni	tun.vxlan_tun_id	tun.vxlan_tun_id	✔	☐	✔	✔	✔	☐	☐
tunnel.vxlan_gbp.vni	tun.vxlan_tun_id	tun.vxlan_tun_id	✔	☐	✔	✔	✔	☐	☐
tunnel.vxlan_gpe.next_proto	tun.vxlan_gpe_next_protocol		☐	☐	☐	☐	☐	☐	☐
tunnel.vxlan_gpe.flags	tun.vxlan_gpe_flags [1]		☐	☐	☐	☐	☐	☐	☐
tunnel.vxlan_gbp.policy_id	tun.vxlan_gbp_group_policy_id		☐	☐	☐	☐	☐	☐	☐

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
tunnel.vxlan.rsvd1		tun.vxlan_tun_rsvd1 [2]	✔	☐	☐	✔	✔	☐	☐
tunnel.vxlan_gpe.rsvd1		tun.vxlan_tun_rsvd1 [2]	✔	☐	☐	✔	✔	☐	☐
tunnel.vxlan_gbp.rsvd1		tun.vxlan_tun_rsvd1 [2]	✔	☐	☐	✔	✔	☐	☐
tunnel.gtp.teid	tun.gtp_teid	tun.gtp_teid	✔	☐	✔	✔	✔	☐	☐
tunnel.gtp_ext_hdr.next_ext	tun.gtp_next_ext_hdr_type	tun.gtp_next_ext_hdr_type	N/A	N/A	N/A	N/A	N/A	N/A	N/A
tunnel.gtp_psc.qfi	tun.gtp_ext_psc_qfi	tun.gtp_ext_psc_qfi	✔	☐	✔	✔	✔	☐	☐
tunnel.esp.spi	tun.esp_spi	tun.esp_spi	✔	☐	✔	✔	✔	☐	☐
tunnel.esp.sn	tun.esp_sn	tun.esp_sn	✔	☐	✔	✔	✔	✔	✔
tunnel.psp.nexthdr	tun.psp.nexthdr	tun.psp.nexthdr	✔	☐	✔	✔	✔	☐	☐
tunnel.psp.hdrextlen	tun.psp.hdrextlen	tun.psp.hdrextlen	✔	☐	✔	✔	✔	☐	☐
tunnel.psp.res_cryptofst	tun.psp.res_cryptofst	tun.psp.res_cryptofst	✔	☐	✔	✔	✔	☐	☐
tunnel.psp.s_d_ver_v	tun.psp.s_d_ver_v	tun.psp.s_d_ver_v	✔	☐	✔	✔	✔	☐	☐
tunnel.psp.spi	tun.psp.spi	tun.psp.spi	✔	☐	✔	✔	✔	☐	☐
tunnel.psp.iv	tun.psp.iv	tun.psp.iv	✔	☐	✔	✔	✔	☐	☐

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
tunnel.psp.vc	tun.psp.vc	tun.psp.vc	✔	☐	✔	✔	✔	☐	☐
tunnel.mpls[0].label	tun.mpls[0].label	tun.mpls[0].label	☐	☐	✔	☐	✔	☐	☐
tunnel.mpls[1].label	tun.mpls[1].label	tun.mpls[1].label	☐	☐	✔	☐	✔	☐	☐
tunnel.mpls[2].label	tun.mpls[2].label	tun.mpls[2].label	☐	☐	✔	☐	✔	☐	☐
tunnel.mpls[3].label	tun.mpls[3].label	tun.mpls[3].label	☐	☐	✔	☐	✔	☐	☐
tunnel.mpls[4].label	tun.mpls[4].label	tun.mpls[4].label	☐	☐	✔	☐	✔	☐	☐
tunnel.geneve.ver_opt_len	tun.geneve.ver_opt_len	tun.geneve.ver_opt_len	☐	☐	☐	☐	☐	☐	☐
tunnel.geneve.o_c	tun.geneve.o_c	tun.geneve.o_c	☐	☐	☐	☐	☐	☐	☐
tunnel.geneve.next_proto	tun.geneve.next_proto	tun.geneve.next_proto	☐	☐	☐	☐	☐	☐	☐
tunnel.geneve.vni	tun.geneve.vni	tun.geneve.vni	✔	☐	✔	✔	✔	☐	☐
tunnel.geneve_opt[i].type	None. See section "Copy Geneve Options" for details.		✔	☐	✔	✔	✔	☐	☐
tunnel.geneve_opt[i].class			✔	☐	✔	✔	✔	☐	☐
tunnel.geneve_opt[i].data			✔	☐	✔	✔	✔	☐	☐
inner.eth.dst_mac	inner.eth.dst_mac		☐	☐	✔	☐	✔	☐	☐

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
inner.eth.src_mac	inner.eth.src_mac		☐	☐	✔	☐	✔	☐	☐
inner.eth.type	inner.eth.type		☐	☐	✔	☐	✔	☐	☐
inner.eth_vlan0.tci	inner.eth_vlan[0].tci		☐	☐	☐	☐	☐	☐	☐
inner.eth_vlan1.tci	inner.eth_vlan[1].tci		☐	☐	☐	☐	☐	☐	☐
inner.ipv4.src_ip	inner.ip4.src_ip		☐	☐	✔	☐	✔	☐	☐
inner.ipv4.dst_ip	inner.ip4.dst_ip		☐	☐	✔	☐	✔	☐	☐
inner.ipv4.dscp_ecn	inner.ip4.dscp_ecn		☐	☐	✔	☐	✔	☐	☐
inner.ipv4.next_proto	inner.ip4.next_proto		☐	☐	☐	☐	☐	☐	☐
inner.ipv4.ttl	inner.ip4.ttl		☐	☐	✔	☐	✔	☐	☐
inner.ipv4.version_ihl	inner.ip4.version_ihl		☐	☐	✔	☐	✔	☐	☐
inner.ipv4.total_len	inner.ip4.total_len		☐	☐	✔	☐	✔	☐	☐
inner.ipv4.identification	inner.ip4.identification		☐	☐	☐	☐	☐	☐	☐
inner.ipv4.flags_fragment_offset	inner.ip4.flags_fragment_offset		☐	☐	☐	☐	☐	☐	☐

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
inner.ipv6.src_ip	inner.ip6.src_ip		☐	☐	✔	☐	✔	☐	☐
inner.ipv6.dst_ip	inner.ip6.dst_ip		☐	☐	✔	☐	✔	☐	☐
inner.ipv6.traffic_class	inner.ip6.traffic_class		☐	☐	✔	☐	✔	☐	☐
inner.ipv6.flow_label	inner.ip6.flow_label		☐	☐	✔	☐	✔	☐	☐
inner.ipv6.next_proto	inner.ip6.next_proto		☐	☐	☐	☐	☐	☐	☐
inner.ipv6.hop_limit	inner.ip6.hop_limit		☐	☐	✔	☐	✔	☐	☐
inner.ipv6.payload_len	inner.ip6.payload_len		☐	☐	✔	☐	✔	☐	☐
inner.udp.src_port	inner.udp.l4_port.src_port		☐	☐	✔	☐	✔	☐	☐
inner.udp.dst_port	inner.udp.l4_port.dst_port		☐	☐	✔	☐	✔	☐	☐
inner.transport.src_port	inner.transport.src_port		☐	☐	✔	☐	✔	☐	☐
inner.transport.dst_port	inner.transport.dst_port		☐	☐	✔	☐	✔	☐	☐
inner.tcp.src_port	inner.tcp.l4_port.src_port		☐	☐	✔	☐	✔	☐	☐

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
`inner.tcp.dst_port`	`inner.tcp.l4_port.dst_port`		☐	☐	✔	☐	✔	☐	☐
`inner.tcp.flags`	`inner.tcp.flags`		☐	☐	☐	☐	☐	☐	☐
`inner.tcp.data_offset`	`inner.tcp.data_offset`		☐	☐	✔	☐	✔	☐	☐
`inner.icmp4.type`	`inner.icmp.type`		☐	☐	☐	☐	☐	☐	☐
`inner.icmp4.code`	`inner.icmp.code`		☐	☐	☐	☐	☐	☐	☐
`inner.icmp4.ident`	`inner.icmp.ident`		☐	☐	☐	☐	☐	☐	☐
`inner.icmp6.type`	`inner.icmp.type`		☐	☐	☐	☐	☐	☐	☐
`inner.icmp6.code`	`inner.icmp.code`		☐	☐	☐	☐	☐	☐	☐

1. `tun.vxlan_gpe_flags` is mandatory in VXLAN-GPE matching since DOCA 2.9 __
2. `tun.vxlan_tun_rsvd1` modifications only work for traffic with the default UDP destination port (i.e., 4789 for VXLAN and VXLAN-GBP and 4790 for VXLAN-GPE) __ __ __

Non-Matchable Field Strings

Users can modify fields which are not included in `doca_flow_match` structure.

Copy Hash Result

Users can copy the the matcher hash calculation into other fields using the `"parser_meta.hash"` string.

Copy GENEVE Options

User can copy GENEVE option type/class/data using the following strings:

- `"tunnel.geneve_opt[i].type"` – Copy from/to option type (only for option configured with `DOCA_FLOW_PARSER_GENEVE_OPT_MODE_MATCHABLE`).
- `"tunnel.geneve_opt[i].class"` – Copy from/to option class (only for option configured with `DOCA_FLOW_PARSER_GENEVE_OPT_MODE_MATCHABLE`).
- `"tunnel.geneve_opt[i].data"` – Copy from/to option data, the bit offset is from the start of the data.

`i` is the index of the option in `tlv_list` array provided in `doca_flow_parser_geneve_opt_create`.

DOCA Flow Connection Tracking

> **Note**
>
> This feature is not supported in this DOCA release. It will be re-enabled in DOCA version 3.0.

This guide provides an overview and configuration instructions for DOCA Flow CT API.

Introduction

DOCA Flow Connection Tracking (CT) is a 5-tuple table which supports the following:

- Track 5-tuple sessions (or 6-tuple when a zone is available)
- Zone based – virtual tables
- Aging (i.e., removes idle connections)
- Sets metadata for a connection
- Bidirectional packet handling
- High rate of connections per second (CPS)

The CT module makes it simple and efficient to track connections by leveraging hardware resources. The module supports both autonomous and managed mode.

Architecture

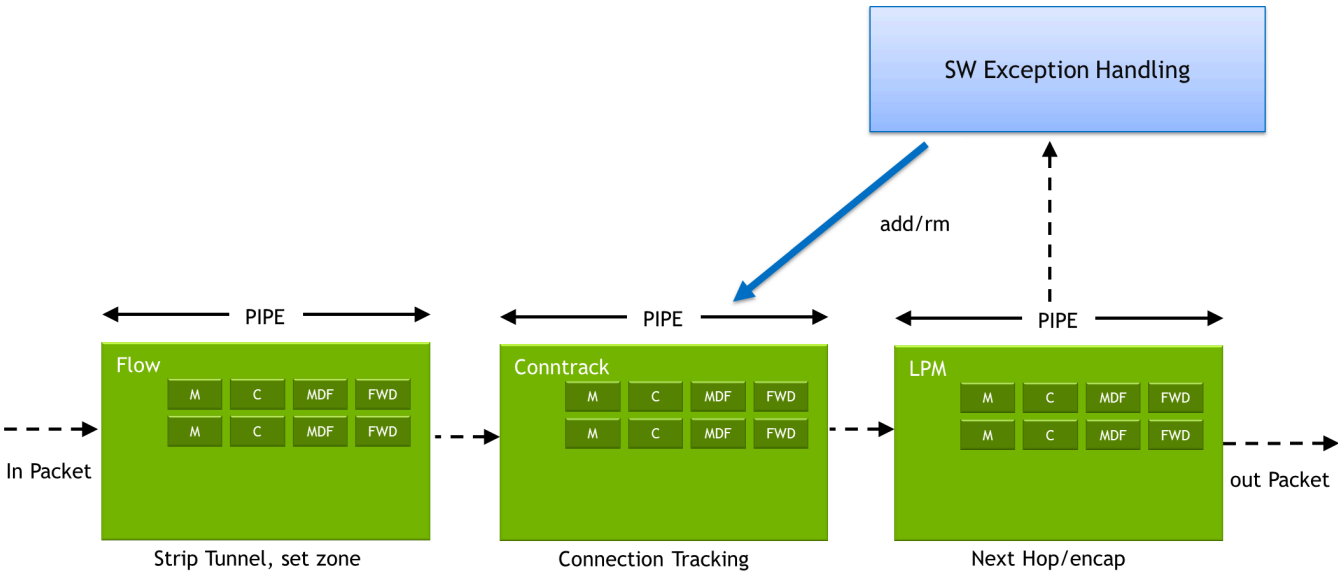
DOCA Flow CT pipe handles non-encapsulated TCP and UDP packets. The CT pipe only supports forward to next pipe or miss to next pipe actions:

- All packets matching known connection 6-tuples are forwarded to the CT's forward pipe
- Non-matching packets are forwarded to the miss pipe

The user application must handle packets accordingly.

The DOCA Flow CT API is built around four major parts:

- CT module manipulation – configuring CT module resources
- CT connection entry manipulation – adding, removing, or updating connection entries
- Callbacks – handling asynchronous entry processing result
- Pipe and entry statistics



Aging

Aging time is a time in seconds that sets the maximum allowed time for a session to be maintained without a packet seen. If that time elapses with no packet being detected, the session is terminated.

To support aging, a dedicated aging thread is started to poll and check counters for all connections.

Autonomous Mode

In this mode, DOCA runs multiple CT workers internally, to handle connections in parallel.

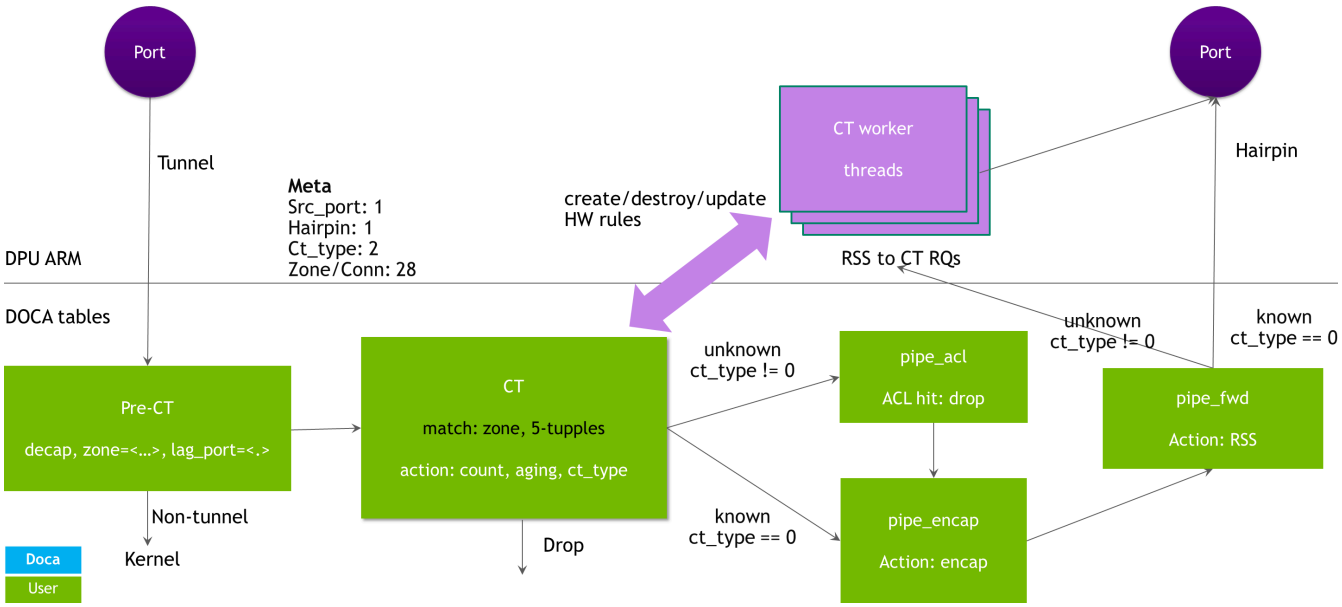
A connection's lifecycle is controlled by the connection state encapsulated in the packet and time-based aging.

CT workers establish and close connections automatically based on the connection's state stored in packet meta.

Packet meta is defined as follows:

```
uint32_t src : 1;       /**< Source port in multi-port E-Switch mode */
uint32_t hairpin : 1;   /**< Subject to forward using hairpin. */
uint32_t type : 2;      /**< CT packet type: New, End or Update */
uint32_t data : 28;     /**< Zone set by user or reserved after CT pipe. */
```

- `data` – CT table matches on packet meta (zone) and 5-tuples
- `type` – can have the following values:
  - `NONE` – (known) if packet hit any connection rule
  - `NEW` – if new TCP or UDP connection
  - `END` – if TCP connection closed
- `src` and `hairpin` – used for forwarding pipe and worker to deliver packet



Managed Mode

The application is responsible for managing the worker threads in this mode, parsing and handling the connection's lifecycle.

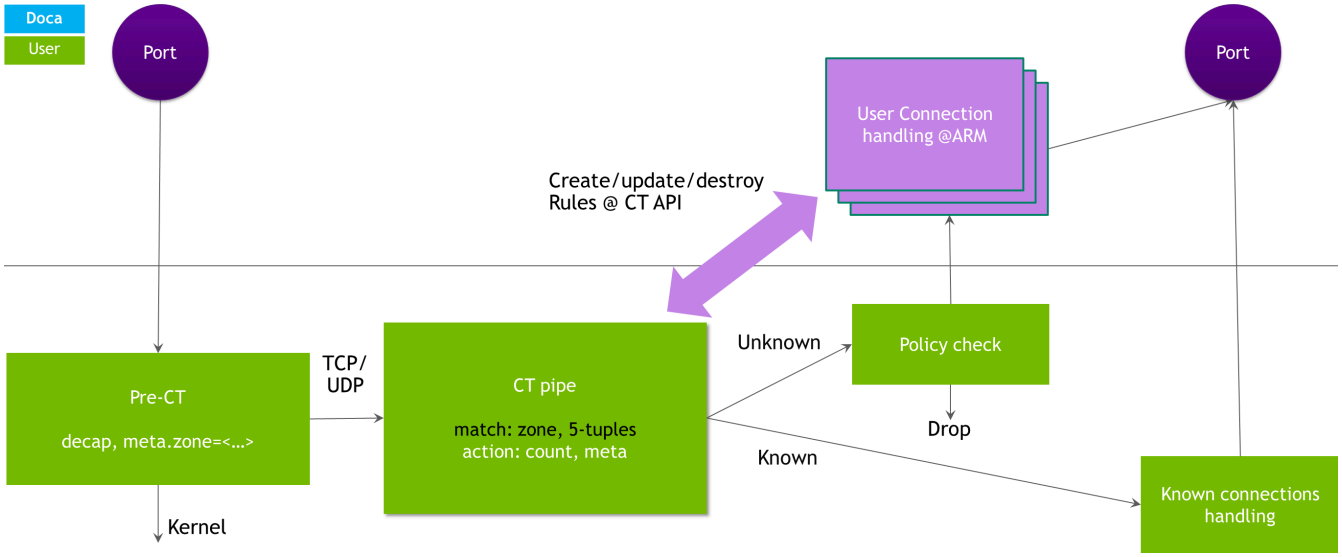
Managed mode uses DOCA Flow CT management APIs to create or destroy the connections.

The CT aging module notifies on aged out connections by calling callbacks.

Users can create connection rules with a different pattern, meta, or counter, for each packet direction.

> **Info**
>
> Users are responsible for defining meta and mask to `match` and `modify`.

Users can create one rule of a connection first, then create another rule using API `doca_flow_ct_entry_add_dir()`.



DOCA Flow API can be used to process CT entries with a CT-dedicated queue.

- `doca_flow_entries_process` – process pipe entries in queue
- `doca_flow_aging_handle` – handle pipe entries aging

> **Info**
>
> Other DOCA Flow APIs like CT entry status query and pipe miss query are not supported.

Prerequisites

DPU

To enable DOCA Flow CT on the DPU, perform the following on the Arm:

1. Enable `iommu.passthrough` in Linux boot commands (or disable SMMU from the DPU BIOS):
    1. Run:

```
sudo vim /etc/default/grub
```

2. Set `GRUB_CMDLINE_LINUX="iommu.passthrough=1"`.

3. Run:

```
sudo update-grub
sudo reboot
```

2. Configure DPU firmware with `LAG_RESOURCE_ALLOCATION=1`:

```
sudo mlxconfig -d <device-id> s LAG_RESOURCE_ALLOCATION=1
```

> **Info**
>
> Retrieve `device-id` from the output of the `mst status -v` command. If, under the MST tab, the value is N/A, run the `mst start` command.

3. Update `/etc/mellanox/mlnx-bf.conf` as follows:

```
ALLOW_SHARED_RQ="no"
```

4. Perform power cycle on the host and Arm sides.

5. If working with a single port, set the DPU into e-switch mode:

```
sudo devlink dev eswitch set pci/<pcie-address> mode
switchdev
sudo devlink dev param set pci/<pcie-address> name
esw_multiport value false cmode runtime
```

Info

Retrieve `pcie-address` from the output of the `mst status -v` command.

6. If working with two PF ports, set the DPU into multi-port e-switch mode (for the 2 PCIe devices):

```
sudo devlink dev param set pci/<pcie-address> name
esw_multiport value true cmode runtime
```

Info

Retrieve `pcie-address` from the output of the `mst status -v` command.

7. Define huge pages (see DOCA Flow prerequisites).

ConnectX

To enable DOCA Flow CT on the NVIDIA® ConnectX®, perform the following:

1. Configure firmware with `LAG_RESOURCE_ALLOCATION=1`:

```
sudo mlxconfig -d <device-id> s LAG_RESOURCE_ALLOCATION=1
```

> **Info**
>
> Retrieve `device-id` from the output of the `mst status -v` command. If, under the MST tab, the value is N/A, run the `mst start` command.

2. Perform power cycle.

3. If working with a single port:

```
sudo devlink dev eswitch set pci/<pcie-address> mode
switchdev
sudo devlink dev param set pci/<pcie-address> name
esw_multiport value false cmode runtime
```

> **Info**
>
> Retrieve `pcie-address` from the output of the `mst status -v` command.

4. If working with two PF ports:

```
sudo devlink dev eswitch set pci/<pcie-address0> mode
switchdev
sudo devlink dev eswitch set pci/<pcie-address1> mode
switchdev
sudo devlink dev param set pci/<pcie-address0> name
esw_multiport value true cmode runtime
sudo devlink dev param set pci/<pcie-address1> name
esw_multiport value true cmode runtime
```

Info

Retrieve `pcie-address` from the output of the `mst status -v` command.

5. Define huge pages (see DOCA Flow prerequisites).

Actions

DOCA Flow CT supports actions based on meta and NAT operations. Each action can be defined as either shared or non-shared.

Note

Action descriptors are not supported.

Shared Actions

Actions that can be shared between entries. Shared actions are predefined and reused in multiple entries.

The user gets a handle per shared action created and uses this handle as a reference to the action where required.

> **Info**
>
> It is user responsibility to track shared actions and to remove them when they become irrelevant.

Shared actions are defined using a control queue (see struct doca_flow_ct_cfg).

Non-shared Actions

Actions provided with their data during entry create/update.

These actions are completely managed by DOCA Flow CT and cannot be reused in multiple flows (i.e., NAT operations).

Action Sets in Pipe Creation

When creating a DOCA Flow CT pipe, users must define action sets, just as they would for any other pipe.

Fields in the CT pipe must be marked as `CHANGEABLE` during pipe creation. This allows the actual criteria for these fields to be specified later during entry creation.

> **Info**

Only actions related to meta and NAT, as defined in struct doca_flow_ct_actions , are supported.

During entry creation or update, different actions can be specified for each direction, allowing variations in action content and/or action type.

Feature Enable

To enable user actions, configure the following parameters:

- User action templates during DOCA Flow CT pipe creation
- Maximum number of user actions (`nb_user_actions` on DOCA Flow CT init)

Using Actions in Autonomous Mode

Init

Configure the following parameters on `doca_flow_ct_init()`:

- `nb_ctrl_queues` – number of control queues for defining shared actions
- `nb_user_actions` – maximum number of actions (shared and non-shared)
- `worker_cb` – callbacks required to communicate with the user

Create DOCA Flow CT Pipe

Configure actions sets on `doca_flow_pipe_create()`.

Create Shared Actions

Use `doca_flow_ct_actions_add_shared()` with one of the control queues.

Shared actions can be added at any time before use.

Implement Worker Callbacks

Callbacks are called from each worker thread to acquire synchronization with the user code and on the first packet of a flow.

On `doca_flow_ct_rule_pkt_cb`:

- Determine how the packet should be treated
- If rules are required, return the actions handles to use

Using Actions in Managed Mode

Init

Configure the following parameters on `doca_flow_ct_init()`:

- `nb_ctrl_queues` – number of control queues for defining shared actions
- `nb_user_actions` – maximum number of user actions. Must align to 64. Both shared control queues and non-shared control queues cache action IDs to speed up ID allocation. Each queue may cache a maximum of 1024 IDs. Users must configure the expected number of actions + total queues * 1024. This number cannot exceed the number of actions hardware supports.

Create DOCA Flow CT Pipe

Configure actions sets on `doca_flow_pipe_create()`.

Create Shared Actions

Use `doca_flow_ct_actions_add_shared()` with one of the control queues.

Shared actions can be added at any time before use.

Add Entry

Entry can be created in one of the following ways:

- Using an action handle of a predefined shared action
- Using action data, which is specific to the flow, not sharable (e.g., for NAT operations)

The entry can have different actions and/or different action types per direction.

Remove Entry

Non-shared actions associated with an entry are implicitly destroyed by DOCA Flow CT.

Shared actions are not destroyed. They can be used by the user until they decide to remove them.

Update Entry

Entry actions can be updated per direction. All combinations of shared/non-shared actions are applicable (e.g., update from shared to non-shared).

Changeable Forward

DOCA Flow CT allows using a different forward pipe per flow direction.

DOCA Flow CT supports the forward pipe in two levels:

- Pipe level – a single forward pipe defined during DOCA Flow CT pipe creation and used for all entries
- Entry level – forward pipe defined during entry create
- DOCA Flow CT operates in one of the two levels

DOCA CT forward in entry level has the following characteristics:

- Supports only `DOCA_FLOW_FWD_PIPE` (up to 4 different forward pipes)
- Supports forward pipe per flow direction (both directions can have same/different forward pipe)
- Must set forward pipes on each entry create (no default forward pipe)

Turn on the feature:

1. Create DOCA Flow CT pipe with forward type = `DOCA_FLOW_FWD_PIPE` and `next_pipe` = `NULL` .
2. Call to `doca_flow_ct_fwd_register` to register forward pipes and get `fwd_handles` in return.

Using Changeable Forward in Managed Mode

1. Initialize DOCA Flow CT (`doca_flow_ct_init`).
2. Register forward pipes (`doca_flow_ct_fwd_register`).
    - Define pipes that can be used for forward
3. Create DOCA Flow CT pipe (`doca_flow_pipe_create`) w ith definition of possible forward pipes.
4. Add entry (`doca_flow_ct_add_entry`).
    - Set origin and/or reply `fwd_handles` returned from `doca_flow_ct_fwd_register` .
5. Update forward for entry direction (`doca_flow_ct_update_entry`).

> **Note**
>
> Updating forward handle requires setting all other parameters with their previous values.

Using Changeable Forward in Autonomous Mode

1. Initialize DOCA Flow CT (`doca_flow_ct_init`).
2. Register forward pipes (`doca_flow_ct_fwd_register`).
   - Define pipes that can be used for forward.
3. Create DOCA Flow CT pipe (`doca_flow_pipe_create`) w ith definition of possible forward pipes.
4. CT workers start to handle traffic.
5. On the first flow packet, `doca_flow_ct_rule_pkt` callback is called.
   - In this callback, determine if the entry should be created, and which actions and/or forward handles should be used for this entry.

> **Info**
>
> Update forward for entry direction is not supported.

API

For the library API reference, refer to DOCA Flow and CT API documentation in the DOCA Library APIs.

Note

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

The following sections provide additional details about the library API.

enum doca_flow_ct_flags

DOCA Flow CT configuration optional flags.

Flag	Description
`DOCA_FLOW_CT_FLAG_STATS = 1u << 0`	Enable internal pipe counters for packet tracking purposes. Call `doca_flow_pipe_dump(<ct_pipe>)` to dump counter values. Each call dumps values changed.
`DOCA_FLOW_CT_FLAG_WORKER_STATS = 1u << 1,`	Enable worker thread internal debug counter periodical dump. Autonomous mode only.
`DOCA_FLOW_CT_FLAG_NO_AGING = 1u << 2,`	Disable aging
`DOCA_FLOW_CT_FLAG_SW_PKT_PARSING = 1u << 3,`	Enable CT worker software packet parsing to support VLAN, IPv6 options, or special tunnel types
`DOCA_FLOW_CT_FLAG_MANAGED = 1u << 4,`	Enable managed mode in which user application is responsible for managing packet handling, and calling the CT API to manipulate CT connection entries
`DOCA_FLOW_CT_FLAG_ASYMMETRIC = 1u << 5,`	Allows different 6-tuple table definitions for the origin and reply directions. Default to symmetric mode, uses same meta and reverse 5-tuples for reply direction. Managed mode only.

Flag	Description
`DOCA_FLOW_CT_FLAG_ASYMMETRIC_COUNTER = 1u << 6,`	Enable different counters for the origin and reply directions. Managed mode only.
`DOCA_FLOW_CT_FLAG_NO_COUNTER = 1u << 7,`	Disable counter and aging to save aging thread CPU cycles
`DOCA_FLOW_CT_FLAG_DEFAULT_MISS = 1u << 8,`	Check TCP SYN flags and UDP in CT miss flow to identify ADD type packets.
`DOCA_FLOW_CT_FLAG_WIRE_TO_WIRE = 1u << 9,`	Hint traffic comes from uplink wire and forwards to uplink wire. **Note** If this flag is set, the direction info must be `DOCA_FLOW_DIRECTION_NETWORK_TO_HOST` .
`DOCA_FLOW_CT_FLAG_CALC_TUN_IP_CHKSUM = 1u << 10,`	Enable hardware to calculate and set the checksum on L3 header (IPv4)
`DOCA_FLOW_CT_FLAG_DUP_FILTER_UDP_ONLY= 1u << 11,`	Apply the connection duplication filter for UDP connections only

enum doca_flow_ct doca_flow_ct_entry_flags

DOCA Flow CT Entry optional flags.

Flag	Description
`DOCA_FLOW_CT_ENTRY_FLAGS_NO_WAIT = (1 << 0)`	Entry is not buffered; send to hardware immediately
`DOCA_FLOW_CT_ENTRY_FLAGS_DIR_ORIGIN = (1 << 1)`	Apply flags to origin direction
`DOCA_FLOW_CT_ENTRY_FLAGS_DIR_REPLY = (1 << 2)`	Apply flags to reply direction
`DOCA_FLOW_CT_ENTRY_FLAGS_IPV6_ORIGIN = (1 << 3)`	Origin direction is IPv6; origin match union in struct `doca_flow_ct_match` is IPv6
`DOCA_FLOW_CT_ENTRY_FLAGS_IPV6_REPLY = (1 << 4)`	Reply direction is IPv6; reply match union in struct `doca_flow_ct_match` is IPv6
`DOCA_FLOW_CT_ENTRY_FLAGS_COUNTER_ORIGIN = (1 << 5)`	Apply counter to origin direction
`DOCA_FLOW_CT_ENTRY_FLAGS_COUNTER_REPLY = (1 << 6)`	Apply counter to reply direction
`DOCA_FLOW_CT_ENTRY_FLAGS_COUNTER_SHARED = (1 << 7)`	Counter is shared for both direction (origin and reply)
`DOCA_FLOW_CT_ENTRY_FLAGS_FLOW_LOG = (1 << 8)`	Enable flow log on entry removed
`DOCA_FLOW_CT_ENTRY_FLAGS_ALLOC_ON_MISS = (1 << 9)`	Allocate on entry not found when calling `doca_flow_ct_entry_prepare()` API
`DOCA_FLOW_CT_ENTRY_FLAGS_DUP_FILTER_ORIGIN = (1 << 10)`	Enable duplication filter on origin direction
`DOCA_FLOW_CT_ENTRY_FLAGS_DUP_FILTER_REPLY = (1 << 11)`	Enable duplication filter on reply direction

enum doca_flow_ct_rule_opr

Options for handling flows in autonomous mode with shared actions. The decision is taken on the first flow packet.

Operation	Description
`DOCA_FLOW_CT_RULE_OK`	Flow should be defined in the CT pipe using the required shared actions handles
`DOCA_FLOW_CT_RULE_DROP`	Flow should not be defined in the CT pipe. The packet should be dropped.
`DOCA_FLOW_CT_RULE_TX_ONLY`	Flow should not be defined in the CT pipe. The packet should be transmitted.

struct direction_cfg

Managed mode configuration for origin or reply direction.

Field	Description
`bool match_inner`	5-tuple match pattern applies to packet inner layer
`struct doca_flow_meta *zone_match_mask`	Mask to indicate meta field and bits to match
`struct doca_flow_meta *meta_modify_mask`	Mask to indicate meta field and bits to modify on connection packet match

struct doca_flow_ct_worker_callbacks

Set of callbacks for using shared actions in autonomous mode.

Field	Description
`doca_flow_ct_sync_acquire_cb worker_init`	Called at the start of a worker thread to sync with the user context
`doca_flow_ct_sync_release_cb worker_release`	Called at the end of a worker thread
`doca_flow_ct_rule_pkt_cb rule_pkt`	Called on the first packet of a flow

struct doca_flow_ct_cfg

DOCA Flow CT configuration.

```
uint32_t nb_arm_queues;
uint32_t nb_ctrl_queues;
uint32_t nb_user_actions;
uint32_t nb_arm_sessions[DOCA_FLOW_CT_SESSION_MAX];
uint32_t flags;
uint16_t aging_core;
uint16_t aging_query_delay_s;
doca_flow_ct_flow_log_cb flow_log_cb;
struct doca_flow_ct_aging_ops *aging_ops;
uint32_t base_core_id;
uint32_t dup_filter_sz;
union {
                /* Managed mode configuration for origin and reply direction. */
                struct direction_cfg direction[2];

                /* Below fields are dedicate for autonomous mode */
                struct {
                                                uint16_t
tcp_timeout_s;
                                                uint16_t
tcp_session_del_s;
                                                uint16_t
udp_timeout_s;
                        enum doca_flow_tun_type tunnel_type;
                        uint16_t vxlan_dst_port;
                        enum doca_flow_ct_hash_type hash_type;
                        uint32_t meta_user_bits;
                        uint32_t meta_action_bits;
                        struct doca_flow_meta *meta_zone_mask;
```

```
                        struct doca_flow_meta
*connection_id_mask;
                        struct doca_flow_ct_worker_callbacks
worker_cb;
                };
        };
```

Where:

Field	Description
`uint32_t nb_arm_queues`	Number of CT queues. In autonomous mode, also the number of worker threads.
`uint32_t nb_ctrl_queues`	Number of CT control queues used for defining shared actions
`uint32_t nb_user_actions`	Maximum number of user actions supported (shared and non-shared) Minimum value is 1K * (`nb_ctrl_queues + nb_arm_queues`)
`uint32_t nb_arm_sessions[DOCA_FLOW_CT_SESSION_MAX]`	Maximum number of IPv4 and IPv6 CT connections
`uint32_t flags`	CT configuration flags
`uint16_t aging_core`	CPU core ID for CT aging thread to bind.
`uint16_t aging_core_delay`	CT aging code delay.
`doca_flow_ct_flow_log_cb flow_log_cb`	Flow log callback function, when set
`struct doca_flow_ct_aging_ops *aging_ops`	User-defined aging logic callback functions. Fallback to default aging logic
`uint32_t base_core_id`	Base core ID for the workers
`uint32_t dup_filter_sz`	Number of connections to cache in the duplication filter

Field	Description
`struct direction_cfg direction`	Managed mode configuration for origin or reply direction
`uint16_t tcp_timeout_s`	TCP timeout in seconds
`uint16_t tcp_session_del_s`	Time to delay or kill TCP session after RST/FIN
`enum doca_flow_tun_type tunnel_type`	Encapsulation tunnel type
`uint16_t vxlan_dst_port`	VXLAN outer UDP destination port in big endian
`enum doca_flow_ct_hash_type hash_type`	Type of connection hash table type: `NONE` or `SYMMETRIC_HASH`
`uint32_t meta_user_bits`	User packet meta bits to be owned by the user
`uint32_t meta_action_bits`	User packet meta bits to be carried by identified connection packet
`struct doca_flow_meta *meta_zone_mask`	Mask to indicate meta field and bits saving zone information
`struct doca_flow_meta *connection_id_mask`	Mask to indicate meta field and bits for CT internal connection ID
`struct doca_flowct_worker_callbacks worker_cb`	Worker callbacks to use shared actions

struct doca_flow_ct_actions

This structure is used in the following cases:

- For defining shared actions. In this case, action data is provided by the user. The action handle is returned by DOCA Flow CT.
- For defining an entry with actions. The structure can be filled with two options:
  - With action handle of a previously created shared action
  - With non-shared action data

DOCA Flow CT action structure.

```
enum doca_flow_resource_type  resource_type;
union {
                                /* Used when creating an entry with a shared action. */
                uint32_t action_handle;

                /* Used when creating an entry with non-shared action or when creating a shared action. */
                struct {
                                uint32_t action_idx;
                                struct doca_flow_meta meta;
                                struct doca_flow_header_l4_port l4_port;
                                union {
                                                struct doca_flow_ct_ip4 ip4;
                                                struct doca_flow_ct_ip6 ip6;
                                 };
                } data;
        };
```

Where:

Field	Description
`enum doca_flow_resource_type resource_type`	Shared/non-shared action
`uint32_t action_handle`	Shared action handle
`uint32_t action_idx`	Actions template index
`struct doca_flow_meta meta`	Modify meta values
`struct doca_flow_header_l4_port l4_port`	UDP or TCP source and destination port
`struct doca_flow_ct_ip4 ip4`	Source and destination IPv4 addresses

Field	Description
`struct doca_flow_ct_ip6 ip6`	Source and destination IPv6 addresses

Info

The value in `meta` , `l4_port` , `ip4` , and `ip6` should start from `bit0`, the least significant bit, regardless of which bits are set in mask. For example, `action_val.meta.u32[0] = DOCA_HTOBE32(0x12)`, `action_mask.meta.u32[0] = DOCA_HTOBE32(0x0000FF00)` sets bits 15-8 to `0x12`.

DOCA Flow Tune Server

This guide provides an overview and configuration instructions for DOCA Flow Tune Server API.

Introduction

DOCA Flow Tune Server (TS) is a DOCA Flow subcomponent that collects predefined internal key performance indicators (KPIs) and pipeline information of a running DOCA Flow application. All information is transferred by an inter-process communication channel (Unix domain socket) to DOCA Flow Tune Tool for further analysis and monitoring.

Prerequisites

DOCA Flow Tune Server API is only available when using the DOCA Flow and DOCA Flow Tune Server trace libraries.

> **Info**
>
> For more detailed information, refer to section "Debug and Trace Features" under DOCA Flow.

Configuration

DOCA Flow Tune Server has a configuration file that allows customizing various settings. The configuration file is divided into different sections so to ease its use.

Config File Default Values

If a configuration file was not provided, DOCA Flow Tune Server uses default values for its mandatory fields. List of all default values can be seen in section "Configuration File Example".

Custom Config File

Instead of using the default configuration values, users may create a file of their own and provide a file path using the `doca_flow_tune_server_cfg_set_cfg_file_path()` API call.

Once used, DOCA Flow Tune Server loads all provided values directly from the file, while the rest of the fields (if any) use their respective default values.

Configuration File Example

```
{
        "network" : {
                "uds_path" : "/tmp/tune_server.sock"
        }
}
```

- `network`
  - `uds_path` – Unix Domain Socket (`AF_UNIX`) path for the tune server to bind to. This socket is used for the inter-process-communication (IPC) channel between DOCA Flow Tune Server and DOCA Flow Tune Tool. Default value is `/tmp/tune_server.sock`.

API

Info

For more detailed information on DOCA Flow API, refer to *DOCA Library APIs*.

The following subsections provide additional details about the library API.

struct doca_flow_tune_server_cfg

Opaque configuration struct to use on configuration API calls.

doca_flow_tune_server_cfg_create

Allocates and creates DOCA Flow Tune Server configuration structure.

```
doca_error_t doca_flow_tune_server_cfg_create(struct
doca_flow_tune_server_cfg **cfg);
```

doca_flow_tune_server_cfg_set_cfg_file_path

Sets the local configuration file path in the opaque configuration struct, for DOCA Flow Tune Server to use when searching for the JSON configuration file.

Providing a JSON configuration file is optional. If a file is not provided, DOCA Flow Tune Server uses internal defaults.

```
doca_error_t doca_flow_tune_server_cfg_set_bind_path(struct
doca_flow_tune_server_cfg *cfg, const char *path);
```

doca_flow_tune_server_cfg_destroy

Destroys and deallocates DOCA Flow Tune Server opaque configuration structure.

Should be called after calling `doca_flow_tune_server_init()`.

```
doca_error_t doca_flow_tune_server_cfg_destroy(struct
doca_flow_tune_server_cfg *cfg);
```

doca_flow_tune_server_init

Starts DOCA Flow Tune Server main thread.

```
doca_error_t doca_flow_tune_server_init(struct
doca_flow_tune_server_cfg *cfg);
```

doca_flow_tune_server_destroy

Stops DOCA Flow Tune Server main thread.

```
void doca_flow_tune_server_destroy(void);
```