NVIDIA DOCA DPI

Programming Guide

MLNX-15-060466 _v1.2 | January 2022

Table of Contents

Chapter 1. Introduction

Deep packet inspection (DPI) is a method of examining the full content of data packets as they traverse a monitored network checkpoint.

DPI provides a more robust mechanism for enforcing network packet filtering as it can be used to identify and block a range of complex threats hiding in network data streams more accurately. This includes:

- Malicious applications
- Malware data exfiltration attempts
- Content policy violations
- Application recognition
- Load balancing

1.1. Intended Audience

This document is intended for software developers writing DPI-based applications such as application recognition (AR), intrusion prevention system (IPS), and intrusion detection system (IDS).

The document assumes familiarity with the TCP/UDP stack and data plane development kit (DPDK).

1.2. Changes and New Features in 1.1

This section provides information regarding the features added and changes made in this software version.

- Added support for IP-based matching
- Added support for non-TCP/UDP streams
- Added support for IP-based signatures
- Added support for action field in signatures

Chapter 2. Setup Configuration

DPI-based application can run either on the host machine, or on the NVIDIA® BlueField® DPU target. As the DPI leverages the Regular Expressions (RegEx) Engine, users must make sure it is enabled.

1. Ensure that your DPU is operating in Ethernet mode, please refer to the DOCA Installation Guide for more information.
2. Ensure that the BlueField DPU is running in embedded CPU function (ECPF) mode (default).

   Note: Refer to DPU Modes of Operation > "Configuring ECPF Mode from Separated Host Mode" under DPU Runtime Guides in the SDK DOCA Developer Zone documentation.

3. The RegEx engine is enabled by default on the DPU. However, to use DPI directly on the host, run:

```
host> sudo /etc/init.d/openibd stop
dpu> echo 1 > /sys/class/net/p0/smart_nic/pf/regex_en
dpu> cat /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
400
# Make sure to allocate 200 additional hugepages
dpu> echo 600 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
dpu> systemctl start mlx-regex
# Verify the service is properly running
dpu> systemctl status mlx-regex
host> sudo /etc/init.d/openibd start
```

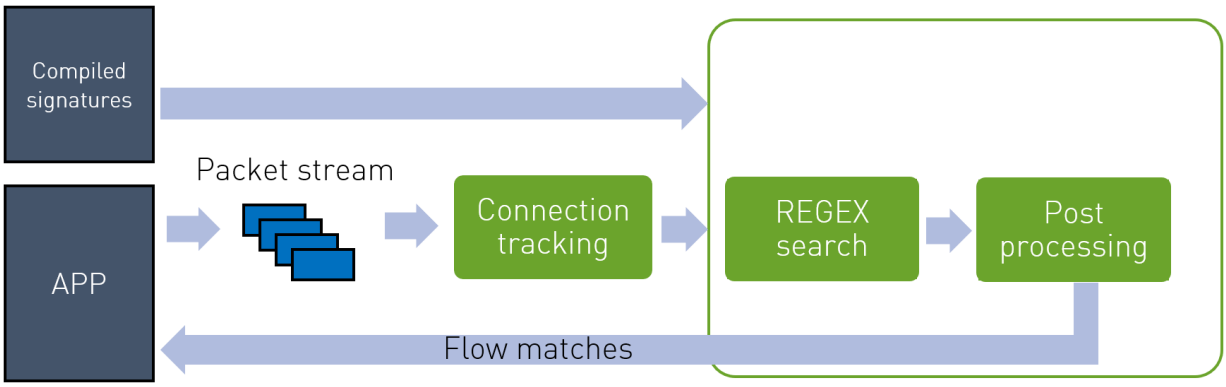
   Note: Commands with the `host` prompt must be run on the host. Commands with the `dpu` prompt must be run on BlueField (Arm).

2.1. Known Issues

- The DOCA DPI library only supports inspection of the following protocols:
  - http 2.0/1.1/1.0
  - TLS/SSL ClientHello and certificate messages
  - DNS
  - FTP
- TCP/UDP stream-based signatures may detect applications on other protocols

Chapter 3. DPI Architecture

The following diagram shows how packets are identified by the connection tracking protocol and then injected into the DPI library for processing.



3.1. Signature Database

The signature database is compiled into a CDO file by the DPI compiler. The CDO file includes:

- Post-processing table
- Compiled RegEx engine rules
- Other signature information

The application may load a new database while the DPI is processing packets.

For more information on DPI compiler, please refer to the DOCA DPI Compiler document.

3.2. DPI Queue

A DPI queue is designed to be used by a worker thread. The DPI queue holds the flow's state. Therefore, all packets from both directions of the flow must be submitted to the same DPI queue "in order". The connection tracking logic will handle out of order packets or retransmission.

3.3. Connection Tracking

For the DPI library to process cross-packet content, each packet must be injected along with a flow context and a direction. Packets from the same flow direction must be injected "in order". A flow direction is usually represented by a 5-tuple, but it can also be a 3-tuple for other protocols.

The connection tracking (CT) logic must handle out of order packets as well as fragmented packets. Once a connection has timed out or terminated, the application must notify the DPI library as well.

Chapter 4. DPI Initialization and Teardown

Before enqueueing packets for processing, the DPI library must be initialized and loaded with signatures by the main thread:

```
struct doca_dpi_ctx *dpi_ctx = doca_dpi_init(doca_dpi_config);
      doca_dpi_load_signatures(dpi_ctx, cdo_filename);
```

The following configuration parameters are available:

- nb_queues – number of DPI queues
- max_packets_per_queue – maximum number of packets to be concurrently processed per queue
- max_sig_match_len – maximum signature length guaranteed to be matched by the DPI library

  For example: A.*B and max_sig_match_len = 4 guarantees to match AxxB but does not guarantee to match AxxxB.

To close the DPI library, the user should call the following function:

```
doca_dpi_destroy(dpi_ctx)
```

Chapter 5. Packet Processing

5.1. Flow Life Cycle

- Once a new flow was detected by the connection tracking SW, the user should call `doca_dpi_flow_create()`
- Every incoming packet classified for this flow should be enqueued by `calling doca_dpi_enqueue()`
- To poll for the results, the application must call `doca_dpi_dequeue()`. The result will contain matching information (if matched).
- When the connection tracking SW detected that the flow was terminated or aged-out, the application should notify the DPI library by calling `doca_dpi_flow_destroy()`

5.2. Enqueueing Packets for Processing

A call to `doca_dpi_enqueue()` may reject packets for processing for the following reasons:

- Packet is empty
- DPI queue is full (`doca_dpi_dequeue()` must be called first)

5.2.1. Packet Ownership

For every mbuf injected the user is not allowed to free the mbuf until the mbuf is dequeued.

If an external attach is used, users must follow the DPDK guidelines for `rte_pktmbuf_attach_extbuf()` to make sure the mbuf is freed when both the user and the DPI free the mbuf.

5.2.2. Flow Matching

A flow may match one or more signatures. The match result will be available to the application on `doca_dpi_dequeue()`. The DPI library will only report the matched signature with the highest priority. Another way to see the match result for a given flow is to use the function `doca_dpi_flow_match_get()`.

The application may query for the application name using `doca_dpi_signature_get()`. To preserve performance, it is not recommended to call these functions while packets are being processed.

It is recommended that the application call `doca_dpi_signatures_get()` after loading the database to acquire a copy of the signature names.

Chapter 6. Performance

6.1. Multithreading

The DPI library is designed to achieve optimal results in a multi-threaded environment. To achieve best performance, it is recommended that both the packet acquisition and the DPI processing will be done by the same thread.

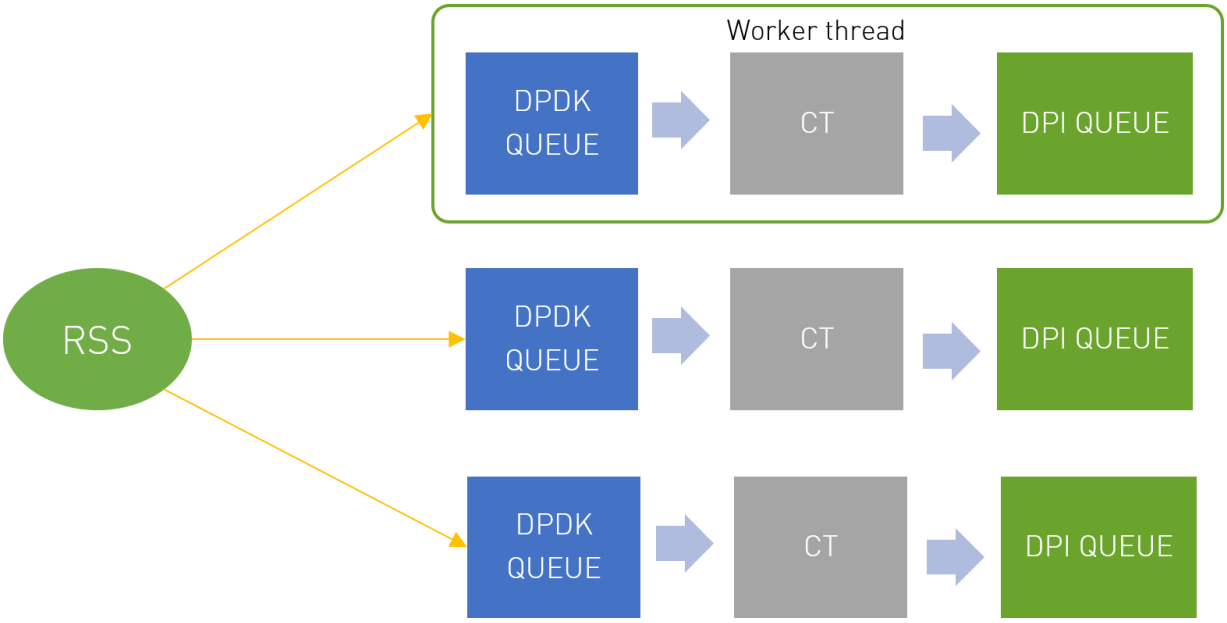
Because some of the DPI work is offloaded to the HW, it is highly recommended that the worker thread will work in a pipeline mode, meaning, it should never wait for a DPI job to be completed but rather go and fetch more packets to be processed. This way the SW can best utilize the CPU while the RegEx accelerator is processing the job.

The following pseudocode shows the recommended way to call the DPI library:

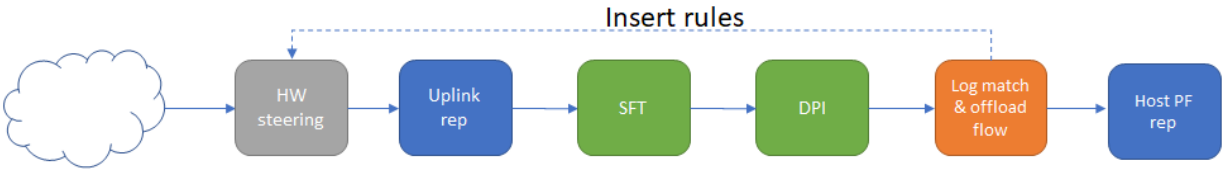
```
while(true) {
   mbufs = rx_burst()
   foreach mbuf in mbufs {
     flow_id = connection_tracking(qid, mbuf)
     if (new flow)
        doca_dpi_flow_create(qid, flow_id, parsing_info)
     status = doca_dpi_enqueue(flow_ctx, mbuf, offset)
     if (status ....)
   }
   while(doca_dpi_dequeue(qid, &result) == DOCA_DPI_DEQUEUE_READY)
     ... inspect result ...
   // At this point processing may not be completed for all packets, so the worker
   // should continue handling more incoming packets.
   }
```

6.2. RSS and RTE_FLOW

The packets of each flow must be submitted exclusively to the same queue, for both directions. This may be achieved by either using a symmetric RSS or manually (using `rte_flow`) directing both directions of the flow to the same DPDK queue.


Worker thread
DPDK QUEUE
CT
DPI QUEUE
RSS
DPDK QUEUE
CT
DPI QUEUE
DPDK QUEUE
CT
DPI QUEUE

Chapter 7. Packet Life Cycle Example

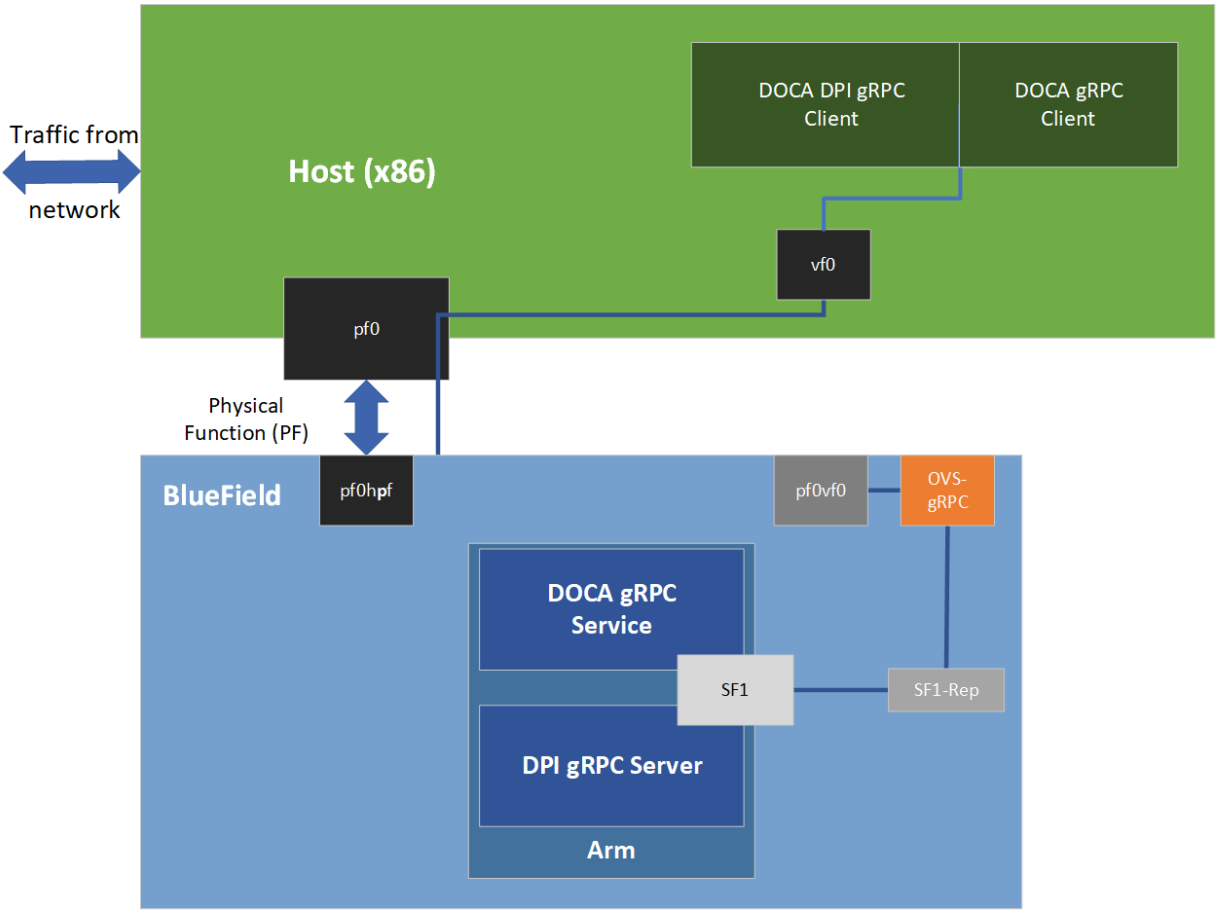


1. The packet is sent to the SFT for processing by calling `sft_process_packet()` to see if the hardware recognizes the flow.
2. If the packet is not marked with a zone ID by the HW, the SW must explicitly inform the SFT about the zone of the packet by using `sft_process_packet_with_zone()`.
3. If the packet is not marked with a flow ID by the HW or the SW, a new flow is created by calling `sft_activate()`.
4. If a new flow ID is assigned by the SFT, `doca_dpi_flow_create()` must be invoked before enqueuing the packet.
5. The packet is then processed by the DPI by calling `doca_dpi_enqueue()`.
6. If the packet is accepted by the DPI for processing, the result is dequeued by calling `doca_dpi_dequeue()`.
7. If a match is found, the result is printed and counted for statistics. The flow is then offloaded (sent directly to the host) because no further inspection is required.
8. To retrieve the match from the DPI engine, `doca_dpi_signature_get()` allows accessing the `sig_data` struct which contains the signature ID and message string. This action might affect DPI performance.
9. When the flow is terminated by the SFT, it should also be destroyed by invoking `doca_dpi_flow_destroy()` with the corresponding flow ID.
10. Additional statistics can be retrieved using `doca_dpi_stat_get()`.

Chapter 8. DOCA DPI gRPC

This section describes the gRPC (Google remote procedure calls) support for DOCA DPI API. The DOCA DPI gRPC-based API, allows users on the host to leverage the HW offload capabilities of the BlueField-2 DPU using gRPC calls from the host itself. For more information about gRPC support in DOCA, refer to the DOCA gRPC User Guide.

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



8.1. Proto-buff

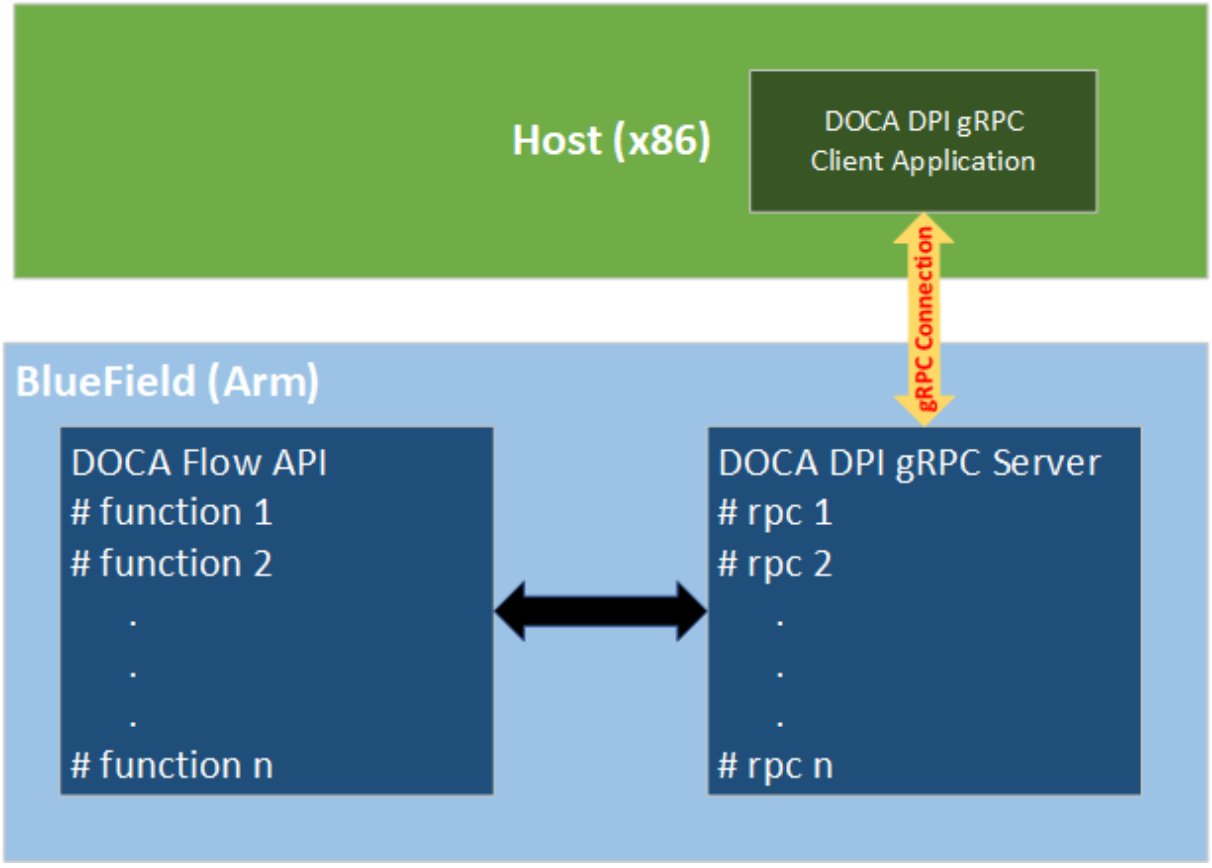
As with every gRPC proto-buff, DOCA DPI gRPC proto-buff defines the service it introduces and the messages used for communication between the client and the server.

Users provide two proto-buff files:

- `doca_dpi_types.proto` – defines message representation of DOCA DPI API structs and enums
- `doca_dpi.proto` – defines the service, its RPCs, the request, and response objects

Each message defined in `doca_dpi_types.proto` with the `DocaDpi` prefix represents exactly one struct or enum defined by DOCA DPI API.

The following figure illustrates how DOCA DPI gRPC server represents the DOCA DPI API.



The proto-buff path for DOCA DPI gRPC is:

- `/opt/mellanox/doca/infrastructure/doca_grpc/doca_dpi/doca_dpi.proto`
- `/opt/mellanox/doca/infrastructure/doca_grpc/doca_dpi/doca_dpi_types.proto`

8.2. Usage

Similarly to the regular DPI API, the gRPC DPI is dependent on the same constrains, and its RPCs must be called in the same manner, except for `doca_dpi_init` which is invoked by the service upon loading instead via gRPC call.

The host must use a connection tracking mechanism to provide the DOCA DPI gRPC service with ordered and unfragmented packets.

The gRPC API replaces the usage of `struct rte_mbuf` packets with `message DocaDpiGenericPacket` which is not protocol-dependent and is detailed in the Enqueue section.

For more information about the sections that follow, please refer to the regular DPI API guide (chapters 1-7) or the `doca_dpi.h` file.

8.2.1. DPI and Flow Context

Unlike direct usage of DOCA DPI API, no `struct doca_dpi_ctx` is passed to the client. Instead, it is saved on the service and used when needed.

`DocaDpi_FlowCtx` contains a unique ID to identify the real flow context instance:

```
message DocaDpi_FlowCtx {
 uint64 unique_id = 1;
}
```

8.2.2. DPI Queues

```
message DocaDpiDequeueParams {
 uint32 uint16_dpi_q = 1; /* The DPI queue from which to dequeue the flows packets.
 */
}
message DocaDpiFlowCreateParams {
         DocaDpiParsingInfo parsing_info = 1;
 uint32 uint16_dpi_q = 2; /* The DPI packets queue the flow will be assigned to. */
}
```

When creating a new flow context, the mandatory parameter `uint16_dpi_q` refers to an inner packets-queue to link the flow to. Its value can be arbitrary. However, once a flow is linked with a queue, all messages of that flow will arrive to the same queue.

When dequeuing a packet, the mandatory parameter `uint16_dpi_q` refers to the inner packets-queue from which to dequeue a packet.

The parameter `uint16_dpi_q` can be anything between 0 and the number of lcores the BlueField DPU has minus 1.

8.2.3. Errors

Errors are returned as:

```
message DocaDpiErrorInfo {
 int64 error_code = 1;
 optional string err_msg = 2;
}
```

An `error_code` with non-zero value indicates that an error occurred. Then, and only then, the error message is presented in the `err_msg` field.

8.2.4. Signature Database

```
rpc DocaDpiLoadSignatures (DocaDpiLoadSignaturesParams) returns (DocaDpiErrorInfo);
message DocaDpiLoadSignaturesParams {
 oneof cdo {
  string cdo_filename = 1;  /* Path, on the DPU, to a cdo file */
  bytes cdo_data = 2;  /* Content of a cdo file. */
 }
}
```

The CDO file must be produced in the same manner as in regular DPI API.

This differs from the regular `doca_dpi_load_signatures` by including the option to send a CDO file instead of a path to a CDO file on BlueField:

- Like the regular `doca_dpi_load_signatures` the argument `cdo_filename` is a path to a CDO file on the DPU.
- Unlike the regular `doca_dpi_load_signatures` an entire CDO file can be sent inside `cdo_data` instead of using a path.

8.2.5. Destruction

```
rpc DocaDpiDestroy (DocaDpiDestroyParams) returns (DocaDpiDpiDestroyResponse);
```

The service can properly exit by calling the RPC method `DocaDpiDestroy`. When invoked, the service destroys all current flow contexts and exits with code 0.

8.2.6. Additional gRPC APIs

```
rpc DocaDpiDequeue (DocaDpiDequeueParams) returns (DocaDpiActionResponse);
rpc DocaDpiFlowCreate (DocaDpiFlowCreateParams) returns (DocaDpiFlowCreateResponse);
rpc DocaDpiFlowDestroy (DocaDpiFlowDestroyParams) returns
 (DocaDpiFlowDestroyResponse);
rpc DocaDpiFlowMatchGet (DocaDpiFlowMatchGetParams) returns (DocaDpiActionResponse);
rpc DocaDpiSignatureGet (DocaDpiSignatureGetParams) returns
 (DocaDpiSignatureGetResponse);
rpc DocaDpiSignaturesGet (DocaDpiSignaturesGetParams) returns
 (DocaDpiSignaturesGetResponse);
rpc DocaDpiStatGet (DocaDpiStatGetParams) returns (DocaDpiStatInfo);
```

These methods work in the same manner as the regular DPI API. For more information on `DocaDpiActionResponse`, see section Matching.

8.3. Multi-Processing and Multithreading

8.3.1. Enqueue

```
rpc DocaDpiEnqueue (DocaDpiEnqueueParams) returns (DocaDpiErrorInfo);
message DocaDpiGenericPacket {
 bytes segment = 1;  /* The packet data, max length is 65535 (0xffff). */
}
```

The proto-buff type `DocaDpiGenericPacket` is used to enqueue a packet with up to 65535 bytes of payload or segment (both are applicable), which can include the actual packet headers.

Enqueue RPC is thread-safe per queue. This means that:

- Multiple processes/threads can use enqueue when the flows are linked to different queues
- Multiple processes/threads cannot use enqueue when the flows are linked to the same queue

8.3.2. Matching

```
rpc DocaDpiDequeue (DocaDpiDequeueParams) returns (DocaDpiActionResponse);
rpc DocaDpiFlowMatchGet (DocaDpiFlowMatchGetParams) returns (DocaDpiActionResponse);
message DocaDpiActionResponse {
 DocaDpiErrorInfo erreno = 1;
 DocaDpiResult result = 2;  /* gRPC message equvalent to doca_dpi_result struct. */
}
```

The same DPI gRPC server can be used by many processes/threads by using one of the following 2 options:

- Use `DocaDpiFlowMatchGet` instead of `DocaDpiDequeue`. Notice that dequeuing is still needed to free up the DPI queue.
- Use a different `uint16_dpi_q` value for each process/thread. By doing so, users can make sure packets from only one processing entity reach a certain queue and that only its packets are dequeued from that queue.

There can be only one DPI server at a time due to the ownership of the HW RegEx accelerator.

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