NVIDIA DOCA RDMA Programming Guide

Programming Guide

MLNX-15-060595 _v2.2.1 | October 2023

Table of Contents

Chapter 1. Introduction

Note: This library is currently supported at beta level only.

DOCA RDMA enables direct access to the memory of remote machines, without interrupting the processing of their CPUs or operating systems. Avoiding CPU interruptions reduces context switching for I/O operations, leading to lower latency and higher bandwidth compared to traditional network communication methods.

DOCA RDMA library provides an API to execute the various RDMA operations.

This document is intended for software developers wishing to improve their applications by utilizing RDMA operations.

Chapter 2. Prerequisites

DOCA RDMA-based applications can run either on the host machine or on the NVIDIA® BlueField® DPU target.

Chapter 3. Architecture

DOCA RDMA consists of two connected sides, passing data between one another. This includes the option for one side to access the remote side's memory if the granted permissions allow it.

The connection between the two sides can either be based on InfiniBand (IB) or based on Ethernet using RoCE. Currently, only reliable connection (RC) transport type is supported.

The different operations that may be executed between the two sides, using DOCA RDMA, are:

- Receive
- Send
- Send with Immediate
- Write
- Write with Immediate
- Read
- Atomic Compare & Swap
- Atomic Fetch & Add

DOCA RDMA relies heavily on the underlying DOCA core architecture for its operation, including the memory map, buffer objects, context and workq. RDMA operations are requested by submitting an RDMA job on the relevant workq. The DOCA RDMA library then executes that operation asynchronously before posting a completion event on the work queue.

Note: Currently, each RDMA context supports only a single workq.

Note: The DOCA RDMA library supports scatter-gather (SG) DOCA buffers in some jobs utilizing the linked list option. For job-specific information, refer to DOCA RDMA Job Structures.

Chapter 4. API

This chapter details of the specific structures and enums related to the DOCA RDMA library.

Refer to Usage to learn how to use DOCA RDMA API (including all RDMA functions) to run a program from start to finish.

4.1. DOCA RDMA Job Structures

The API for DOCA RDMA consists of 4 unique DOCA RDMA unique job structures that can be used the execute a total of 7 different DOCA RDMA jobs. This section overviews the different job structures, their expected inputs, and results.

Each DOCA RDMA job structure includes a `doca_job` structure as its base:

```
struct doca_job {
    int type;                   /**< Defines the type of the job. */
    int flags;                  /**< Job submission flags (see `enum
 doca_job_flags`). */
    struct doca_ctx *ctx;       /**< Doca CTX targeted by the job. */
    union doca_data user_data;  /**< Job identifier provided by user. Will be
 returned back on completion. */
};
```

For each job submitted using DOCA RDMA, the following applies:

- It is expected that the `flags` field value is `DOCA_JOB_FLAGS_NONE` (part of `enum doca_job_flags`), since there are currently no jobs that use flags in DOCA RDMA.
- The `ctx` field should point to a valid DOCA RDMA context. The context can be retrieved once the RDMA instance is created using `doca_rdma_as_ctx()`.
- The `user_data` field can hold whatever value the user desires and is returned untouched to the user on completion of the given job.

Note: Most DOCA RDMA operations are not atomic and therefore it is imperative that the application handle synchronization appropriately. Moreover, successful completion of a write job, with or without immediate, does not guarantee the data tp be written to the remote address.

Note: All buffers used in DOCA RDMA jobs must remain valid until the job result is retrieved.

4.1.1. DOCA RDMA Receive

This job should be submitted prior to an expected submission of a send/send with immediate/write with immediate job on the remote side.

```
struct doca_rdma_job_recv {
    struct doca_job base;       /**< Common job data */
    struct doca_buf *dst_buff;  /**< Destination data buffer,
                                 *  chain len must not exceed recv_buf_chain_len
 property
                                 */
};
```

To execute an DOCA RDMA receive job, the value of `base.type` field should be set to `DOCA_RDMA_JOB_RECV` (part of the `enum doca_rdma_job_types`).

The destination buffer (`dst_buff`) should point to a local memory address. Upon success, the received message is appended after the data section in the destination buffer, as it was prior to the job submission, and the data length is increased by the received data length.

The given destination buffer/chain of buffers (given in `dst_buff`) must have a total length sufficient for the expected message size or the job will fail.

The destination buffer is not mandatory and may be NULL when the requested DOCA RDMA job on the remote side is "write with immediate" or when the remote side is sending an empty message, with or without immediate (may be relevant when wanting to keep a connection alive).

Note: For the DOCA RDMA receive job, the length of each buffer is considered as the length from the end of the data section until the end of the buffer, as this is the available memory that can be written to in each buffer. The data length is increased in each buffer if data is written to it once the job is successfully completed. For more information, refer to the NVIDIA DOCA Core Programming Guide.

Note: The total length of the message must not exceed the device's `max_message_size` or 2GB (whichever is lower). The number of chained buffers must also not exceed the `recv_buf_chain_len` property of the RDMA instance. Refer to Usage to understand how to retrieve `max_message_size` and `recv_buf_chain_len`.

4.1.2. DOCA RDMA Send

This job should be submitted to transfer a message to the remote side, with or without immediate data, and while the remote side is expecting a message and had submitted a receive job beforehand.

```
struct doca_rdma_job_send {
    struct doca_job base;                           /**< Common job data */
    struct doca_buf const *src_buff;                /**< Source data buffer */
    doca_be32_t immediate_data;                     /**< Immediate data */
    struct doca_rdma_addr const *rdma_peer_addr;    /**< Optional: For RDMA context
 of type UD or DC */
};
```

To execute a DOCA RDMA send or send with immediate job, the value of the `base.type` field should be set to `DOCA_RDMA_JOB_SEND`/`DOCA_RDMA_JOB_SEND_IMM` respectively (part of `enum doca_rdma_job_types`).

The total length of the given source buffer/chain of buffers (in `src_buff`) may not exceed the expected message size on the remote side or the job will fail.

The source buffer is not mandatory and may be NULL when wishing to send an empty message, with or without immediate (may be relevant when wishing to keep a connection alive).

Note: For the purpose of the DOCA RDMA send job, the length of each buffer is considered as its data length.

Note: The total length of the message must not exceed the `max_message_size` device capability or 2GB (whichever is lower). Refer to Usage to understand how to retrieve `max_message_size`.

The `immediate_data` field is a 32-bit value sent to the remote side, out-of-band, and should be in Big-Endian format. This value is transferred only when the job type is `DOCA_RDMA_JOB_SEND_IMM`, and is received by the remote side only once a receive job is completed successfully.

Currently, the `rdma_peer_addr` field is not in use as DC and UD transport types are not yet supported.

4.1.3. DOCA RDMA Read/Write

These jobs should be submitted when wishing to access (read or write) data from remote memory (i.e., the memory on the remote side of the connection).

```
struct doca_rdma_job_read_write {
    struct doca_job base;                               /**< Common job data */
    struct doca_buf *dst_buff;                          /**< Destination data buffer */
    struct doca_buf const *src_buff;                    /**< Source data buffer */
    doca_be32_t immediate_data;                         /**< Immediate data for write
 with imm */
    struct doca_rdma_addr const *rdma_peer_addr;        /**< Optional: For RDMA context
 of type DC */
};
```

Note that for each read or write job submitted using DOCA RDMA, the following applies:

- The source buffer (`src_buff`) is not mandatory and may be NULL when wishing to read or write zero bytes (might be relevant when wishing to keep a connection alive). In such a case, the destination buffer may be NULL as well.
- Currently, the `rdma_peer_addr` field is not in use as DC transport type is not yet supported.

4.1.3.1. DOCA RDMA Read

To execute a DOCA RDMA read job, the value of the `base.type` field should be set to `DOCA_RDMA_JOB_READ` (part of the `enum doca_rdma_job_types`).

The destination buffer (`dst_buff`) should point to a local memory address. Upon success, the read data is appended after the data section in the destination buffer, as it was prior to the job submission, and the data length is increased by the read data length.

The source buffer should point to a remote memory address from which the data should be read. The data is read only from the data section of the source buffer.

Note: For the DOCA RDMA read job:

- The length of the source buffer is considered its data length. The length of data read from the source buffer depends on its data length yet can not exceed the total length of the given destination buffer/chain of buffers. That is, the actual length read depends on the minimal length between the source and destination.
- The length of each destination buffer is considered as the length from the end of the data section until the end of the buffer, as this is the available memory that can be written to in each buffer.

Note: The given source buffer length must not exceed the `max_message_size` device capability or 2GB (whichever is lower). Refer to Usage to understand how to retrieve `max_message_size`.

The `immediate_data` field is ignored.

4.1.3.2. DOCA RDMA Write

To execute a DOCA RDMA write or write with immediate job, the value of `base.type` field should be set to `DOCA_RDMA_JOB_WRITE`/`DOCA_RDMA_JOB_WRITE_IMM` respectively (part of the `enum doca_rdma_job_types`).

The destination buffer (`dst_buff`) should point to a remote memory address. Upon success, the written data is appended after the data section in the destination buffer, as it was prior to the job submission, and the data length is increased by the written data length.

The source buffer should point to a local memory address from which the data should be read. The data is read only from the data section of the source buffer.

Note: For the purpose of the DOCA RDMA write job:

- The length of each buffer is considered as its data length
- The length of the destination buffer is considered as the length from the end of the data section until the end of the buffer, as this is the available memory that can be written to
- The length of data written to the destination buffer depends on the total length of the given source buffer/chain of buffers

Note: The total length of the given source buffer/chain of buffers must be not exceed the `max_message_size` device capability or 2GB (whichever is lower). Refer to Usage to understand how to retrieve `max_message_size`.

The `immediate_data` field is a 32-bit value sent to the remote side, out-of-band, and should be in a Big-Endian format. This value is transferred only when the job type is `DOCA_RDMA_JOB_WRITE_IMM` and is received by the remote side only once a receive job is completed successfully.

Note: A write with immediate job succeeds only if the remote side is expecting the immediate and had submitted a receive job beforehand.

4.1.4. DOCA RDMA Atomic

These jobs should be submitted when wishing to execute an 8-byte atomic operation on the remote memory, the memory on the remote side.

```
struct doca_rdma_job_atomic {
    struct doca_job base;                                   /**< Common job data */
    struct doca_buf *cmp_or_add_dest_buff;                  /**< Destination data buffer */
    struct doca_buf *result_buff;                           /**< Result of the atomic
 operation:
                                                             *  remote original data before
 add, or remote original data
                                                             *  before compare
                                                             */
    uint64_t swap_or_add_data;                              /**< For add, the increment
 value
                                                             *  for cmp, the new value to
 swap
                                                             */
    uint64_t cmp_data;                                      /**< Value to compare for
 compare and swap */
    struct doca_rdma_addr const *rdma_peer_addr;            /**< Optional: For RDMA context
 of type DC */
};
```

For each atomic job submitted using DOCA RDMA, the following applies:

- The destination buffer (`cmp_or_add_dest_buff`) should point to a remote memory address and its data section must begin in a memory address aligned to 8 bytes. Only the first 8 bytes following the data address are considered for atomic operations.
- The result buffer (`result_buff`) should point to a local memory address and, upon success, the original value of the destination buffer (before executing the atomic operation) is written to it. The result is written to the first 8 bytes following the data address.
- Currently, the `rdma_peer_addr` field is not in use as DC transport type is not yet supported.

4.1.4.1. DOCA RDMA Atomic Compare and Swap

To execute a DOCA RDMA atomic compare and swap job, the value of `base.type` field should be set to `DOCA_RDMA_JOB_ATOMIC_CMP_SWP` (part of the `enum doca_rdma_job_types`).

The compare data field (`cmp_data`) is a 64-bit value that is compared to the value in the destination buffer (the first 64-bit following the beginning of the data section of the buffer).

- If the compared values are equal, the value in the destination is swapped with the 64-bit value in the jobs swap data field (`swap_or_add_data`)
- If the compared values are not equal, the value in the destination value remains unchanged

4.1.4.2. DOCA RDMA Atomic Fetch and Add

When wishing to execute a DOCA RDMA atomic fetch and add job, the value of `base.type` field should be set to `DOCA_RDMA_JOB_ATOMIC_FETCH_ADD` (part of the `enum doca_rdma_job_types`).

The value in the destination is increased by the 64-bit value in the job's add data field (`swap_or_add_data`).

The compare data field (`cmp_data`) is ignored.

4.2. DOCA RDMA Job Result Structure

Once a job is submitted and its progress is successfully retrieved, the `doca_rdma_result` struct is updated as part of the `doca_event` returned (see Waiting for Job Completion for more information).

```
struct doca_rdma_result {
    doca_error_t result;                    /**< Operation result */
    enum doca_rdma_opcode_t opcode;         /**< Opcode in case of
 doca_rdma_job_recv completion */
    struct doca_rdma_addr *rdma_peer_addr;  /**< Peer Address for UD and DC */
    doca_be32_t immediate_data;             /**< Immediate data, valid only if
 opcode indicates */
    /** 'dst_buff' data positioning will get updated on RECV and READ ops */
};
```

The `result` field holds a `doca_error_t` representing the result of the job.

The `rdma_peer_addr` field is currently not in use as DC and UD transport types are not yet supported.

The following fields are valid only when the `doca_rdma_result` returns a successful completion of a receive job:

- The `opcode` field represents which job has been submitted by the remote side that required there to be a receive job
- The `immediate_data` field is valid only when the `opcode` field value is `DOCA_RDMA_OPCODE_RECV_SEND_WITH_IMM` or `DOCA_RDMA_OPCODE_RECV_WRITE_WITH_IMM`. This holds the 32-bit immediate data sent from the remote side in Big-Endian format.

4.3. DOCA RDMA State Enum

These values describe the state of the RDMA instance at any point:

```
enum doca_rdma_state {
    DOCA_RDMA_STATE_RESET = 0,
    DOCA_RDMA_STATE_INIT,
```

```
    DOCA_RDMA_STATE_CONNECTED,
    DOCA_RDMA_STATE_ERROR,
};
```

DOCA_RDMA_STATE_RESET
The initial state of any RDMA instance. This state can be returned to, at any time, by calling `doca_ctx_stop()`.

DOCA_RDMA_STATE_INIT
The RDMA instance is initialized (`doca_ctx_start()` has been called) and is ready to be connected (i.e., `doca_rdma_export()` and `doca_rdma_connect()` may be called).

DOCA_RDMA_STATE_CONNECTED
The RDMA instance is connected to another RDMA instance (`doca_rdma_connect()` has been called) and communication between the peers is possible.

DOCA_RDMA_STATE_ERROR
The RDMA instance is in an error state. Trying to communicate between the peers would result in an error. Both sides should be reset (i.e. call `doca_ctx_stop()`).

4.4. DOCA RDMA Transport Type Enum

This enum includes the possible transport types in RDMA:

```
enum doca_rdma_transport_type {
    DOCA_RDMA_TRANSPORT_RC, /**< RC transport */
    DOCA_RDMA_TRANSPORT_DC, /**< DC transport, currently not supported */
};
```

Note: Currently, only RC transport is supported.

Chapter 5. Usage

The following subsections go through the various stages required to initialize, execute, and clean up RDMA operations.

Note that the DOCA RDMA library relies on the use of `doca_ctx` and `doca_workq` to execute RDMA jobs. The following explanations regarding the flow and use of DOCA RDMA, require users to be familiar with these objects (as well as other DOCA Core objects such as `doca_dev`, `doca_mmap`, `doca_buf_inventory`, `doca_buf`, etc). For more information, see NVIDIA DOCA Core Programming Guide.

5.1. Preparation

The following section describes the necessary steps before executing any RDMA operation.

The order in which the following subsections are presented is non-biding. The user may perform whichever initialization process suits their needs best.

5.1.1. Selecting and Opening a DOCA Device

To execute RDMA operations, a device must be chosen. To choose a device, users may iterate over all DOCA devices (via `doca_devinfo_list_create()`) and query each for its capabilities relevant to RDMA operations, using `doca_rdma_get_*(struct doca_devinfo *, ...)` functions, and check whether the device is suitable for the RDMA job type that would be performed, using `doca_rdma_job_get_supported()`.

5.1.2. Setting up and Initializing DOCA RDMA Context

To use DOCA RDMA:

1. Create an RDMA instance using `doca_rdma_create()` and acquire its context using `doca_rdma_as_ctx()`. The state of a newly created RDMA instance is `DOCA_RDMA_STATE_RESET`.
2. The chosen device must be added to the RDMA context, using `doca_ctx_dev_add()`.

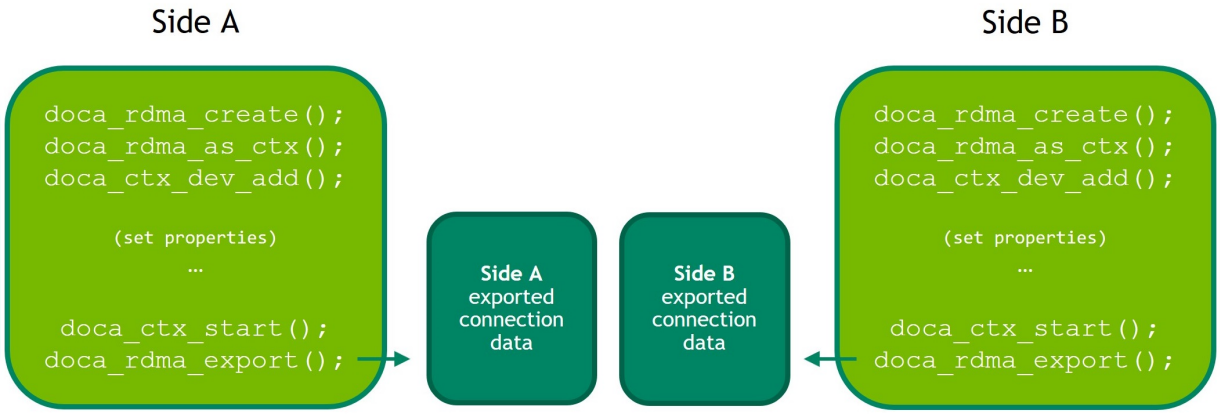
3. (Optional) Edit the default properties of the RDMA instance and query its properties using the `doca_rdma_set_<property>()` and `doca_rdma_get_<property>(struct doca_rdma *, …)` functions respectively.

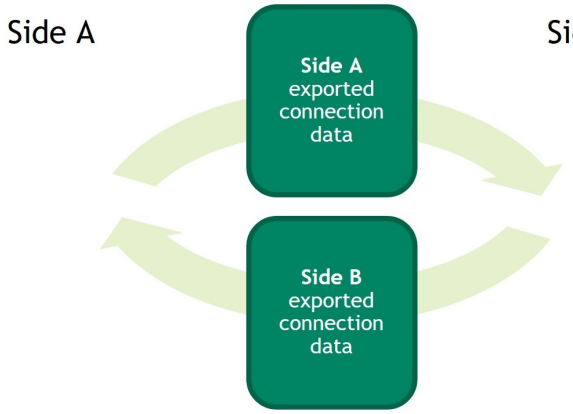
   Note: Some RDMA operations require certain permissions to be set. For more information, refer to Summary of Necessary Permissions for RDMA Operations.

4. Start the RDMA context by using `doca_ctx_start()`. Once started, the RDMA instance moves to state `DOCA_RDMA_STATE_INIT`.
5. Export each RDMA instance to the remote side to a blob by using `doca_rdma_export()`.
6. Transfer the blob to the opposite side out-of-band (OOB) and provide it as input to the `doca_rdma_connect()` function on that side. Connecting an RDMA instance moves its state to `DOCA_RDMA_STATE_CONNECTED` and it is ready to start running jobs.

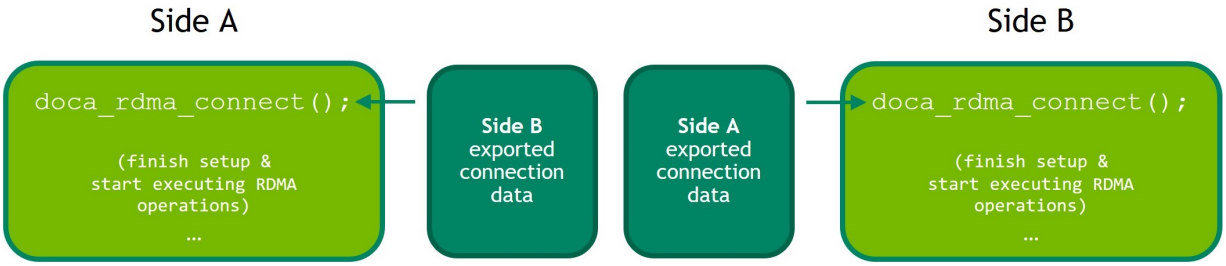
Step 1: Initiate the RDMA instance, and when ready, export it



Step 2: Transfer the exported connection data out-of-band



Step 3: Connect the two RDMA instances



5.1.3. Creating and Initializing DOCA Core Objects

DOCA RDMA requires several DOCA core objects to be created as specified in the following subsections.

5.1.3.1. WorkQ

Executing any RDMA operation requires creating a work queue using `doca_workq_create()`.

A workq can work in two different modes:

- The default polling mode where the program must check whether a job has finished its execution until receiving confirmation
- The event-driven mode where the program may receive a notification once the job is done

To set the workq to event-driven mode, use `doca_workq_set_event_driven_enable()`. Then call `doca_workq_get_event_handle()` to retrieve the workq event handle to be used by `epoll` (or other Linux wait-for-event interfaces) to wait on events.

Once the RDMA context is started, the workq may be added to it by calling `doca_ctx_workq_add()`.

5.1.3.2. Memory Map

Executing any job in which data is passed between the peers requires creating a memory map (MMAP) on each side using `doca_mmap_create()`.

1. Add the chosen device to the memory map using `doca_mmap_dev_add()`.
2. Set the relevant memory map properties. For example, setting the memory range of the MMAP is mandatory and can be done using `doca_mmap_set_memrang()`.
3. Set the MMAP's permissions according to the required RDMA operations using `doca_mmap_set_permissions()`:
   - To execute RDMA operations, the MMAP's permissions must include `DOCA_ACCESS_LOCAL_READ_WRITE` (from `enum doca_access_flags`)
   - To allow remote access to the memory region of the MMAP, one must set the relevant RDMA permission from the `enum doca_access_flags`, according to the required RDMA operations

   > For more information about the required permissions, refer to Summary of Necessary Permissions for RDMA Operations.
4. Start the MMAP so it is ready to use by calling `doca_mmap_start()`.

To allow remote memory access:

1. Export the MMAP using `doca_mmap_export_rdma()` and pass it to the remote side (the side requesting the remote RDMA operation).
2. The remote side must create an MMAP from the exported blob (referred to as remote MMAP from here on) using `doca_mmap_create_from_export()`.

Both steps may be done later (even after RDMA jobs such as send/receive have executed) but they are necessary for allowing one side (or both) to request remote operations.

5.1.3.3. Buffer Inventory

Executing any job in which data is passed between the peers requires the requester to create a buffer inventory using `doca_buf_inventory_create()` and start it using `doca_buf_inventory_start()`.

5.1.4. Summary of Necessary Permissions for RDMA Operations

Summary of the necessary permissions of RDMA and MMAP for each RDMA operation:

DOCA RDMA Job Type	Minimal Permissions				Should Export MMAP?[(a)]
	Requester Side		Responder Side		
	RDMA	MMAP	RDMA	MMAP	
Read	–	Local Read Write	RDMA Read	Local Read Write | RDMA Read	Yes
Write/ Write with Immediate	–	Local Read Write	RDMA Write	Local Read Write | RDMA Write	Yes
Atomic (Fetch and Add, Compare and Swap)	–	Local Read Write	RDMA Atomic	Local Read Write | RDMA Atomic	Yes
Send/ Send with Immediate	–	Local Read Write	–	Local Read Write	No
Receive	Depending on the received job	Local Read Write	Not relevant		

Note: [(a)] Responder side never requires exporting MMAP.

5.1.5. Constructing DOCA Buffers

Before setting up and submitting an RDMA operation, users must construct the relevant DOCA buffers for the desired job by calling `doca_buf_inventory_buf_by_addr()`, providing addresses that exist within the memory region registered with the given memory map (local or remote).

The data address and length of the DOCA buffers may need to be set using `doca_buf_set_data()` as this field may affect how many bytes are transferred and where data will be written to. For more information on the affect of the data section on each job, refer to DOCA RDMA Job Structures.

5.2. RDMA Job Cycle

Once the preparations are complete as described in Preparation, RDMA jobs can be executed on the RDMA instance.

The following subsections describe the process of submitting a job and retrieving its result.

This cycle can be repeated for each desired job and these subsections may be executed in bulk; first by constructing all the desired jobs, then submitting them, and finally retrieving their result, all under the limitation of the workq depth and the queue size.

5.2.1. Constructing and Executing DOCA RDMA Operation

To begin an RDMA operation, enqueue an RDMA job on the previously created work queue object:

1. Create the DOCA RDMA job struct that contains the relevant job details. For further information about the different jobs and how to fill the job struct, refer to DOCA RDMA Job Structures.
2. Call `doca_workq_submit()` to submit the RDMA operation.

5.2.2. Waiting for Job Completion

To retrieve an RDMA operation result using `doca_workq_progress_retrieve()`, the user must provide a `doca_event` structure. This structure should point to an allocated `doca_rdma_result` struct in its `result.ptr` field.

It is the user's responsibility to allocate and manage the `doca_event` structure as well as the `doca_rdma_result`.

Code example for result preparation and retrieval:

```
struct doca_event event = {0};
struct doca_rdma_result rdma_result;
memset(&rdma_result, 0, sizeof(rdma_result));

event.result.ptr = (void *)(&rdma_result);
doca_workq_progress_retrieve(workq, &event, DOCA_WORKQ_RETRIEVE_FLAGS_NONE);
```

According to the workq mode, users may detect when the RDMA operation has been completed (via `doca_workq_progress_retrieve()`):

- Workq operating in polling mode – periodically poll the workq until the API call indicates that a valid event has been received (i.e., `DOCA_SUCCESS` returned).
- Workq operating in event mode – while `doca_workq_progress_retrieve()` does not return success as a result, perform the following loop:
  1. Arm the workq `doca_workq_event_handle_arm()`.
  2. Wait for an event using the event handle (e.g., using `epoll_wait()`).

3. Once the thread wakes up, call `doca_workq_event_handle_clear()`.

Regardless of the operating mode, once the event is successfully retrieved, the result of the operation can be read in the provided `rdma_result` structure. For further information about the fields of the result structure, refer to DOCA RDMA Job Result Structure.

For some of the operations, the buffer's data length field may be updated according to the written data. For further information, refer to DOCA RDMA Job Structures.

5.2.3. Error Handling

If any RDMA job fails to run, and its result is returned with an error status, the RDMA instance itself moves to an error state (`DOCA_RDMA_STATE_ERROR`). Once in error state, no more jobs can be submitted until an error recovery flow is performed.

To recover an RDMA instance from the error state, the context must be stopped using `doca_ctx_stop()`, restarted using `doca_ctx_start()`, and connected again as explained under Setting up and Initializing DOCA RDMA Context, including exporting the connection information and passing it between the peers.

Note: To stop the context, the workq must be removed beforehand using `doca_ctx_workq_rm()` and added after restarting the context using `doca_ctx_workq_add()`.

5.3. Clean-up

This section describes the necessary steps to release all the resources allocated for executing RDMA operations.

The order in which the following subsections are presented is non-biding. The user may perform whichever clean up process suits their needs best.

5.3.1. Buffer and Buffer Inventory

1. Destroy all the buffers created during the run using `doca_buf_refcount_rm()` regardless of whether the operation is successful or not.
2. Only after all the buffers from the inventory are destroyed, destroy the buffer inventory using `doca_buf_inventory_destroy()`.

5.3.2. Memory Map

Both the memory map set with a local memory range and the memory map set with a remote memory range (remote MMAP), if created, must be destroyed using `doca_mmap_destroy()`.

5.3.3. WorkQ

1. Remove the workq from the RDMA context using `doca_ctx_workq_rm()`. If the workq has been set to event-driven mode, do not forget to clean up any resources created (apart from DOCA resources) to support wait-for-event.
2. Destroy the workq with `doca_workq_destroy()`.

5.3.4. DOCA RDMA Context

1. Stop the context using `doca_ctx_stop()` .
2. Destroy the context using `doca_rdma_destroy()`.

Chapter 6. DOCA RDMA Samples

This chapter describes RDMA samples based on the DOCA RDMA library. These samples illustrate how to use the DOCA RDMA API to execute RDMA operations.

6.1. Running the Samples

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA applications.
2. To build a given sample:
   ```
   cd /opt/mellanox/doca/samples/doca_rdma/<sample_name>
   meson build
   ninja -C build
   ```
3. RDMA sample arguments:
   - Common arguments:

Argument	Description
`-d, --device`	IB device name (optional). If not provided, then a random IB device is assigned.
`-ld, --local-descriptor-path`	Local descriptor file path that includes the local connection information to be copied to the remote program.
`-re, --remote-descriptor-path`	Remote descriptor file path that includes the remote connection information to be copied from the remote program.
`-m, --mmap-descriptor-path`	Remote descriptor file path that includes the remote mmap connection information to be copied from the remote program.
`-g, --gid-index`	GID index for DOCA RDMA (optional).

   - Sample-specific arguments:

Sample	Argument	Description
RDMA Read Responder	`-r, --read-string`	String to read (optional). If not provided, then "Hi DOCA RDMA!" is defined.
RDMA Send RDMA Send Immediate	`-s, --send-string`	String to send (optional). If not provided, then "Hi DOCA RDMA!" is defined.
RDMA Write Requester RDMA Write Immediate Requester	`-w, --write-string`	String to write (optional). If not provided, then "Hi DOCA RDMA!" is defined.

4. For additional information per sample, use the `-h` option:
```
./build/doca_<sample_name> -h
```

6.2. Samples

Each sample presents a connection between two peers, transferring data from one to another, using a different RDMA operation in each sample. For more information on the available RDMA operations, refer to DOCA RDMA Job Structures.

Each sample is comprised of two executables, each running on a peer.

The samples can run on either DPU or host, as long as the chosen peers have a connection between them.

Note: Prior to running the samples, ensure that the chosen devices, selected by the device name and the GID index, are set correctly and have a connection between one another. In each sample, it is the user's responsibility to copy the descriptors between the peers.

6.2.1. RDMA Read

6.2.1.1. RDMA Read Requester

This sample illustrates how to read from a remote peer (the responder) using DOCA RDMA.

The sample logic includes:

1. Locating DOCA device.
2. Initializing necessary DOCA core structures.
3. Adding a device to a DOCA memory map set with a memory range.
4. Initializing a DOCA RDMA object and setting up permissions.
5. Converting a DOCA RDMA object to DOCA context and adding a device.
6. Adding RDMA context to work queue.
7. Connecting RDMA. The user is responsible for copying the descriptors between the two sides.

8. Constructing DOCA buffers.
9. Setting and submitting an RDMA read job to the work queue.
10. Waiting and retrieving RDMA job from the queue once it is done.
11. Checking transferred data by printing the data read from the responder.
12. Destroying all RDMA and DOCA core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_read_requester/rdma_read_requester_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_read_requester/rdma_read_requester_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_read_requester/meson.build`

6.2.1.2. RDMA Read Responder

This sample illustrates how to set up a remote peer for a DOCA RDMA read request.

The sample logic includes:

1. Locating DOCA device.
2. Initializing necessary DOCA core structures.
3. Adding a device to a DOCA memory map set with a memory range.
4. Initializing a DOCA RDMA object and setting up permissions.
5. Converting a DOCA RDMA object to DOCA context and adding a device.
6. Adding RDMA context to work queue.
7. Exporting memory map of RDMA.
8. Connecting RDMA. The user is responsible for copying the descriptors between the two sides.
9. Destroying all RDMA and DOCA core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_read_responder/rdma_read_responder_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_read_responder/rdma_read_responder_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_read_responder/meson.build`

6.2.2. RDMA Write

6.2.2.1. RDMA Write Requester

This sample illustrates how to write to a remote peer (the responder) using DOCA RDMA.

The sample logic includes:

1. Locating DOCA device.

2. Initializing necessary DOCA core structures.
3. Adding a device to a DOCA memory map set with a memory range.
4. Initializing a DOCA RDMA object and setting up permissions.
5. Converting a DOCA RDMA object to DOCA context and adding a device.
6. Adding RDMA context to work queue.
7. Connecting RDMA. The user is responsible for copying the descriptors between the two sides.
8. Constructing DOCA buffers.
9. Setting and submitting an RDMA write job to the work queue.
10. Waiting and retrieving RDMA job from the queue once it is done.
11. Destroying all RDMA and DOCA core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_requester/rdma_write_requester_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_requester/rdma_write_requester_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_requester/meson.build`

6.2.2.2. RDMA Write Responder

This sample illustrates how to set up a remote peer for a DOCA RDMA read request.

The sample logic includes:

1. Locating DOCA device.
2. Initializing necessary DOCA core structures.
3. Adding a device to a DOCA memory map set with a memory range.
4. Initializing a DOCA RDMA object and setting up permissions.
5. Converting a DOCA RDMA object to DOCA context and adding a device.
6. Adding RDMA context to work queue.
7. Exporting memory map of RDMA.
8. Connecting RDMA. The user is responsible for copying the descriptors between the two sides.
9. Checking transferred data by printing the data sent by the requester on the DOCA mmap.
10. Destroying all RDMA and DOCA core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_responder/rdma_write_responder_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_responder/rdma_write_responder_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_responder/meson.build`

6.2.3. RDMA Write Immediate

6.2.3.1. RDMA Write Immediate Requester

This sample illustrates how to write to a remote peer (the responder) using DOCA RDMA along with a 32-bit immediate value which is sent OOB.

The sample logic includes:

1. Locating DOCA device.
2. Initializing necessary DOCA core structures.
3. Adding a device to a DOCA memory map set with a memory range.
4. Initializing a DOCA RDMA object and setting up permissions.
5. Converting a DOCA RDMA object to DOCA context and adding a device.
6. Adding RDMA context to work queue.
7. Connecting RDMA. The user is responsible for copying the descriptors between the two sides.
8. Constructing DOCA buffers.
9. Setting and submitting an RDMA write with immediate job to the work queue.
10. Waiting and retrieving RDMA job from the queue once it is done.
11. Destroying all RDMA and DOCA core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_requester/rdma_write_immediate_requester_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_requester/rdma_write_immediate_requester_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_requester/meson.build`

6.2.3.2. RDMA Write Immediate Responder

This sample illustrates how the set up a remote peer for a DOCA RDMA write request whilst receiving a 32-bit immediate value from the peer's OOB.

Note: The responder must submit a receive job.

The sample logic includes:

1. Locating DOCA device.
2. Initializing necessary DOCA core structures.
3. Adding a device to a DOCA memory map set with a memory range.
4. Initializing a DOCA RDMA object and setting up permissions.
5. Converting a DOCA RDMA object to DOCA context and adding a device.

6. Adding RDMA context to work queue.
7. Exporting memory map of RDMA.
8. Connecting RDMA. The user is responsible for copying the descriptors between the two sides.
9. Setting and submitting an RDMA write with immediate job to the work queue.
10. Waiting and retrieving RDMA job from the queue once it is done.
11. Checking transferred data by printing the data sent by the requester on the DOCA mmap.
12. Destroying all RDMA and DOCA core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_responder/rdma_write_responder_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_responder/rdma_write_responder_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_responder/meson.build`

6.2.4. RDMA Send and Receive

6.2.4.1. RDMA Send

This sample illustrates how to send a message to a remote peer using DOCA RDMA.

The sample logic includes:

1. Locating DOCA device.
2. Initializing necessary DOCA core structures.
3. Adding a device to a DOCA memory map set with a memory range.
4. Initializing a DOCA RDMA object and setting up permissions.
5. Converting a DOCA RDMA object to DOCA context and adding a device.
6. Adding RDMA context to work queue.
7. Connecting RDMA. The user is responsible for copying the descriptors between the two sides.
8. Constructing DOCA buffers.
9. Setting and submitting an RDMA send job to the work queue.
10. Waiting and retrieving RDMA job from the queue once it is done.
11. Destroying all RDMA and DOCA core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_send/rdma_send_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_send/rdma_send_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_send/meson.build`

6.2.4.2. RDMA Receive

This sample illustrates how the remote peer can receive a message sent by the peer (the sender).

The sample logic includes:

1. Locating DOCA device.
2. Initializing necessary DOCA core structures.
3. Adding a device to a DOCA memory map set with a memory range.
4. Initializing a DOCA RDMA object and setting up permissions.
5. Converting a DOCA RDMA object to DOCA context and adding a device.
6. Adding RDMA context to work queue.
7. Connecting RDMA. The user is responsible for copying the descriptors between the two sides.
8. Constructing DOCA buffers.
9. Setting and submitting an RDMA receive job to the work queue.
10. Waiting and retrieving RDMA job from the queue once it is done.
11. Checking transferred data.
12. Destroying all RDMA and DOCA core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_receive/rdma_receive_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_receive/rdma_receive_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_receive/meson.build`

6.2.5. RDMA Send and Receive with Immediate

6.2.5.1. RDMA Send with Immediate

This sample illustrates how to send a message to a remote peer using DOCA RDMA along with a 32-bit immediate value which is sent OOB.

The sample logic includes:

1. Locating DOCA device.
2. Initializing necessary DOCA core structures.
3. Adding a device to a DOCA memory map set with a memory range.
4. Initializing a DOCA RDMA object and setting up permissions.
5. Converting a DOCA RDMA object to DOCA context and adding a device.
6. Adding RDMA context to work queue.
7. Connecting RDMA. The user is responsible for copying the descriptors between the two sides.
8. Constructing DOCA buffers.

9. Setting and submitting an RDMA send with immediate job to the work queue.
10. Waiting and retrieving RDMA job from the queue once it is done.
11. Destroying all RDMA and DOCA core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_send_immediate/rdma_send_immediate_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_send_immediate/rdma_send_immediate_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_send_immediate/meson.build`

6.2.5.2. RDMA Receive with Immediate

This sample illustrates how the remote peer can receive a message sent by the peer (the sender) while also receiving a 32-bit immediate value from the peer's OOB.

The sample logic includes:

1. Locating DOCA device.
2. Initializing necessary DOCA core structures.
3. Adding a device to a DOCA memory map set with a memory range.
4. Initializing a DOCA RDMA object and setting up permissions.
5. Converting a DOCA RDMA object to DOCA context and adding a device.
6. Adding RDMA context to work queue.
7. Connecting RDMA. The user is responsible for copying the descriptors between the two sides.
8. Constructing DOCA buffers.
9. Setting and submitting an RDMA receive with immediate job to the work queue.
10. Waiting and retrieving RDMA job from the queue once it is done.
11. Checking transferred data.
12. Destroying all RDMA and DOCA core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_receive_immediate/rdma_receive_immediate_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_receive_immediate/rdma_receive_immediate_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_receive_immediate/meson.build`

Notice

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

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

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

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

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

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

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

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

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF 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