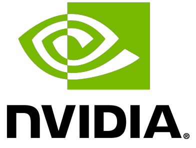


NVIDIA DOCA Comm Channel

Programming Guide

MLNX-15-060545 _v1.4 | September 2022

Table of Contents

Chapter 1. Introduction... 1
Chapter 2. Prerequisites... 2
Chapter 3. API...3
3.1. doca_comm_channel_ep_t.. 3
3.2. doca_comm_channel_init_attr.. 3
3.3. doca_comm_channel_ep_create..3
3.4. doca_comm_channel_addr_t.. 4
3.5. doca_comm_channel_ep_listen... 4
3.6. doca_comm_channel_ep_connect.. 4
3.7. doca_comm_channel_ep_sendto... 4
3.8. doca_comm_channel_rp_recvfrom...5
3.9. doca_comm_channel_ep_disconnect... 5
3.10. doca_comm_channel_ep_destroy.. 6
Chapter 4. Limitations.. 7
4.1. Message Size...7
4.2. Queue Size..7
4.3. Service Name.. 7
4.4. Multi-client... 7
4.5. Threads... 8
4.6. Devices..8
Chapter 5. Usage... 9
5.1. Objects.. 9
5.1.1. Endpoint... 9
5.1.2. Peer_address...9
5.2. Endpoint Initialization... 9
5.3. Connection Flow...10
5.4. Data Transfer Flow... 10
5.4.1. Blocking Mode.. 11
5.4.2. Non-blocking Mode... 11
5.5. Disconnection Flow...11

Chapter 1. Introduction

The DOCA Comm Channel provides a secure, network-independent communication channel between the host and the DPU.

The communication channel allows the host to control services on the DPU or to activate certain offloads.

The DOCA Comm Channel is reliable, message-based, and supports multi-client connections to a single service. The API allows communication between a client using any PF/VF/SF on the host to a service on the DPU.

Chapter 2. Prerequisites

Both the host and the DPU must meet the following criteria:

- Firmware version 24.33.0458 and higher
- DOCA Base 5.5-1.0.3.2 and higher

The client is supported both for Linux and Windows, while the server only has Linux support.

Chapter 3. API

3.1. doca_comm_channel_ep_t

This struct represents a Comm Channel endpoint either on the client or service side. The endpoint is needed for every other Comm Channel API function.

```
struct doca_comm_channel_ep_t;
```

3.2. doca_comm_channel_init_attr

This struct contains the attributes for the newly created endpoint in `doca_comm_channel_ep_create()`. See further information under Limitations.

```
 struct doca_comm_channel_init_attr {
      uint32_t    cookie;
      uint32_t    flags;
      uint32_t    maxmsgs;
      uint32_t    msgsize;
      }
```

cookie
: Cookie returned when polling the `event_channel`.

flags
: 0 or `DOCA_CC_INIT_FLAG_NONBLOCK`. Toggles non-blocking mode in created endpoint.

maxmsgs
: queue size for created endpoint.

msgsize
: maximal message size in created endpoint.

3.3. doca_comm_channel_ep_create

This function is used to create and initialize the endpoint used for all Comm Channel functions.

```
doca_error_t doca_comm_channel_ep_create(
      struct doca_comm_channel_init_attr *attr, struct doca_comm_channel_ep_t **ep);
```

attr
: Endpoint attributes structure.

ep

Output parameter, pointer to created endpoint object.

3.4. doca_comm_channel_addr_t

Referred to as `peer_address`, this struct represents a connection and whichrepresents a connection and can be used to identify the source of a received message. It. Itis required to send a message using `doca_comm_channel_ep_sendto()`.

```
struct doca_comm_channel_addr_t;
```

3.5. doca_comm_channel_ep_listen

Used to listen on service endpoint, this function can only be called on the DPU. The service listens on all available devices. Calling listen allows clients to connect to the service.

```
doca_error_t doca_comm_channel_ep_listen(struct doca_comm_channel_ep_t
 *local_ep, const char *name);
```

local_ep

Endpoint to listen on.

name

The name for the service to listen on. Clients must provide the same name to connect to the service.

3.6. doca_comm_channel_ep_connect

Used to create a connection between a client and a service, this function can only be called on the host.

The client connects to the first device it sees.

```
doca_error_t doca_comm_channel_ep_connect(
struct doca_comm_channel_ep_t *local_ep, const char *name,
struct doca_comm_channel_addr_t **peer_addr);
```

local_ep

Endpoint to listen on.

name

The name for the service to listen on. Clients must provide the same name to connect to the service.

peer_addr

Output parameter. Contains the pointer to the new connection.

3.7. doca_comm_channel_ep_sendto

This function is used to send a packet of data from one side to the other (service to the client or vice versa). This function can be called either in blocking or non-blocking mode. Refer to chapter Usage for more details.

```
doca_error_t doca_comm_channel_ep_sendto(struct doca_comm_channel_ep_t *local_ep,
```

```
const void *msg, size_t len, int flags,
struct doca_comm_channel_addr_t *peer_addr);
```

local_ep
: Endpoint to send the message from.

msg
: Pointer to the buffer that contains the data to send.

len
: Length of data to be sent.

flags
: 0 or `DOCA_CC_MSG_FLAG_DONTWAIT`, runs on blocking/non-blocking mode respectively.

peer_addr
: Peer address to send the message to (see also `struct doca_comm_channel_addr_t`).

3.8. doca_comm_channel_rp_recvfrom

This function is used to receive a packet of data on either the service or the host. This function can be called either in blocking or non-blocking mode. Refer to chapter Usage for more details.

```
doca_error_t doca_comm_channel_ep_recvfrom(
struct doca_comm_channel_ep_t *local_ep, void *msg, size_t *len, int flags, struct
 doca_comm_channel_addr_t **peer_addr);
```

local_ep
: Endpoint to send the receive the message on.

msg
: Buffer to write the message to.

len
: Size of the given buffer

flags
: 0 or `DOCA_CC_MSG_FLAG_DONTWAIT`, runs on blocking or non-blocking mode respectively.

peer_addr
: Output parameter. Handle to `peer_addr` that represents the connection the message arrived from.

3.9. doca_comm_channel_ep_disconnect

This function disconnects an endpoint from a specific `peer_address`. The disconnection is one-sided, and the other side is unaware of it. New connections can be created afterwards. Refer to chapter Usage for more details.

```
doca_error_t doca_comm_channel_ep_disconnect(
struct doca_comm_channel_ep_t *local_ep,
struct doca_comm_channel_addr_t *peer_addr);
```

local_ep
: Endpoint to disconnect.

peer_addr
: Connection to disconnect from.

3.10. doca_comm_channel_ep_destroy

This function disconnects all connections of the endpoint, destroys the endpoint object, and frees all related resources.

```
doca_error_t doca_comm_channel_ep_destroy(struct doca_comm_channel_ep_t *ep);
```

`local_ep`
: Endpoint to destroy.

Chapter 4. Limitations

4.1. Message Size

When creating an endpoint using `doca_comm_channel_ep_create()`, the user must provide the maximal message size value.

This value must be between 256 and 4080 bytes. If a smaller value is provided, it is raised internally to 256 bytes.

4.2. Queue Size

The maximal queue size (as provided in `doca_comm_channel_ep_create(), maxmsgs`) cannot exceed 8191 messages.

The minimal queue size isat least 16 messages. If a smaller value is provided, it is raised internally to 16.

4.3. Service Name

The service name must be between one character and 120 bytes long, including the terminating null byte.

4.4. Multi-client

A single service on the DPU can serve multiple clients, but a client can only connect to a single service.

In addition, two clients cannot connect to the same service from the same VF. If two clients are present on the same VF, each must connect to a different service.

4.5. Threads

The DOCA Comm Channel is not thread-safe. To use a single endpoint over multiple threads, one must use locks. Different endpoints can be used over different threads.

4.6. Devices

DOCA Comm Channel does not allow the user to choose the devices used. The Comm Channel listens on all devices available on the service side. On the client's side, the endpoint uses the first available device.

Chapter 5. Usage

5.1. Objects

While working with DOCA Comm Channel, one must maintain two objects:

- `struct doca_comm_channel_ep_t`, referred to as "endpoint"
- `struct doca_comm_channel_addr_t`, referred to as "peer_address"

5.1.1. Endpoint

The endpoint object represents the endpoint of the Comm Channel, either on the client or service side. The endpoint is created by calling the `doca_comm_channel_ep_create()` function. It is required for every other Comm Channel function.

5.1.2. Peer_address

The `peer_address` structure represents a connection. It is created when a new connection is made (i.e., client calls `doca_comm_channel_ep_connect()` or a service receives a connection through `doca_comm_channel_ep_recvfrom()`). Refer to section Connection Flow for more details.

The peer_address structure can be used to identify the source of a received message and is needed to send a message using `doca_comm_channel_ep_sendto()`.

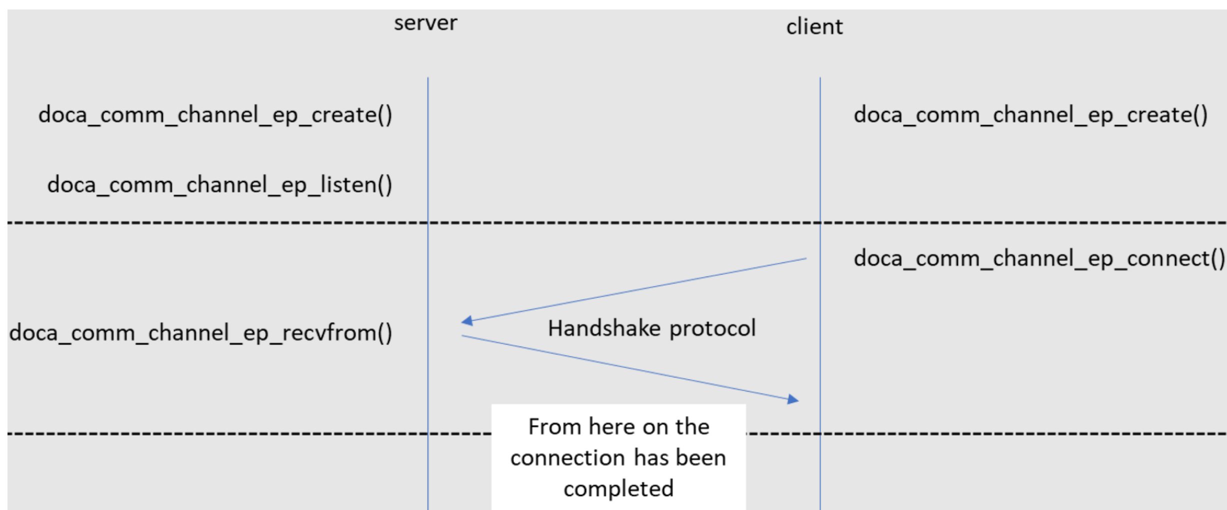
The `peer_address` has an identifier named "user data" that can be set by the user using `doca_comm_channel_peer_addr_user_data_set()`, and retrieved using `doca_comm_channel_peer_addr_user_data_get()`. The default value for user data is 0.

5.2. Endpoint Initialization

To start using the DOCA Comm Channel, the user should initialize a `doca_comm_channel_init_attr` struct that contains the endpoint's configuration attributes and use it to create an endpoint object, using the `doca_comm_channel_ep_create()` function.

5.3. Connection Flow

The following diagram illustrates the process of establishing a connection between the host and a service.



1. After initializing the endpoint on the service side, one should call `doca_comm_channel_ep_listen()` with a legal service name (see Limitations) to start listening.
2. After the service starts listening and the client endpoint is created, the client calls `doca_comm_channel_ep_connect()` with the same service name used for listening. As part of the connect function, the client starts a handshake protocol with the server. If connect is called before the service is listening, or the handshake process fails, then the connect function fails.
3. The service receiving new connections is done using `doca_comm_channel_ep_recvfrom()`. No indication is given that a new connection is made. The server keeps waiting to receive packets. If the handshake fails or is done for an existing client, then the receive function fails.

5.4. Data Transfer Flow

After a connection is established between client and service, both sides can send and receive data using the `doca_comm_channel_ep_sendto()` and `doca_comm_channel_ep_recvfrom()` functions, respectively.

If multiple clients are connected to the same service, then the `doca_comm_channel_ep_recvfrom()` function reads the messages in the order of their arrival, regardless of their source.

To send a message, the endpoint needs to obtain the target's peer_address object. This restriction leads to the client being the one that must start the communication (not including the handshake) for the server to obtain the client's peer_address object and send data back.

The user can choose between working in blocking or non-blocking mode. When running in non-blocking mode, the send/receive function returns immediately, either in success or failure. When running in blocking mode, if the conditions to send/receive are not met, the function waits internally for the requirements to be met.

5.4.1. Blocking Mode

If `doca_comm_channel_ep_ sendto()` or `doca_comm_channel_ep_ recvfrom()` are called without the `DOCA_CC_MSG_FLAG_DONTWAIT` flag and also the sending/receiving endpoint is not created with the `DOCA_CC_INIT_FLAG_NONBLOCK` flag, then those functions run in blocking mode. And if the other side is not ready to receive (for send), or a new message is not available to receive, then the function is blocked until the receiver is ready or a new message arrives.

5.4.2. Non-blocking Mode

If the `doca_comm_channel_ep_ sendto()` or `doca_comm_channel_ep_ recvfrom()` functions are called with the `DOCA_CC_MSG_FLAG_DONTWAIT` flag, or the endpoint is created with the `DOCA_CC_INIT_FLAG_NONBLOCK` flag, then:

- The send function runs in non-blocking mode and returns with `DOCA_ERROR_AGAIN` if the other side is not ready to receive
- The receive function runs in non-blocking mode and returns with `DOCA_ERROR_AGAIN` if no new message is available

5.5. Disconnection Flow

Disconnection can occur specifically by using `doca_comm_channel_ep_disconnect()` or when destroying the whole endpoint.

Disconnection is one-sided, which means that the other side is unaware of the channel being closed and experiences errors when sending data. It is up to the application to synchronize the connection teardown.

It is possible to perform another handshake and establish a new channel connection after disconnection.

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

© 2022 NVIDIA Corporation & affiliates. All rights reserved.

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

NVIDIA.