DOCA UROM RDMO Application Guide

Table of contents

Introduction

System Design

- Bootstrap Procedure
- Memory Management
- RDMO UROM Worker Operation

Application Architecture

- UROM RDMO Worker Component
- Init
- RQ Create
- RQ Destroy
- MR Register
- MR Deregister
- Command Format
- Append
- Flush
- Scatter

DOCA Libraries

Compiling the Application

- Compiling All Applications
- Compiling Only the Current Application
- Troubleshooting

Running the Application

Host Application Execution

RDMO DPU Plugin Component

Command Line Flags

Troubleshooting

Application Code Flow

References

This guide provides a DOCA Remote Direct Memory Operation implementation on top of NVIDIA® BlueField® DPU using Unified Communication X (UCX) .

Introduction

A remote direct memory operation (RDMO) is conceptionally an active message which is executed outside the context of the target process.

An RDMO involves the following entities:

- Target – establishes a connection to the server to use as the control path. The target interacts with the server to define target endpoints and memory regions. The target exchanges endpoint and memory region information with an initiator to facilitate its connection.

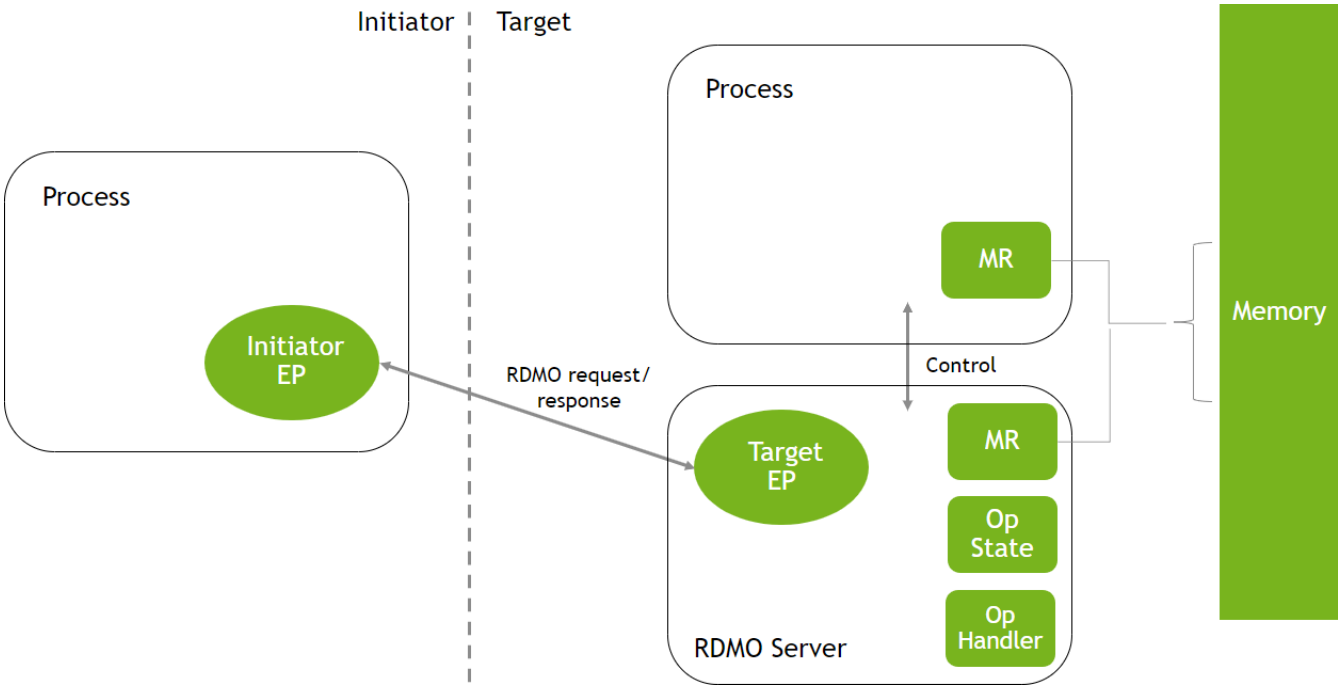
- Initiator – establishes a connection to the server to use as the data path. An RDMO is initiated by sending an RDMO command with an optional payload to the server. The server parses the commands and runs an associated RDMO handler. An RDMO handler interacts with the target process by performing one-sided memory accesses to target-defined memory regions.

- Server – responsible for executing RDMOs asynchronously from the target process. The server implements an RDMO handler for each supported operation. RDMO handlers may maintain a state within the server for optimization.

The DOCA UROM RDMO application includes the above three entities, split into the following parts:

- BlueField side – the implementation of RDMO plugin component to be loaded by the DOCA UROM worker (which is the RDMO server)

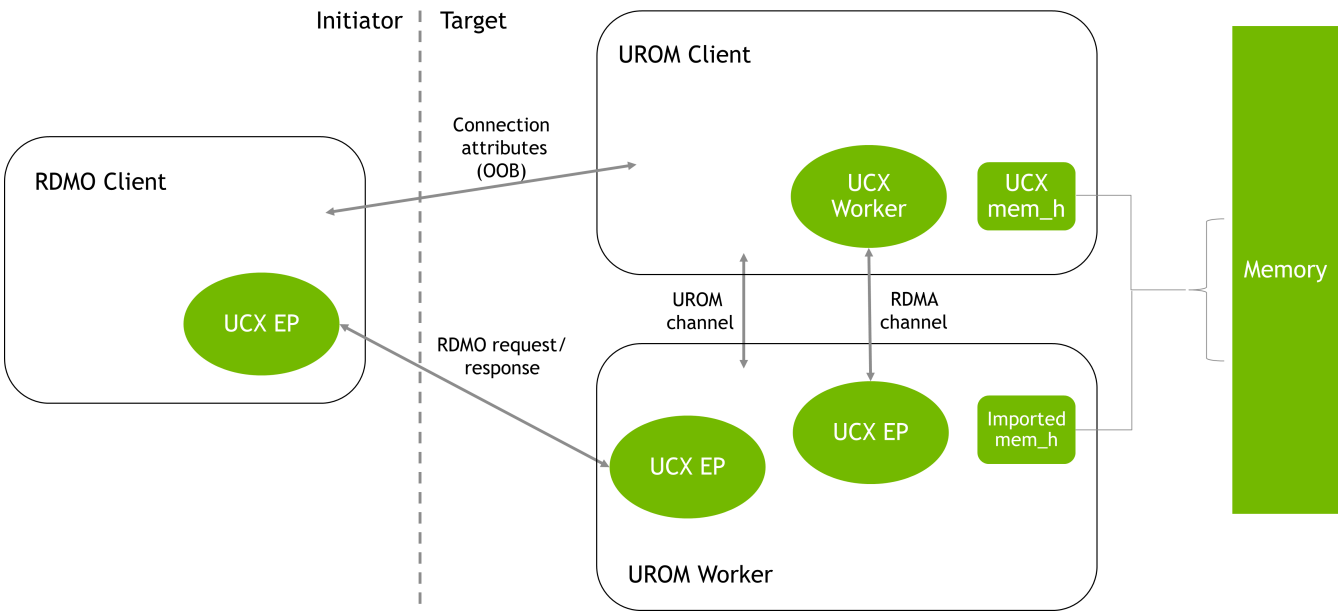
- Host side – host application that runs using two modes: target and initiator



RDMOs are designed to take advantage of extra computing resources on a platform. While application processes run on the primary compute resources, an RDMO server can run on idle resources on the same host or be offloaded to run on a separate device (i.e., BlueField).

System Design

The application demonstrates the implementation of RDMO operations as a DOCA UROM worker plugin component. A target process would use the DOCA UROM API to create a worker with RDMO capabilities. An initiator process establishes an RDMO connection to the UROM worker. The plugin uses UCX as its transport.



Bootstrap Procedure

To connect the RDMO initiator and target, on the target side, UROM is used to retrieve an address for each created RDMO worker. This address would need to be delivered to the RDMO initiator side for connection establishment. The initiator address is obtained from the UCX worker created explicitly by the RDMO application. Both addresses are exchanged over the out-of-band (OOB) network and used to establish the connection:

- On the RDMO initiator side, a UCX endpoint is created using UCX API
- On the RDMO target side, the initiator's address is communicated to the RDMO worker using the UROM command channel

Memory Management

UROM returns an identifier (ID) for each memory region imported to the RDMO plugin component. This ID is used to refer to a target memory region in RDMO requests. It must be exchanged with the initiator process OOB.

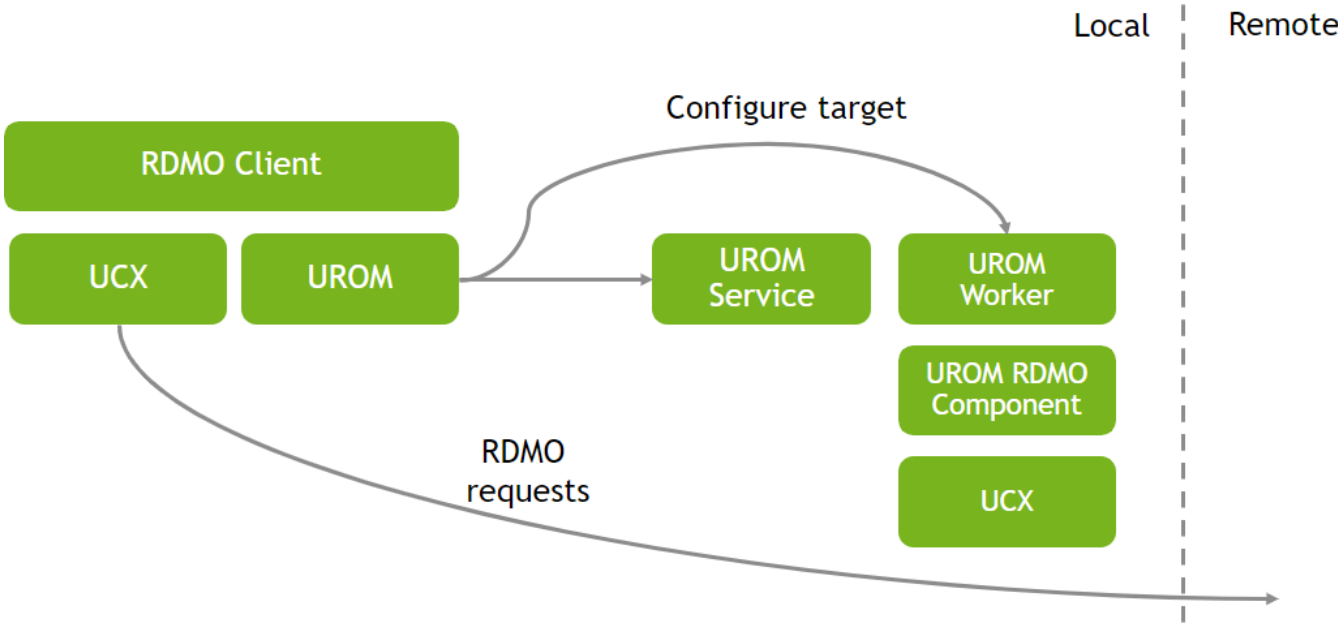
RDMO UROM Worker Operation

Communication between the RDMO initiator and worker is implemented on top of UCX active messages. The worker's active message handler is the entry point that identifies

the type of the RDMO operation based on the RDMO request header. The request is then forwarded to the corresponding RDMO operation handler which determines the operation parameters by inspecting the operation-specific sub-header in the request.

UCX active messages support eager and rendezvous protocols. When using a rendezvous protocol, the worker can choose whether to pull data to the server or move it directly to a target memory using a UCX-imported memory handle.

An RDMO operation handler may perform any combination of computation, initiator and target memory accesses, server state updates, or responses.



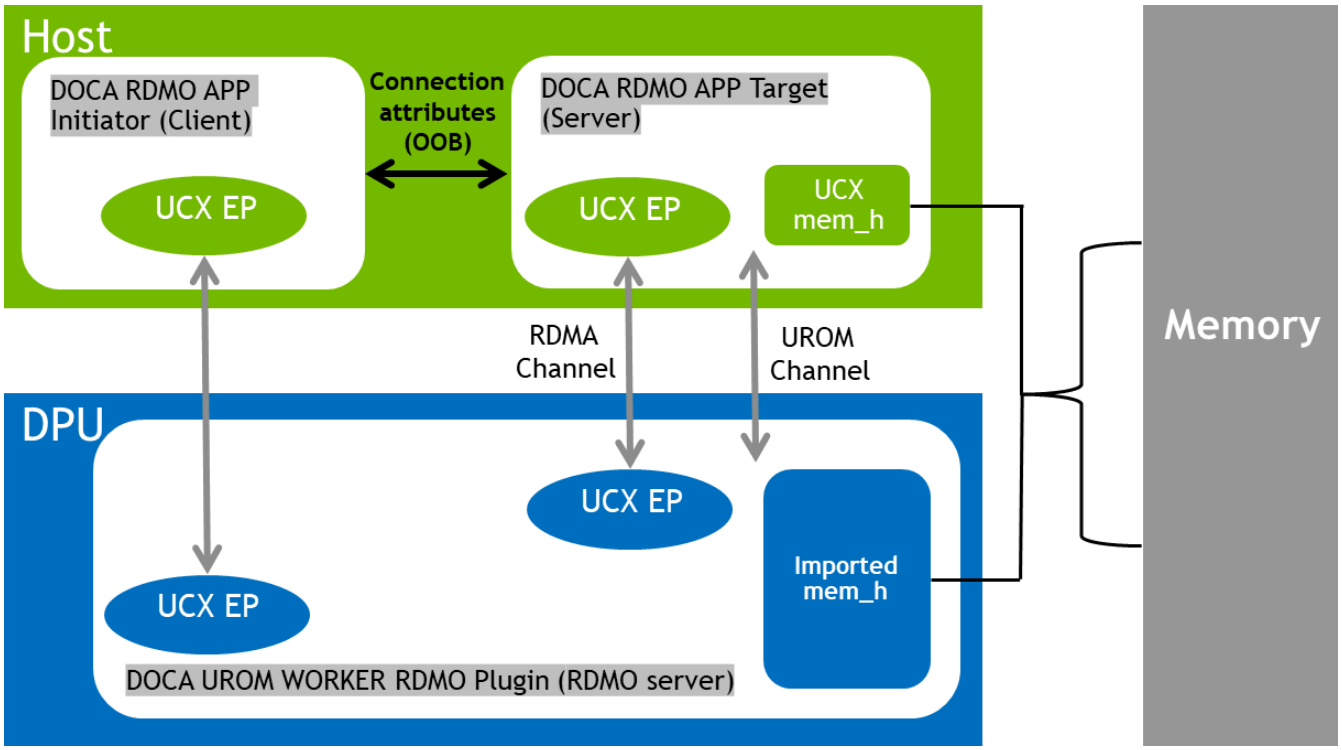
The RDMO client uses UROM to instantiate an RDMO worker and to configure target endpoints and memory regions. The client uses UCX directly to connect endpoints to the RDMO server. The client uses UCX to send formatted RDMO messages.

Application Architecture

DOCA's UROM RDMO application implementation uses UCX to support data exchange between endpoints. It utilizes UCX's sockaddr-based connection establishment and the UCX active messages (AM) API for communications, and UCX is responsible for all RDMO communications (control and data path) .

The RDMO server application initiates a DOCA UROM worker RDMO component via the DOCA UROM service and shares the UROM worker UCX EP with the DOCA UROM RDMO client application. The RDMO server application imports memory regions into the UROM worker to facilitate RDMA operations from the BlueField on host memory.

The RDMO client application performs RDMO operations via the DOCA UROM worker. Upon receiving the UCX EP address from the server, the client application initially establishes a connection with the worker. It then proceeds to request the worker to execute the operation without the server application's awareness.



UROM RDMO Worker Component

The UROM RDMO worker plugin component defines a small set of commands to enable the target to:

- Establish a UCX communication channel between the client and the worker
- Create a UCX endpoint capable of receiving RDMO request
- Import memory regions that can be used as a source or target for RDMA initiated by the worker

The set of commands are:

```
enum urom_worker_rdmo_cmd_type {
        UROM_WORKER_CMD_RDMO_CLIENT_INIT,
        UROM_WORKER_CMD_RDMO_RQ_CREATE,
```

```
        UROM_WORKER_CMD_RDMO_RQ_DESTROY,
        UROM_WORKER_CMD_RDMO_MR_REG,
        UROM_WORKER_CMD_RDMO_MR_DEREG,
};
```

The a ssociated notification types are:

```
enum urom_worker_rdmo_notify_type {
        UROM_WORKER_NOTIFY_RDMO_CLIENT_INIT,
        UROM_WORKER_NOTIFY_RDMO_RQ_CREATE,
        UROM_WORKER_NOTIFY_RDMO_RQ_DESTROY,
        UROM_WORKER_NOTIFY_RDMO_MR_REG,
        UROM_WORKER_NOTIFY_RDMO_MR_DEREG,
};
```

Init

The C lient Init command initializes the client to receive RDMOs. This includes establishing a connection between worker and host to allow the RDMO worker to access client memory.

The command is of type `UROM_WORKER_CMD_RDMO_CLIENT_INIT` . Command format:

```
struct urom_worker_rdmo_cmd_client_init {
        uint64_t id;
        void *addr;
        uint64_t addr_len;
};
```

- `id` – client ID used to identify the target process in RDMO commands
- `addr` – pointer to the client's UCP worker address to use for a worker-to-host connection

- `addr_len` – length of the address

This command returns a notification of type `UROM_WORKER_NOTIFY_RDMO_CLIENT_INIT`. Notification format:

```
struct urom_worker_rdmo_notify_client_init {
        void *addr;
        uint64_t addr_len;
```

- `addr` – pointer to the component's UCP worker address to use for initiator-to-server connections
- `addr_len` – length of the address

RQ Create

This Receive Queue (RQ) Create command creates and connects a new endpoint on the server. The endpoint may be targeted by formatted RDMO messages.

This command is of type `UROM_WORKER_CMD_RDMO_RQ_CREATE`. C ommand format:

```
struct urom_worker_rdmo_cmd_rq_create {
        void *addr;
    uint64_t addr_len;
};
```

- `addr` – the UCP worker address to use to connect the new endpoint
- `addr_len` – the length of address

The command returns a notification of type `UROM_WORKER_NOTIFY_RDMO_RQ_CREATE`. N otification format:

```
struct urom_worker_rdmo_notify_rq_create {
        uint64_t rq_id;
};
```

- `rq_id` – the RQ ID to use to destroy the RQ

RQ Destroy

The RQ Destroy command destroys an RQ.

The RQ Destroy command is of type `UROM_WORKER_CMD_RDMO_RQ_DESTROY` . C ommand format:

```
struct urom_worker_rdmo_cmd_rq_destroy {
        uint64_t rq_id;
};
```

- `rq_id` – the ID of a previously created RQ

The RQ destroy command returns a notification of type `UROM_WORKER_NOTIFY_RDMO_RQ_DESTROY` . N otification format:

```
struct urom_worker_rdmo_notify_rq_destroy {
        uint64_t rq_id;
};
```

- `rq_id` – the destroyed receive queue id

MR Register

The Memory Region (MR) Register command registers a UCP memory handle with the RDMO component. An MR must be registered with the RDMO component before use in RDMOs.

The command is of type `UROM_WORKER_CMD_RDMO_MR_REG`. Command format:

```
struct urom_worker_rdmo_cmd_mr_reg {
        uint64_t va;
        uint64_t len;
        void *packed_rkey;
        uint64_t packed_rkey_len;
        void *packed_memh;
        uint64_t packed_memh_len;
};
```

- `va` – the virtual address of the MR
- `len` – the length of the MR
- `packed_rkey` – pointer to the UCP packed R-key for the MR
- `packed_rkey_len` – the length of `packed_rkey`
- `packed_mem_h` – pointer to the UCP-packed memory handle for the MR. The memory handle must be packed with flag `UCP_MEMH_PACK_FLAG_EXPORT`.
- `packed_memh_len` – the length of `packed_memh`

The command returns a notification of type `UROM_WORKER_NOTIFY_RDMO_MR_REG` . Notification format:

```
struct urom_worker_rdmo_notify_mr_reg {
        uint64_t rkey;
};
```

- `rkey` – t he ID used in RDMOs to refer to the MR

MR Deregister

The MR deregister command deregisters an MR from the RDMO component.

The command is of type `UROM_WORKER_CMD_RDMO_MR_DEREG` . Command format:

```
struct urom_worker_rdmo_cmd_mr_dereg {
        uint64_t rkey;
};
```

- `rkey` – the ID of a previously registered MR

The command returns a notification of type `UROM_WORKER_NOTIFY_RDMO_MR_DEREG` . Notification format:

```
struct urom_worker_rdmo_notify_mr_dereg {
        uint64_t rkey;
};
```

- `rkey` – the deregistered memory region remote key

Command Format

An RDMO is initiated by sending an RDMO request via UCP active message to a UROM RDMO worker server.

The RDMO request format is:

The RDMO header identifies the operation type and flags, modifying how the RDMO is processed. The operation (op) header includes arguments specific to the operation type. Optionally, the operation type may include an arbitrary-sized payload.

RDMO header format:

```
struct urom_rdmo_hdr {
        uint32_t id;
        uint32_t op_id;
        uint32_t flags;
};
```

- `id` – the client ID
- `op_id` – the RDMO operation type ID
- `flags` – flags modifying how the RDMO is processed by the server

Valid flag values:

```
enum urom_rdmo_req_flags {
    UROM_RDMO_REQ_FLAG_FENCE,
};
```

- `UROM_RDMO_REQ_FLAG_FENCE` – Complete all outstanding RDMO requests on the connection before executing this request. This flag is required to implement a flush operation that guarantees remote completion.

Optionally, an operation may return a response to the initiator.

Response header format:

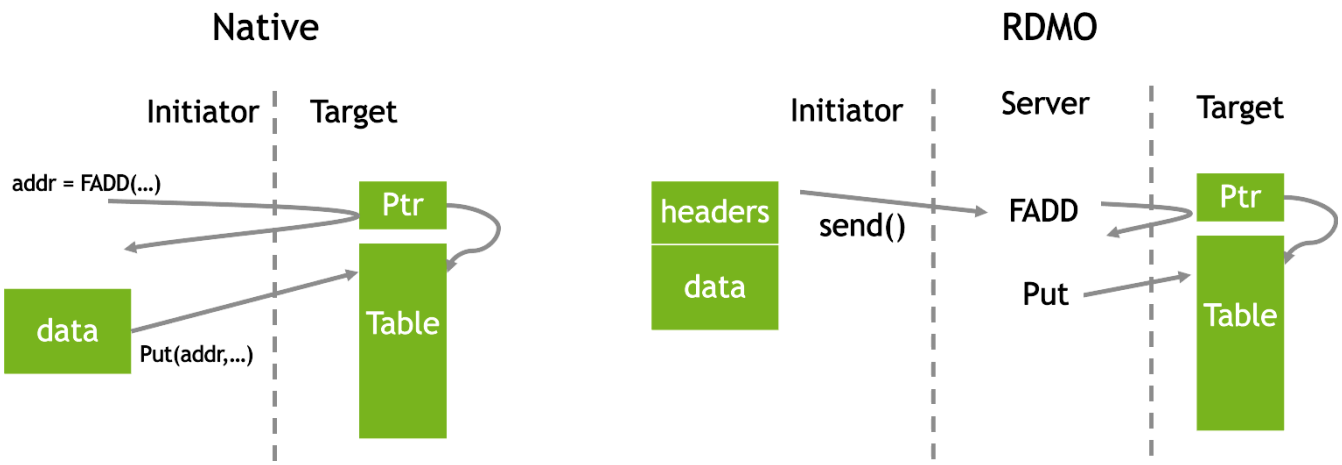
```
struct urom_rdmo_rsp_hdr {
    uint16_t op_id;
};
```

- `op_id` – the RDMO response type ID

Append

RDMO Append atomically appends data to a queue in remote memory. This can be achieved in a one-sided programming model with a Fetching-Add operation to the location of a pointer in remote memory, followed by a Put to the fetched address. RDMO Append allows these dependent operations to be offloaded to the target.

The following diagram provides a comparison of native and RDMO approaches to the Append operation:



Combining two dependent operations into a single RDMO allows the non-blocking implementation of Append, as the initiator does not need to wait between the Fetching Atomic and the data write operations. Using RDMO, the initiator can create a pipeline of operations and achieve a higher message rate.

The rate at which the RDMO server can perform operations on the target memory is expected to be a bottleneck. To improve the rate, the following optimizations can be looked at:

- The result of the Fetch-and-ADD (FADD) after the initial Append is performed can be cached in the server. Subsequent Appends can re-use the cached value, eliminating the atomic FADD operation. The modified pointer value is required to be synchronized during the flush command.

- For small Append sizes, the Append data can be cached in the RDMO server and coalesced into a single Put. As a result, the server requires, on average, a single Put access to target memory to execute several RDMOs.

- To avoid extra memory usage and lost bandwidth for large Append operations, the RDMO server may initiate direct transfers from the initiator to the target memory bypassing the acceleration device memory.

The Append operation uses an operation of type `UROM_RDMO_OP_APPEND`. Append header format:

```
struct urom_rdmo_append_hdr {
    uint64_t ptr_addr;
    uint16_t ptr_rkey;
    uint16_t data_rkey;
};
```

- `ptr_addr` – the address of the queue pointer in target memory
- `ptr_rkey` – the R-key used to access `ptr_addr`
- `data_rkey` – the R-key used to access the queue data

The RDMO payload is the local data buffer.

Flush

RDMO Flush is used to implement synchronization between the initiator and server. On execution, Flush sends a response message back to the initiator. Flush can be used to guarantee remote completion of a previously issued RDMO.

To achieve this, the initiator sends an in-order Flush command including the RDMO flag `UROM_RDMO_REQ_FLAG_FENCE`. This flag causes the server to complete all previously received RDMOs before executing the Flush. To complete previous operations, the server must write any cached data and make it visible in the target memory. Once complete, the server executes the Flush. Flush sends a response to the initiator. When the initiator receives the flush message, the result of all previously sent RDMOs is guaranteed to be visible in the target memory.

The Flush operation uses operation type `UROM_RDMO_OP_FLUSH`. Flush header format:

```
struct urom_rdmo_flush_hdr {
    uint64_t flush_id;
};
```

- `flush_id` – local ID used to track completion

Flush returns a response with the following header format:

```
struct urom_rdmo_flush_rsp_hdr {
    uint64_t    flush_id;
};
```

- `flush_id` – the ID of the completed Flush

Flush requests and responses do not include a payload.

Scatter

RDMO Scatter is used to support aggregating non-contiguous memory Puts. A n RDMO may be defined to map non-contiguous virtual addresses into a single memory region using a network interface at the target platform, and then return a memory key for this region. The initiator may then perform Puts to this memory region, which are scattered by target hardware. Alternatively, an RDMO may be defined to post an IOV Receive. The initiator could then post a matching Send to scatter data at the target.

The Scatter operation uses operation type `UROM_RDMO_OP_SCATTER`. Scatter header format:

```
struct urom_rdmo_scatter_hdr {
        uint64_t count; /* Number of IOVs in the payload */
```

```
};
```

- `count` – Number of IOVs in the RDMO payload

IOVs are packed into the Scatter request payload, descriptor followed by data:

```
struct urom_rdmo_scatter_iov {
        uint64_t addr; /* Scattered data address */
        uint64_t rkey; /* Data remote key */
        uint16_t len;  /* Data length */
};
```

- `addr` – scattered data address
- `rkey` – data remote key
- `len` – data length

DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA UROM
- UCX framework DOCA driver

Refer to their respective programming guide for more information.

Compiling the Application

Info

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

Tip

For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The sources of the application can be found under the application's directory: `/opt/mellanox/doca/applications/urom_rdmo/`.

Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

Info

On the host, `doca_urom_rdmo` is created under `/tmp/build/urom_rdmo/host/`. On the BlueField side, the RDMO

worker plugin `worker_rdmo.so` is created under `/tmp/build/urom_rdmo/dpu/`.

Compiling Only the Current Application

To directly build only the UROM RDMO application (host) or plugin (DPU):

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -
Denable_urom_rdmo=true
ninja -C /tmp/build
```

Info

On the host, `doca_urom_rdmo` is created under `/tmp/build/urom_rdmo/host/`. On the BlueField side, the RDMO worker plugin `worker_rdmo.so` is created under `/tmp/build/urom_rdmo/dpu/`.

Alternatively, one can set the desired flags in the `meson_options.txt` file instead of providing them in the compilation command line:

1. Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt`:
   - Set `enable_all_applications` to `false`
   - Set `enable_urom_rdmo` to `true`
2. Run the following compilation commands :

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

Info

On the host, `doca_urom_rdmo` is created under `/tmp/build/urom_rdmo/host/`. On the BlueField side, the RDMO worker plugin `worker_rdmo.so` is created under `/tmp/build/urom_rdmo/dpu/`.

Troubleshooting

Refer to the DOCA Troubleshooting for any issue encountered with the compilation of the application .

Running the Application

Host Application Execution

The UROM RDMO application is provided in source form; therefore, a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_urom_rdmo [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
```

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

Program Flags:
  -d, --device <IB device name>       IB device name.
  -s, --server-name <server name>     server name.
  -m, --mode {server, client}         Set mode type {server,
client}
```

Info

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
./doca_urom_rdmo -h
```

Info

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application with server mode:

```
./doca_urom_rdmo -d mlx5_0 -m server
```

3. CLI example for running the application with client mode:

```
./doca_urom_rdmo -m clinet -s <server_host_name>
```

4. The application also supports a JSON-based deployment mode, in which all command-line arguments are provided through a JSON file:

```
./doca_urom_rdmo --json [json_file]
```

For example:

```
./doca_urom_rdmo --json ./urom_rdmo_params.json
```

RDMO DPU Plugin Component

The UROM RDMO plugin component is provided in source form, hence a compilation is required before the application can be executed in order when spawning UROM worker could load the plugin in runtime and it is compiled as `.so` file.

The plugin exposes the following symbols:

- Get DOCA worker plugin interface for RDMO plugin:

```
doca_error_t urom_plugin_get_iface(struct urom_plugin_iface
```

```
*iface);
```

- Get the RDMO plugin version which will be used to verify that the host and DPU plugin versions are compatible:

```
doca_error_t urom_plugin_get_version(uint64_t *version);
```

Command Line Flags

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
General flags	`h`	`help`	Print a help synopsis	N/A
	`v`	`version`	Print program version information	N/A
	`l`	`log-level`	Set the log level for the application: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70 (requires compilation with `TRACE` log level support)	`"log-level" : 60`

Flag Type	Short Flag	Long Flag/JSON Key	Description	JSON Content
	N/A	`sdk-log-level`	Set the log level for the program: • DISABLE=10 • CRITICAL=20 • ERROR=30 • WARNING=40 • INFO=50 • DEBUG=60 • TRACE=70	`"sdk-log-level" : 40`
	`j`	`json`	Parse all command flags from an input JSON file	N/A
Program flags	`d`	`device`	DOCA UROM IB device name	`"device" : "mlx5_0"`
	`s`	`server-name`	RDMO server name	`"server-name" : "<host-name>-oob"`
	`m`	`mode`	RDMO application mode [server, client]	`"mode" : "client"`

Info

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

Troubleshooting

Refer to the DOCA Troubleshooting for any issue encountered with the installation or execution of the DOCA applications.

Application Code Flow

1. Parse application argument.
    1. Initialize arg parser resources and register DOCA general parameters.

        ```
        doca_argp_init();
        ```

    2. Register UROM RDMO application parameters.

        ```
        register_urom_rdmo_params();
        ```

    3. Parse the arguments.

        ```
        doca_argp_start();
        ```

2. Run main logic:
    - If the application mode is server:
        1. Create UROM objects and spawn UROM worker on the BlueField.
        2. Initialize UCP with features: `UCP_FEATURE_AM`, `UCP_FEATURE_EXPORTED_MEMH`.
        3. Create a UCP worker and query the worker address

4. Initialize the RDMO worker client with the command `UROM_WORKER_CMD_RDMO_CLIENT_INIT`.

5. Send UROM RDMO worker address to the initiator via OOB channel and receive the intiator's UCP worker address

6. Create a UCP memory handle and register it with the RDMO server using the command `UROM_WORKER_CMD_RDMO_MR_REG`. Receive an R-key in return.

7. Send the RDMO key to the initiator

8. Create an RDMO RQ by passing the initiator's UCP worker address to the UROM command `UROM_WORKER_CMD_RDMO_RQ_CREATE`.

9. Wait till the RDMO append operation is done and next validate the memory data.

10. Wait till the RDMO scatter operation is done and next validate the memory data.

11. Destroy the UCP resources.

12. Destroy UROM RDMO worker and UROM objects.

- If the application mode is client:

  1. Create UCP worker using UCX API directly.

  2. Receive the UROM RDMO worker address via OOB channel and send the initiator's UCP worker address.

  3. Create a UCP endpoint using the RDMO worker address.

  4. Install an Active Message handler on the endpoint to receive RDMO responses.

  5. Send an RDMO requests via UCP Active Message protocol with the header pointing to the serialized RDMO and Op headers, and data pointing to the payload. The request parameter flag: `UCP_AM_SEND_FLAG_REPLY` will be set to allow the RDMO server to identify the sender.

  6. Once the RDMO operations are done, Destroy UCP resources.

3. Arg parser destroy.

```
doca_argp_destroy();
```

References

- `/opt/mellanox/doca/applications/urom_rdmo/`
- `/opt/mellanox/doca/applications/urom_rdmo/urom_rdmo_params.json`

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 and the NVIDIA logo are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.
