NVIDIA DOCA FLOW

Programming Guide

MLNX-15-060488 _v1.2 | January 2022

Table of Contents

Chapter 1. Introduction

DOCA flow is the most fundamental API for building generic execution pipes in HW.

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

Using DOCA flow API, it is easy to develop HW-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

The execution pipe may include packet modification actions:

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

The execution pipe may also have monitoring actions:

- Count
- Policers
- Mirror

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

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

This document is intended for software developers writing network function application that focus on packet processing such as gateways. The document assumes familiarity with network stack and DPDK.

Chapter 2. DOCA Flow Architecture

2.1. Init

Before using any DOCA flow, it is mandatory to call DOCA flow initialization.

```
int doca_flow_init(const struct doca_flow_cfg *cfg, struct doca_flow_error *error);
```

The struct `doca_flow_cfg` contains the following elements:

- `total_sessions` – refers to the estimated scale of HW rules
- `queues` – the number of HW acceleration controls queues. It is expected that the same core always uses the same `queue_id`. In cases where multiple cores are accessing the API using the same `queue_id`, it is up to the application to use locks between different cores/threads.
- `is_hairpin` – the fwd is a hairpin queue while it is set to true
- `aging` – aging is handled by DOCA flow while it is set to true

2.2. Port

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 origin port, it will start the HW 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. The purpose of this step is to attach user application ports to the DOCA flow ports.

```
struct doca_flow_port *doca_flow_port_start(const struct doca_flow_port_cfg *cfg,
                                            struct doca_flow_error *error);
```

The struct `doca_flow_port_cfg` contains the following elements:

- `port_id` – chosen by the user. IDs must start with 0 and be consecutive.
- `type` – depends on underlying API
- `devargs` – a string containing the exact configuration needed according to the type

- `priv_data_size` – per port, users may define private data where application-specific info can be stored

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";
```

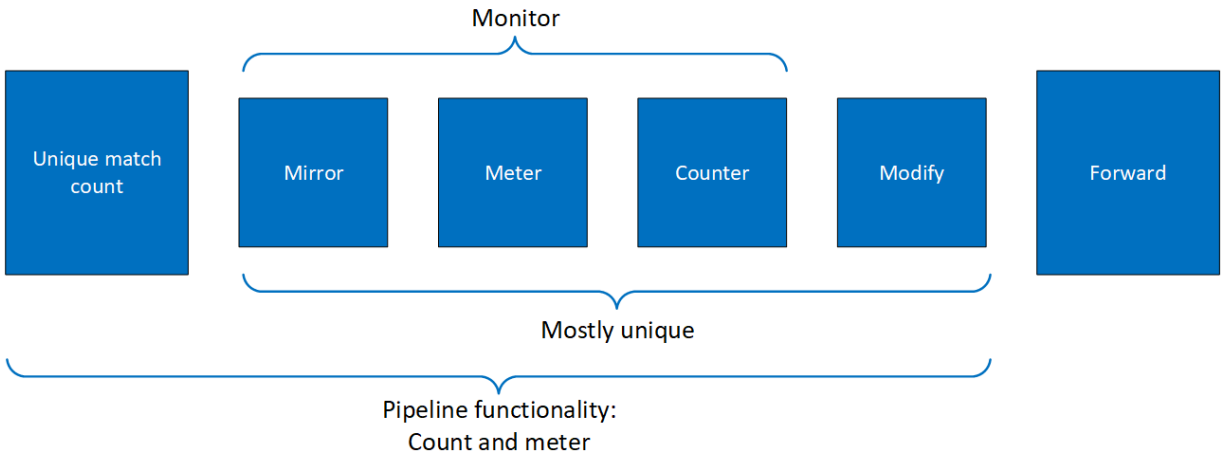
The `devargs` parameter points to a string that has the numeric value of the DPDK `port_id` in decimal. The port must be configured and started before calling this API. Mapping the DPDK port to the DOCA port is required to synchronize application ports with HW ports.

2.3. Pipe

Pipe is a template that defines packet processing without adding any specific HW 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 HW to efficiently build the execution pipe. After the pipe is created, specific entries can be added. Only a subset of the pipe can be used (e.g. skipping the monitor completely, just using the counter, etc).

2.3.1. Setting Pipe Match

Match is a mandatory field when creating a pipe. Using the following struct, users must define the fields that should be matched on the pipe.

The struct `doca_flow_match` contains the following elements:

- `flags` – match items which are no value needed
- `out_src_mac` – outer source MAC address
- `out_dst_mac` – outer destination MAC address
- `out_eth_type` – outer Ethernet layer type
- `vlan_id` – outer VLAN ID
- `out_src_ip` – outer source IP address
- `out_dst_ip` – outer destination IP address
- `out_l4_type` – outer layer 4 protocol type
- `out_src_port` – outer layer 4 source port
- `out_dst_port` – outer layer 4 destination port
- `tun` – tunnel info
- `in_src_ip` – inner source IP address if tunnel is used
- `in_dst_ip` – inner destination IP address if tunnel is used
- `in_l4_type` – inner layer 4 protocol type if tunnel is used
- `in_src_port` – inner layer 4 source port if tunnel is used
- `in_dst_port` – inner layer 4 destination port if tunnel is used

For each field, users choose whether the field is:

- Ignored (wild card) – the value of the field is ignored
- Constant – all entries in the pipe must have the same value for this field. Users should not put a value for each entry.
- Changeable – per entry, the user must provide the value to match

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

The match field type can be defined either implicitly or explicitly.

2.3.1.1. Implicit Match

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

2.3.1.1.1. Ignored Fields

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

2.3.1.1.2. Constant Fields

These are fields that have a constant value. For example, as shown in the following, the tunnel type is VXLAN.

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

These fields only need to be configured once, not once per new pipeline entry.

2.3.1.1.3. Changeable Fields

These are fields that may change per entry. For example, the following shows an inner 5-tuple which are set with a full mask.

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

If this is the constant value required by user, then they should set zero on the field when adding a new entry.

2.3.1.1.4. Example

The following is an example of a match on the VXLAN tunnel, where for each entry there is a specific IPv4 destination address, and an inner 5-tuple.

```
static void build_underlay_overlay_match(struct doca_flow_match *match)
{
    //outer
    match->out_dst_ip.ipv4_addr = 0xffffffff;
    match->out_l4_type = DOCA_PROTO_UDP;
    match->out_dst_port = DOCA_VXLAN_DEFAULT_PORT;
    match->tun.type = DOCA_FLOW_TUN_VXLAN;
    match->tun.vxlan_tun_id = 0xffffffff;
    //inner
    match->in_dst_ip.ipv4_addr = 0xffffffff;
    match->in_src_ip.ipv4_addr = 0xffffffff;
    match->in_src_ip.type = DOCA_FLOW_IP4_ADDR;
    match->in_l4_type = DOCA_PROTO_TCP;
    match->in_src_port = 0xffff;
    match->in_dst_port = 0xffff;
}
```

2.3.1.2. Explicit Match

Users may provide a mask on a match. In this case, there are two `doca_flow_match` items: The first will contain constant values, and the second will contain masks.

2.3.1.2.1. Ignored Fields

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

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

2.3.1.2.2. Constant Fields

These are fields that have a constant value. For example, as shown in the following, the tunnel type is VXLAN and the mask should be full.

```
match.tun.type = DOCA_FLOW_TUN_VXLAN;
        match_mask.tun.type = 0xffffffff;
```

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

2.3.1.2.3. Changeable Fields

These are fields that may change per entry (e.g. inner 5-tuple). Their value should be zero and the mask should be full.

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

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

2.3.2. Setting Pipe Actions

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

The struct `doca_flow_actions` contains the following elements:

- `decap` – decap while it is set to true
- `mod_src_mac` – modify source MAC address
- `mod_dst_mac` – modify destination MAC address
- `mod_src_ip` – modify source IP address
- `mod_dst_ip` – modify destination IP address
- `mod_src_port` – modify layer 4 source port
- `mod_dst_port` – modify layer 4 destination port
- `del_ttl` – decrease TTL value while it is set to true
- `has_encap` – encap while it is set to true
- `encap` – encap data information

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 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 is set by the user per entry

  ```
  match_mask.in_dst_ip.ipv4_addr = 0xffffffff;
  ```
- Boolean fields – Boolean values, encap and decap are considered as constant values. It is not allowed to generate actions with `encap=true` and to then have an entry without an encap value.

For example:

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

2.3.3. Setting Pipe Monitor

If a 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 struct `doca_flow_monitor` contains the following elements:

- `flags` – indicate actions to be included
- `id` – meter ID
- `cir` – committed information rate
- `cbs` – committed burst size
- `aging` – aging time in seconds
- `user_data` – aging user data input

Where:

- Committed information rate (CIR) – defines maximum bandwidth
- Committed burst size (CBS) – defines maximum local burst size

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

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

The monitor also includes the aging configuration, if the aging time is set, this entry ages out if timeout passes without any matching on the entry. User data is used to map user usage. If the `user_data` field is set, when the entry ages out, query API returns this `user_data`. If `user_data` is not configured by application, the aged pipe entry handle is returned.

For example:

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

Refer to Pipe Entry Aged Query for more information.

2.3.4. Setting Pipe Forwarding

The FORWARDING "action" is the last action in a pipe, and it directs where the packet goes next. Users may configure one of the following destinations:

- Send to software (representor)
- Send to wire

- Jump to next pipe
- Drop packets

The FORWARDING action may be set for pipe create, but it can also be unique per entry.

A pipe can be defined with constant forwarding (e.g., always send packets on a specific port). In this case, all entries will have the exact same forwarding. If forwarding is not defined when a pipe is created, users must define forwarding per entry. In this instance, pipes may have different forwarding actions.

The struct `doca_flow_fwd` contains the following elements:

- `type` – indicates the forwarding type
- `rss_flags` – RSS offload types
- `rss_queues` – RSS queues array
- `num_of_queues` – number of queues
- `rss_mark` – mark ID of each queue
- `port_id` – destination port ID
- `next_pipe` – next pipe pointer

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

- `DOCA_FLOW_FWD_RSS` – forwards packets to RSS
- `DOCA_FLOW_FWD_PORT` – forwards packets to port
- `DOCA_FLOW_FWD_PIPE` – forwards packets to another pipe
- `DOCA_FLOW_FWD_DROP` – drops packets

The `rss_flags` include the RSS fields defined in the following enum:

- `DOCA_FLOW_RSS_IP` – RSS by IP header
- `DOCA_FLOW_RSS_UDP` – RSS by UDP header
- `DOCA_FLOW_RSS_TCP` – RSS by TCP header

The following is an RSS forwarding example:

```
fwd->type = DOCA_FLOW_FWD_RSS;
fwd->rss_queues = queues;
fwd->rss_flags = DOCA_FLOW_RSS_IP | DOCA_FLOW_RSS_UDP;
fwd->num_of_queues = 4;
fwd->rss_mark = 0x1234;
```

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.

MARK is an optional parameter that may be communicated to the software. If MARK is set and the packet arrives to the software, the value can be examined using the software API. When DPDK is used, MARK is placed on the struct `rte_mbuf`. (See "Action: MARK" section in

official DPDK documentation.) When using the Kernel, the MARK value is placed on the struct `sk_buff` MARK field.

The `port_id` is given in struct `doca_flow_port_cfg`.

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

Example code for forwarding to port:

```
struct doca_flow_fwd *fwd = malloc(sizeof(struct doca_flow_fwd));
memset(fwd, 0, sizeof(struct doca_flow_fwd));
fwd->type = DOCA_FLOW_FWD_PORT;
fwd->port_id = port_cfg->port_id;
```

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

2.3.5. Pipe Create

Once all parameters are defined, the create function is called.

```
struct doca_flow_pipe *
doca_flow_create_pipe(const struct doca_flow_pipe_cfg *cfg,
                      const struct doca_flow_fwd *fwd,
                      const struct doca_flow_fwd *fwd_miss,
                      struct doca_flow_error *error);
```

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.

It is possible skip optional fields. For example, `fwd` and `fwd_miss` are not mandatory, and in pipe configuration some of the fields might be zeroed when not used. 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 any entry is added to PIPE2.

2.3.6. Pipe Entry (doca_flow_pipe_add_entry)

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

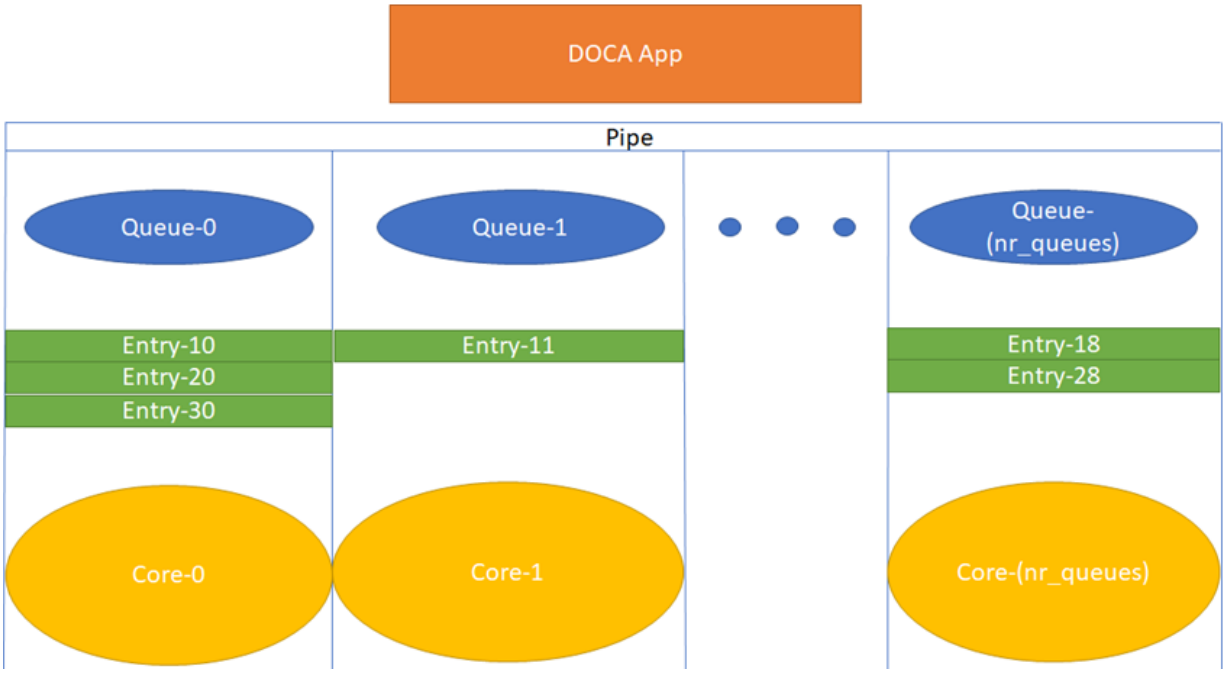
When adding an entry, users should define the values that are not constant among all entries in the pipe. And if FORWARDING is not defined then that is also mandatory.

```
struct doca_flow_pipe_entry *
doca_flow_pipe_add_entry(uint16_t pipe_queue,
                         struct doca_flow_pipe *pipe,
```

```
                                const struct doca_flow_match *match,
                                const struct doca_flow_actions *actions,
                                const struct doca_flow_monitor *monitor,
                                const struct doca_flow_fwd *fwd,
                                struct doca_flow_error *error);
```

2.3.6.1. Pipe Entry Queue

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 required to use similar architecture as 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 HW.



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

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

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

2.3.6.2. Pipe Entry Counting

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

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

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

The struct `doca_flow_query` contains the following elements:

- total_bytes – total bytes hit
- total_ptks – total packets hit

```
int doca_flow_query(struct doca_flow_pipe_entry *entry, struct doca_flow_query
 *query_stats);
```

2.3.6.3. Pipe Entry Aged Query

This query is used to get the aged-out entries by the time quota in microseconds. The entry handle or the `user_data` input are returned by this API.

Since the number of flows can be very large, the query of aged flows is limited by a quota in microseconds. This means that it may return without all flows and requires the user to call it again. When the query has gone over all flows, a full cycle is done.

The function returns:

- > 0 – the number of aged flows filled in entries array
- 0 – no aged entries in current call, but cycle is not over
- -1 – full cycle of this query is done

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

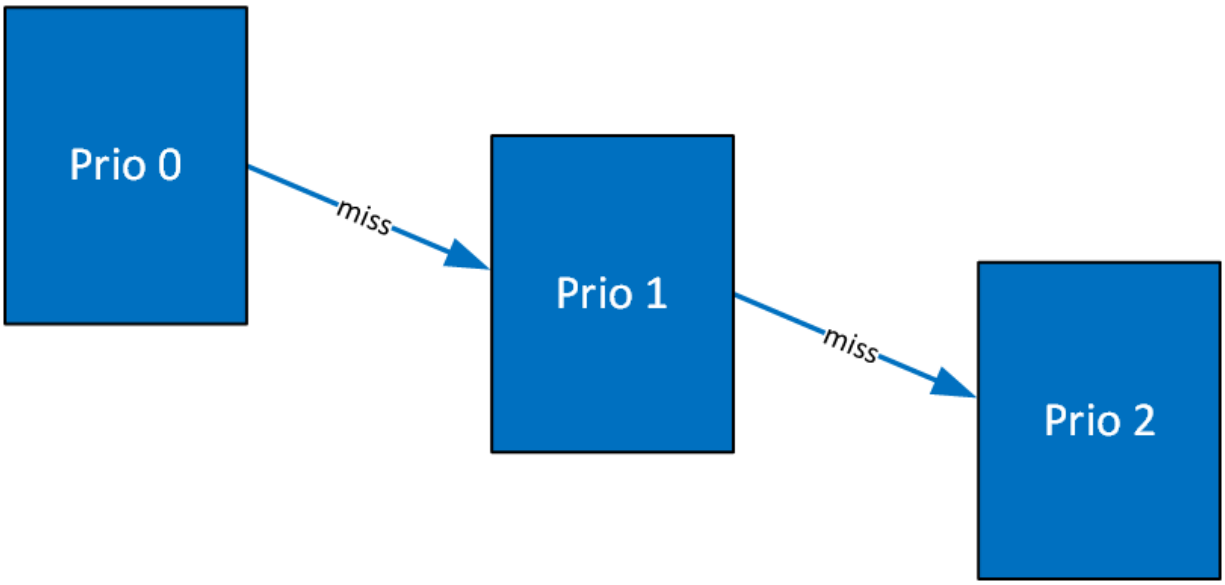
```
int doca_flow_handle_aging(uint16_t queue,
                     uint64_t quota,
                           struct doca_flow_aged_query *entries,
                           int len);
```

2.3.7. Miss Pipe and Control Pipe

Note: On the root table, the user is not allowed to enter overlapping matches. If they do so, the match behavior is unpredictable. Multiple root tables are supported, but it is the user's responsibility to make sure entries do not overlap between root tables.

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 lookup on entries associated with pipe X+1.

The following figure illustrates the HW table structure:



The first lookup is performed on the table with priority 0. If no hits are found, then jump to the next table and perform another lookup.

The way to implement miss-pipe in DOCA flow is to use miss-pipe in FWD. In `struct doca_flow_fwd`, the field `miss_pipe` signifies that, when creating a pipe, if a miss-pipe is configured then on if a packet does not match the specific pipe, steering should jump to `miss_pipe`.

`miss_pipe` is defined as `doca_flow_pipe` and created by `doca_flow_create_pipe`. 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 `miss_pipe`.

```
struct doca_flow_pipe *
doca_flow_create_pipe(const struct doca_flow_pipe_cfg *cfg,
                      const struct doca_flow_fwd *fwd,
                      const struct doca_flow_fwd *fwd_miss,
                      struct doca_flow_error *error)
```

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_create_pipe`, 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 fowrarding 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. The control pipe is created without any configuration except for the port. Then the user can add different matches with different forwarding and priorities when there are conflicts.

```
struct doca_flow_pipe *
doca_flow_create_control_pipe(struct doca_flow_port *port,
                              struct doca_flow_error *error)
```

The user can add a control entry using:

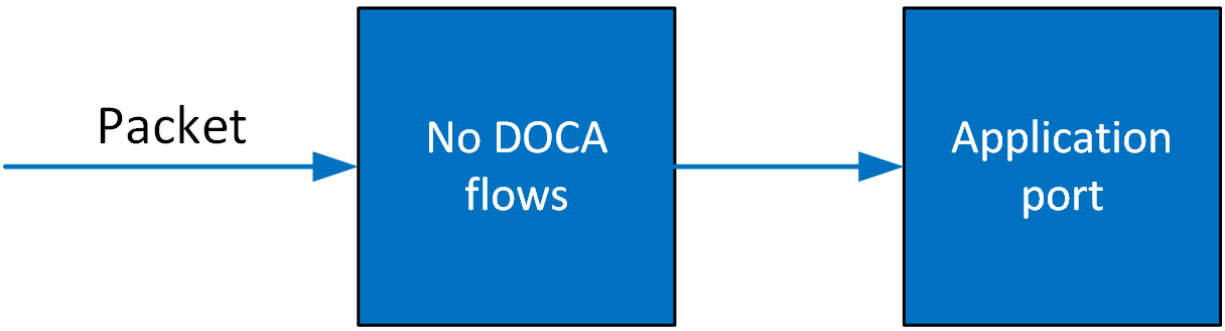
```
struct doca_flow_pipe_entry*
doca_flow_control_pipe_add_entry(uint16_t pipe_queue,
                                 uint8_t priority,
                                 struct doca_flow_pipe *pipe,
                                 const struct doca_flow_match *match,
                                 const struct doca_flow_match *match_mask,
                                 const struct doca_flow_fwd *fwd,
                                 struct doca_flow_error *error)
```

`priority` must be defined as higher than the lowest priority and lower than the highest one.

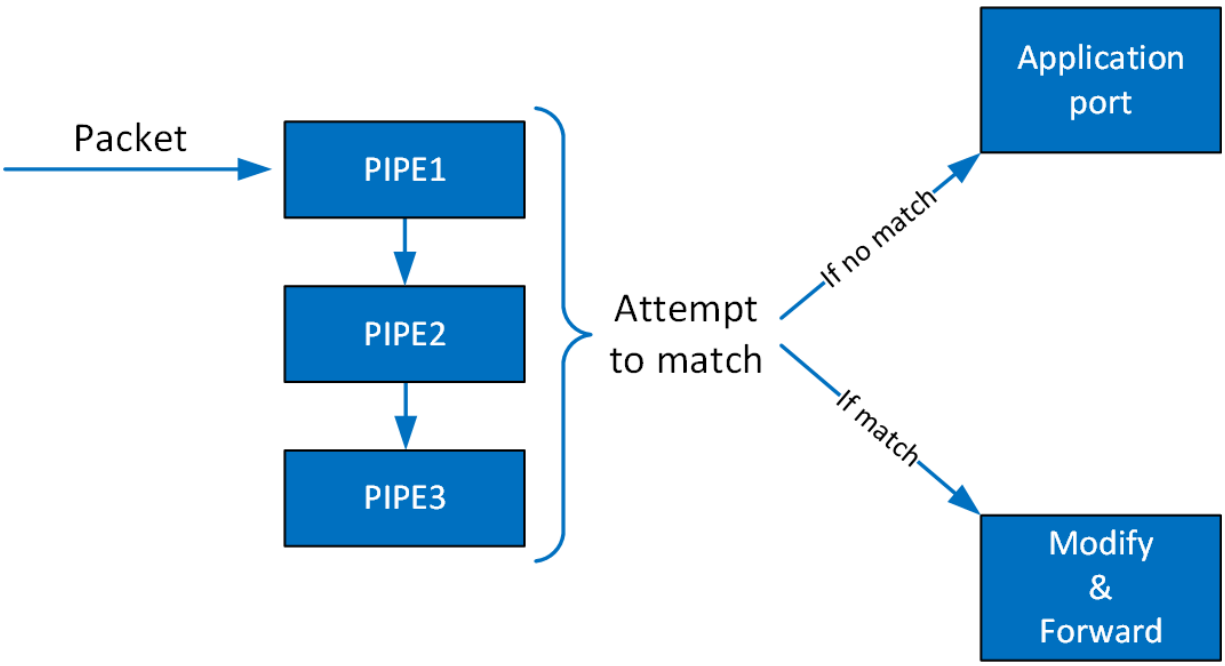
The other parameters represent the same meaning of the parameters in `doca_flow_create_pipe`. In the example above, a control entry for VXLAN is created. The VLXAN packets with `ipB` hit the control entry.

Chapter 3. 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.



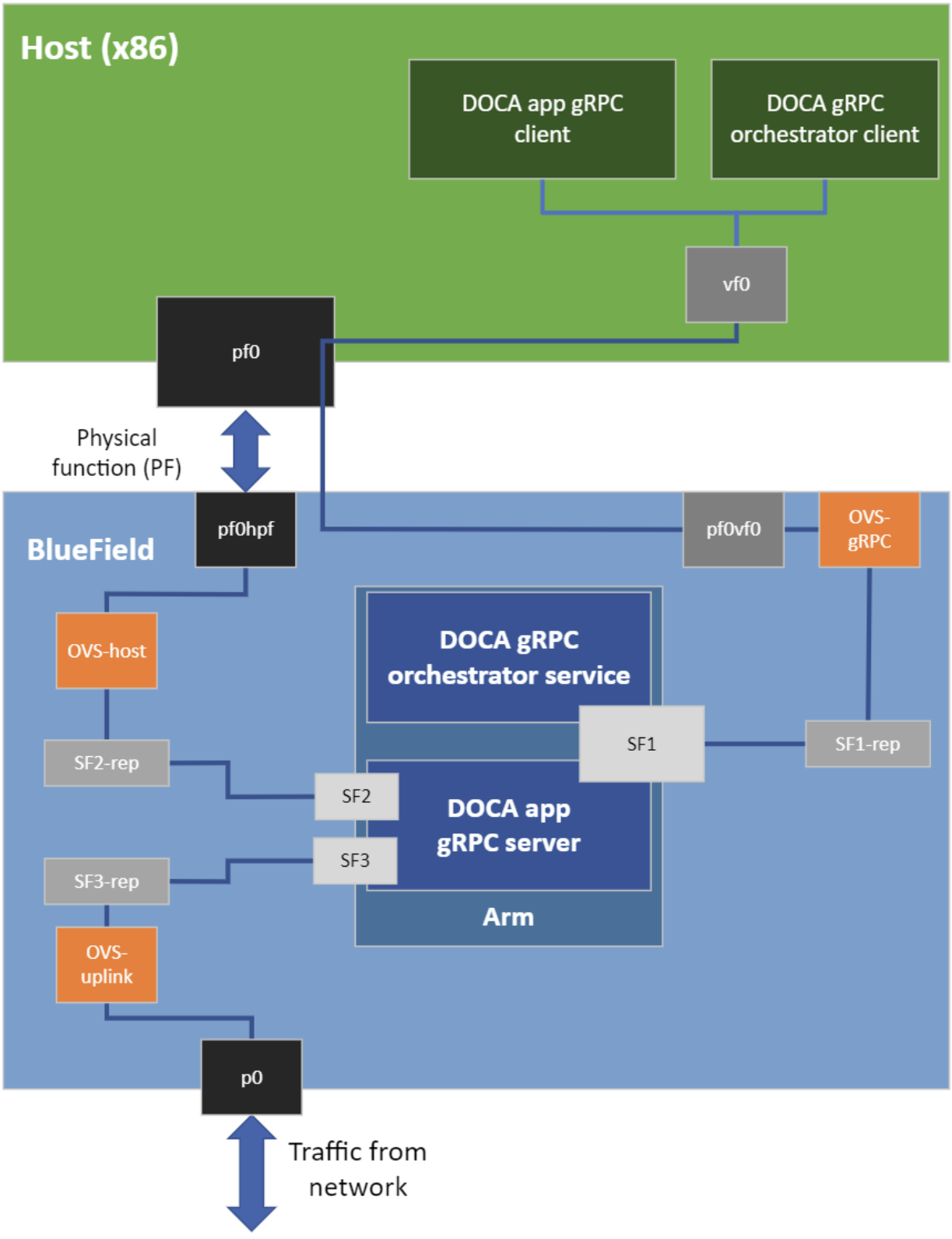
If the packet is forwarded in RSS, the packet is forwarded to software according to the RSS definition. If the packet is forwarded to a port, the packet is redirected back to the wire. If the packet is forwarded to the next pipe, then the software attempts to match it with the next pipe.

Note that the number of pipes impacts performance. The longer the number of matches and actions the packet goes through, the longer it takes the HW to process it. When there is a very large number of entries, the HW needs to access the main memory to retrieve the entry context which increases latency.

Chapter 4. DOCA Flow gRPC

The DOCA flow gRPC-based API allows users on the host to leverage the HW offload capabilities of the BlueField DPU using gRPC calls from the host itself. Refer to NVIDIA DOCA gRPC Infrastructure User Guide for more information about DOCA gRPC support.

The following figure illustrates the DOCA flow gRPC server-client communication.


Host (x86)
DOCA app gRPC client
DOCA gRPC orchestrator client
vf0
pf0
Physical function (PF)
BlueField
pf0hpf
pf0vf0
OVS-gRPC
OVS-host
DOCA gRPC orchestrator service
SF1
SF1-rep
SF2-rep
SF2
DOCA app gRPC server
SF3
SF3-rep
Arm
OVS-uplink
p0
Traffic from network

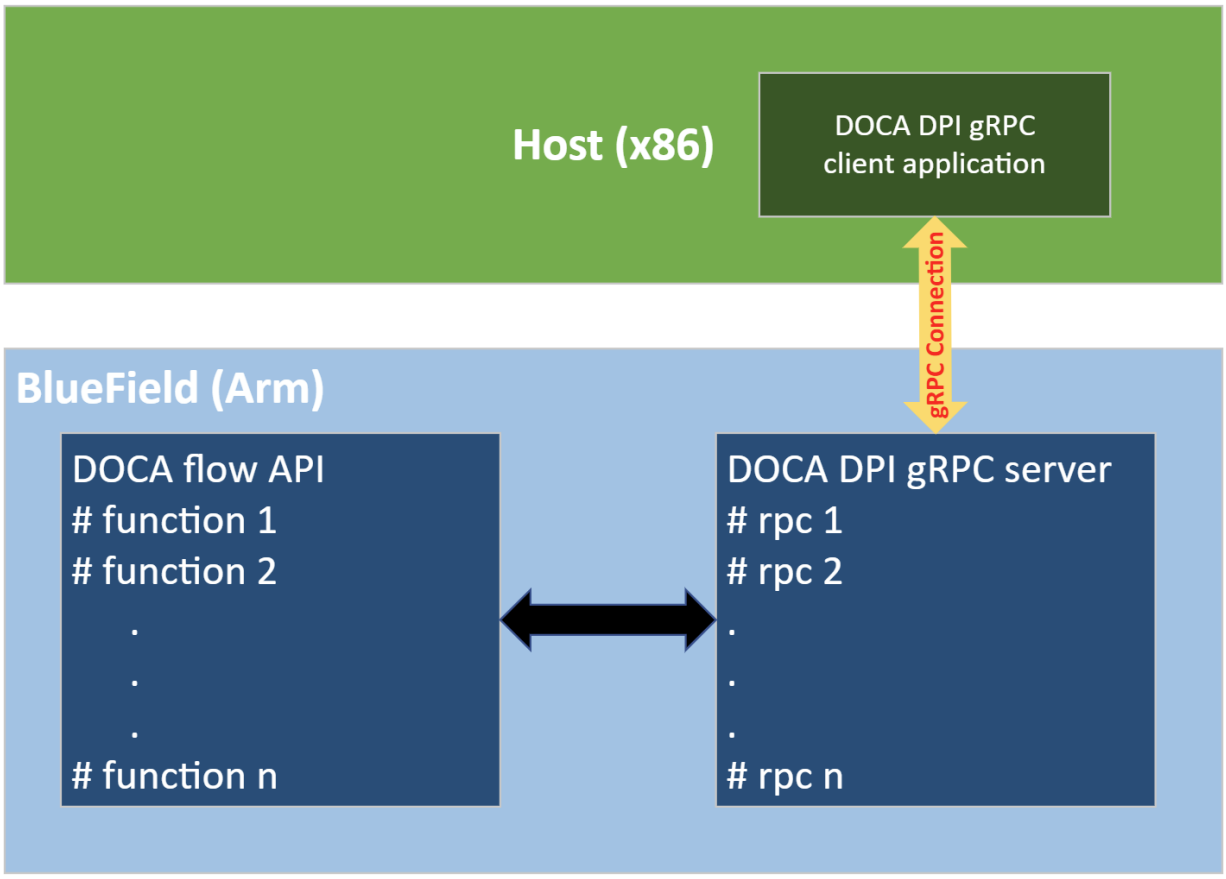
4.1. Proto-Buff

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

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

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

Each message defined in the proto-buff represents exactly one struct defined by DOCA flow API. The following figure illustrates how DOCA flow gRPC server represents the DOCA flow API.



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

4.2. Usage

The gRPC flow is dependant on the same constraints as the regular flow API, and its RPCs must be called in the same manner. Each struct defined in DOCA flow is replaced with its corresponding message in DOCA flow gRPC.

The arguments of each function in DOCA flow API are replaced with one request message that contains all needed arguments, which are represented as messages or scalar types, for the specific function.

Pointers to structs used in DOCA flow are replaced with unique IDs in the corresponding messages. For example, in `DocaFlowFwd` the `next_pipe` field is an ID representing the field `next_pipe` in `struct doca_flow_fwd` which is a pointer to struct `doca_flow_pipe`. The same holds for pointers to entries.

For more information about the sections that follow, refer to the regular flow API guide or the `doca_flow.h` file.

4.2.1. Response Message

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

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

4.2.2. Init

Before using any DOCA flow method, it is mandatory to call the DOCA flow initialization method:

```
rpc DocaFlowInit(DocaFlowCfg) returns (DocaFlowResponse);
```

In case of success, the success field is set to true. Otherwise, the error field will contain the error information. The `DocaFlowCfg` message represents the `doca_flow_cfg` struct and contains the following elements:

```
/** DOCA Flow configuration */
message DocaFlowCfg {
        uint32 total_sessions = 1;      /**< Total number of sessions */
        uint32 queues = 2;              /**< Queue ID for each offload thread */
        bool is_hairpin = 3;            /**< When true, the fwd will be hairpin */
        bool aging = 4;                 /**< When true, aging is handled by DOCA */
}
```

4.2.3. Port

The `DocaFlowPort` message represents the `doca_flow_port` struct and contains the `port_id`:

```
/** DOCA Flow port */
message DocaFlowPort {
        uint32 port_id = 1;
}
```

The service for starting the DOCA flow ports:

```
rpc DocaFlowPortStart(DocaFlowPortCfg) returns (DocaFlowResponse);
```

In case of success, the success field in `DocaFlowResponse` is set to true. Otherwise, the error field will contain the error information.

The `DocaFlowPortCfg` message represents the `doca_flow_port_cfg` struct and contains the following elements:

```
/** DOCA Flow port configuration for DOCA Flow port start */
message DocaFlowPortCfg {
        uint32 port_id = 1;             /**< DPDK port ID */
        DocaFlowPortType type = 2;      /**< Port mapping type */
        string devargs = 3;             /**< specific per port type cfg */
        uint32 priv_data_size = 4;      /**< user private data size */
}
```

4.2.4. Pipe

4.2.4.1. Setting Pipe Match

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

The `DocaFlowMatch` message contains fields of types `DocaFlowIPAddress` and `DocaFlowTun`. These types are messages which are also defined in the `doca_flow_service.proto` file.

As described before, the match field type can be defined either implicitly or explicitly.

4.2.4.1.1. Implicit Match Example

To match implicitly, the following should be considered. The following is an example of a match on the VXLAN tunnel, where for each entry there is a specific IPv4 destination address, and an inner 5-tuple.

```
ip4_full_mask = doca_flow_service_pb2.DocaFlowIpAddress(type = IP4_ADDR,ipv4_address
 = 0xffffffff)
tun = doca_flow_service_pb2.DocaFlowTun(type = TUN_VXLAN , vxlan_tun_id =
 0xffffffff)
match = doca_flow_service_pb2.DocaFlowMatch(out_dst_ip = ip4_full_mask,
       out_l4_type = PROTO_UDP, out_dst_port=VXLAN_DEFAULT_PORT, tun = tun,
       in_src_ip = ip4_full_mask, in_dst_ip = ip4_full_mask,
       in_l4_type = PROTO_TCP, in_src_port = 0xffff , in_dst_port = 0xffff)
```

4.2.4.1.2. Explicit Match Example

Users may provide a mask on match. In this case, there are two `DocaFlowMatch` items: The first contains constant values and the second contains masks. These are fields that have a

constant value. For example, as shown in the following, the tunnel type is VXLAN and the mask should be full:

```
match.tun.type#= DOCA_FLOW_TUN_VXLAN;
match_mask.tun.type#= 0xffffffff;
```

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

4.2.4.2. Setting Pipe Actions

The `DocaFlowActions` message represents `doca_flow_actions` struct.

4.2.4.3. Setting Pipe Monitor

If a policer is 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 `DocaFlowMonitor` message represents the `doca_flow_monitor`.

4.2.4.4. Setting Pipe Forwarding

The `DocaFlowFwd` message represents `doca_flow_fwd` struct.

4.2.5. Pipe Create

The `DocaFlowCreatePipeRequest` message contains all the necessary information for pipe creation as DOCA flow API suggests:

```
message DocaFlowCreatePipeRequest {
        DocaFlowPipeCfg cfg = 1;            /**< the pipe configurations */
        DocaFlowFwd fwd = 2;                /**< the pipe's FORWARDING component  */
        DocaFlowFwd fwd_miss = 3;           /**< The FORWARDING miss component */
}
```

Once all parameters are defined, a "create pipe" service can be called:

```
rpc DocaFlowCreatePipe (DocaFlowCreatePipeRequest) returns (DocaFlowResponse);
```

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

In case of success, the success field in `DocaFlowResponse` is set to true and the `pipe_id` field will hold the pipe ID of the created pipe. This ID should be given when adding entries to pipe. Otherwise, the error field is filled accordingly.

4.2.6. Pipe Entry

`DocaFlowPipeAddEntryRequest` contains all the necessary information for adding entry to the pipe, that is:

```
message DocaFlowPipeAddEntryRequest{
        uint32 port_id = 1;                 /**< the port ID to add the entry to */
        uint32 pipe_queue = 2;              /**< the pipe queue */
        uint32 pipe_id = 3;                 /**< the pipe ID to add the entry to */
        DocaFlowMatch match = 4;            /**< matcher for the entry */
        DocaFlowActions actions = 5;        /**< actions for the entry */
        DocaFlowMonitor monitor = 6;        /**< monitor for the entry */
        DocaFlowFwd fwd = 7;                /**< The entry's FORWARDING component */
}
```

Once all parameters are defined, the "add entry" to pipe service can be called.

```
rpc DocaFlowPipeAddEntry(DocaFlowPipeAddEntryRequest) returns (DocaFlowResponse);
```

If successful, the success field in `DocaFlowResponse` is set to true and the `entry_id` field will hold the entry ID of the added entry. Otherwise, error field is filled accordingly.

4.2.6.1. Pipe Entry Queue

`DocaFlowRemoveEntryRequest` ontains all the necessary information for removing entry from a pipe, that is:

```
message DocaFlowRemoveEntryRequest{
        uint32 pipe_queue = 1;              /**< the pipe queue of the entry to
 remove */
        uint32 entry_id = 2;                /**< the entry ID to be removed */
        uint32 pipe_id = 3;                 /**< the pipe ID that the entry is
 attached to */
        uint32 port_id = 4;                 /**< the port ID that the entry is
 attached to */
}
```

Once all parameters are defined, the "remove entry" service can be called.

```
rpc DocaFlowRemoveEntry(DocaFlowRemoveEntryRequest) returns (DocaFlowResponse);
```

If successful, the success field in `DocaFlowResponse` is set to true. Otherwise, error field is filled accordingly.

4.2.6.2. Pipe Entry Counter

When a counter is present, it is possible to query the flow and get the counter's data using the following service:

```
rpc DocaFlowQuery(DocaFlowEntryQueryRequest) returns (DocaFlowResponse);
```

`DocaFlowEntryQueryRequest` contains all the following fields:

```
message DocaFlowEntryQueryRequest{
        uint32 port_id = 1;                 /**< the port ID the entry is attached to
 */
        uint32 pipe_id = 2;                 /**< the pipe ID that the entry is
 attached to */
        uint32 entry_id = 3;                /**< the entry ID */
}
```

If successful, the `success` field in `DocaFlowResponse` is set to `true` and the `query_res` field will hold the query result of the entry. If a failure occurs, the service returns `false` in the success field in `DocaFlowResponse` message, and the error is filled accordingly.

4.2.6.3. Pipe Entry Aged Query

`DocaFlowHandleAgingRequest` contains all the necessary information for calling aging handler in DOCA flow API.

`DocaFlowHandleAgingRequest` contains all the following fields:

```
message DocaFlowHandleAgingRequest{
        uint32 queue = 1;       /**< the queue identifier */
        uint64 quota = 2;       /**< the max time quota in micro seconds for this
 function to handle aging *
        uint64 user_data = 3;   /**< the user input context, otherwish the
 doca_flow_pipe_entry pointer */
        uint32 len = 4;         /**< the user input length of entries array */
}
```

Once all parameters are defined, the aging handler service can be called.

```
rpc DocaFlowHandleAging(DocaFlowHandleAgingRequest) returns (DocaFlowResponse);
```

If successful, the success field in `DocaFlowResponse` is set to true and the `handle_aging_res` field will hold the result of the handler. This field, `handle_aging_res`, is a field of type `DocaFlowHandleAgingRes` and contains the following fields:

```
message DocaFlowHandleAgingRes {
        uint32 res = 1;                 /***< the result of the handler, number of
 handled aged entries */
        repeated uint32 entries = 2;    /***< the aged entries' IDs */
}
```

4.2.6.4. Miss Pipe and Control Pipe

As described in earlier sections, it is possible to create a special type of pipe called "control pipe". In addition, users may add entries to the control pipe. The service for creating the control pipe:

```
rpc DocaFlowCreateControlPipe(DocaFlowPort) returns (DocaFlowResponse);
```

If successful, the `success` field in `DocaFlowResponse` is set to `true` and the `pipe_id` field will hold the pipe ID of the created control. Otherwise, error field is filled accordingly.

The service for adding entries to the control pipe is the following:

```
rpc DocaFlowControlPipeAddEntry(DocaFlowControlPipeAddEntryRequest) returns
 (DocaFlowResponse);
```

If successful, the `success` field in `DocaFlowResponse` is set to `true` and the `entry_id` field will hold the entry ID of the added entry. Otherwise, error field is filled accordingly.

The `DocaFlowControlPipeAddEntryRequest` contains the needed arguments for adding entries to the control pipe.

```
message DocaFlowControlPipeAddEntryRequest{
        uint32 port_id = 1;                 /**< the port ID to add the entry to */
        uint32 priority = 2;                /**< he priority of the added entry to
 the filter pipe */
        uint32 pipe_queue = 3;              /**< the pipe queue */
        uint32 pipe_id = 4;                 /**< the pipe ID to add the entry to */
        DocaFlowMatch match = 5;            /**< matcher for the entry */
        DocaFlowMatch match_mask = 6;       /**< matcher mask for the entry */
        DocaFlowFwd fwd = 7;                /**< The entry's FORWARDING component */
}
```

To use the "miss pipe", the user must define a `miss_fwd` component in the `DocaFlowCreatePipeRequest` when calling the `DocaFlowCreatePipe` service. When defining the `miss_fwd` component, the user must make sure the definition of `miss_fwd` is `FORWARDING` to a next pipe. For more information about the miss pipe, see Miss Pipe and Control Pipe.

Note: The default value of each field in a created object of a specific message is zeroed. This may be considered in many cases (e.g., when defining a match for a pipe).

For more information, and deployment instructions for the gRPC-enabled DOCA flow server, refer to NVIDIA DOCA gRPC Infrastructure User Guide.

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation nor any of its direct or indirect subsidiaries and affiliates (collectively: "NVIDIA") make no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assume no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks

NVIDIA, the NVIDIA logo, and Mellanox are trademarks and/or registered trademarks of Mellanox Technologies Ltd. and/or NVIDIA Corporation in the U.S. and in other countries. The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a world¬wide basis. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
http://www.nvidia.com