DOCA Drivers
# Table of contents

**DOCA UCX**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLX Drivers (MLNX_OFED)</td>
<td>24</td>
</tr>
<tr>
<td>InfiniBand Network</td>
<td>24</td>
</tr>
<tr>
<td>InfiniBand Interface</td>
<td>25</td>
</tr>
<tr>
<td>NVIDIA SM</td>
<td>25</td>
</tr>
<tr>
<td>QoS - Quality of Service</td>
<td>87</td>
</tr>
<tr>
<td>IP over InfiniBand (IPoIB)</td>
<td>92</td>
</tr>
<tr>
<td>Advanced Transport</td>
<td>105</td>
</tr>
<tr>
<td>Optimized Memory Access</td>
<td>107</td>
</tr>
<tr>
<td>NVIDIA PeerDirect</td>
<td>113</td>
</tr>
<tr>
<td>CPU Overhead Distribution</td>
<td>114</td>
</tr>
<tr>
<td>Out-of-Order (OOO) Data Placement</td>
<td>114</td>
</tr>
<tr>
<td>IB Router</td>
<td>115</td>
</tr>
<tr>
<td>MAD Congestion Control</td>
<td>117</td>
</tr>
<tr>
<td>Storage Protocols</td>
<td>118</td>
</tr>
<tr>
<td>SRP - SCSI RDMA Protocol</td>
<td>119</td>
</tr>
<tr>
<td>iSCSI Extensions for RDMA (iSER)</td>
<td>134</td>
</tr>
<tr>
<td>Lustre</td>
<td>137</td>
</tr>
<tr>
<td>NVME-oF - NVM Express over Fabrics</td>
<td>138</td>
</tr>
<tr>
<td>Virtualization</td>
<td>139</td>
</tr>
<tr>
<td>Single Root IO Virtualization (SR-IOV)</td>
<td>139</td>
</tr>
<tr>
<td>SR-IOV Live Migration</td>
<td>163</td>
</tr>
<tr>
<td>Enabling Paravirtualization</td>
<td>175</td>
</tr>
<tr>
<td>VXLAN Hardware Stateless Offloads</td>
<td>176</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Q-in-Q Encapsulation per VF in Linux (VST)</td>
<td>178</td>
</tr>
<tr>
<td>802.1Q Double-Tagging</td>
<td>181</td>
</tr>
<tr>
<td>Scalable Functions</td>
<td>183</td>
</tr>
<tr>
<td>Resiliency</td>
<td>183</td>
</tr>
<tr>
<td>Reset Flow</td>
<td>183</td>
</tr>
<tr>
<td>Docker Containers</td>
<td>187</td>
</tr>
<tr>
<td>Docker Using SR-IOV</td>
<td>187</td>
</tr>
<tr>
<td>Kubernetes Using SR-IOV</td>
<td>188</td>
</tr>
<tr>
<td>Kubernetes with Shared HCA</td>
<td>188</td>
</tr>
<tr>
<td>HPC-X</td>
<td>188</td>
</tr>
<tr>
<td>Fast Driver Unload</td>
<td>188</td>
</tr>
</tbody>
</table>
This section describes underlying drivers included in DOCA and includes the following pages:

- **DOCA UCX**

- **MLX Drivers (MLNX_OFED)**
DOCA UCX

This guide provides instructions for developing applications on top of the UCX library.

Introduction

Unified Communication X (UCX) is an optimized point-to-point communication framework.

UCX exposes a set of abstract communication primitives that utilize the best available hardware resources and offloads, such as active messages, tagged send/receive, remote memory read/write, atomic operations, and various synchronization routines. The supported hardware types include RDMA (InfiniBand and RoCE), TCP, GPUs, and shared memory.

UCX facilitates rapid development by providing a high-level API, masking the low-level details, while maintaining high-performance and scalability.

UCX implements best practices for transfer of messages of all sizes, based on the accumulated experience gained from applications running on the world's largest datacenters and supercomputers.

Prerequisites

UCX runtime libraries are installed as part of the DOCA installation.

UCX is used the same way from the host and the DPU side.

Any active network device available on the system might be used by UCX, including network devices that might be unreachable to the remote peer.

If one of the destinations is not reachable via a certain network device (e.g., a BlueField cannot reach another BlueField via `tmfifo_net0`), UCX communication may fail.

To resolve this, use the UCX environment variable `UCX_NET_DEVICES` to specify which devices UCX can use. For example:
Or:

```
env UCX_NET_DEVICES=mlx5_2:1,mlx5_3:1 <UCX-program>
```

Using the command `show_gids` on the BlueField one can obtain the mlx device name and the port of an SF. Then that can be used to limit the UCX network interfaces and allow IB. For example:

```
dpu> show_gids

<table>
<thead>
<tr>
<th>DEV</th>
<th>PORT</th>
<th>INDEX</th>
<th>GID</th>
<th>IPv4</th>
<th>VER</th>
<th>DEV</th>
</tr>
</thead>
<tbody>
<tr>
<td>mlx5_2</td>
<td>1</td>
<td>0</td>
<td>fe80:0000:0000:0000:0000:0052:72ff:fe63:1651</td>
<td>v2 enp3s0f0s0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mlx5_3</td>
<td>1</td>
<td>0</td>
<td>fe80:0000:0000:0000:0000:0032:6bff:fe13:f13a</td>
<td>v2 enp3s0f1s0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dpu> env UCX_NET_DEVICES=mlx5_2:1,mlx5_3:1 <UCX-program>
```

When RDMACM is not available, it is also required to list the Ethernet devices in the `UCX_NET_DEVICES` configuration, so they could be used for TCP-based connection establishment. For example:

```
dpu> env UCX_NET_DEVICES=enp3s0f0s0,enp3s0f1s0,mlx5_2:1,mlx5_3:1 <UCX-program>
```

**Architecture**

The following image describes the software layers of UCX middleware.

On the upper layer, various applications that utilize high-speed communications are built on top of the UCX high-level API (UCP).

UCP layer implements the business logic to utilize, combine, and manipulate different transports to achieve the best possible performance for different use cases. This logic
decides which transports must be used for each message, which types of basic hardware communication primitives to use, how to fragment messages, etc.

UCT, the transport API, is a hardware abstraction layer that brings different types of communication devices to a common denominator. There are multiple communication primitives defined by UCT API, but each transport service may implement only some of them—preferably the ones that are natively supported by the underlying hardware. UCT users (e.g., UCP) are expected to handle the missing communication primitives defined by UCT API but not implemented by a transport service.

**UCP Objects**

This section describes the high-level communication objects that are used by most applications written on top of UCX.

**UCP Context (ucp_context_h)**

The context is the top-level object and it defines the scope of all other UCX objects. It is possible to create multiple contexts in the same process to have a complete separation of hardware and memory resources.

**UCP Worker (ucp_worker_h)**
The worker represents a communication state and its associated network resources. It is responsible for sending and processing incoming messages and handling all network-related events. All point-to-point connections are created in the scope of a particular worker.

A worker object can be defined to support usage from multiple threads. However, due to lock contention, the performance is better when a given worker is used most of the time from one thread.

The worker progresses communications either by active polling, waiting for asynchronous events, or a combination of both.

**UCP Endpoint (ucp_ep_h)**

The endpoint represents a connection from a local worker to a remote worker. That remote worker may be created in any place that is reachable by one of the communication networks supported by UCT layer. That could be, for example, on a different host in the fabric, the same host, on the DPU, or even in the same process.

**UCP Listener (ucp_listener_h)**

The listener binds to a network port number on the underlying operating system, and dispatches incoming connection requests. The incoming connection request can be used to create a matching endpoint on the server (passive) side or rejected and released.

**UCP Request (ucp_request_h)**

The request object is created by one of the non-blocking communications primitives in a case where the operation could not be completed immediately in-place. The application is expected to check the request for completion, either by testing it directly, or by associating a custom callback with the request.
API

This section describes the main UCX APIs for high-speed communications. For the full reference, refer to UCX API specification.

UCX exposes two kinds of API: the high-level UCP API and the low-level UCT (transport) API. For most applications, it is recommended to use only the UCP API, since it relieves much of the burden of handling each transport’s capabilities, limitations, and performance traits.

Many of the APIs accept a structure pointer with a field_mask as an argument. This method is used to provide backward ABI/API compatibility: If new function arguments are introduced, they are added as new fields in the struct, so the function signature does not change. In addition, field_mask specifies which struct fields are valid from the caller’s (user application) perspective. UCX only accesses the fields enabled by this bitmask and uses default values for the remaining struct fields.

Some APIs require passing user-defined callbacks as a method to get notifications about specific events. Unless otherwise specified, such callbacks are called from the context of the ucp_worker_progress() call (detailed below), and are expected to complete quickly or defer some of their tasks to another thread (to avoid timeouts and starvation of processing from other network events).

The pkg-config (*.pc file) for the UCX library is named ucx.

The following sections provide additional details about the library API.

**ucs_status_t**

An enum type that holds all UCX error codes.

**ucp_init**
**ucp_init**

```c
ucs_status_t ucp_init(const ucp_params_t *params, const ucp_config_t *config, ucp_context_h *context_p)
```

- **params [in]** – points to a structure with optional parameters. All fields are optional except features, which must be set.

- **config [in]** – optional, can be NULL for default behavior. Configuration can be obtained by calling `ucp_config_read()`.

**Note**

The supported configuration options can change between UCX versions. The full list can be obtained by running the `ucx_info` CLI tool:

```
ucx_info -c -f
```

- **context_p [out]** – a pointer to a location in memory for the created UCP context

The function returns an error code as defined by `ucs_status_t`.

This function creates a new UCP top-level context and returns it by value in the `context_p` argument.

**ucp_cleanup**

```c
void ucp_cleanup(ucp_context_h context_p)
```

- **context_p [in]** – a UCP context instance

This function destroys a previously created context. Prior to calling this function, any other resources created on this context (e.g., workers or endpoints) must be destroyed.
**ucp_worker_create**

```c
ucs_status_t ucp_worker_create(ucp_context_h context, const ucp_worker_params_t *params, ucp_worker_h *worker_p)
```

- **context [in]** – an existing UCP context
- **params [in]** – points to a structure with configuration parameters. All fields are optional. Commonly, only the field `thread_mode` is used. Possible `thread_mode` values are as follows:
  - **UCS_THREAD_MODE_SINGLE** – only one specific thread (typically, the one that created the worker) is used to access the worker and its associated endpoints.
  - **UCS_THREAD_MODE_SERIALIZED** – multiple threads can access the worker and its associated endpoints, but only one at a time. This implies an exclusion mechanism (e.g., locking) implemented in the application. Sometimes, more expensive bus flushing instructions are needed with serialized mode, compared to single thread mode.
  - **UCS_THREAD_MODE_MULTI** – multiple threads can access the worker at any given time. UCX takes care of the locking internally. As of version 1.12, it is implemented as a global lock on the worker.
- **worker_p [out]** – a pointer to a location in memory for the created worker

The function returns an error code as defined by `ucs_status_t`.

This function creates a new UCP worker on a previously created context and returns it by value in the `worker_p` argument.

**Note**

When `ucp_worker_create()` succeeds, the caller is still expected to check the actual thread mode the worker was created with by calling `ucp_worker_query()` API, and take the necessary actions (for example,
ucp_worker_destroy

```c
void ucp_worker_destroy(ucp_worker_h worker)
```

- `context_p [in]` – an UCP worker instance

This function destroys a previously created worker. Prior to calling this function, all associated endpoints and listeners must be destroyed.

Destroying the worker may cause communication errors on any remote peer that has an open endpoint to this worker. These errors are handled according to that endpoint's error handling configuration (detailed in section ucp_ep_create).

ucp_listener_create

```c
ucs_status_t ucp_listener_create(ucp_worker_h worker, const ucp_listener_params_t *params, ucp_listener_h *listener_p)
```

- `worker [in]` – an existing UCP worker
- `params [in]` – points to a structure with configuration parameters. The fields `sockaddr` and `conn_handler` are mandatory, but the rest of the fields are optional.
  - `sockaddr` – specifies IPv4/IPv6 address to listen for connections. The semantics are similar to the built-in `bind()` function. `INADDR_ANY/INADDR6_ANY` can be used to listen on all network interfaces. If the port number is set to 0, a random unused port is selected. The actual port number can be obtained by calling the `ucp_listener_query()` API.
- conn_handler – a callback for handling incoming connection requests along with an associated user-defined argument. The callback type is defined as:

```
void (*ucp_listener_conn_callback_t) (ucp_conn_request_h conn_request, void *arg)
```

Whenever a remote endpoint is created through this listener, this callback is called on the listener side with a new conn_request object representing the incoming connection, and the user-defined argument arg that is passed to ucp_listener_create().

The callback is expected to process this connection request by either creating an endpoint for it (pass conn_request as a parameter to ucp_ep_create, including on a different worker), or rejecting and destroying it (call ucp_listener_reject). This does not have to happen immediately. The callback may put the connection request on an internal application queue and process it later.

- listener_p [out] – a pointer to a location in memory for the created listener

The function returns an error code as defined by ucs_status_t.

This function creates a new listener object to accept incoming connections on a specific network port, and returns it by value in the listener_p argument.

**ucp_listener_destroy**

```
void ucp_listener_destroy(ucp_listener_h listener_p)
```

- listener_p [in] – a listener instance

This function destroys a previously created listener. Prior to calling this function, any connection requests that were reported by conn_handler are expected to be processed. Pending connection requests that have not been reported to the application yet, or new connection requests that arrive after this function is called, are rejected.
ucp_ep_create

```c
ucp_status_t ucp_ep_create(ucp_worker_t worker, const ucp_ep_params_t *params, ucp_ep_t *ep_p)
```

- **worker [in]** – an existing UCP worker
- **params [in]** – Points to a structure with configuration parameters. A *creation mode* field must be set. Other fields are optional. Commonly used fields are described in the following subsections.
- **ep_p [in]** – a pointer to a location in memory for the created endpoint

The function returns an error code as defined by `ucs_status_t`.

This function creates a new connection to a remote peer and returns it by value in the `ep_p` parameter. The new endpoint can be used for communication immediately after it is created, though some operations may be queued internally and sent after the underlying connection is established.

### Create Modes (ucp_ep_params_t)

There are three ways the endpoint can be created:

- **Client connects to a remote listener**

  In this case, the sockaddr field specifies the remote IPv4/IPv6 address and port number. The flags field must be enabled and must include the `UCP_EP_PARAMS_FLAGS_CLIENT_SERVER` flag. Optionally, from UCX version 1.13 on, the local_sockaddr field may be used to specify a local source device address to bind to.

- **Server creates an endpoint due to an incoming connection request**

  In this case, the `conn_request` field must be set to this connection request. Such endpoint can optionally be created on a different worker, not the same one this connection request was accepted on.

- **Create an endpoint to a specific worker address**
In this case, the field `address` must be set to point to a remote worker’s address. That address (and its length) must be obtained on the remote side by calling `ucp_worker_query()` and sent using an application-defined method (e.g., TCP socket, or other existing communication mechanism). The internal structure of the address is opaque and may change in different versions.

**User-Defined Error Handling (ucp_ep_params_t)**

By default, unexpected errors on the connection (e.g., network disconnection or aborted remote process) generate a fatal failure. To enable graceful error handling, several parameters must be set during endpoint creation:

- The `err_mode` field must be set to `UCP_ERR_HANDLING_MODE_PEER`. This guarantees that send requests are always completed (successfully or error). Otherwise, network errors are considered fatal and abort the application without giving it a chance to perform cleanup or fallback flows.

- The `err_handler.cb` field must be set to a user-defined callback which is called if a connection error occurs. The error handler is defined as follows:

```c
void (*ucp_err_handler_cb_t)(void *arg, ucp_ep_h ep, ucs_status_t status)
```

The callback parameters are the user-defined argument (passed in `user_data`), the endpoint handle on which the error happened, and the error code.

After this callback, no more communications should be done on the endpoint. The application is expected to close the endpoint.

- The `user_data` field must be set to a user-defined argument passed to the `err_handler` callback

**ucs_status_ptr_t**
typedef void* ucs_status_ptr_t;

This function is commonly used as a return value for non-blocking operations.

The return value of ucs_status_ptr_t combines a status code and a request pointer which may be one of the following:

- A NULL pointer indicating that the operation has completed successfully in-place. The user-provided callback, if there is one, is not called.

- An error status, that can be detected by the UCS_PTR_IS_ERR(status) macro and extracted by UCS_PTR_STATUS(status).

- Otherwise, the status is a request pointer which can also be detected by the UCS_PTR_IS_PTR(status) macro. This means that the communication operation has started (or was queued) but not yet completed. The completion is reported by calling the user-provided callback (in ucp_request_param_t) or through an explicit check on the request status by calling ucp_request_check_status().

ucp_ep_close_nbx

```c
ucs_status_ptr_t ucp_ep_close_nbx(ucp_ep_h ep, const ucp_request_param_t *param)
```

- ep [in] – an existing UCP endpoint

- param [in] – points to a structure that defines how the closing operation is performed. The flags field of the param structure specifies which method to use to close the endpoint:
  
  - UCP_EP_CLOSE_MODE_FORCE – close the endpoint immediately without attempting to flush outstanding operation. Some requests already completed on the transport level may complete successfully, others may be completed with an error status. In the latter case, it is not known whether they have reached the destination process or completed there.
Closing an endpoint this way is equivalent to calling `close()` on a TCP socket and can generate a connection error on the remote side. Therefore, to use this mode, both the local and remote endpoints must be created with the `err_mode` parameter set to `UCP_ERR_HANDLING_MODE_PEER`.

- `UCP_EP_CLOSE_MODE_FLUSH` – synchronize with the remote peer and flush outstanding operations. Some operations may be canceled and complete with the status `UCS_ERR_CANCELED`. However, it is guaranteed that they did not complete on the remote peer as well.

The function returns a status pointer to check the operation's status. `NULL` means success.

This function starts the process of closing a previously created endpoint. The function is non-blocking, and the returned value is a status pointer used to indicate when the endpoint is fully destroyed. For more information, refer to section Communications.

**ucp_request_param_t**

```c
struct ucp_request_param_t {
    uint32_t op_attr_mask;
    uint32_t flags;
    union ucp_request_param_t cb;
    void *user_data;
    ucp_datatype_t datatype;
    /* Some other fields that are rarely used */
    ...
}
```

- `op_attr_mask [in]` – mask of enabled fields and several control flags.
- `cb [in]` – callback for when the operation is completed.
- `user_data [in]` – user-defined argument passed to the completion callback.
• datatype [in] – may be used to specify a custom data layout for the data buffer (not user_data) that is provided to the communication API. If this parameter is not set, the data buffer is treated as a contiguous byte buffer.

The fields of ucp_request_param_t specify several common attributes and flags that are used to control how the communications request is allocated and completed. This is aimed to optimize different use-cases.

**ucp_worker_progress**

```c
unsigned ucp_worker_progress(ucp_worker_h worker)
```

• worker [in] – an existing UCP worker

The function returns a non-zero value if any communication has been progressed. Otherwise, it returns zero.

This function progresses outstanding communications on the worker. This includes polling hardware and shared memory queues, calling callbacks, pushing pending operations to the network devices, advancing the state of complex protocols, progressing connection establishment process, and more.

Though some transports, such as RDMA, offload do much of the heavy lifting, the initiation and completion of communication operations still must be performed explicitly by the process. UCX does not spawn additional progress threads. Instead, it is expected that the upper-layer application spawns its own progress thread, as needed, to call ucp_worker_progress().

**Note**

This function cannot be used from inside a callback.
ucp_am_send_nbx

```
ucs_status_ptr_t ucp_am_send_nbx(ucp_ep_h ep, unsigned id, const void *header,
                                 size_t header_length, const void *buffer,
                                 size_t count, const ucp_request_param_t *param)
```

- **ep [in]** – connection to send the active message on. Previously returned from ucp_ep_create().

- **id [in]** – active message identifier. This is an arbitrary 16-bit integer value defined by the application and used to select the active message callback to call on the receiver side. This allows handling different types of messages by different callback functions.

- **header [in]** – pointer to a user-defined header for an active message

- **header_length [in]** – length of the header to send. Usually, the header is small and, in any case, it should be no larger than the `max_am_header` worker attribute, as returned from ucp_worker_query(). The header size could vary depending on the available transports and is usually expected to be at least 256 bytes.

- **buffer [in]** – pointer to the active message payload

- **count [in]** – number of elements in the payload buffer. By default, each element is a single byte, so this is the byte-length of the buffer. Other data layouts, such as IO vector (IOV) list, could be specified by `param->datatype`.

- **param [in]** – additional parameters controlling request completion semantics. The relevant field is only `flags` and it can be set to a combination of the following flags:
  - `UCP_AM_SEND_FLAG_REPLY` – force passing `reply_ep` to the callback on the receiver side. This can increase the internal header size and add some overhead.
  - `UCP_AM_SEND_FLAG_EAGER` – force using eager protocol (details below).
  - `UCP_AM_SEND_FLAG_RNDV` – force using rendezvous protocol (details below).

The active message can be sent either by the eager or rendezvous protocol. Eager protocol means the data buffer is available on the receiver immediately.
during the callback, while the rendezvous protocol requires fetching the data using an additional call to \texttt{ucp\_am\_recv\_data\_nbx()}, allowing it to be placed directly to an application-selected buffer. By default, smaller messages are sent via eager protocol, and larger messages use rendezvous protocol. This can be overridden using \texttt{UCP\_AM\_SEND\_FLAG\_EAGER} or \texttt{UCP\_AM\_SEND\_FLAG\_RNDV}.

\begin{quote}
\begin{center}
\textbf{Note}
\end{center}
\begin{quote}
\texttt{UCP\_AM\_SEND\_FLAG\_EAGER} and \texttt{UCP\_AM\_SEND\_FLAG\_RNDV} are mutually exclusive.
\end{quote}
\end{quote}

The function returns a status pointer to check the operation's status. \texttt{NULL} means success.

This function initiates sending of an active message from the initiator side. As a result, a designated callback (registered by \texttt{ucp\_worker\_set\_am\_recv\_handler}) is called on the receiver side to handle this message. The function is non-blocking, so if the send operation is not completed immediately, a request handle is returned.

\textbf{ucp\_worker\_set\_am\_recv\_handler}

\begin{verbatim}
ucs\_status\_t ucp\_worker\_set\_am\_recv\_handler(ucp\_worker\_h worker, const\ ucp\_am\_handler\_param\_t *param)
\end{verbatim}

- \texttt{worker} \texttt{[in]} – an existing UCP worker.
- \texttt{param} \texttt{[in]} – set callback configurations. See more below.

The function returns a non-zero value if any communication has been progressed. Otherwise, it returns zero.

This function registers a callback for processing active messages on the given worker.

The following are the mandatory fields to set in \texttt{param}:

\begin{itemize}
  \item \texttt{ucp\_am\_recv\_data\_callback}
  \item \texttt{ucp\_am\_recv\_error\_callback}
  \item \texttt{ucp\_am\_recv\_forward\_callback}
\end{itemize}
• **id** – active message identifier to bind with the registered callback. Callback is invoked when receiving incoming messages with the same ID.

• **arg** – a user-defined argument to pass to the active message callback.

• **cb** – a user-defined callback to invoke when an active message arrives. The callback is defined as:

```c
ucs_status_t (*ucp_am_recv_callback_t)(void *arg, const void *header,
    size_t header_length, void *data,
    size_t length,
    const ucp_am_recv_param_t *param)
```

The following are the parameters passed from UCX to the callback:

• **arg** – the same user-defined argument passed to `ucp_worker_set_am_recv_handler`.

• **header** – points to the active message header as defined by the sender side while sending the active message. The header should be consumed by the callback since it is not valid after the callback returns.

• **header_length** – valid size of the buffer pointer by `header`.

• **data** – pointer to the data or an opaque handle that can be used to fetch the data according to the `UCP_AM_RECV_ATTR_FLAG_RNDV` flag in the field `param->recv_attr`. When flag is on, this is an opaque handle.

• **length** – length of the active message data (even if the data argument is an opaque handle and not the actual data).

• **param** – pointer to additional parameters of the incoming message. The relevant fields are:
  
  • **recv_attr** – flags providing more information about the incoming message.
  
  • **reply_ep** – if `UCP_AM_RECV_ATTR_FIELD_REPLY_EP` is set in `recv_attr`, then this field holds a handle to an endpoint that can be used to send replies to the active message sender.
The callback is expected to return UCS_OK if the message data has been consumed or if UCP_AM_RECV_ATTR_FLAG_RNDV is set in recv_attr. Otherwise, the if UCP_AM_RECV_ATTR_FLAG_DATA is set in recv_attr, the callback is allowed to keep the data for later processing (by adding it to an internal application queue, for example). In this case, the callback should return UCS_INPROGRESS as indication that the data should persist.

When a message arrives with UCP_AM_RECV_ATTR_FLAG_RNDV flag, the function ucp_am_recv_data_nbx must be used to fetch the data from the sender.

**ucp_am_recv_data_nbx**

```c
ucs_status_ptr_t ucp_am_recv_data_nbx(ucp_worker_h worker, void *data_desc,
                                       void *buffer, size_t count,
                                       const ucp_request_param_t *param)
```

- **worker [in]** – UCP worker object to use for initiating the receive operation.

  **Note**
  
  The connection handle (endpoint) is not needed.

- **data_desc [in]** – handle for the data to receive. Obtained from the data argument for the active message callback.

- **buffer [in]** – receive buffer for the incoming data.

- **count [in]** – number of elements in the payload buffer. By default, each element is a single byte, so this is the byte-length of the buffer. Other data layouts, such as the IOV list, may be specified by param->datatype.

- **param [in]** – additional parameters that control request allocation and completion reporting. No specific flags are needed for this function.
The function returns a status pointer to check the operation's status. NULL means success.

This function is used for rendezvous active messages. The function initiates the process of fetching data from the sender side into an application-defined receive buffer. It is expected to be used when an active message callback is called with the UCP_AM_RECV_ATTR_FLAG_RNDV flag set in params->recv_attr field.

**UCX Best Practices**

**Initialization**

An application using UCX will usually create one global context (ucp_context_h) then create one or more workers (ucp_worker_h). Each worker consumes some memory for send/receive buffers, so it is not recommended to create too many workers. The rule of thumb is that the number of workers should be roughly tied to the number of CPU cores/threads.

The mapping of workers to threads is defined by the application's use case, for example:

- A single-threaded application does not need more than one worker
- A simple implementation of a multi-threaded application can create one or more workers in multi-threaded mode. These workers can be used by any thread.
- A multi-threaded application with a strong affinity between the thread and CPU core can create a dedicated worker per thread. These workers can be created in a single-threaded mode.
- Applications with many threads can implement a pool of workers and use one randomly or assign some to threads temporarily.

**Note**
To initiate communications, the application should create endpoints (ucp_ep_h) connected to the remote peers. There are two main methods to create an endpoint: Either by connecting directly to a remote worker’s address, or by creating a listener object (ucp_listener_h) and connecting to remote IP address and port. These methods are described in more detail in the ucp_ep_create() section.

Communications

After initializing the UCP context, worker, and endpoints, the application can start using the endpoint for communications. Usually, endpoints are associated with application-level object that represents a connection.

Most communication operations follow a similar pattern: A non-blocking function (with _nbx suffix) receives a pointer to the ucp_request_param_t structure and returns ucs_status_ptr_t. Using a struct pointer allows extending the operations and while maintaining backward compatibility.

There are several types of communication methods supported by UCP intended for different kinds of applications. The recommended method for most applications is active messages which mean that the initiator can send arbitrary data to the responder, and the responder invokes a callback that can access this data.
MLX Drivers (MLNX_OFED)

Note

It is recommended to enable the "above 4G decoding" BIOS setting for features that require a large amount of PCIe resources (e.g., SR-IOV with numerous VFs, PCIe Emulated Switch, Large BAR Requests).

The chapter contains the following sections:

- InfiniBand Network
- Storage Protocols
- Virtualization
- Resiliency
- Docker Containers
- HPC-X
- Fast Driver Unload

InfiniBand Network

The chapter contains the following sections:

- InfiniBand Interface
- NVIDIA SM
- QoS - Quality of Service
- IP over InfiniBand (IPoIB)
- Advanced Transport
- Optimized Memory Access
- NVIDIA PeerDirect
- CPU Overhead Distribution
- Out-of-Order (OOO) Data Placement
- IB Router
- MAD Congestion Control

InfiniBand Interface

Port Type Management

For information on port type management of ConnectX-4 and above adapter cards, please refer to Port Type Management/VPI Cards Configuration section.

RDMA Counters

- RDMA counters are available only through sysfs located under:
  - `#/sys/class/infiniband/<device>/ports/*/hw_counters/`
  - `#/sys/class/infiniband/<device>/ports/*/counters`

For mlx5 port and RDMA counters, refer to the Understanding mlx5 Linux Counters Community post.

NVIDIA SM

NVIDIA SM is an InfiniBand compliant Subnet Manager (SM). It is provided as a fixed flow executable called "opensm", accompanied by a testing application called osmtest. NVIDIA

**OpenSM Application**

OpenSM is an InfiniBand compliant Subnet Manager and Subnet Administrator that runs on top of the NVIDIA OFED stack. OpenSM performs the InfiniBand specification's required tasks for initializing InfiniBand hardware. One SM must be running for each InfiniBand subnet.

OpenSM defaults were designed to meet the common case usage on clusters with up to a few hundred nodes. Thus, in this default mode, OpenSM will scan the IB fabric, initialize it, and sweep occasionally for changes.

OpenSM attaches to a specific IB port on the local machine and configures only the fabric connected to it. (If the local machine has other IB ports, OpenSM will ignore the fabrics connected to those other ports). If no port is specified, opensm will select the first "best" available port. opensm can also present the available ports and prompt for a port number to attach to.

By default, the OpenSM run is logged to/var/log/opensm.log. All errors reported in this log file should be treated as indicators of IB fabric health issues. (Note that when a fatal and non-recoverable error occurs, OpenSM will exit). opensm.log should include the message "SUBNET UP" if OpenSM was able to set up the subnet correctly.

**Syntax**

```
opensm [OPTIONS]
```

For the complete list of OpenSM options, please run:

```
opensm --help / -h / -?
```

**Environment Variables**

The following environment variables control OpenSM behavior:
• OSM_TMP_DIR - controls the directory in which the temporary files generated by OpenSM are created. These files are: opensm-subnet.lst, opensm.f dbs, and opensm.mcf dbs. By default, this directory is /var/log.

• OSM_CACHE_DIR - opensm stores certain data to the disk such that subsequent runs are consistent. The default directory used is /var/cache/opensm. The following file is included in it:

    guid2lid – stores the LID range assigned to each GUID

**Signaling**

When OpenSM receives a HUP signal, it starts a new heavy sweep as if a trap has been received or a topology change has been found.

Also, SIGUSR1 can be used to trigger a reopen of /var/log/opensm.log for logrotate purposes.

**Running OpenSM as Daemon**

OpenSM can also run as daemon. To run OpenSM in this mode, enter:

```
host1# service opensmd start
```

**osmtest**

osmtest is a test program for validating the InfiniBand Subnet Manager and Subnet Administrator. osmtest provides a test suite for opensm. It can create an inventory file of all available nodes, ports, and PathRecords, including all their fields. It can also verify the existing inventory with all the object fields and matches it to a pre-saved one.

osmtest has the following test flows:
- Multicast Compliancy test
- Event Forwarding test
- Service Record registration test
- RMPP stress test
- Small SA Queries stress test

For further information, please refer to the tool's man page.

**Partitions**

OpenSM enables the configuration of partitions (PKeys) in an InfiniBand fabric. By default, OpenSM searches for the partitions configuration file under the name `/etc/opensm/partitions.conf`. To change this filename, you can use opensm with the `--Pconfig` or `-P` flags.

The default partition is created by OpenSM unconditionally, even when a partition configuration file does not exist or cannot be accessed.

The default partition has a P_Key value of 0x7fff. The port out of which runs OpenSM is assigned full membership in the default partition. All other end-ports are assigned partial membership.

**Note**

- Adding a new partition to the partition.conf file, does not require SM restart, but signalling SM process via a HUP signal (e.g. `pkill -HUP opensm`).
- The default partition cannot be removed.
Adjustments to the Port GUIDs, including additions, removals, or membership alterations (denoted as "<PortGUID>=[full|limited|both]" in the "Partition Definition") can be applied with a HUP signal to the Subnet Manager process (e.g. pkill -HUP opensm).

⚠️ **Warning**

Performing changes in the ipoib_bc_flags (ipoib/sl/scope/rate/mtu) and mgroup flags of an existing partition requires a restart of the Subnet Manager to take effect.

---

### File Format

#### Note

Line content followed after ' #' character is comment and ignored by parser.

---

#### General File Format

- `<Partition Definition>:\n<newline>\n<Partition Properties>`

- `<Partition Definition>`:

  ```
  [PartitionName]=PKey[,indx0][,ipoib_bc_flags][,defmember=full|limited]
  ```

  where:
<table>
<thead>
<tr>
<th>PartitionName</th>
<th>String, will be used with logging. When omitted empty string will be used.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKey</td>
<td>P_Key value for this partition. Only low 15 bits will be used. When omitted will be auto-generated.</td>
</tr>
<tr>
<td>indx0</td>
<td>Indicates that this pkey should be inserted in block 0 index 0.</td>
</tr>
<tr>
<td>ipoib_bc_flags</td>
<td>Used to indicate/specify IPoIB capability of this partition.</td>
</tr>
<tr>
<td>defmember=full</td>
<td>limited</td>
</tr>
</tbody>
</table>

ipoib_bc_flags are:

<table>
<thead>
<tr>
<th>ipoib</th>
<th>Indicates that this partition may be used for IPoIB, as a result the IPoIB broadcast group will be created with the flags given, if any.</th>
</tr>
</thead>
<tbody>
<tr>
<td>rate=&lt;val&gt;</td>
<td>Specifies rate for this IPoIB MC group (default is 3 (10GBps))</td>
</tr>
<tr>
<td>mtu=&lt;val&gt;</td>
<td>Specifies MTU for this IPoIB MC group (default is 4 (2048))</td>
</tr>
<tr>
<td>sl=&lt;val&gt;</td>
<td>Specifies SL for this IPoIB MC group (default is 0)</td>
</tr>
<tr>
<td>scope=&lt;val&gt;</td>
<td>Specifies scope for this IPoIB MC group (default is 2 (link local))</td>
</tr>
</tbody>
</table>

- **<Partition Properties>:**

  \[<Port list> | <MCast Group>]* | <Port list>

  - **<Port List>:**

  <Port Specifier>[,<Port Specifier>]

    - **<Port Specifier>:**

      <PortGUID>[=full | limited | both]
<table>
<thead>
<tr>
<th>PortGUID</th>
<th>GUID of partition member EndPort. Hexadecimal numbers should start from 0x, decimal numbers are accepted too.</th>
</tr>
</thead>
<tbody>
<tr>
<td>full, limited</td>
<td>Indicates full and/or limited membership for this both port. When omitted (or unrecognized) limited membership is assumed. Both indicate full and limited membership for this port.</td>
</tr>
</tbody>
</table>

- `<MCast Group>`:

```
mgid=gid[,mgroup_flag]*
```

where:

<table>
<thead>
<tr>
<th>mgid=gid</th>
<th>gid specified is verified to be a Multicast address IP groups are verified to match the rate and mtu of the broadcast group. The P_Key bits of the mgid for IP groups are verified to either match the P_Key specified in by &quot;Partition Definition&quot; or if they are 0x0000 the P_Key will be copied into those bits.</th>
</tr>
</thead>
<tbody>
<tr>
<td>mgroup_flag</td>
<td>Specifies rate for this MC group (default is 3 (10GBps))</td>
</tr>
<tr>
<td>rate= &lt;val&gt;</td>
<td>Specifies MTU for this MC group (default is 4 (2048))</td>
</tr>
<tr>
<td>mtu= &lt;val&gt;</td>
<td>Specifies SL for this MC group (default is 0)</td>
</tr>
<tr>
<td>sl=&lt;val&gt;</td>
<td>Specifies scope for this MC group (default is 2 (link local)). Multiple scope settings are permitted for a partition. NOTE: This overwrites the scope nibble of the specified mgid. Furthermore specifying multiple scope settings will result in multiple MC groups being created.</td>
</tr>
<tr>
<td>scope= &lt;val&gt;</td>
<td>Specifies the Q_Key for this MC group (default: 0x0b1b for IP groups, 0 for other groups)</td>
</tr>
<tr>
<td>qkey= &lt;val&gt;</td>
<td>Specifies tclass for this MC group (default is 0)</td>
</tr>
<tr>
<td>tclass= &lt;val&gt;</td>
<td>Specifies FlowLabel for this MC group (default is 0)</td>
</tr>
<tr>
<td>FlowLabel =&lt;val&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Note that values for rate, MTU, and scope should be specified as defined in the IBTA specification (for example, mtu=4 for 2048). To use 4K MTU, edit that entry to "mtu=5" (5 indicates 4K MTU to that specific partition).

PortGUIDs list:
There are some useful keywords for PortGUID definition:

- 'ALL_CAS' means all Channel Adapter end ports in this subnet
- 'ALL_VCAS' means all virtual end ports in the subnet
- 'ALL_SWITCHES' means all Switch end ports in this subnet
- 'ALL ROUTERS' means all Router end ports in this subnet
- 'SELF' means subnet manager's port. An empty list means that there are no ports in this partition

Notes:

- White space is permitted between delimiters ('=', ',', ':', ';').
- PartitionName does not need to be unique, PKey does need to be unique. If PKey is repeated then those partition configurations will be merged and the first PartitionName will be used (see the next note).
- It is possible to split partition configuration in more than one definition, but then PKey should be explicitly specified (otherwise different PKey values will be generated for those definitions).

Examples:

```
Default=0x7fff : ALL, SELF=full ;
Default=0x7fff : ALL, ALL_SWITCHES=full, SELF=full ;

NewPartition , ipoib : 0x123456=full, 0x3456789034=limi, 0x2134af2306 ;

YetAnotherOne = 0x300 : SELF=full ;
YetAnotherOne = 0x300 : ALL=limited ;
```
The following rule is equivalent to how OpenSM used to run prior to the partition manager:

```
Default=0x7fff,ipoib:
mgid=ff12:401b::0707,sl=1 # random IPv4 group
mgid=ff12:601b::16 # MLDv2-capable routers
mgid=ff12:401b::16 # IGMP
mgid=ff12:601b::2 # All routers
mgid=ff12::1,sl=1,Q_Key=0xDEADBEEF,rate=3,mtu=2 # random group
ALL=full;
```

**Effect of Topology Changes**

If a link is added or removed, OpenSM may not recalculate the routes that do not have to change. A route has to change if the port is no longer UP or no longer the MinHop. When routing changes are performed, the same algorithm for balancing the routes is invoked.

In the case of using the file-based routing, any topology changes are currently ignored. The 'file' routing engine just loads the LFTs from the file specified, with no reaction to real topology. Obviously, this will not be able to recheck LIDs (by GUID) for disconnected nodes, and LFTs for non-existent switches will be skipped. Multicast is not affected by 'file' routing engine (this uses min hop tables).

**Routing Algorithms**
OpenSM offers the following routing engines:

1. **Min Hop Algorithm**

   Based on the minimum hops to each node where the path length is optimized.

2. **UPDN Algorithm**

   Based on the minimum hops to each node, but it is constrained to ranking rules. This algorithm should be chosen if the subnet is not a pure Fat Tree, and a deadlock may occur due to a loop in the subnet.

3. **Fat-tree Routing Algorithm**

   This algorithm optimizes routing for a congestion-free "shift" communication pattern. It should be chosen if a subnet is a symmetrical Fat Tree of various types, not just a K-ary-N-Tree: non-constant K, not fully staffed, and for any CBB ratio. Similar to UPDN, Fat Tree routing is constrained to ranking rules.

4. **DOR Routing Algorithm**

   Based on the Min Hop algorithm, but avoids port equalization except for redundant links between the same two switches. This provides deadlock free routes for hypercubes when the fabric is cabled as a hypercube and for meshes when cabled as a mesh.

5. **Torus-2QoS Routing Algorithm**

   Based on the DOR Unicast routing algorithm specialized for 2D/3D torus topologies. Torus-2QoS provides deadlock-free routing while supporting two quality of service (QoS) levels. Additionally, it can route around multiple failed fabric links or a single failed fabric switch without introducing deadlocks, and without changing path SL values granted before the failure.

6. **Routing Chains**

   Allows routing configuration of different parts of a single InfiniBand subnet by different routing engines. In the current release, minhop/updn/ftree/dor/torus-2QoS/pqft can be combined.
MINHOP/UPDN/DOR routing algorithms are comprised of two stages:

1. MinHop matrix calculation. How many hops are required to get from each port to each LID. The algorithm to fill these tables is different if you run standard (min hop) or Up/Down. For standard routing, a "relaxation" algorithm is used to propagate min hop from every destination LID through neighbor switches. For Up/Down routing, a BFS from every target is used. The BFS tracks link direction (up or down) and avoid steps that will perform up after a down step was used.

2. Once MinHop matrices exist, each switch is visited and for each target LID a decision is made as to what port should be used to get to that LID. This step is common to standard and Up/Down routing. Each port has a counter counting the number of target LIDs going through it. When there are multiple alternative ports with same MinHop to a LID, the one with less previously assigned ports is selected.

If LMC > 0, more checks are added. Within each group of LIDs assigned to same target port:

1. Use only ports which have same MinHop

2. First prefer the ones that go to different systemImageGuid (then the previous LID of the same LMC group)

3. If none, prefer those which go through another NodeGuid

4. Fall back to the number of paths method (if all go to same node).

**Min Hop Algorithm**

The Min Hop algorithm is invoked by default if no routing algorithm is specified. It can also be invoked by specifying '-R minhop'.

The Min Hop algorithm is divided into two stages: computation of min-hop tables on every switch and LFT output port assignment. Link subscription is also equalized with the

**Note**

Please note that LASH Routing Algorithm is not supported.
ability to override based on port GUID. The latter is supplied by:

```
-i <equalize-ignore-guids-file>
-ignore-guids <equalize-ignore-guids-file>
```

This option provides the means to define a set of ports (by GUIDs) that will be ignored by the link load equalization algorithm.

LMC awareness routes based on a (remote) system or on a switch basis.

## UPDN Algorithm

The UPDN algorithm is designed to prevent deadlocks from occurring in loops of the subnet. A loop-deadlock is a situation in which it is no longer possible to send data between any two hosts connected through the loop. As such, the UPDN routing algorithm should be sent if the subnet is not a pure Fat Tree, and one of its loops may experience a deadlock (due, for example, to high pressure).

The UPDN algorithm is based on the following main stages:

1. Auto-detect root nodes - based on the CA hop length from any switch in the subnet, a statistical histogram is built for each switch (hop num vs the number of occurrences). If the histogram reflects a specific column (higher than others) for a certain node, then it is marked as a root node. Since the algorithm is statistical, it may not find any root nodes. The list of the root nodes found by this auto-detect stage is used by the ranking process stage.

### Note

The user can override the node list manually.
2. Ranking process - All root switch nodes (found in stage 1) are assigned a rank of 0. Using the BFS algorithm, the rest of the switch nodes in the subnet are ranked incrementally. This ranking aids in the process of enforcing rules that ensure loop-free paths.

3. Min Hop Table setting - after ranking is done, a BFS algorithm is run from each (CA or switch) node in the subnet. During the BFS process, the FDB table of each switch node traversed by BFS is updated, in reference to the starting node, based on the ranking rules and GUID values.

At the end of the process, the updated FDB tables ensure loop-free paths through the subnet.

**UPDN Algorithm Usage**

Activation through OpenSM:

- Use `-R updn` option (instead of old `-u`) to activate the UPDN algorithm.

- Use `-a <root_guid_file>` for adding an UPDN GUID file that contains the root nodes for ranking. If the `-a` option is not used, OpenSM uses its auto-detect root nodes algorithm.

Notes on the GUID list file:

- A valid GUID file specifies one GUID in each line. Lines with an invalid format will be discarded

- The user should specify the root switch GUIDs

**Fat-tree Routing Algorithm**
The fat-tree algorithm optimizes routing for "shift" communication pattern. It should be chosen if a subnet is a symmetrical or almost symmetrical fat-tree of various types. It supports not just K-ary-N-Trees, by handling for non-constant K, cases where not all leafs (CAs) are present, any Constant Bisectional Ratio (CBB) ratio. As in UPDN, fat-tree also prevents credit-loop-dead-locks.

If the root GUID file is not provided ('a' or '-root_guid_file' options), the topology has to be pure fat-tree that complies with the following rules:

- Tree rank should be between two and eight (inclusively)
- Switches of the same rank should have the same number of UP-going port groups, unless they are root switches, in which case they shouldn't have UP-going ports at all.
  
  Note: Ports that are connected to the same remote switch are referenced as 'port group'.
- Switches of the same rank should have the same number of DOWN-going port groups, unless they are leaf switches.
- Switches of the same rank should have the same number of ports in each UP-going port group.
- Switches of the same rank should have the same number of ports in each DOWN-going port group.
- All the CAs have to be at the same tree level (rank).

If the root GUID file is provided, the topology does not have to be pure fat-tree, and it should only comply with the following rules:

- Tree rank should be between two and eight (inclusively)
- All the Compute Nodes have to be at the same tree level (rank). Note that non-compute node CAs are allowed here to be at different tree ranks.
  
  Note: List of compute nodes (CNs) can be specified using ‘-u’ or ‘--cn_guid_file’ OpenSM options.

Topologies that do not comply cause a fallback to min-hop routing. Note that this can also occur on link failures which cause the topology to no longer be a "pure" fat-tree.
Note that although fat-tree algorithm supports trees with non-integer CBB ratio, the routing will not be as balanced as in case of integer CBB ratio. In addition to this, although the algorithm allows leaf switches to have any number of CAs, the closer the tree is to be fully populated, the more effective the "shift" communication pattern will be. In general, even if the root list is provided, the closer the topology to a pure and symmetrical fat-tree, the more optimal the routing will be.

The algorithm also dumps the compute node ordering file (opensm-ftree-ca-order.dump) in the same directory where the OpenSM log resides. This ordering file provides the CN order that may be used to create efficient communication pattern, that will match the routing tables.

Routing between non-CN Nodes

The use of the io_guid_file option allows non-CN nodes to be located on different levels in the fat tree. In such case, it is not guaranteed that the Fat Tree algorithm will route between two non-CN nodes. In the scheme below, N1, N2, and N3 are non-CN nodes. Although all the CN have routes to and from them, there will not necessarily be a route between N1, N2 and N3. Such routes would require to use at least one of the switches the wrong way around.

To solve this problem, a list of non-CN nodes can be specified by `--io_guid_file` option. These nodes will be allowed to use switches the wrong way around a specific number of times (specified by `--max_reverse_hops`). With the proper `max_reverse_hops` and `io_guid_file` values, you can ensure full connectivity in the Fat Tree. In the scheme above, with a `max_reverse_hops` of 1, routes will be instantiated between N1<->N2 and N2<->N3. With a `max_reverse_hops` value of 2, N1, N2 and N3 will all have routes between them.
Using max_reverse_hops creates routes that use the switch in a counter-stream way. This option should never be used to connect nodes with high bandwidth traffic between them! It should only be used to allow connectivity for HA purposes or similar. Also having routes the other way around can cause credit loops.

**Activation through OpenSM**

Use `-R ftree` option to activate the fat-tree algorithm.

---

**Note**

LMC > 0 is not supported by fat-tree routing. If this is specified, the default routing algorithm is invoked instead.

---

**DOR Routing Algorithm**

The Dimension Order Routing algorithm is based on the Min Hop algorithm and so uses shortest paths. Instead of spreading traffic out across different paths with the same shortest distance, it chooses among the available shortest paths based on an ordering of dimensions. Each port must be consistently cabled to represent a hypercube dimension or a mesh dimension. Paths are grown from a destination back to a source using the lowest dimension (port) of available paths at each step. This provides the ordering necessary to avoid deadlock. When there are multiple links between any two switches, they still represent only one dimension and traffic is balanced across them unless port equalization is turned off. In the case of hypercubes, the same port must be used throughout the fabric to represent the hypercube dimension and match on both ends of the cable. In the case of meshes, the dimension should consistently use the same pair of ports, one port on one end of the cable, and the other port on the other end, continuing along the mesh dimension.

Use `-R dor` option to activate the DOR algorithm.
Torus-2QoS Routing Algorithm

Torus-2QoS is a routing algorithm designed for large-scale 2D/3D torus fabrics. The torus-2QoS routing engine can provide the following functionality on a 2D/3D torus:

- Free of credit loops routing
- Two levels of QoS, assuming switches support 8 data VLs
- Ability to route around a single failed switch, and/or multiple failed links, without:
  - introducing credit loops
  - changing path SL values
- Very short run times, with good scaling properties as fabric size increases

Unicast Routing

Torus-2 QoS is a DOR-based algorithm that avoids deadlocks that would otherwise occur in a torus using the concept of a dateline for each torus dimension. It encodes into a path SL which datelines the path crosses as follows:

```c
sl = 0;
for (d = 0; d < torus_dimensions; d++)
/* path_crosses_dateline(d) returns 0 or 1 */
sl |= path_crosses_dateline(d) << d;
```

For a 3D torus, that leaves one SL bit free, which torus-2 QoS uses to implement two QoS levels. Torus-2 QoS also makes use of the output port dependence of switch SL2VL maps to encode into one VL bit the information encoded in three SL bits. It computes in which torus coordinate direction each inter-switch link "points", and writes SL2VL maps for such ports as follows:

```c
for (sl = 0; sl < 16; sl++)
/* cdir(port) reports which torus coordinate direction a switch port * "points" in, and returns 0, 1, or 2 */
```
Thus, on a pristine 3D torus, i.e., in the absence of failed fabric switches, torus-2 QoS consumes 8 SL values (SL bits 0-2) and 2 VL values (VL bit 0) per QoS level to provide deadlock-free routing on a 3D torus. Torus-2 QoS routes around link failure by "taking the long way around" any 1D ring interrupted by a link failure. For example, consider the 2D 6x5 torus below, where switches are denoted by [+a-zA-Z]:

For a pristine fabric the path from S to D would be S-n-T-r-D. In the event that either link S-n or n-T has failed, torus-2QoS would use the path S-m-p-o-T-r-D.

Note that it can do this without changing the path SL value; once the 1D ring m-S-n-T-o-p-m has been broken by failure, path segments using it cannot contribute to deadlock, and the x-direction dateline (between, say, x=5 and x=0) can be ignored for path segments on that ring. One result of this is that torus-2QoS can route around many simultaneous link failures, as long as no 1D ring is broken into disjoint segments. For example, if links n-T and T-o have both failed, that ring has been broken into two disjoint segments, T and o-p-m-S-n. Torus-2QoS checks for such issues, reports if they are found, and refuses to route such fabrics.

Note that in the case where there are multiple parallel links between a pair of switches, torus-2QoS will allocate routes across such links in a round-robin fashion, based on ports at the path destination switch that are active and not used for inter-switch links. Should a link that is one of several such parallel links fail, routes are redistributed across the remaining links. When the last of such a set of parallel links fails, traffic is rerouted as described above.
Handling a failed switch under DOR requires introducing into a path at least one turn that would be otherwise "illegal", i.e. not allowed by DOR rules. Torus-2QoS will introduce such a turn as close as possible to the failed switch in order to route around it. In the above example, suppose switch T has failed, and consider the path from S to D. Torus-2QoS will produce the path S-n-l-r-D, rather than the S-n-T-r-D path for a pristine torus, by introducing an early turn at n. Normal DOR rules will cause traffic arriving at switch I to be forwarded to switch r; for traffic arriving from I due to the "early" turn at n, this will generate an "illegal" turn at I.

Torus-2QoS will also use the input port dependence of SL2VL maps to set VL bit 1 (which would be otherwise unused) for y-x, z-x, and z-y turns, i.e., those turns that are illegal under DOR. This causes the first hop after any such turn to use a separate set of VL values, and prevents deadlock in the presence of a single failed switch. For any given path, only the hops after a turn that is illegal under DOR can contribute to a credit loop that leads to deadlock. So in the example above with failed switch T, the location of the illegal turn at I in the path from S to D requires that any credit loop caused by that turn must encircle the failed switch at T. Thus the second and later hops after the illegal turn at I (i.e., hop I-r-D) cannot contribute to a credit loop because they cannot be used to construct a loop encircling T. The hop I-r uses a separate VL, so it cannot contribute to a credit loop encircling T. Extending this argument shows that in addition to being capable of routing around a single switch failure without introducing deadlock, torus-2QoS can also route around multiple failed switches on the condition they are adjacent in the last dimension routed by DOR. For example, consider the following case on a 6x6 2D torus:

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>-D-</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-I-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Suppose switches T and R have failed, and consider the path from S to D. Torus-2QoS will generate the path S-n-q-l-u-D, with an illegal turn at switch I, and with hop I-u using a VL with bit 1 set. As a further example, consider a case that torus-2QoS cannot route without...
deadlock: two failed switches adjacent in a dimension that is not the last dimension routed by DOR; here the failed switches are O and T:

In a pristine fabric, torus-2QoS would generate the path from S to D as S-n-O-T-r-D. With failed switches O and T, torus-2QoS will generate the path S-n-I-q-r-D, with an illegal turn at switch I, and with hop I-q using a VL with bit 1 set. In contrast to the earlier examples, the second hop after the illegal turn, q-r, can be used to construct a credit loop encircling the failed switches.

**Multicast Routing**

Since torus-2QoS uses all four available SL bits, and the three data VL bits that are typically available in current switches, there is no way to use SL/VL values to separate multicast traffic from unicast traffic. Thus, torus-2QoS must generate multicast routing such that credit loops cannot arise from a combination of multicast and unicast path segments. It turns out that it is possible to construct spanning trees for multicast routing that have that property. For the 2D 6x5 torus example above, here is the full-fabric spanning tree that torus-2QoS will construct, where "x" is the root switch and each "+" is a non-root switch:
For multicast traffic routed from root to tip, every turn in the above spanning tree is a legal DOR turn. For traffic routed from tip to root, and some traffic routed through the root, turns are not legal DOR turns. However, to construct a credit loop, the union of multicast routing on this spanning tree with DOR unicast routing can only provide 3 of the 4 turns needed for the loop. In addition, if none of the above spanning tree branches crosses a dateline used for unicast credit loop avoidance on a torus, and if multicast traffic is confined to SL 0 or SL 8 (recall that torus-2QoS uses SL bit 3 to differentiate QoS level), then multicast traffic also cannot contribute to the "ring" credit loops that are otherwise possible in a torus. Torus-2QoS uses these ideas to create a master spanning tree. Every multicast group spanning tree will be constructed as a subset of the master tree, with the same root as the master tree. Such multicast group spanning trees will in general not be optimal for groups which are a subset of the full fabric. However, this compromise must be made to enable support for two QoS levels on a torus while preventing credit loops. In the presence of link or switch failures that result in a fabric for which torus-2QoS can generate credit-loop-free unicast routes, it is also possible to generate a master spanning tree for multicast that retains the required properties. For example, consider that same 2D 6x5 torus, with the link from (2,2) to (3,2) failed. Torus-2QoS will generate the following master spanning tree:
Two things are notable about this master spanning tree. First, assuming the x dateline was between x=5 and x=0, this spanning tree has a branch that crosses the dateline. However, just as for unicast, crossing a dateline on a 1D ring (here, the ring for y=2) that is broken by a failure cannot contribute to a torus credit loop. Second, this spanning tree is no longer optimal even for multicast groups that encompass the entire fabric. That, unfortunately, is a compromise that must be made to retain the other desirable properties of torus-2QoS routing. In the event that a single switch fails, torus-2QoS will generate a master spanning tree that has no "extra" turns by appropriately selecting a root switch. In the 2D 6x5 torus example, assume now that the switch at (3,2) (i.e., the root for a pristine fabric), fails. Torus-2QoS will generate the following master spanning tree for that case:

```
4 + + + + + +
|   |   |   |   |   |
3 + + + + + +
|   |   |   |   |   |
2 + + + + + +
|   |   |   |   |   |
1 + + + + + +
|   |   |   |   |   |
```

Assuming the dateline was between y=4 and y=0, this spanning tree has a branch that crosses a dateline. However, this cannot contribute to credit loops as it occurs on a 1D ring (the ring for x=3) that is broken by failure, as in the above example.

**Torus Topology Discovery**

The algorithm used by torus-2QoS to construct the torus topology from the undirected graph representing the fabric requires that the radix of each dimension be configured via torus-2QoS.conf. It also requires that the torus topology be "seeded"; for a 3D torus this requires configuring four switches that define the three coordinate directions of the torus. Given this starting information, the algorithm is to examine the cube formed by the eight switch locations bounded by the corners (x,y,z) and (x+1,y+1,z+1). Based on switches already placed into the torus topology at some of these locations, the algorithm examines 4-loops of inter-switch links to find the one that is consistent with a face of the
cube of switch locations and adds its switches to the discovered topology in the correct locations.

Because the algorithm is based on examining the topology of 4-loops of links, a torus with one or more radix-4 dimensions requires extra initial seed configuration. See torus-2QoS.conf(5) for details. Torus-2QoS will detect and report when it has an insufficient configuration for a torus with radix-4 dimensions.

In the event the torus is significantly degraded, i.e., there are many missing switches or links, it may happen that torus-2QoS is unable to place into the torus some switches and/or links that were discovered in the fabric, and will generate a warning in that case. A similar condition occurs if torus-2QoS is misconfigured, i.e., the radix of a torus dimension as configured does not match the radix of that torus dimension as wired, and many switches/links in the fabric will not be placed into the torus.

**Quality Of Service Configuration**

OpenSM will not program switches and channel adapters with SL2VL maps or VL arbitration configuration unless it is invoked with -Q. Since torus-2QoS depends on such functionality for correct operation, always invoke OpenSM with -Q when torus-2QoS is in the list of routing engines. Any quality of service configuration method supported by OpenSM will work with torus-2QoS, subject to the following limitations and considerations. For all routing engines supported by OpenSM except torus-2QoS, there is a one-to-one correspondence between QoS level and SL. Torus-2QoS can only support two quality of service levels, so only the high-order bit of any SL value used for unicast QoS configuration will be honored by torus-2QoS. For multicast QoS configuration, only SL values 0 and 8 should be used with torus-2QoS.

Since SL to VL map configuration must be under the complete control of torus-2QoS, any configuration via qos_sl2vl, qos_swe_sl2vl, etc., must and will be ignored, and a warning will be generated. Torus-2QoS uses VL values 0-3 to implement one of its supported QoS levels, and VL values 4-7 to implement the other. Hard-to-diagnose application issues may arise if traffic is not delivered fairly across each of these two VL ranges. Torus-2QoS will detect and warn if VL arbitration is configured unfairly across VLS in the range 0-3, and also in the range 4-7. Note that the default OpenSM VL arbitration configuration does not meet this constraint, so all torus-2QoS users should configure VL arbitration via qos_vlarb_high, qos_vlarb_low, etc.

**Operational Considerations**
Any routing algorithm for a torus IB fabric must employ path SL values to avoid credit loops. As a result, all applications run over such fabrics must perform a path record query to obtain the correct path SL for connection setup. Applications that use rdma_cm for connection setup will automatically meet this requirement.

If a change in fabric topology causes changes in path SL values required to route without credit loops, in general, all applications would need to repath to avoid message deadlock. Since torus-2QoS has the ability to reroute after a single switch failure without changing path SL values, repathing by running applications is not required when the fabric is routed with torus-2QoS.

Torus-2QoS can provide unchanging path SL values in the presence of subnet manager failover provided that all OpenSM instances have the same idea of dateline location. See torus-2QoS.conf(5) for details. Torus-2QoS will detect configurations of failed switches and links that prevent routing that is free of credit loops and will log warnings and refuse to route. If "no_fall-back" was configured in the list of OpenSM routing engines, then no other routing engine will attempt to route the fabric. In that case, all paths that do not transit the failed components will continue to work, and the subset of paths that are still operational will continue to remain free of credit loops. OpenSM will continue to attempt to route the fabric after every sweep interval and after any change (such as a link up) in the fabric topology. When the fabric components are repaired, full functionality will be restored. In the event OpenSM was configured to allow some other engine to route the fabric if torus-2QoS fails, then credit loops and message deadlock are likely if torus-2QoS had previously routed the fabric successfully. Even if the other engine is capable of routing a torus without credit loops, applications that built connections with path SL values granted under torus-2QoS will likely experience message deadlock under routing generated by a different engine, unless they repath. To verify that a torus fabric is routed free of credit loops, use ibdmchk to analyze data collected via ibdiagnet -vlr.

**Torus-2QoS Configuration File Syntax**

The file torus-2QoS.conf contains configuration information that is specific to the OpenSM routing engine torus-2QoS. Blank lines and lines where the first non-whitespace character is "#" are ignored. A token is any contiguous group of non-whitespace characters. Any tokens on a line following the recognized configuration tokens described below are ignored.
Either torus or mesh must be the first keyword in the configuration and sets the topology that torus-2QoS will try to construct. A 2D topology can be configured by specifying one of \texttt{x_radix}, \texttt{y_radix}, or \texttt{z_radix} as 1. An individual dimension can be configured as mesh (open) or torus (looped) by suffixing its radix specification with one of \texttt{m}, \texttt{M}, \texttt{t}, or \texttt{T}. Thus, “mesh 3T 4 5” and “torus 3 4M 5M” both specify the same topology.

Note that although torus-2QoS can route mesh fabrics, its ability to route around failed components is severely compromised on such fabrics. A failed fabric components very likely to cause a disjoint ring; see UNICAST ROUTING in torus-2QoS(8).

These keywords are used to seed the torus/mesh topology. For example, "xp_link 0x2000 0x2001" specifies that a link from the switch with node GUID 0x2000 to the switch with node GUID 0x2001 would point in the positive \texttt{x} direction, while "xm_link 0x2000 0x2001" specifies that a link from the switch with node GUID 0x2000 to the switch with node GUID 0x2001 would point in the negative \texttt{x} direction. All the link keywords for a given seed must specify the same “from” switch.

In general, it is not necessary to configure both the positive and negative directions for a given coordinate; either is sufficient. However, the algorithm used for topology discovery needs extra information for torus dimensions of radix four (see TOPOLOGY DISCOVERY in torus-2QoS(8)). For such cases, both the positive and negative coordinate directions must be specified.

Based on the topology specified via the torus/mesh keyword, torus-2QoS will detect and log when it has insufficient seed configuration.
In order for torus-2QoS to provide the guarantee that path SL values do not change under any conditions for which it can still route the fabric, its idea of dateline position must not change relative to physical switch locations. The dateline keywords provide the means to configure such behavior.

The dateline for a torus dimension is always between the switch with coordinate 0 and the switch with coordinate radix-1 for that dimension. By default, the common switch in a torus seed is taken as the origin of the coordinate system used to describe switch location. The position parameter for a dateline keyword moves the origin (and hence the dateline) the specified amount relative to the common switch in a torus seed.

If any of the switches used to specify a seed were to fail torus-2QoS would be unable to complete topology discovery successfully. The next_seed keyword specifies that the following link and dateline keywords apply to a new seed specification.

For maximum resiliency, no seed specification should share a switch with any other seed specification. Multiple seed specifications should use dateline configuration to ensure that torus-2QoS can grant path SL values that are constant, regardless of which seed was used to initiate topology discovery.

portgroup_max_ports max_ports - This keyword specifies the maximum number of parallel inter-switch links, and also the maximum number of host ports per switch, that torus-2QoS can accommodate. The default value is 16. Torus-2QoS will log an error message during topology discovery if this parameter needs to be increased. If this keyword appears multiple times, the last instance prevails.

port_order p1 p2 p3 ... - This keyword specifies the order in which CA ports on a destination switch are visited when computing routes. When the fabric contains switches connected with multiple parallel links, routes are distributed in a round-robin fashion across such links, and so changing the order that CA ports are visited changes the distribution of routes across such links. This may be advantageous for some specific traffic patterns.
The default is to visit CA ports in increasing port order on destination switches. Duplicate values in the list will be ignored.

Example:

```bash
# Look for a 2D (since x radix is one) 4x5 torus.
torus 1 4 5
# y is radix-4 torus dimension, need both
# yp_link and yp_link configuration.
yp_link 0x200000 0x200005 # sw @ y=0,z=0 -> sw @ y=1,z=0
ym_link 0x200000 0x20000f # sw @ y=0,z=0 -> sw @ y=3,z=0
# z is not radix-4 torus dimension, only need one of
# zm_link or zp_link configuration.
zp_link 0x200000 0x200001 # sw @ y=0,z=0 -> sw @ y=0,z=1
next_seed
yp_link 0x20000b 0x200010 # sw @ y=2,z=1 -> sw @ y=3,z=1
ym_link 0x20000b 0x200006 # sw @ y=2,z=1 -> sw @ y=1,z=1
zp_link 0x20000b 0x20000c # sw @ y=2,z=1 -> sw @ y=2,z=2
y_dateline -2 # Move the dateline for this seed
z_dateline -1 # back to its original position.
# If OpenSM failover is configured, for maximum resiliency
# one instance should run on a host attached to a switch
# from the first seed, and another instance should run
# on a host attached to a switch from the second seed.
# Both instances should use this torus-2QoS.conf to ensure
# path SL values do not change in the event of SM failover.
# port_order defines the order on which the ports would be
# chosen for routing.
port_order 7 10 8 11 9 12 25 28 26 29 27 30
```

### Routing Chains

The routing chains feature is offering a solution that enables one to configure different parts of the fabric and define a different routing engine to route each of them. The routings are done in a sequence (hence the name "chains") and any node in the fabric that is configured in more than one part is left with the routing updated by the last routing engine it was a part of.
Configuring Routing Chains

To configure routing chains:

1. Define the port groups.
2. Define topologies based on previously defined port groups.
3. Define configuration files for each routing engine.
4. Define routing engine chains over previously defined topologies and configuration files.

Defining Port Groups

The basic idea behind the port groups is the ability to divide the fabric into sub-groups and give each group an identifier that can be used to relate to all nodes in this group. The port groups is a separate feature from the routing chains but is a mandatory prerequisite for it. In addition, it is used to define the participants in each of the routing algorithms.

Defining a Port Group Policy File

In order to define a port group policy file, set the parameter 'pgrp_policy_file' in the OpenSM configuration file.

pgrp_policy_file /etc/opensm/conf/port_groups_policy_file

Configuring a Port Group Policy

The port groups policy file details the port groups in the fabric. The policy file should be composed of one or more paragraphs that define a group. Each paragraph should begin with the line 'port-group' and end with the line 'end-port-group'.

For example:
Port Group Qualifiers

Note

Unlike the port group's beginning and end which do not require a colon, all qualifiers must end with a colon (':'). Also - a colon is a predefined mark that must not be used inside qualifier values. The inclusion of a colon in the name or the use of a port group will result in the policy's failure.

Rule Qualifier

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Each group must have a name. Without a name qualifier, the policy fails.</td>
<td>name: grp1</td>
</tr>
<tr>
<td>use</td>
<td>'use' is an optional qualifier that one can define in order to describe the usage of this port group (if undefined, an empty string is used as a default).</td>
<td>use: first port group</td>
</tr>
</tbody>
</table>

There are several qualifiers used to describe a rule that determines which ports will be added to the group. Each port group may include one or more rules out of the rules described in the below table (at least one rule must be defined for each port group).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>guid list</td>
<td>Comma separated list of GUIDs to include in the group.</td>
<td>port-guid: 0x283, 0x286, 0x289</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>If no specific physical ports were configured, all physical ports of the guid are chosen. However, for each guid, one can detail specific physical ports to be included in the group. This can be done using the following syntax:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Specify a specific port in a guid to be chosen port-guid: 0x283@3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Specify a specific list of ports in a guid to be chosen port-guid: 0x286@1/5/7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Specify a specific range of ports in a guid to be chosen port-guid: 0x289@2-5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Specify a list of specific ports and ports ranges in a guid to be chosen port-guid: 0x289@2-5/7/9-13/18</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Complex rule                                                                                                                  port-guid: 0x283@5-8/12/14, 0x286, 0x289/6/8/12</td>
<td></td>
</tr>
<tr>
<td>port-guid-range</td>
<td>It is possible to configure a range of guids to be chosen to the group. However, while using the range qualifier, it is impossible to detail specific physical ports. Note: A list of ranges cannot be specified. The below example is invalid and will cause the policy to fail: port-guid-range: 0x283-0x289, 0x290-0x295</td>
<td>port-guid-range: 0x283-0x289</td>
</tr>
<tr>
<td>port-name</td>
<td>One can configure a list of hostnames as a rule. Hosts with a node description that is built out of these hostnames will be chosen. Since the node description contains the network card index as well, one might also specify a network card index and a physical port to be chosen. For example, the given configuration will cause only physical port 2 of a host with the node description 'kuku HCA-1' to be chosen. port and hca_idx parameters are optional. If the port is unspecified, all physical ports are chosen. If hca_idx is unspecified, all card numbers are chosen. Specifying a hostname is mandatory. One can configure a list of hostname/ port/hca_idx sets in the same qualifier as follows:</td>
<td>port-name: hostname=kuku; port=2; hca_idx=1</td>
</tr>
</tbody>
</table>
## Predefined Port Groups

There are 3 predefined, automatically created port groups that are available for use, yet cannot be defined in the policy file (if a group in the policy is configured with the name of one of these predefined groups, the policy fails) -

- **ALL** - a group that includes all nodes in the fabric
- **ALL_SWITCHES** - a group that includes all switches in the fabric

- **ALL_CAS** - a group that includes all HCAs in the fabric

- **ALL_ROUTERS** - a group that includes all routers in the fabric (supported in OpenSM starting from v4.9.0)

### Port Groups Policy Examples

```plaintext
port-group
name: grp3
use: Subtract of groups grp1 and grp2
subtract-rule: grp1, grp2
end-port-group

port-group
name: grp1
port-guid: 0x281, 0x282, 0x283
end-port-group

port-group
name: grp2
port-guid-range: 0x282-0x286
port-name: hostname=server1 port=1
end-port-group

port-group
name: grp4
port-name: hostname=kika port=1 hca_idx=1
end-port-group

port-group
name: grp3
union-rule: grp3, grp4
end-port-group
```

### Defining a Topologies Policy File
In order to define a topology policy file, set the parameter 'topo_policy_file' in the OpenSM configuration file.

```bash
topo_policy_file /etc/opensm/conf/topo_policy_file.cfg
```

**Configuring a Topology Policy**

The topologies policy file details a list of topologies. The policy file should be composed of one or more paragraphs which define a topology. Each paragraph should begin with the line 'topology' and end with the line 'end-topology'.

For example:

```plaintext
topology
...topology qualifiers...
end-topology
```

**Topology Qualifiers**

**Note**

Unlike topology and end-topology which do not require a colon, all qualifiers must end with a colon (':'). Also - a colon is a predefined mark that must not be used inside qualifier values. An inclusion of a column in the qualifier values will result in the policy's failure.

All topology qualifiers are mandatory. Absence of any of the below qualifiers will cause the policy parsing to fail.

Topology Qualifiers
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Topology ID. Legal Values – any positive value. Must be unique.</td>
<td>id: 1</td>
</tr>
<tr>
<td>sw-grp</td>
<td>Name of the port group that includes all switches and switch ports to be used in this topology.</td>
<td>sw-grp: ys_switches</td>
</tr>
<tr>
<td>hca-grp</td>
<td>Name of the port group that includes all HCA's to be used in this topology.</td>
<td>hca-grp: ys_hosts</td>
</tr>
</tbody>
</table>

**Configuration File per Routing Engine**

Each engine in the routing chain can be provided by its own configuration file. Routing engine configuration file is the fraction of parameters defined in the main OpenSM configuration file.

Some rules should be applied when defining a particular configuration file for a routing engine:

- Parameters that are not specified in specific routing engine configuration file are inherited from the main OpenSM configuration file.

- The following configuration parameters are taking effect only in the main OpenSM configuration file:
  - qos and qos_* settings like (vl_arb, sl2vl, etc.)
  - lmc
  - routing_engine

**Defining a Routing Chain Policy File**

In order to define a port group policy file, set the parameter ‘rch_policy_file’ in the OpenSM configuration file.
**First Routing Engine in the Chain**

The first unicast engine in a routing chain must include all switches and HCAs in the fabric (topology id must be 0). The path-bit parameter value is path-bit 0 and it cannot be changed.

**Configuring a Routing Chains Policy**

The routing chains policy file details the routing engines (and their fallback engines) used for the fabric's routing. The policy file should be composed of one or more paragraphs which defines an engine (or a fallback engine). Each paragraph should begin with the line 'unicast-step' and end with the line 'end-unicast-step'.

For example:

```
unicast-step
...routing engine qualifiers...
end-unicast-step
```

**Routing Engine Qualifiers**

**Note**

Unlike unicast-step and end-unicast-step which do not require a colon, all qualifiers must end with a colon (':'). Also - a colon is a predefined mark that must not be used inside qualifier values. An
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
| id        | 'id' is mandatory. Without an ID qualifier for each engine, the policy fails.  
  - Legal values – size_t value (0 is illegal).  
  - The engines in the policy chain are set according to an ascending id order, so it is highly crucial to verify that the id that is given to the engines match the order in which you would like the engines to be set. | is: 1 |
| engine    | This is a mandatory qualifier that describes the routing algorithm used within this unicast step. Currently, on the first phase of routing chains, legal values are minhop/ftree/updn. | engine: minhop |
| use       | This is an optional qualifier that enables one to describe the usage of this unicast step. If undefined, an empty string is used as a default. | use: ftree routing for yellow stone nodes |
| config    | This is an optional qualifier that enables one to define a separate OpenSM config file for a specific unicast step. If undefined, all parameters are taken from main OpenSM configuration file. | config: /etc/config/opensm2.cfg |
| topology  | Define the topology that this engine uses.  
  - Legal value – id of an existing topology that is defined in topologies policy (or zero that represents the entire fabric and not a specific topology).  
  - Default value – If unspecified, a routing engine will relate to the entire fabric (as if topology zero was defined).  
  - Notice: The first routing engine (the engine with the lowest id) MUST be configured with topology: 0 (entire fabric) or else, the routing chain parser will fail. | topology: 1 |
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
| fallback-to| This is an optional qualifier that enables one to define the current unicast step as a fallback to another unicast step. This can be done by defining the id of the unicast step that this step is a fallback to.  
  - If undefined, the current unicast step is not a fallback.  
  - If the value of this qualifier is a non-existent engine id, this step will be ignored.  
  - A fallback step is meaningless if the step it is a fallback to did not fail.  
  - It is impossible to define a fallback to a fallback step (such definition will be ignored) |         |
| path-bit   | This is an optional qualifier that enables one to define a specific lid offset to be used by the current unicast step. Setting lmc > 0 in main OpenSM configuration file is a prerequisite for assigning specific path-bit for the routing engine. Default value is 0 (if path-bit is not specified) | Path-bit: 1 |

**Dump Files per Routing Engine**

Each routing engine on the chain will dump its own data files if the appropriate log_flags is set (for instance 0x43).

The files that are dumped by each engine are:

- opensm-lid-matrix.dump
- opensm-lfts.dump
- opensm.fdb
- opensm-subnet.lst

These files should contain the relevant data for each engine topology.
- Each engine concatenates its ID and routing algorithm name in its dump files names, as follows:
  - opensm-lid-matrix.2.minhop.dump
  - opensm.f dbs.3.ftree
  - opensm-subnet.4.updn.lst
- In case that a fallback routing engine is used, both the routing engine that failed and the fallback engine that replaces it, dump their data.

If, for example, engine 2 runs ftree and it has a fallback engine with 3 as its id that runs minhop, one should expect to find 2 sets of dump files, one for each engine:
  - opensm-lid-matrix.2.ftree.dump
  - opensm-lid-matrix.3.minhop.dump
  - opensm.f dbs.2.ftree
  - opensm.f dbs.3.munhop

**Unicast Routing Cache**

Unicast routing cache prevents routing recalculation (which is a heavy task in a large cluster) when no topology change was detected during the heavy sweep, or when the topology change does not require new routing calculation (for example, when one or more CAs/RTRs/leaf switches going down, or one or more of these nodes coming back after being down).
Quality of Service Management in OpenSM

When Quality of Service (QoS) in OpenSM is enabled (using the ‘-Q’ or ‘--qos’ flags), OpenSM looks for a QoS Policy file. During fabric initialization and at every heavy sweep, OpenSM parses the QoS policy file, applies its settings to the discovered fabric elements, and enforces the provided policy on client requests. The overall flow for such requests is as follows:

- The request is matched against the defined matching rules such that the QoS Level definition is found

- Given the QoS Level, a path(s) search is performed with the given restrictions imposed by that level

There are two ways to define QoS policy:

- Advanced – the advanced policy file syntax provides the administrator various ways to match a PathRecord/MultiPathRecord (PR/MPR) request, and to enforce various QoS constraints on the requested PR/MPR

- Simple – the simple policy file syntax enables the administrator to match PR/MPR requests by various ULPs and applications running on top of these ULPs

**Advanced QoS Policy File**

The QoS policy file has the following sections:
1. Port Groups (denoted by port-groups) - this section defines zero or more port groups that can be referred later by matching rules (see below). Port group lists ports by:

- Port GUID

- Port name, which is a combination of NodeDescription and IB port number

- PKey, which means that all the ports in the subnet that belong to partition with a given PKey belong to this port group

- Partition name, which means that all the ports in the subnet that belong to partition with a given name belong to this port group

- Node type, where possible node types are: CA, SWITCH, ROUTER, ALL, and SELF (SM's port).

2. QoS Setup (denoted by qos-setup) - this section describes how to set up SL2VL and VL Arbitration tables on various nodes in the fabric. However, this is not supported in OFED. SL2VL and VLArb tables should be configured in the OpenSM options file (default location - /var/cache/opensm/opensm.opts).

3. QoS Levels (denoted by qos-levels) - each QoS Level defines Service Level (SL) and a few optional fields:

- MTU limit

- Rate limit

- PKey

- Packet lifetime

When path(s) search is performed, it is done with regards to restriction that these QoS Level parameters impose. One QoS level that is mandatory to define is a DEFAULT QoS level. It is applied to a PR/MPR query that does not match any existing match rule. Similar to any other QoS Level, it can also be explicitly referred by any match rule.

- QoS Matching Rules (denoted by qos-match-rules) - each PathRecord/MultiPathRecord query that OpenSM receives is matched against the
set of matching rules. Rules are scanned in order of appearance in the QoS policy file such as the first match takes precedence.

Each rule has a name of QoS level that will be applied to the matching query. A default QoS level is applied to a query that did not match any rule.

Queries can be matched by:

- Source port group (whether a source port is a member of a specified group)
- Destination port group (same as above, only for destination port)
- PKey
- QoS class
- Service ID

To match a certain matching rule, PR/MPR query has to match ALL the rule's criteria. However, not all the fields of the PR/MPR query have to appear in the matching rule.

For instance, if the rule has a single criterion - Service ID, it will match any query that has this Service ID, disregarding rest of the query fields. However, if a certain query has only Service ID (which means that this is the only bit in the PR/MPR component mask that is on), it will not match any rule that has other matching criteria besides Service ID.

**Simple QoS Policy Definition**

Simple QoS policy definition comprises of a single section denoted by qos-ulps. Similar to the advanced QoS policy, it has a list of match rules and their QoS Level, but in this case a match rule has only one criterion - its goal is to match a certain ULP (or a certain application on top of this ULP) PR/MPR request, and QoS Level has only one constraint - Service Level (SL).

The simple policy section may appear in the policy file in combine with the advanced policy, or as a stand-alone policy definition. See more details and list of match rule criteria below.
Policy File Syntax Guidelines

- Leading and trailing blanks, as well as empty lines, are ignored, so the indentation in the example is just for better readability.

- Comments are started with the pound sign (#) and terminated by EOL.

- Any keyword should be the first non-blank in the line, unless it's a comment.

- Keywords that denote section/subsection start have matching closing keywords.

- Having a QoS Level named "DEFAULT" is a must - it is applied to PR/MPR requests that did not match any of the matching rules.

- Any section/subsection of the policy file is optional.

Examples of Advanced Policy Files

As mentioned earlier, any section of the policy file is optional, and the only mandatory part of the policy file is a default QoS Level.

Here is an example of the shortest policy file:

```plaintext
qos-levels
    qos-level
        name: DEFAULT
        sl: 0
    end-qos-level
end-qos-levels
```

Port groups section is missing because there are no match rules, which means that port groups are not referred anywhere, and there is no need defining them. And since this policy file doesn't have any matching rules, PR/MPR query will not match any rule, and OpenSM will enforce default QoS level. Essentially, the above example is equivalent to not having a QoS policy file at all.
The following example shows all the possible options and keywords in the policy file and their syntax:

```plaintext
#  
# See the comments in the following example.  
# They explain different keywords and their meaning.  
#  
port-groups

   port-group  
      name: Storage  
      # "use" is just a description that is used for logging  
      use: SRP Targets  
      port-guid: 0x10000000000001, 0x10000000000005-0x1000000000FFFA  
      port-guid: 0x1000000000FFFF  
   end-port-group

   port-group  
      name: Virtual Servers  
      port-name: "vs1 HCA-1/P1, vs2 HCA-1/P1"  
   end-port-group

   port-group  
      name: Partitions  
      partition: Part1  
      pkey: 0x1234  
   end-port-group

   port-group  
      name: CAs and SM  
      node-type: CA, SELF  
   end-port-group

end-port-groups
```
# This section of the policy file describes how to set up SL2VL and VL Arbitration tables on various nodes in the fabric.
# However, this is not supported in OFED - the section is parsed and ignored. SL2VL and VLArb tables should be configured in the OpenSM options file (by default - /var/cache/opensm/opensm.opts).

qos-setup

qos-levels

# Having a QoS Level named "DEFAULT" is a must - it is applied to PR/MPR requests that didn't match any of the matching rules.
qos-level
  name: DEFAULT
  use: default QoS Level
  sl: 0
end-qos-level

# the whole set: SL, MTU-Limit, Rate-Limit, PKey, Packet Lifetime
qos-level
  name: WholeSet
  sl: 1
  mtu-limit: 4
  rate-limit: 5
  pkey: 0x1234
  packet-life: 8
end-qos-level

end-qos-levels

# Match rules are scanned in order of their appearance in the policy file.
# First matched rule takes precedence.
qos-match-rules

# matching by single criteria: QoS class
qos-match-rule
  use: by QoS class
  qos-class: 7-9,11
  # Name of qos-level to apply to the matching PR/MPR
  qos-level-name: WholeSet
end-qos-match-rule

# show matching by destination group and service id
qos-match-rule
Simple QoS Policy - Details and Examples

Simple QoS policy match rules are tailored for matching ULPs (or some application on top of a ULP) PR/MPR requests. This section has a list of per-ULP (or per-application) match rules and the SL that should be enforced on the matched PR/MPR query.

Match rules include:

- Default match rule that is applied to PR/MPR query that didn't match any of the other match rules
- IPoIB with a default PKey
- IPoIB with a specific PKey
- Any ULP/application with a specific Service ID in the PR/MPR query
- Any ULP/application with a specific PKey in the PR/MPR query
• Any ULP/application with a specific target IB port GUID in the PR/MPR query

Since any section of the policy file is optional, as long as basic rules of the file are kept (such as no referring to nonexistent port group, having default QoS Level, etc), the simple policy section (qos-ulps) can serve as a complete QoS policy file.

The shortest policy file in this case would be as follows:

```
qos-ulps
  default: 0 #default SL
end-qos-ulps
```

It is equivalent to the previous example of the shortest policy file, and it is also equivalent to not having policy file at all. Below is an example of simple QoS policy with all the possible keywords:

```
qos-ulps
  default: 0 # default SL
  sdp, port-num 30000: 0 # SL for application running on
    # top of SDP when a destination
    # TCP/IPport is 30000
  sdp, port-num 10000-20000: 0
  sdp: 1 # default SL for any other
    # application running on top of SDP
  rds: 2 # SL for RDS traffic
  ipoib, pkey 0x0001: 0 # SL for IPoIB on partition with
    # pkey 0x0001
  ipoib: 4 # default IPoIB partition,
    # pkey=0x7FFF
  any, service-id 0x6234:6 # match any PR/MPR query with a
    # specific Service ID
  any, pkey 0x0ABC: 6 # match any PR/MPR query with a
    # specific PKey
  srp, target-port-guid 0x1234: 5 # SRP when SRP Target is located
    # on a specified IB port GUID
  any, target-port-guid 0x0ABC-0xFFFFF: 6 # match any PR/MPR query
    # with a specific target port GUID
end-qos-ulps
```
Similar to the advanced policy definition, matching of PR/MPR queries is done in order of appearance in the QoS policy file such as the first match takes precedence, except for the "default" rule, which is applied only if the query didn't match any other rule. All other sections of the QoS policy file take precedence over the qos-ulps section. That is, if a policy file has both qos-match-rules and qos-ulps sections, then any query is matched first against the rules in the qos-match-rules section, and only if there was no match, the query is matched against the rules in qos-ulps section.

Note that some of these match rules may overlap, so in order to use the simple QoS definition effectively, it is important to understand how each of the ULPs is matched.

**IPoIB**

IPoIB query is matched by PKey or by destination GID, in which case this is the GID of the multicast group that OpenSM creates for each IPoIB partition.

Default PKey for IPoIB partition is 0x7fff, so the following three match rules are equivalent:

```
ipoib:<SL>ipoib, pkey 0x7fff : <SL>
any, pkey 0x7fff : <SL>
```

**SRP**

Service ID for SRP varies from storage vendor to vendor, thus SRP query is matched by the target IB port GUID. The following two match rules are equivalent:

```
srp, target-port-guid 0x1234 : <SL>
any, target-port-guid 0x1234 : <SL>
```

Note that any of the above ULPs might contain target port GUID in the PR query, so in order for these queries not to be recognized by the QoS manager as SRP, the SRP match rule (or any match rule that refers to the target port GUID only) should be placed at the end of the qos-ulps match rules.
MPI

SL for MPI is manually configured by an MPI admin. OpenSM is not forcing any SL on the MPI traffic, which explains why it is the only ULP that did not appear in the qos-ulps section.

SL2VL Mapping and VL Arbitration

OpenSM cached options file has a set of QoS related configuration parameters, that are used to configure SL2VL mapping and VL arbitration on IB ports. These parameters are:

- Max VLs: the maximum number of VLs that will be on the subnet
- High limit: the limit of High Priority component of VL Arbitration table (IBA 7.6.9)
- VLArb low table: Low priority VL Arbitration table (IBA 7.6.9) template
- VLArb high table: High priority VL Arbitration table (IBA 7.6.9) template
- SL2VL: SL2VL Mapping table (IBA 7.6.6) template. It is a list of VLs corresponding to SLs 0-15 (Note that VL15 used here means drop this SL).

There are separate QoS configuration parameters sets for various target types: CAs, routers, switch external ports, and switch's enhanced port 0. The names of such parameters are prefixed by "qos_<type>_" string. Here is a full list of the currently supported sets:

- qos_ca_ —QoS configuration parameters set for CAs.
- qos_rtr_ —parameters set for routers.
- qos_sw0_ —parameters set for switches' port 0.
- qos_swe_ —parameters set for switches' external ports.

Here's the example of typical default values for CAs and switches' external ports (hard-coded in OpenSM initialization):
VL arbitration tables (both high and low) are lists of VL/Weight pairs. Each list entry contains a VL number (values from 0-14), and a weighting value (values 0-255), indicating the number of 64 byte units (credits) which may be transmitted from that VL when its turn in the arbitration occurs. A weight of 0 indicates that this entry should be skipped. If a list entry is programmed for VL15 or for a VL that is not supported or is not currently configured by the port, the port may either skip that entry or send from any supported VL for that entry.

Note, that the same VLs may be listed multiple times in the High or Low priority arbitration tables, and, further, it can be listed in both tables. The limit of high-priority VLArb table (qos_high_limit) indicates the number of high-priority packets that can be transmitted without an opportunity to send a low-priority packet. Specifically, the number of bytes that can be sent is high_limit times 4K bytes.

A high_limit value of 255 indicates that the byte limit is unbounded.

Note

If the 255 value is used, the low priority VLs may be starved.

A value of 0 indicates that only a single packet from the high-priority table may be sent before an opportunity is given to the low-priority table.

Keep in mind that ports usually transmit packets of size equal to MTU. For instance, for 4KB MTU a single packet will require 64 credits, so in order to achieve effective VL
arbitration for packets of 4KB MTU, the weighting values for each VL should be multiples of 64.

Below is an example of SL2VL and VL Arbitration configuration on subnet:

```
qos_ca_max_vls 15
qos_ca_high_limit 6
qos_ca_vlarb_high 0:4
qos_ca_vlarb_low 0:0,1:64,2:128,3:192,4:0,5:64,6:64,7:64
qos_ca_sl2vl 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,7
qos_swe_max_vls 15
qos_swe_high_limit 6
qos_swe_vlarb_high 0:4
qos_swe_vlarb_low 0:0,1:64,2:128,3:192,4:0,5:64,6:64,7:64
qos_swe_sl2vl 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,7
```

In this example, there are 8 VLs configured on subnet: VL0 to VL7. VL0 is defined as a high priority VL, and it is limited to 6 x 4KB = 24KB in a single transmission burst. Such configuration would suit VL that needs low latency and uses small MTU when transmitting packets. Rest of VLs are defined as low priority VLs with different weights, while VL4 is effectively turned off.

**Deployment Example**

The figure below shows an example of an InfiniBand subnet that has been configured by a QoS manager to provide different service levels for various ULPs.

QoS Deployment on InfiniBand Subnet Example
QoS Configuration Examples

The following are examples of QoS configuration for different cluster deployments. Each example provides the QoS level assignment and their administration via OpenSM configuration files.

Typical HPC Example: MPI and Lustre

Assignment of QoS Levels

- MPI
  - Separate from I/O load
  - Min BW of 70%
- Storage Control (Lustre MDS)
  - Low latency
- Storage Data (Lustre OST)
○ Min BW 30%

Administration

- MPI is assigned an SL via the command line
  
  host1# mpirun -sl 0

- OpenSM QoS policy file

```plaintext
qos-ulps
  default :0 # default SL (for MPI)
  any, target-port-guid OST1,OST2,OST3,OST4 :1 # SL for Lustre OST
  any, target-port-guid MDS1,MDS2 :2 # SL for Lustre MDS
end-qos-ulps
```

Note: In this policy file example, replace OST* and MDS* with the real port GUIDs.

- OpenSM options file

```plaintext
qos_max_vls 8
qos_high_limit 0
qos_vlarb_high 2:1
qos_vlarb_low 0:96,1:224
qos_sl2vl 0,1,2,3,4,5,6,7,15,15,15,15,15,15,15,15
```

EDC SOA (2-tier): IPoIB and SRP

The following is an example of QoS configuration for a typical enterprise data center (EDC) with service oriented architecture (SOA), with IPoIB carrying all application traffic and SRP used for storage.

QoS Levels
• Application traffic
  o IPoIB (UD and CM) and SDP
  o Isolated from storage
  o Min BW of 50%

• SRP
  o Min BW 50%
  o Bottleneck at storage nodes

Administration

• OpenSM QoS policy file

```
qos-ulps
  default :0
  ipoib :1
  sdp :1
  srp, target-port-guid SRPT1,SRPT2,SRPT3 :2
end-qos-ulps
```

Note: In this policy file example, replace SRPT* with the real SRP Target port GUIDs.

• OpenSM options file

```
qos_max_vls 8
qos_high_limit 0
qos_vlarb_high 1:32,2:32
qos_vlarb_low 0:1,
qos_sl2vl 0,1,2,3,4,5,6,7,15,15,15,15,15,15,15,15
```
EDC (3-tier): IPoIB, RDS, SRP

The following is an example of QoS configuration for an enterprise data center (EDC), with IPoIB carrying all application traffic, RDS for database traffic, and SRP used for storage.

**QoS Levels**

- Management traffic (ssh)
  - IPoIB management VLAN (partition A)
  - Min BW 10%
- Application traffic
  - IPoIB application VLAN (partition B)
  - Isolated from storage and database
  - Min BW of 30%
- Database Cluster traffic
  - RDS
  - Min BW of 30%
- SRP
  - Min BW 30%
  - Bottleneck at storage nodes

**Administration**

- OpenSM QoS policy file

```bash
qos-ulps
```
Note: In the following policy file example, replace SRPT* with the real SRP Initiator port GUIDs.

- **OpenSM options file**

```plaintext
default:0
ipoib, pkey 0x8001:1
ipoib, pkey 0x8002:2
rds:3
srp, target-port-guid SRPT1, SRPT2, SRPT3:4
end-qos-ulps
```

- **Partition configuration file**

```plaintext
qos_max_vl 8
qos_high_limit 0
qos_vlarb_high 1:32,2:96,3:96,4:96
qos_vlarb_low 0:1
qos_sl2vl 0,1,2,3,4,5,6,7,15,15,15,15,15,15,15,15
Default=0x7fff,ipoib : ALL=full;PartA=0x8001, sl=1, ipoib : ALL=full;
```

**Enhanced QoS**

Enhanced QoS provides a higher resolution of QoS at the service level (SL). Users can configure rate limit values per SL for physical ports, virtual ports, and port groups, using enhanced_qos_policy_file configuration parameter.

Valid values of this parameter:

- Full path to the policy file through which Enhanced QoS Manager is configured
- "null" - to disable the Enhanced QoS Manager (default value)
Enhanced QoS Policy File

The policy file is comprised of three sections:

- **BW_NAMES:** Used to define bandwidth setting and name (currently, rate limit is the only setting). Bandwidth names can be used in BW_RULES and VPORT_BW_RULES sections.

  Bandwidth names are defined using the syntax:

  `<name> = <rate limit in 1Mbps units>`

  Example: `My_bandwidth = 50`

- **BW_RULES:** Used to define the rules that map the bandwidth setting to a specific SL of a specific GUID.

  Bandwidth rules are defined using the syntax:

  `<guid>|<port group name> = <sl id>:<bandwidth name>, <sl id>:<bandwidth name>…`

  Examples:

  `0x2c90000000025 = 5:My_bandwidth, 7:My_bandwidth`
  
  `Port_grp1 = 3:My_bandwidth, 9:My_bandwidth`

- **VPORT_BW_RULES:** Used to define the rules that map the bandwidth setting to a specific SL of a specific virtual port GUID.

  Bandwidth rules are defined using the syntax:

  `<guid>= <sl id>:<bandwidth name>, <sl id>:<bandwidth name>…`  

  Examples:

  `0x2c90000000026= 5:My_bandwidth, 7:My_bandwidth`

**Note**

To enable Enhanced QoS Manager, QoS must be enabled in OpenSM.
Special Keywords

- Keyword “all” allows setting a rate limit of all SLs to some BW for a specific physical or virtual port. It is possible to combine “all” with specific SL rate limits.

Example:

0x2c90000000025 = all:BW1,SL3:BW2In this case, SL3 will be assigned BW2 rate limit, while the rest of SLs get BW1 rate limit.

- "default" is a well-known name which can be used to define a default rule used for any GUID with no defined rule.

If no default rule is defined, any GUID without a specific rule will be configured with unlimited rate limit for all SLs.

Keyword “all” is also applicable to the default rule. Default rule is local to each section.

Special Subnet Manager Configuration Options

New SM configuration option enhanced_qos_vport0_unlimit_default_rl was added to opensm.conf.

The possible values for this configuration option are:

- TRUE: For specific virtual port0 GUID, SLs not mentioned in bandwidth rule will be set to unlimited bandwidth (0) regardless of the default rule of the VPORT_BW_RULES section.

  Virtual port0 GUIDs not mentioned in VPORT_BW_SECTION will be set to unlimited BW on all SLs.

- FALSE: The GUID of virtual port0 is treated as any other virtual port in VPORT_BW_SECTION.

  SM should be signaled by HUP once the option is changed.
Default: TRUE

Notes

- When rate limit is set to 0, it means that the bandwidth is unlimited.

- Any unspecified SL in a rule will be set to 0 (unlimited) rate limit automatically if no default rule is specified.

- Failure to complete policy file parsing leads to an undefined behavior. User must confirm no relevant error messages in SM log in order to ensure Enhanced QoS Manager is configured properly.

- A file with only 'BW_NAMES' and 'BW_RULES' keywords configures the network with an unlimited rate limit.

- HCA physical port GUID can be specified in BW_RULES and VPORT_BW_RULES sections.

- In BW_RULES section, the rate limit assigned to a specific SL will limit the total BW that can be sent through the PF on a given SL.

- In VPORT_BW_RULES section, the rate limit assigned to a specific SL will limit only the traffic sent from the IB interface corresponding to the physical port GUID (virtual port0 IB interface). The traffic sent from other virtual IB interfaces will not be limited if no specific rules are defined.

Policy File Example

All physical ports in the fabric are with a rate limit of 50Mbps on SL1, except for GUID 0x2c90000000025, which is configured with rate limit of 25Mbps on SL1. In this example, the traffic on SLs (other than SL1) is unlimited.

All virtual ports in the fabric (except virtual port0 of all physical ports) will be rate-limited to 15Mbps for all SLs because of the default rule of VPORT_BW_RULES section.

Virtual port GUID 0x2c90000000026 is configured with a rate limit of 10Mbps on SL3. The rest of the SLs on this virtual port will get a rate limit of 15 Mbps because of the default
Adaptive Routing Manager and Self-Healing Networking

Adaptive Routing Manager supports advanced InfiniBand features; Adaptive Routing (AR) and Self-Healing Networking.

For information on how to set up AR and Self-Healing Networking, please refer to HowTo Configure Adaptive Routing and Self-Healing Networking Community post.

DOS MAD Prevention

DOS MAD prevention is achieved by assigning a threshold for each agent's RX. Agent's RX threshold provides a protection mechanism to the host memory by limiting the agents' RX with a threshold. Incoming MADs above the threshold are dropped and are not queued to the agent’s RX.

To enable DOS MAD Prevention:
1. Go to `/etc/modprobe.d/mlnx.conf`.

2. Add to the file the option below.

```
ib_umad enable_rx_threshold 1
```

The threshold value can be controlled from the user-space via `libibumad`.

To change the value, use the following API:

```
int umad_update_threshold(int fd, int threshold);
```

@fd: file descriptor, agent's RX associated to this fd.
@threshold: new threshold value

---

**IB Router Support in OpenSM**

In order to enable the IB router in OpenSM, the following parameters should be configured:

**IB Router Parameters for OpenSM**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtr_pr_flow_lab</td>
<td>Defines whether the SM should create alias GUIDs required for router support for each port. Defines flow label value to use in response for path records related to the router.</td>
<td>0 (Disabled)</td>
</tr>
<tr>
<td>rtr_pr_tclass</td>
<td>Defines TClass value to use in response for path records related to the router</td>
<td>0</td>
</tr>
<tr>
<td>rtr_pr_sl</td>
<td>Defines sl value to use in response for path records related to router.</td>
<td>0</td>
</tr>
<tr>
<td>rtr_p_mtu</td>
<td>Defines MTU value to use in response for path records related to the router.</td>
<td>4 (IB_MTU_LEN_2048)</td>
</tr>
</tbody>
</table>
### OpenSM Activity Report

OpenSM can produce an activity report in a form of a dump file which details the different activities done in the SM. Activities are divided into subjects. The OpenSM Supported Activities table below specifies the different activities currently supported in the SM activity report.

Reporting of each subject can be enabled individually using the configuration parameter `activity_report_subjects`:

- Valid values:

  Comma separated list of subjects to dump. The current supported subjects are:

  - "mc" - activity IDs 1, 2 and 8
  - "prtn" - activity IDs 3, 4, and 5
  - "virt" - activity IDs 6 and 7
  - "routing" - activity IDs 8-12

  Two predefined values can be configured as well:

  - "all" - dump all subjects
  - "none" - disable the feature by dumping none of the subjects

  Default value: "none"

---

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtr_pr_rate</td>
<td>Defines rate value to use in response for path records related to the router.</td>
<td>16 (IB_PATH_RECORD_RATE_100_GBS)</td>
</tr>
<tr>
<td>Activity ID</td>
<td>Activity Name</td>
<td>Additional Fields</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>mcm_member</td>
<td>• MLid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• MGid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Port Guid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Join State</td>
</tr>
<tr>
<td>2</td>
<td>mcg_change</td>
<td>• MLid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• MGid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Change</td>
</tr>
<tr>
<td>3</td>
<td>prtn_guid_add</td>
<td>• Port Guid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• PKey</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Block index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Pkey Index</td>
</tr>
<tr>
<td>4</td>
<td>prtn_create</td>
<td>-PKey</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Prtn Name</td>
</tr>
<tr>
<td>5</td>
<td>prtn_delete</td>
<td>• PKey</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Delete</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>port_virt_discover</td>
<td>• Port Guid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Top Index</td>
</tr>
<tr>
<td>7</td>
<td>vport_state_change</td>
<td>• Port Guid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• VPort Guid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• VPort Index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• VNode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Guid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• VPort State</td>
</tr>
</tbody>
</table>
### Activity ID 8

<table>
<thead>
<tr>
<th>Activity Name</th>
<th>Additional Fields</th>
<th>Comments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcg_tree_calc</td>
<td>mlid</td>
<td></td>
<td>MCast group tree calculated</td>
</tr>
</tbody>
</table>

### Activity ID 9

<table>
<thead>
<tr>
<th>Activity Name</th>
<th>Additional Fields</th>
<th>Comments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>routing_succeed</td>
<td>routing engine</td>
<td>Routing done successfully</td>
<td></td>
</tr>
</tbody>
</table>

### Activity ID 10

<table>
<thead>
<tr>
<th>Activity Name</th>
<th>Additional Fields</th>
<th>Comments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>routing_failed</td>
<td>routing engine</td>
<td>Routing failed</td>
<td></td>
</tr>
</tbody>
</table>

### Activity ID 11

<table>
<thead>
<tr>
<th>Activity Name</th>
<th>Additional Fields</th>
<th>Comments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ucast_cache_invali-</td>
<td></td>
<td>ucast cache invalidated</td>
<td></td>
</tr>
</tbody>
</table>

### Activity ID 12

<table>
<thead>
<tr>
<th>Activity Name</th>
<th>Additional Fields</th>
<th>Comments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ucast_cache_routing</td>
<td></td>
<td>ucast cache routing done</td>
<td></td>
</tr>
</tbody>
</table>

### Offsweep Balancing

When working with minhop/dor/updn, subnet manager can re-balance routing during idle time (between sweeps).

- offsweep_balancing_enabled - enables/disables the feature. Examples:
  - offsweep_balancing_enabled = TRUE
  - offsweep_balancing_enabled = FALSE (default)

- offsweep_balancing_window - defines window of seconds to wait after sweep before starting the re-balance process. Applicable only if offsweep_balancing_enabled=TRUE. Example:
  - offsweep_balancing_window = 180 (default)

### QoS - Quality of Service

Quality of Service (QoS) requirements stem from the realization of I/O consolidation over an IB network. As multiple applications and ULPs share the same fabric, a means is needed to control their use of network resources.
The basic need is to differentiate the service levels provided to different traffic flows, such that a policy can be enforced and can control each flow utilization of fabric resources.

The InfiniBand Architecture Specification defines several hardware features and management interfaces for supporting QoS:

Up to 15 Virtual Lanes (VL) carry traffic in a non-blocking manner

- Arbitration between traffic of different VLs is performed by a two-priority-level weighted round robin arbiter. The arbiter is programmable with a sequence of (VL, weight) pairs and a maximal number of high priority credits to be processed before low priority is served.

- Packets carry class of service marking in the range 0 to 15 in their header SL field.

- Each switch can map the incoming packet by its SL to a particular output VL, based on a programmable table VL=SL-to-VL-MAP(in-port, out-port, SL).

- The Subnet Administrator controls the parameters of each communication flow by providing them as a response to Path Record (PR) or MultiPathRecord (MPR) queries.
DiffServ architecture (IETF RFC 2474 & 2475) is widely used in highly dynamic fabrics. The following subsections provide the functional definition of the various software elements that enable a DiffServ-like architecture over the NVIDIA OFED software stack.

**QoS Architecture**

QoS functionality is split between the SM/SA, CMA and the various ULPs. We take the "chronology approach" to describe how the overall system works.

1. The network manager (human) provides a set of rules (policy) that define how the network is being configured and how its resources are split to different QoS-Levels. The policy also define how to decide which QoS-Level each application or ULP or service use.

2. The SM analyzes the provided policy to see if it is realizable and performs the necessary fabric setup. Part of this policy defines the default QoS-Level of each partition. The SA is enhanced to match the requested Source, Destination, QoS-Class, Service-ID, PKey against the policy, so clients (ULPs, programs) can obtain a policy enforced QoS. The SM may also set up partitions with appropriate IPoIB broadcast group. This broadcast group carries its QoS attributes: SL, MTU, RATE, and Packet Lifetime.

3. IPoIB is being setup. IPoIB uses the SL, MTU, RATE and Packet Lifetime available on the multicast group which forms the broadcast group of this partition.

4. MPI which provides non IB based connection management should be configured to run using hard coded SLs. It uses these SLs for every QP being opened.

5. ULPs that use CM interface (like SRP) have their own pre-assigned Service-ID and use it while obtaining PathRecord/MultiPathRecord (PR/MPR) for establishing connections. The SA receiving the PR/MPR matches it against the policy and returns the appropriate PR/MPR including SL, MTU, RATE and Lifetime.

6. ULPs and programs (e.g. SDP) use CMA to establish RC connection provide the CMA the target IP and port number. ULPs might also provide QoS-Class. The CMA then creates Service-ID for the ULP and passes this ID and optional QoS-Class in the PR/MPR request. The resulting PR/MPR is used for configuring the connection QP.

**PathRecord and Multi Path Record Enhancement for QoS:**
As mentioned above, the PathRecord and MultiPathRecord attributes are enhanced to carry the Service-ID which is a 64bit value. A new field QoS-Class is also provided.

A new capability bit describes the SM QoS support in the SA class port info. This approach provides an easy migration path for existing access layer and ULPs by not introducing new set of PR/MPR attributes.

**Supported Policy**

The QoS policy, which is specified in a stand-alone file, is divided into the following four subsections:

**Port Group**

A set of CAs, Routers or Switches that share the same settings. A port group might be a partition defined by the partition manager policy, list of GUIDs, or list of port names based on NodeDescription.

**Fabric Setup**

Defines how the SL2VL and VLArb tables should be set up.

**Note**

In OFED this part of the policy is ignored. SL2VL and VLArb tables should be configured in the OpenSM options file (opensm.opts).

**QoS-Levels Definition**

This section defines the possible sets of parameters for QoS that a client might be mapped to. Each set holds SL and optionally: Max MTU, Max Rate, Packet Lifetime and Path Bits.
Matching Rules

A list of rules that match an incoming PR/MPR request to a QoS-Level. The rules are processed in order such as the first match is applied. Each rule is built out of a set of match expressions which should all match for the rule to apply. The matching expressions are defined for the following fields:

- SRC and DST to lists of port groups
- Service-ID to a list of Service-ID values or ranges
- QoS-Class to a list of QoS-Class values or ranges

CMA Features

The CMA interface supports Service-ID through the notion of port space as a prefix to the port number, which is part of the sockaddr provided to rdma_resolve_add(). The CMA also allows the ULP (like SDP) to propagate a request for a specific QoS-Class. The CMA uses the provided QoS-Class and Service-ID in the sent PR/MPR.

IPoIB

IPoIB queries the SA for its broadcast group information and uses the SL, MTU, RATE and Packet Lifetime available on the multicast group which forms this broadcast group.

SRP
The current SRP implementation uses its own CM callbacks (not CMA). So SRP fills in the Service-ID in the PR/MPR by itself and use that information in setting up the QP.

SRP Service-ID is defined by the SRP target I/O Controller (it also complies with IBTA Service-ID rules). The Service-ID is reported by the I/O Controller in the ServiceEntries DMA attribute and should be used in the PR/MPR if the SA reports its ability to handle QoS PR/MPRs.

**IP over InfiniBand (IPoIB)**

**Upper Layer Protocol (ULP)**

The IP over IB (IPoIB) ULP driver is a network interface implementation over InfiniBand. IPoIB encapsulates IP datagrams over an InfiniBand Datagram transport service. The IPoIB driver, ib_ipoib, exploits the following capabilities:

- VLAN simulation over an InfiniBand network via child interfaces
- High Availability via Bonding
- Varies MTU values:
  - up to 4k in Datagram mode
- Uses any ConnectX® IB ports (one or two)
- Inserts IP/UDP/TCP checksum on outgoing packets
- Calculates checksum on received packets
- Support net device TSO through ConnectX® LSO capability to defragment large data-grams to MTU quantas.

IPoIB also supports the following software based enhancements:

- Giant Receive Offload
- NAPI
• Ethtool support

Enhanced IPoIB

Enhanced IPoIB feature enables offloading ULP basic capabilities to a lower vendor specific driver, in order to optimize IPoIB data path. This will allow IPoIB to support multiple stateless offloads, such as RSS/TSS, and better utilize the features supported, enabling IPoIB datagram to reach peak performance in both bandwidth and latency.

Enhanced IPoIB supports/performs the following:

• Stateless offloads (RSS, TSS)

• Multi queues

• Interrupt moderation

• Multi partitions optimizations

• Sharing send/receive Work Queues

• Vendor specific optimizations

• UD mode only

Port Configuration

The physical port MTU (indicates the port capability) default value is 4k, whereas the IPoIB port MTU ("logical" MTU) default value is 2k as it is set by the OpenSM.

To change the IPoIB MTU to 4k, edit the OpenSM partition file in the section of IPoIB setting as follow:

```
Default=0xffff, ipoib, mtu=5 : ALL=full;
```

where:

"mtu=5" indicates that all IPoIB ports in the fabric are using 4k MTU, ("mtu=4" indicates 2k MTU)
**IPoIB Configuration**

Unless you have run the installation script `mlnxofedinstall` with the flag `-n`, then IPoIB has not been configured by the installation. The configuration of IPoIB requires assigning an IP address and a subnet mask to each HCA port, like any other network adapter card (i.e., you need to prepare a file called `ifcfg-ib<n>` for each port). The first port on the first HCA in the host is called interface `ib0`, the second port is called `ib1`, and so on.

IPoIB configuration can be based on DHCP or on a static configuration that you need to supply (see below). You can also apply a manual configuration that persists only until the next reboot or driver restart (see below).

**IPoIB Configuration Based on DHCP**

Setting an IPoIB interface configuration based on DHCP is performed similarly to the configuration of Ethernet interfaces. In other words, you need to make sure that IPoIB configuration files include the following line:

- For RedHat:
  
  ```
  BOOTPROTO= dhcp
  ```

- For SLES:
  
  ```
  BOOTPROTO= 'dhcp'
  ```

**Note**

If IPoIB configuration files are included, `ifcfg-ib<n>` files will be installed under:

- `/etc/sysconfig/network-scripts/` on a RedHat machine
- `/etc/sysconfig/network/` on a SuSE machine.
Standard DHCP fields holding MAC addresses are not large enough to contain an IPoIB hardware address. To overcome this problem, DHCP over InfiniBand messages convey a client identifier field used to identify the DHCP session. This client identifier field can be used to associate an IP address with a client identifier value, such that the DHCP server will grant the same IP address to any client that conveys this client identifier.

The length of the client identifier field is not fixed in the specification. For the *NVIDIA OFED for Linux* package, it is recommended to have IPoIB use the same format that FlexBoot uses for this client identifier.

---

**Note**

A patch for DHCP may be required for supporting IPoIB. For further information, please see the REAME file available under the docs/dhcp/ directory.

**Note**

Red Hat Enterprise Linux 7 supports assigning static IP addresses to InfiniBand IPoIB interfaces. However, as these interfaces do not have a normal hardware Ethernet address, a different method of specifying a unique identifier for the IPoIB interface must be used. The standard is to use the option dhcp-client-identifier= construct to specify the IPoIB interface’s dhcp-client-identifier field. The DHCP server host construct supports at most one hardware Ethernet and one dhcp-client-identifier entry per host stanza. However, there may be more than one fixed-address entry and the DHCP server will automatically respond with an address that is appropriate for the network that the DHCP request was received on.
DHCP Server

In order for the DHCP server to provide configuration records for clients, an appropriate configuration file needs to be created. By default, the DHCP server looks for a configuration file called dhcpd.conf under /etc. You can either edit this file or create a new one and provide its full path to the DHCP server using the -cf flag (See a file example at docs/dhcpd.conf).

The DHCP server must run on a machine which has loaded the IPoIB module. To run the DHCP server from the command line, enter:

```
dhcp <IB network interface name> -d
```

Example:

```
host1# dhcpd ib0 -d
```

DHCP Client (Optional)

- **Note**
  
  A DHCP client can be used if you need to prepare a diskless machine with an IB driver.

In order to use a DHCP client identifier, you need to first create a configuration file that defines the DHCP client identifier.

Then run the DHCP client with this file using the following command:
Example of a configuration file for the ConnectX (PCI Device ID 26428), called dhclient.conf:

```
Example of a configuration file for InfiniHost III Ex (PCI Device ID 25218), called dhclient.conf:

In order to use the configuration file, run

```

```

host1# dhclient –cf dhclient.conf ib1

Static IPoIB Configuration

If you wish to use an IPoIB configuration that is not based on DHCP, you need to supply the installation script with a configuration file (using the ‘-n’ option) containing the full IP configuration. The IPoIB configuration file can specify either or both of the following data for an IPoIB interface:

- A static IPoIB configuration
An IPoIB configuration based on an Ethernet configuration

See your Linux distribution documentation for additional information about configuring IP addresses.

The following code lines are an excerpt from a sample IPoIB configuration file:

```plaintext
# Static settings; all values provided by this file
IPADDR_ib0=10.4.3.175
NETMASK_ib0=255.255.0.0
NETWORK_ib0=10.4.0.0
BROADCAST_ib0=10.4.255.255
ONBOOT_ib0=1

# Based on eth0; each '*' will be replaced with a corresponding octet
# from eth0.
LAN_INTERFACE_ib0=eth0
IPADDR_ib0=10.4.*.*
NETMASK_ib0=255.255.0.0
NETWORK_ib0=10.4.0.0
BROADCAST_ib0=10.4.255.255
ONBOOT_ib0=1

# Based on the first eth<n> interface that is found (for n=0,1,...);
# each '*' will be replaced with a corresponding octet from eth<n>.
LAN_INTERFACE_ib0=
IPADDR_ib0=10.4.*.*
NETMASK_ib0=255.255.0.0
NETWORK_ib0=10.4.0.0
BROADCAST_ib0=10.4.255.255
ONBOOT_ib0=1
```

Manually Configuring IPoIB

ℹ️ Note

This manual configuration persists only until the next reboot or driver restart.
To manually configure IPoIB for the default IB partition (VLAN), perform the following steps:

1. Configure the interface by entering the ifconfig command with the following items:
   - The appropriate IB interface (ib0, ib1, etc.)
   - The IP address that you want to assign to the interface
   - The netmask keyword
   - The subnet mask that you want to assign to the interface

   The following example shows how to configure an IB interface:

   ```
   host1$ ifconfig ib0 10.4.3.175 netmask 255.255.0.0
   ```

2. (Optional) Verify the configuration by entering the ifconfig command with the appropriate interface identifier `ib#` argument.

   The following example shows how to verify the configuration:

   ```
   host1$ ifconfig ib0
   ib0 Link encap:UNSPEC  HWaddr 80-00-04-04-FE-80-00-00-00-00-00-00-00-00
   inet addr:10.4.3.175  Bcast:10.4.255.255  Mask:255.255.0.0
   UP BROADCAST MULTICAST  MTU:65520  Metric:1
   RX packets:0  errors:0  dropped:0  overruns:0  frame:0
   TX packets:0  errors:0  dropped:0  overruns:0  carrier:0
   collisions:0  txqueuelen:128
   RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
   ```

3. Repeat the first two steps on the remaining interface(s).

**Sub-interfaces**
You can create sub-interfaces for a primary IPoIB interface to provide traffic isolation. Each such sub-interface (also called a child interface) has a different IP and network addresses from the primary (parent) interface. The default Partition Key (PKey), ff:ff, applies to the primary (parent) interface.

This section describes how to:

- Create a subinterface
- Remove a subinterface

## Creating a Subinterface

In the following procedure, ib0 is used as an example of an IB sub-interface.

*To create a child interface (sub-interface), follow this procedure:*

1. Decide on the PKey to be used in the subnet (valid values can be 0 or any 16-bit unsigned value). The actual PKey used is a 16-bit number with the most significant bit set. For example, a value of 1 will give a PKey with the value 0x8001.

2. Create a child interface by running:

   ```
   host1$ echo <PKey> > /sys/class/net/<IB subinterface>/create_child
   ```

   **Example:**

   ```
   host1$ echo 1 > /sys/class/net/ib0/create_child
   ```

   This will create the interface ib0.8001.

3. Verify the configuration of this interface by running:
Using the example of the previous step:

```
host1$ ifconfig <subinterface>.<subinterface PKey>
```

4. As can be seen, the interface does not have IP or network addresses. To configure those, you should follow the manual configuration procedure described in "Manually Configuring IPoIB" section above.

5. To be able to use this interface, a configuration of the Subnet Manager is needed so that the PKey chosen, which defines a broadcast address, be recognized.

**Removing a Subinterface**

*To remove a child interface (subinterface), run:*

```
echo <subinterface PKey> /sys/class/net/<ib_interface>/delete_child
```

Using the example of the second step from the previous chapter:

```
echo 0x8001 > /sys/class/net/ib0/delete_child
```
Note that when deleting the interface you must use the PKey value with the most significant bit set (e.g., 0x8000 in the example above).

**Verifying IPoIB Functionality**

To verify your configuration and IPoIB functionality are successful, perform the following steps:

1. Verify the IPoIB functionality by using the `ifconfig` command.
   
   The following example shows how two IB nodes are used to verify IPoIB functionality. In the following example, IB node 1 is at 10.4.3.175, and IB node 2 is at 10.4.3.176:
   
   ```
   host1# ifconfig ib0 10.4.3.175 netmask 255.255.0.0
   host2# ifconfig ib0 10.4.3.176 netmask 255.255.0.0
   ```

2. Enter the ping command from 10.4.3.175 to 10.4.3.176.

3. The following example shows how to enter the ping command:
   
   ```
   host1# ping -c 5 10.4.3.176
   PING 10.4.3.176 (10.4.3.176) 56(84) bytes of data.
   64 bytes from 10.4.3.176: icmp_seq=0 ttl=64 time=0.079 ms
   64 bytes from 10.4.3.176: icmp_seq=1 ttl=64 time=0.044 ms
   64 bytes from 10.4.3.176: icmp_seq=2 ttl=64 time=0.055 ms
   64 bytes from 10.4.3.176: icmp_seq=3 ttl=64 time=0.049 ms
   64 bytes from 10.4.3.176: icmp_seq=4 ttl=64 time=0.065 ms
   --- 10.4.3.176 ping statistics ---
   5 packets transmitted, 5 received, 0% packet loss, time 3999ms rtt min/avg/max/mdev =
   0.044/0.058/0.079/0.014 ms, pipe 2
   ```

**Bonding IPoIB**
To create an interface configuration script for the ibX and bondX interfaces, you should use the standard syntax (depending on your OS).

Bonding of IPoIB interfaces is accomplished in the same manner as would bonding of Ethernet interfaces: via the Linux Bonding Driver.

- Network Script files for IPoIB slaves are named after the IPoIB interfaces (e.g: ifcfg-ib0)
- The only meaningful bonding policy in IPoIB is High-Availability (bonding mode number 1, or active-backup)
- Bonding parameter "fail_over_mac" is meaningless in IPoIB interfaces, hence, the only supported value is the default: 0

For a persistent bonding IPoIB Network configuration, use the same Linux Network Scripts semantics, with the following exceptions/ additions:

- In the bonding master configuration file (e.g: ifcfg-bond0), in addition to Linux bonding semantics, use the following parameter: MTU=65520

**COND**

![Note](image)

For IPoIB slaves, use MTU=2044. If you do not set the correct MTU or do not set MTU at all, performance of the interface might decrease.

Dynamically Connected Transport (DCT)

- In the bonding slave configuration file (e.g: ifcfg-ib0), use the same Linux Network Scripts semantics. In particular: DEVICE=ib0
- In the bonding slave configuration file (e.g: ifcfg-ib0.8003), the line TYPE=InfiniBand is necessary when using bonding over devices configured with partitions (p_key)
- For RHEL users:
In `/etc/modprobe.b/bond.conf` add the following lines:

```
alias bond0 bonding
```

- For SLES users:

  It is necessary to update the `MANDATORY_DEVICES` environment variable in
  `/etc/sysconfig/network/config` with the names of the IPoIB slave devices (e.g. `ib0, ib1, etc.`). Otherwise, bonding master may be created before IPoIB slave interfaces
  at boot time.

  It is possible to have multiple IPoIB bonding masters and a mix of IPoIB bonding
  master and Ethernet bonding master. However, It is NOT possible to mix Ethernet
  and IPoIB slaves under the same bonding master.

**Note**

Restarting `openibd` does no keep the bonding configuration via
Network Scripts. You have to restart the network service in order to
bring up the bonding master. After the configuration is saved, restart
the network service by running: `/etc/init.d/network restart`.

**Dynamic PKey Change**

Dynamic PKey change means the PKey can be changed (add/removed) in the SM
database and the interface that is attached to that PKey is updated immediately without
the need to restart the driver.

If the PKey is already configured in the port by the SM, the child-interface can be used
immediately. If not, the interface will be ready to use only when SM adds the relevant
PKey value to the port after the creation of the child interface. No additional
configuration is required once the child-interface is created.

**Precision Time Protocol (PTP) over IPoIB**
This feature allows for accurate synchronization between the distributed entities over the network. The synchronization is based on symmetric Round Trip Time (RTT) between the master and slave devices.

This feature is enabled by default, and is also supported over PKey interfaces.

For more on the PTP feature, refer to Running Linux PTP with ConnectX-4/ConnectX-5/ConnectX-6 Community post.

For further information on Time-Stamping, follow the steps in "Time-Stamping Service".

**One Pulse Per Second (1PPS) over IPoIB**

1PPS is a time synchronization feature that allows the adapter to be able to send or receive 1 pulse per second on a dedicated pin on the adapter card using an SMA connector (SubMiniature version A). Only one pin is supported and could be configured as 1PPS in or 1PPS out.

For further information, refer to HowTo Test 1PPS on NVIDIA Adapters Community post.

**Advanced Transport**

**Atomic Operations**

**Atomic Operations in mlx5 Driver**

To enable atomic operation with this endianness contradiction, use the `ibv_create_qp` to create the QP and set the `IBV_QP_CREATE_ATOMIC_BE_REPLY` flag on `create_flags`.

**XRC - eXtended Reliable Connected Transport Service for InfiniBand**

XRC allows significant savings in the number of QPs and the associated memory resources required to establish all to all process connectivity in large clusters.
It significantly improves the scalability of the solution for large clusters of multicore end-nodes by reducing the required resources.

For further details, please refer to the "Annex A14 Supplement to InfiniBand Architecture Specification Volume 1.2.1"

A new API can be used by user space applications to work with the XRC transport. The legacy API is currently supported in both binary and source modes, however it is deprecated. Thus we recommend using the new API.

The new verbs to be used are:

- `ibv_open_xrcd/ibv_close_xrcd`
- `ibv_create_srq_ex`
- `ibv_get_srq_num`
- `ibv_create_qp_ex`
- `ibv_open_qp`

Please use `ibv_xsrq_pingpong` for basic tests and code reference. For detailed information regarding the various options for these verbs, please refer to their appropriate man pages.

**Dynamically Connected Transport (DCT)**

Dynamically Connected transport (DCT) service is an extension to transport services to enable a higher degree of scalability while maintaining high performance for sparse traffic. Utilization of DCT reduces the total number of QPs required system wide by having Reliable type QPs dynamically connect and disconnect from any remote node. DCT connections only stay connected while they are active. This results in smaller memory footprint, less overhead to set connections and higher on-chip cache utilization and hence increased performance. DCT is supported only in mlx5 driver.

**Note**
MPI Tag Matching and Rendezvous Offloads

Note

Supported in ConnectX®-5 and above adapter cards.

Tag Matching and Rendezvous Offloads is a technology employed by NVIDIA to offload the processing of MPI messages from the host machine onto the network card. Employing this technology enables a zero copy of MPI messages, i.e. messages are scattered directly to the user's buffer without intermediate buffering and copies. It also provides a complete rendezvous progress by NVIDIA devices. Such overlap capability enables the CPU to perform the application's computational tasks while the remote data is gathered by the adapter.

For more information Tag Matching Offload, please refer to the Understanding MPI Tag Matching and Rendezvous Offloads (ConnectX-5) Community post.

Optimized Memory Access

Memory Region Re-registration

Memory Region Re-registration allows the user to change attributes of the memory region. The user may change the PD, access flags or the address and length of the memory region. Memory region supports contagious pages allocation. Consequently, it de-registers memory region followed by register memory region. Where possible, resources are reused instead of de-allocated and reallocated.
Example:

```c
int ibv_rereg_mr(struct ibv_mr *mr, int flags, struct ibv_pd *pd, void *addr, size_t length, uint64_t access, struct ibv_rereg_mr_attr *attr);
```

<table>
<thead>
<tr>
<th>@mr:</th>
<th>The memory region to modify.</th>
</tr>
</thead>
<tbody>
<tr>
<td>@flags:</td>
<td>A bit-mask used to indicate which of the following properties of the memory region are being modified. Flags should be one of: IBV_REREG_MR_CHANGE_TRANSLATION /* Change translation (location and length) <em>/ IBV_REREG_MR_CHANGE_PD /</em> Change protection domain <em>/ IBV_REREG_MR_CHANGE_ACCESS /</em> Change access flags */</td>
</tr>
<tr>
<td>@pd:</td>
<td>If IBV_REREG_MR_CHANGE_PD is set in flags, this field specifies the new protection domain to associated with the memory region, otherwise, this parameter is ignored.</td>
</tr>
<tr>
<td>@addr:</td>
<td>If IBV_REREG_MR_CHANGE_TRANSLATION is set in flags, this field specifies the start of the virtual address to use in the new translation, otherwise, this parameter is ignored.</td>
</tr>
<tr>
<td>@length:</td>
<td>If IBV_REREG_MR_CHANGE_TRANSLATION is set in flags, this field specifies the length of the virtual address to use in the new translation, otherwise, this parameter is ignored.</td>
</tr>
<tr>
<td>@access:</td>
<td>If IBV_REREG_MR_CHANGE_ACCESS is set in flags, this field specifies the new memory access rights, otherwise, this parameter is ignored. Could be one of the following: IBV_ACCESS_LOCAL_WRITE IBV_ACCESS_REMOTE_WRITE IBV_ACCESS_REMOTE_READ IBV_ACCESS_ALLOCATE_MR /* Let the library allocate the memory for the user, tries to get contiguous pages */</td>
</tr>
<tr>
<td>@attr:</td>
<td>Future extensions</td>
</tr>
</tbody>
</table>

ibv_rereg_mr returns 0 on success, or the value of an errno on failure (which indicates the error reason). In case of an error, the MR is in undefined state. The user needs to call ibv_dereg_mr in order to release it.
Please note that if the MR (Memory Region) is created as a Shared MR and a translation is requested, after the call, the MR is no longer a shared MR. Moreover, Re-registration of MRs that uses NVIDIA PeerDirect™ technology are not supported.

**Memory Window**

Memory Window allows the application to have a more flexible control over remote access to its memory. It is available only on physical functions/native machines. The two types of Memory Windows supported are: type 1 and type 2B.

Memory Windows are intended for situations where the application wants to:

- Grant and revoke remote access rights to a registered region in a dynamic fashion with less of a performance penalty
- Grant different remote access rights to different remote agents and/or grant those rights over different ranges within registered region

For further information, please refer to the InfiniBand specification document.

**Note**

Memory Windows API cannot co-work with peer memory clients (PeerDirect).

**Query Capabilities**

Memory Windows are available if and only the hardware supports it. To verify whether Memory Windows are available, run `ibv_query_device`.

For example:

```c
struct ibv_device_attr device_attr = {comp_mask = IBV_DEVICE_ATTR_RESERVED - 1};
ibv_query_device(context, & device_attr);
if (device_attr.exp_device_cap_flags & IBV_DEVICE_MEM_WINDOW ||
    device_attr.exp_device_cap_flags & IBV_DEVICE_MW_TYPE_2B) {
```
Memory Window Allocation

Allocating memory window is done by calling the `ibv_alloc_mw` verb.

```c
/* Memory window is supported */

int type_mw = IBV_MW_TYPE_2; / IBV_MW_TYPE_1
int mw = ibv_alloc_mw(pd, type_mw);
```

Binding Memory Windows

After being allocated, memory window should be bound to a registered memory region. Memory Region should have been registered using the `IBV_ACCESS_MW_BIND` access flag.

For further information on how to bind memory windows, please see `rdma-core man page`.

Invalidating Memory Window

Before rebinding Memory Window type 2, it must be invalidated using `ibv_post_send` - see `here`.

Deallocating Memory Window

Deallocating memory window is done using the `ibv_dealloc_mw` verb.

```c
ibv_dealloc_mw(mw);
```
User-Mode Memory Registration (UMR)

User-mode Memory Registration (UMR) is a fast registration mode which uses send queue. The UMR support enables the usage of RDMA operations and scatters the data at the remote side through the definition of appropriate memory keys on the remote side.

UMR enables the user to:

- Create indirect memory keys from previously registered memory regions, including creation of KLM's from previous KLM's. There are not data alignment or length restrictions associated with the memory regions used to define the new KLM's.
- Create memory regions, which support the definition of regular non-contiguous memory regions.

On-Demand-Paging (ODP)

On-Demand-Paging (ODP) is a technique to alleviate much of the shortcomings of memory registration. Applications no longer need to pin down the underlying physical pages of the address space, and track the validity of the mappings. Rather, the HCA requests the latest translations from the OS when pages are not present, and the OS invalidates translations which are no longer valid due to either non-present pages or mapping changes. ODP does not support contiguous pages.

ODP can be further divided into 2 subclasses: Explicit and Implicit ODP.

- Explicit ODP
  
  In Explicit ODP, applications still register memory buffers for communication, but this operation is used to define access control for IO rather than pin-down the pages. ODP Memory Region (MR) does not need to have valid mappings at registration time.

- Implicit ODP
  
  In Implicit ODP, applications are provided with a special memory key that represents their complete address space. This all IO accesses referencing this key (subject to the access rights associated with the key) does not need to register any virtual address range.
Query Capabilities

On-Demand Paging is available if both the hardware and the kernel support it. To verify whether ODP is supported, run `ibv_query_device`.

For further information, please refer to the `ibv_query_device` manual page.

Registering ODP Explicit and Implicit MR

ODP Explicit MR is registered after allocating the necessary resources (e.g. PD, buffer), while ODP implicit MR registration provides an implicit lkey that represents the complete address space.

For further information, please refer to the `ibv_reg_mr` manual page.

De-registering ODP MR

ODP MR is deregistered the same way a regular MR is deregistered:

```
ibv_dereg_mr(mr);
```

Advice MR Verb

The driver can pre-fetch a given range of pages and map them for access from the HCA. The advice MR verb is applicable for ODP MRs only.

For further information, please refer to the `ibv_advise_mr` manual page.

ODP Statistics
To aid in debugging and performance measurements and tuning, ODP support includes an extensive set of statistics.

For further information, please refer to rdma-statistics manual page.

**Inline-Receive**

The HCA may write received data to the Receive CQE. Inline-Receive saves PCIe Read transaction since the HCA does not need to read the scatter list. Therefore, it improves performance in case of short receive-messages.

On poll CQ, the driver copies the received data from CQE to the user's buffers.

Inline-Receive is enabled by default and is transparent to the user application. To disable it globally, set MLX5_SCATTER_TO_CQE environment variable to the value of 0. Otherwise, disable it on a specific QP using mlx5dv_create_qp() with MLX5DV_QP_CREATE_DISABLE_SCATTER_TO_CQE.

For further information, please refer to the manual page of mlx5dv_create_qp().

**NVIDIA PeerDirect**

NVIDIA PeerDirect™ uses an API between IB CORE and peer memory clients, (e.g. GPU cards) to provide access to an HCA to read/write peer memory for data buffers. As a result, it allows RDMA-based (over InfiniBand/RoCE) application to use peer device computing power, and RDMA interconnect at the same time without copying the data between the P2P devices.

For example, PeerDirect is being used for GPUDirect RDMA.

Detailed description for that API exists under MLNX OFED installation, please see docs/readme_and_user_manual/PEER_MEMORY_API.txt.

**PeerDirect Async**

Mellanox PeerDirect Async sub-system gives PeerDirect hardware devices, such as GPU cards, dedicated AS accelerators, and so on, the ability to take control over HCA in critical
path offloading CPU. To achieve this, there is a set of verb calls and structures providing application with abstract description of operation sequences intended to be executed by peer device.

**Relaxed Ordering (RSYNC)**

> **Note**
> This feature is only supported on ConnectX-5 adapter cards and above.

In GPU systems with relaxed ordering, RSYNC callback will be invoked to ensure memory consistency. The registration and implementation of the callback will be done using an external module provided by the system vendor. Loading the module will register the callback in MLNX_OFED to be used later to guarantee memory operations order.

**CPU Overhead Distribution**

When creating a CQ using the `ibv_create_cq()` API, a "comp_vector" argument is sent. If the value set for this argument is 0, while the CPU core executing this verb is not equal to zero, the driver assigns a completion EQ with the least CQs reporting to it. This method is used to distribute CQs amongst available completions EQ. To assign a CQ to a specific EQ, the EQ needs to be specified in the `comp_vector` argument.

**Out-of-Order (OOO) Data Placement**

> **Note**
> This feature is only supported on:

- ConnectX-5 adapter cards and above
- RC and XRC QPs
Overview

In certain fabric configurations, InfiniBand packets for a given QP may take up different paths in a network from source to destination. This results into packets being received in an out-of-order manner. These packets can now be handled instead of being dropped, in order to avoid retransmission, by:

- Achieving better network utilization
- Decreasing latency

Data will be placed into host memory in an out-of-order manner when out-of-order messages are received.

For information on how to set up out-of-order processing by the QP, please refer to HowTo Configure Adaptive Routing and SHIELD Community post.

IB Router

IB router provides the ability to send traffic between two or more IB subnets thereby potentially expanding the size of the network to over 40k end-ports, enabling separation and fault resilience between islands and IB subnets, and enabling connection to different topologies used by different subnets.

The forwarding between the IB subnets is performed using GRH lookup. The IB router's basic functionality includes:

- Removal of current L2 LRH (local routing header)
- Routing
- table lookup – using GID from GRH
- Building new LRH according to the destination according to the routing table
The DLID in the new LRH is built using simplified GID-to-LID mapping (where LID = 16 LSB bits of GID) thereby not requiring to send for ARP query/lookup.

**Local Unicast GID Format**

![Local Unicast GID Format Diagram]

For this to work, the SM allocates an alias GID for each host in the fabric where the alias GID = {subnet prefix[127:64], reserved[63:16], LID[15:0]. Hosts should use alias GIDs in order to transmit traffic to peers on remote subnets.

**Host-to-Host IB Router Unicast Flow**

- For information on the architecture and functionality of IB Router, refer to [IB Router Architecture and Functionality Community post](#).
- For information on IB Router configuration, refer to [HowTo Configure IB Routers Community post](#).
MAD Congestion Control

The SA Management Datagrams (MAD) are General Management Packets (GMP) used to communicate with the SA entity within the InfiniBand subnet. SA is normally part of the subnet manager, and it is contained within a single active instance. Therefore, congestion on the SA communication level may occur.

Congestion control is done by allowing max_outstanding MADs only, where outstanding MAD means that it has no response yet. It also holds a FIFO queue that holds the SA MADs that their sending is delayed due to max_outstanding overflow.

The length of the queue is queue_size and meant to limit the FIFO growth beyond the machine memory capabilities. When the FIFO is full, SA MADs will be dropped, and the drops counter will increment accordingly.

When time expires (time_sa_mad) for a MAD in the queue, it will be removed from the queue and the user will be notified of the item expiration.

This feature is implemented per CA port.

The SA MAD congestion control values are configurable using the following sysfs entries:

```bash
/sys/class/infiniband/mlx5_0/mad_sa_cc/
  1
    drops
    max_outstanding
    queue_size
    time_sa_mad
  2
    drops
    max_outstanding
    queue_size
    time_sa_mad
```

To print the current value:
To

change the current value:

```bash
echo 32 > /sys/class/infiniband/mlx5_0/mad_sa_cc/1/max_outstanding
cat /sys/class/infiniband/mlx5_0/mad_sa_cc/1/max_outstanding
32
```

To

reset the drops counter:

```bash
echo 0 > /sys/class/infiniband/mlx5_0/mad_sa_cc/1/drops
```

Parameters' Valid Ranges

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Range</th>
<th>Default Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MIN</td>
<td>MAX</td>
</tr>
<tr>
<td>max_outstanding</td>
<td>1</td>
<td>2^20</td>
</tr>
<tr>
<td>queue_size</td>
<td>16</td>
<td>2^20</td>
</tr>
<tr>
<td>time_sa_mad</td>
<td>1 milliseconds</td>
<td>10000</td>
</tr>
</tbody>
</table>

Storage Protocols

There are several storage protocols that use the advantage of InfiniBand and RDMA for performance reasons (high throughput, low latency and low CPU utilization). In this chapter we will discuss the following protocols:
• **SCSI RDMA Protocol (SRP)** is designed to take full advantage of the protocol off-load and RDMA features provided by the InfiniBand architecture.

• **iSCSI Extensions for RDMA (iSER)** is an extension of the data transfer model of iSCSI, a storage networking standard for TCP/IP. It uses the iSCSI components while taking the advantage of the RDMA protocol suite. ISER is implemented on various storage targets such as TGT, LIO, SCST and out of scope of this manual.

For various ISER targets configuration steps, troubleshooting and debugging, as well as other implementation of storage protocols over RDMA (such as Ceph over RDMA, nbdX and more) refer to Storage Solutions on the Community website.

• **Lustre** is an open-source, parallel distributed file system, generally used for large-scale cluster computing that supports many requirements of leadership class HPC simulation environments.

• **NVM Express™ over Fabrics (NVME-oF)**
  
  ○ NVME-oF is a technology specification for networking storage designed to enable NVMe message-based commands to transfer data between a host computer and a target solid-state storage device or system over a network such as Ethernet, Fibre Channel, and InfiniBand. Tunneling NVMe commands through an RDMA fabric provides a high throughput and a low latency. This is an alternative to the SCSi based storage networking protocols.

  ○ NVME-oF Target Offload is an implementation of the new NVME-oF standard Target (server) side in hardware. Starting from ConnectX-5 family cards, all regular IO requests can be processed by the HCA, with the HCA sending IO requests directly to a real NVMe PCI device, using peer-to-peer PCI communications. This means that excluding connection management and error flows, no CPU utilization will be observed during NVME-oF traffic.

For further information, please refer to Storage Solutions on the Community website (enterprise-support.nvidia.com/s/).

**SRP - SCSI RDMA Protocol**

The SCSI RDMA Protocol (SRP) is designed to take full advantage of the protocol offload and RDMA features provided by the InfiniBand architecture. SRP allows a large body of SCSI software to be readily used on InfiniBand architecture. The SRP Initiator controls the
connection to an SRP Target in order to provide access to remote storage devices across an InfiniBand fabric. The kSRP Target resides in an IO unit and provides storage services.

**SRP Initiator**

This SRP Initiator is based on open source from OpenFabrics ([www.openfabrics.org](http://www.openfabrics.org)) that implements the SCSI RDMA Protocol-2 (SRP-2). SRP-2 is described in Document # T10/1524-D available from [http://www.t10.org](http://www.t10.org).

The SRP Initiator supports

- Basic SCSI Primary Commands -3 (SPC-3)
- Basic SCSI Block Commands -2 (SBC-2)
- Basic functionality, task management and limited error handling

**Note**

This package, however, does not include an SRP Target.

**Loading SRP Initiator**

*To load the SRP module either:*

- Execute the `modprobe ib_srp` command after the OFED driver is up.

  or

1. Change the value of `SRP_LOAD` in `/etc/infiniband/openib.conf` to “yes”.

2. Run `/etc/init.d/openibd` restart for the changes to take effect.
**Note**

When loading the `ib_srp` module, it is possible to set the module parameter `srp_sg_tablesize`. This is the maximum number of gather/scatter entries per I/O (default: 12).

---

**SRP Module Parameters**

When loading the SRP module, the following parameters can be set (viewable by the "modinfo ib_srp" command):

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cmd_sg_entries</code></td>
<td>Default number of gather/scatter entries in the SRP command (default is 12, max 255)</td>
</tr>
<tr>
<td><code>allow_ext_sg</code></td>
<td>Default behavior when there are more than <code>cmd_sg_entries</code> S/G entries after mapping; fails the request when false (default false)</td>
</tr>
<tr>
<td><code>topspin_workarounds</code></td>
<td>Enable workarounds for Topspin/Cisco SRP target bugs</td>
</tr>
<tr>
<td><code>reconnect_delay</code></td>
<td>Time between successive reconnect attempts. Time between successive reconnect attempts of SRP initiator to a disconnected target until dev_loss_tmo timer expires (if enabled), after that the SCSI target will be removed</td>
</tr>
<tr>
<td><code>fast_io_fail_tmo</code></td>
<td>Number of seconds between the observation of a transport layer error and failing all I/O. Increasing this timeout allows more tolerance to transport errors, however, doing so increases the total failover time in case of serious transport failure. Note: <code>fast_io_fail_tmo</code> value must be smaller than the value of <code>reconnect_delay</code></td>
</tr>
<tr>
<td><code>dev_loss_tmo</code></td>
<td>Maximum number of seconds that the SRP transport should insulate transport layer errors. After this time has been exceeded the SCSI target is removed. Normally it is advised to set this to -1 (disabled) which will never remove the scsi_host. In deployments where different SRP targets are connected and disconnected frequently, it may be required to enable this</td>
</tr>
</tbody>
</table>
timeout in order to clean old scsi_hosts representing targets that no longer exists

Constraints between parameters:

- `dev_loss_tmo`, `fast_io_fail_tmo`, `reconnect_delay` cannot be all disabled or negative values.
- `reconnect_delay` must be positive number.
- `fast_io_fail_tmo` must be smaller than SCSI block device timeout.
- `fast_io_fail_tmo` must be smaller than `dev_loss_tmo`.

**SRP Remote Ports Parameters**

Several SRP remote ports parameters are modifiable online on existing connection.

- **To modify `dev_loss_tmo` to 600 seconds:**
  
  ```
  echo 600 > /sys/class/srp_remote_ports/port-xxx/dev_loss_tmo
  ```

- **To modify `fast_io_fail_tmo` to 15 seconds:**
  
  ```
  echo 15 > /sys/class/srp_remote_ports/port-xxx/fast_io_fail_tmo
  ```

- **To modify `reconnect_delay` to 10 seconds:**
  
  ```
  echo 20 > /sys/class/srp_remote_ports/port-xxx/reconnect_delay
  ```

**Manually Establishing an SRP Connection**
The following steps describe how to manually load an SRP connection between the Initiator and an SRP Target. "Automatic Discovery and Connection to Targets" section explains how to do this automatically.

- Make sure that the ib_srp module is loaded, the SRP Initiator is reachable by the SRP Target, and that an SM is running.

- To establish a connection with an SRP Target and create an SRP (SCSI) device for that target under /dev, use the following command:

```bash
echo -n id_ext=[GUID value],ioc_guid=[GUID value],dgid=[port GID value],
pkey=ffff,service_id=[service[0] value] > \
/sys/class/infiniband_srp/srp-mlx[hca number]-[port number]/add_target
```

See “SRP Tools - ibsrpdm, srp_daemon and srpd Service Script” section for instructions on how the parameters in this echo command may be obtained.

**Notes:**

- Execution of the above “echo” command may take some time

- The SM must be running while the command executes

- It is possible to include additional parameters in the echo command:
  - `max_cmd_per_lun` - Default: 62
  - `max_sect` (short for max_sectors) - sets the request size of a command
  - `io_class` - Default: 0x100 as in rev 16A of the specification (In rev 10 the default was 0xff00)
  - `tl_retry_count` - a number in the range 2..7 specifying the IB RC retry count. Default: 2
  - `comp_vector`, a number in the range 0..n-1 specifying the MSI-X completion vector. Some HCA's allocate multiple (n) MSI-X vectors per HCA port. If the IRQ affinity masks of these interrupts have been configured such that each MSI-X interrupt is handled by a different CPU then the comp_vector parameter can be used to spread the SRP completion workload over multiple CPU's.
- cmd_sg_entries, a number in the range 1..255 that specifies the maximum number of data buffer descriptors stored in the SRP_CMD information unit itself. With allow_ext_sg=0 the parameter cmd_sg_entries defines the maximum S/G list length for a single SRP_CMD, and commands whose S/G list length exceeds this limit after S/G list collapsing will fail.

- initiator_ext - see "Multiple Connections from Initiator InfiniBand Port to the Target" section.

- To list the new SCSI devices that have been added by the echo command, you may use either of the following two methods:
  - Execute “fdisk -l”. This command lists all devices; the new devices are included in this listing.
  - Execute “dmesg” or look at /var/log/messages to find messages with the names of the new devices.

### SRP sysfs Parameters

Interface for making ib_srp connect to a new target. One can request ib_srp to connect to a new target by writing a comma-separated list of login parameters to this sysfs attribute. The supported parameters are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id_ext</td>
<td>A 16-digit hexadecimal number specifying the eight byte identifier extension in the 16-byte SRP target port identifier. The target port identifier is sent by ib_srp to the target in the SRP_LOGIN_REQ request.</td>
</tr>
<tr>
<td>ioc_guid</td>
<td>A 16-digit hexadecimal number specifying the eight byte I/O controller GUID portion of the 16-byte target port identifier.</td>
</tr>
<tr>
<td>dgid</td>
<td>A 32-digit hexadecimal number specifying the destination GID.</td>
</tr>
<tr>
<td>pkey</td>
<td>A four-digit hexadecimal number specifying the InfiniBand partition key.</td>
</tr>
<tr>
<td>service_id</td>
<td>A 16-digit hexadecimal number specifying the InfiniBand service ID used to establish communication with the SRP target. How to find out the value of the service ID is specified in the documentation of the SRP target.</td>
</tr>
<tr>
<td>max_sect</td>
<td>A decimal number specifying the maximum number of 512-byte sectors to be transferred via a single SCSI command.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>max_cmd_per_lun</td>
<td>A decimal number specifying the maximum number of outstanding commands for a single LUN.</td>
</tr>
<tr>
<td>io_class</td>
<td>A hexadecimal number specifying the SRP I/O class. Must be either 0xff00 (rev 10) or 0x0100 (rev 16a). The I/O class defines the format of the SRP initiator and target port identifiers.</td>
</tr>
<tr>
<td>initiator_ext</td>
<td>A 16-digit hexadecimal number specifying the identifier extension portion of the SRP initiator port identifier. This data is sent by the initiator to the target in the SRP_LOGIN_REQ request.</td>
</tr>
<tr>
<td>cmd_sg_entries</td>
<td>A number in the range 1..255 that specifies the maximum number of data buffer descriptors stored in the SRP_CMD information unit itself. With allow_ext_sg=0 the parameter cmd_sg_entries defines the maximum S/G list length for a single SRP_CMD, and commands whose S/G list length exceeds this limit after S/G list collapsing will fail.</td>
</tr>
<tr>
<td>allow_ext_sg</td>
<td>Whether ib_srp is allowed to include a partial memory descriptor list in an SRP_CMD instead of the entire list. If a partial memory descriptor list has been included in an SRP_CMD the remaining memory descriptors are communicated from initiator to target via an additional RDMA transfer. Setting allow_ext_sg to 1 increases the maximum amount of data that can be transferred between initiator and target via a single SCSI command. Since not all SRP target implementations support partial memory descriptor lists the default value for this option is 0.</td>
</tr>
<tr>
<td>sg_tablesize</td>
<td>A number in the range 1..2048 specifying the maximum S/G list length the SCSI layer is allowed to pass to ib_srp. Specifying a value that exceeds cmd_sg_entries is only safe with partial memory descriptor list support enabled (allow_ext_sg=1).</td>
</tr>
<tr>
<td>comp_vector</td>
<td>A number in the range 0..n-1 specifying the MSI-X completion vector. Some HCA's allocate multiple (n) MSI-X vectors per HCA port. If the IRQ affinity masks of these interrupts have been configured such that each MSI-X interrupt is handled by a different CPU then the comp_vector parameter can be used to spread the SRP completion workload over multiple CPU's.</td>
</tr>
<tr>
<td>tl_retry_count</td>
<td>A number in the range 2..7 specifying the IB RC retry count.</td>
</tr>
</tbody>
</table>

**SRP Tools - ibsrpdm, srp_daemon and srpd Service Script**

DOCA Drivers
The OFED distribution provides two utilities: ibsrpdm and srp_daemon:

- They detect targets on the fabric reachable by the Initiator (Step 1)
- Output target attributes in a format suitable for use in the above “echo” command (Step 2)
- A service script srpd which may be started at stack startup

The utilities can be found under /usr/sbin/, and are part of the srptools RPM that may be installed using the OFED installation. Detailed information regarding the various options for these utilities are provided by their man pages.

Below, several usage scenarios for these utilities are presented.

**ibsrpdm**

ibsrpdm has the following tasks:

1. Detecting reachable targets.

   1. To detect all targets reachable by the SRP initiator via the default umad device (/sys/class/infiniband_mad/umad0), execute the following command:

   ```
   ibsrpdm
   ```

   This command will result into readable output information on each SRP Target detected. Sample:

   ```
   IO Unit Info:
   port LID: 0103
   port GID: fe800000000000000002c90200402bd5
   change ID: 0002
   max controllers: 0x10
   controller[ 1]
     GUID: 0002c90200402bd4
     vendor ID: 0002c9
     device ID: 005a44
     IO class: 0100
   ```
2. To detect all the SRP Targets reachable by the SRP Initiator via another umad device, use the following command:

```
ibsrpdm -d <umad device>
```

2. Assisting in SRP connection creation.

1. To generate an output suitable for utilization in the “echo” command in “Manually Establishing an SRP Connection” section, add the ‘-c’ option to ibsrpdm:

```
ibsrpdm -c
```

Sample output:

```
id_ext=200400A0B81146A1,ioc_guid=0002c90200402bd4,dgid=fe8000000000000000002c90200402bd5,pkey=ffff,service_id=200400a0b81146a1
```

2. To establish a connection with an SRP Target using the output from the ‘ibsrpdm -c’ example above, execute the following command:

```
echo -n id_ext=200400A0B81146A1,ioc_guid=0002c90200402bd4,dgid=fe8000000000000000002c90200402bd5,pkey=ffff,service_id=200400a0b81146a1 > /sys/class/infiniband_srpsrp-mlx5_0-1/add_target
```

The SRP connection should now be up; the newly created SCSI devices should appear in the listing obtained from the ‘fdisk -l’ command.
3. Discover reachable SRP Targets given an InfiniBand HCA name and port, rather than by just running `/sys/class/infiniband_mad/umad<N>` where `<N>` is a digit.

**srpd**

The srpd service script allows automatic activation and termination of the srp_daemon utility on all system live InfiniBand ports.

**srp_daemon**

srp_daemon utility is based on ibsrpdm and extends its functionality. In addition to the ibsrpdm functionality described above, srp_daemon can:

- Establish an SRP connection by itself (without the need to issue the “echo” command described in “Manually Establishing an SRP Connection” section)
- Continue running in background, detecting new targets and establishing SRP connections with them (daemon mode)
- Discover reachable SRP Targets given an infiniband HCA name and port, rather than just by `/dev/umad<N>` where `<N>` is a digit
- Enable High Availability operation (together with Device-Mapper Multipath)
- Have a configuration file that determines the targets to connect to:
  1. srp_daemon commands equivalent to ibsrpdm:

```
"srp_daemon -a -o" is equivalent to "ibsrpdm"
"srp_daemon -c -a -o" is equivalent to "ibsrpdm -c"
```

**Note:** These srp_daemon commands can behave differently than the equivalent ibsrpdm command when `/etc/srp_daemon.conf` is not empty.

2. srp_daemon extensions to ibsrpdm.
- To discover SRP Targets reachable from the HCA device `<InfiniBand HCA name>` and the port `<port num>`, (and to generate output suitable for `echo`), execute:

```
host1# srp_daemon -c -a -o -i `<InfiniBand HCA name>` -p `<port number>`
```

**Note:** To obtain the list of InfiniBand HCA device names, you can either use the `ibstat` tool or run `ls /sys/class/infiniband`.

- To both discover the SRP Targets and establish connections with them, just add the `-e` option to the above command.

- Executing `srp_daemon` over a port without the `-a` option will only display the reachable targets via the port and to which the initiator is not connected. If executing with the `-e` option it is better to omit `-a`.

- It is recommended to use the `-n` option. This option adds the `initiator_ext` to the connecting string (see "Multiple Connections from Initiator InfiniBand Port to the Target" section).

- `srp_daemon` has a configuration file that can be set, where the default is `/etc/srp_daemon.conf`. Use the `-f` to supply a different configuration file that configures the targets `srp_daemon` is allowed to connect to. The configuration file can also be used to set values for additional parameters (e.g., `max_cmd_per_lun`, `max_sect`).

- A continuous background (daemon) operation, providing an automatic ongoing detection and connection capability. See "Automatic Discovery and Connection to Targets" section.

---

**Automatic Discovery and Connection to Targets**

- Make sure the `ib_srp` module is loaded, the SRP Initiator can reach an SRP Target, and that an SM is running.

- To connect to all the existing Targets in the fabric, run “`srp_daemon -e -o`”. This utility will scan the fabric once, connect to every Target it detects, and then exit.
To connect to all the existing Targets in the fabric and to connect to new targets that will join the fabric, execute srp_daemon -e. This utility continues to execute until it is either killed by the user or encounters connection errors (such as no SM in the fabric).

To execute SRP daemon as a daemon on all the ports:

- `srp_daemon.sh` (found under `/usr/sbin`). `srp_daemon.sh` sends its log to `/var/log/srp_daemon.log`.
- Start the srpd service script, run service srpd start

For the changes in openib.conf to take effect, run:

```
/etc/init.d/openibd restart
```

**Multiple Connections from Initiator InfiniBand Port to the Target**

Some system configurations may need multiple SRP connections from the SRP Initiator to the same SRP Target: to the same Target IB port, or to different IB ports on the same Target HCA.

In case of a single Target IB port, i.e., SRP connections use the same path, the configuration is enabled using a different `initiator_ext` value for each SRP connection. The `initiator_ext` value is a 16-hexadecimal-digit value specified in the connection command.
Also in case of two physical connections (i.e., network paths) from a single initiator IB port to two different IB ports on the same Target HCA, there is need for a different initiator_ext value on each path. The conventions is to use the Target port GUID as the initiator_ext value for the relevant path.

If you use srp_daemon with -n flag, it automatically assigns initiator_ext values according to this convention. For example:

```
id_ext=200500A0B81146A1,ioc_guid=0002c90200402bec,\
dgid=fe80000000000000000000002c90200402bed,pkey=ffff,\
service_id=200500a0b81146a1,initiator_ext=ed2b400002c90200
```

**Notes:**

- It is recommended to use the -n flag for all srp_daemon invocations.
- ibsrpd does not have a corresponding option.
- srp_daemon.sh always uses the -n option (whether invoked manually by the user, or automatically at startup by setting SRP_DAEMON_ENABLE to yes).

**High Availability (HA)**

High Availability works using the Device-Mapper (DM) multipath and the SRP daemon. Each initiator is connected to the same target from several ports/HCAs. The DM multipath is responsible for joining together different paths to the same target and for failover between paths when one of them goes offline. Multipath will be executed on newly joined SCSI devices.

Each initiator should execute several instances of the SRP daemon, one for each port. At startup, each SRP daemon detects the SRP Targets in the fabric and sends requests to the ib_srp module to connect to each of them. These SRP daemons also detect targets that subsequently join the fabric, and send the ib_srp module requests to connect to them as well.

**Operation**
When a path (from port1) to a target fails, the ib_srp module starts an error recovery process. If this process gets to the reset_host stage and there is no path to the target from this port, ib_srp will remove this scsi_host. After the scsi_host is removed, multipath switches to another path to this target (from another port/HCA).

When the failed path recovers, it will be detected by the SRP daemon. The SRP daemon will then request ib_srp to connect to this target. Once the connection is up, there will be a new scsi_host for this target. Multipath will be executed on the devices of this host, returning to the original state (prior to the failed path).

**Manual Activation of High Availability**

Initialization - execute after each boot of the driver:

1. Execute `modprobe dm-multipath`

2. Execute `modprobe ib-srp`

3. Make sure you have created file `/etc/udev/rules.d/91-srp.rules` as described above

4. Execute for each port and each HCA:

   ```
   srp_daemon -c -e -R 300 -i <InfiniBand HCA name> -p <port number>
   ```

   This step can be performed by executing `srp_daemon.sh`, which sends its log to `/var/log/srp_daemon.log`.

Now it is possible to access the SRP LUNs on `/dev/mapper/`.

ℹ️ **Note**

It is possible for regular (non-SRP) LUNs to also be present; the SRP LUNs may be identified by their names. You can configure the `/etc/multipath.conf` file to change multipath behavior.
**Note**

It is also possible that the SRP LUNs will not appear under `/dev/mapper/`. This can occur if the SRP LUNs are in the black-list of multipath. Edit the ‘blacklist’ section in `/etc/multipath.conf` and make sure the SRP LUNs are not blacklisted.

---

**Automatic Activation of High Availability**

- Start srpd service, run:

  ```
  service srpd start
  ```

- From the next loading of the driver it will be possible to access the SRP LUNs on `/dev/mapper/`

**Note**

It is possible that regular (not SRP) LUNs are also present. SRP LUNs may be identified by their name.

- It is possible to see the output of the SRP daemon in `/var/log/srp_daemon.log`

---

**Shutting Down SRP**
SRP can be shutdown by using “rmmod ib_srp”, or by stopping the OFED driver (“/etc/init.d/openibd stop”), or as a by-product of a complete system shutdown.

Prior to shutting down SRP, remove all references to it. The actions you need to take depend on the way SRP was loaded. There are three cases:

1. Without High Availability

   When working without High Availability, you should unmount the SRP partitions that were mounted prior to shutting down SRP.

2. After Manual Activation of High Availability

   If you manually activated SRP High Availability, perform the following steps:

   1. Unmount all SRP partitions that were mounted.
   2. Stop service srpd (Kill the SRP daemon instances).
   3. Make sure there are no multipath instances running. If there are multiple instances, wait for them to end or kill them.
   4. Run: multipath -F

3. After Automatic Activation of High Availability

   If SRP High Availability was automatically activated, SRP shutdown must be part of the driver shutdown (“/etc/init.d/openibd stop”) which performs Steps 2-4 of case b above. However, you still have to unmount all SRP partitions that were mounted before driver shutdown.

**iSCSI Extensions for RDMA (iSER)**

iSCSI Extensions for RDMA (iSER) extends the iSCSI protocol to RDMA. It permits data to be transferred directly into and out of SCSI buffers without intermediate data copies.

iSER uses the RDMA protocol suite to supply higher bandwidth for block storage transfers (zero time copy behavior). To that fact, it eliminates the TCP/IP processing overhead while preserving the compatibility with iSCSI protocol.
There are three target implementation of ISER:

- Linux SCSI target framework (tgt)
- Linux-IO target (LIO)
- Generic SCSI target subsystem for Linux (SCST)

Each one of those targets can work in TCP or iSER transport modes.

iSER also supports RoCE without any additional configuration required. To bond the RoCE interfaces, set the fail_over_mac option in the bonding driver (see "Bonding IPoIB").

RDMA/RoCE is located below the iSER block on the network stack. In order to run iSER, the RDMA layer should be configured and validated (over Ethernet or InfiniBand). For troubleshooting RDMA, please refer to "HowTo Enable, Verify and Troubleshoot RDMA" on the Community website.
iSER Initiator

The iSER initiator is controlled through the iSCSI interface available from the iscsi-initiator-utils package.

To discover and log into iSCSI targets, as well as access and manage the open-iscsi database use the iscsiadm utility, a command-line tool.

To enable iSER as a transport protocol use "$ iser" as a parameter of the iscsiadm command.

Example for discovering and connecting targets over iSER:

```
iscsiadm -m discovery -o new -o old -t st -I iser -p <ip:port> -l
```

Note that the target implementation (e.g. LIO, SCST, TGT) does not affect the initiator process and configuration.

iSER Targets

Note

Setting the iSER target is out of scope of this manual. For guidelines of how to do so, please refer to the relevant target documentation (e.g. stgt, targetcli).

Targets settings such as timeouts and retries are set the same as any other iSCSI targets.
If targets are set to auto connect on boot, and targets are unreachable, it may take a long time to continue the boot process if timeouts and max retries are set too high.

For various configuration, troubleshooting and debugging examples, refer to Storage Solutions on the Community website.

**Lustre**

Lustre is an open-source, parallel distributed file system, generally used for large-scale cluster computing that supports many requirements of leadership class HPC simulation environments.

Lustre Compilation for MLNX_OFED:

*Note*

This procedure applies to RHEL/SLES OSs supported by Lustre. For further information, please refer to Lustre Release Notes.

➤ *To compile Lustre version 2.4.0 and higher:*

```
$ ./configure --with-o2ib=/usr/src/ofa_kernel/default/
$ make rpms
```

```
$ EXTRA_LNET_INCLUDE="-I/usr/src/ofa_kernel/default/include/ -include /usr/src/ofa_kernel/default/include/linux/compat-2.6.h" ./configure --with-o2ib=/usr/src/ofa_kernel/default/
```
For full installation example, refer to HowTo Install NVIDIA OFED driver for Lustre Community post.

**NVME-oF - NVM Express over Fabrics**

**NVME-oF**

NVME-oF enables NVMe message-based commands to transfer data between a host computer and a target solid-state storage device or system over a network such as Ethernet, Fibre Channel, and InfiniBand. Tunneling NVMe commands through an RDMA fabric provides a high throughput and a low latency.

For information on how to configure NVME-oF, please refer to the HowTo Configure NVMe over Fabrics Community post.

**Note**

The --with-nvmf installation option should not be specified, if nvme-tcp kernel module is used. In this case, the native Inbox nvme-tcp kernel module will be loaded.

**NVME-oF Target Offload**

**Note**

This feature is only supported for ConnectX-5 adapter cards family and above.
NVME-oF Target Offload is an implementation of the new NVME-oF standard Target (server) side in hardware. Starting from ConnectX-5 family cards, all regular IO requests can be processed by the HCA, with the HCA sending IO requests directly to a real NVMe PCI device, using peer-to-peer PCI communications. This means that excluding connection management and error flows, no CPU utilization will be observed during NVME-oF traffic.

- For instructions on how to configure NVME-oF target offload, refer to HowTo Configure NVME-oF Target Offload Community post.

- For instructions on how to verify that NVME-oF target offload is working properly, refer to Simple NVMe-oF Target Offload Benchmark Community post.

Virtualization

The chapter contains the following sections:

- **Single Root IO Virtualization (SR-IOV)**
- **Enabling Paravirtualization**
- **VXLAN Hardware Stateless Offloads**
- **Q-in-Q Encapsulation per VF in Linux (VST)**
- **802.1Q Double-Tagging**
- **Scalable Functions**

**Single Root IO Virtualization (SR-IOV)**

Single Root IO Virtualization (SR-IOV) is a technology that allows a physical PCIe device to present itself multiple times through the PCIe bus. This technology enables multiple virtual instances of the device with separate resources. NVIDIA adapters are capable of exposing up to 127 virtual instances (Virtual Functions (VFs) for each port in the NVIDIA ConnectX® family cards. These virtual functions can then be provisioned separately. Each VF can be seen as an additional device connected to the Physical Function. It shares the same resources with the Physical Function, and its number of ports equals those of the Physical Function.
SR-IOV is commonly used in conjunction with an SR-IOV enabled hypervisor to provide virtual machines direct hardware access to network resources hence increasing its performance.

In this chapter we will demonstrate setup and configuration of SR-IOV in a Red Hat Linux environment using ConnectX® VPI adapter cards.

**System Requirements**

To set up an SR-IOV environment, the following is required:

- MLNX_OFED Driver
- A serverblade with an SR-IOV-capable motherboard BIOS
- Hypervisor that supports SR-IOV such as: Red Hat Enterprise Linux Server Version 6
- NVIDIA ConnectX® VPI Adapter Card family with SR-IOV capability

**Setting Up SR-IOV**

Depending on your system, perform the steps below to set up your BIOS. The figures used in this section are for illustration purposes only. For further information, please refer to the appropriate BIOS User Manual:

1. Enable "SR-IOV" in the system BIOS.
2. Enable "Intel Virtualization Technology".

3. Install a hypervisor that supports SR-IOV.

4. Depending on your system, update the /boot/grub/grub.conf file to include a similar command line load parameter for the Linux kernel.

For example, to Intel systems, add:

```
default=0
timeout=5
splashimage=(hd0,0)/grub/splash.xpm.gz
hiddenmenu
title Red Hat Enterprise Linux Server (4.x.x)
    root (hd0,0)
    kernel /vmlinuz-4.x.x ro root=/dev/VolGroup00/LogVol00 rhgb quiet
    intel_iommu=on
    initrd /initrd-4.x.x.img
```

**Note:** Please make sure the parameter "intel_iommu=on" exists when updating the /boot/grub/grub.conf file, otherwise SR-IOV cannot be loaded.

Some OSs use /boot/grub2/grub.cfg file. If your server uses such file, please edit this file instead (add "intel_iommu=on" for the relevant menu entry at the end of the line that starts with "linux16").
Configuring SR-IOV (Ethernet)

To set SR-IOV in Ethernet mode, refer to HowTo Configure SR-IOV for ConnectX-4/ConnectX-5/ConnectX-6 with KVM (Ethernet) Community Post.

Configuring SR-IOV (InfiniBand)

1. Install the MLNX_OFED driver for Linux that supports SR-IOV.

2. Check if SR-IOV is enabled in the firmware.

   mlxconfig -d /dev/mst/mt4115_pciconf0 q

   Device #1:
   ----------
   Device type: Connect4
   PCI device: /dev/mst/mt4115_pciconf0
   Configurations: Current
   SRIOV_EN 1
   NUM_OF_VFS 8

   Note
   
   If needed, use mlxconfig to set the relevant fields:

   mlxconfig -d /dev/mst/mt4115_pciconf0 set SRIOV_EN=1 NUM_OF_VFS=16

3. Reboot the server.

4. Write to the sysfs file the number of Virtual Functions you need to create for the PF. You can use one of the following equivalent files:
- A standard Linux kernel generated file that is available in the new kernels.

Note: This file will be generated only if IOMMU is set in the grub.conf file (by adding \texttt{intel_iommu=on}, as seen in the fourth step under \textit{Setting Up SR-IOV}).

- A file generated by the mlx5_core driver with the same functionality as the kernel generated one.

Note: This file is used by old kernels that do not support the standard file. In such kernels, using \texttt{sriov_numvfs} results in the following error: \texttt{bash: echo: write error: Function not implemented}.

The following rules apply when writing to these files:

- If there are no VFs assigned, the number of VFs can be changed to any valid value (0 - max #VFs as set during FW burning)

- If there are VFs assigned to a VM, it is not possible to change the number of VFs

- If the administrator unloads the driver on the PF while there are no VFs assigned, the driver will unload and SRI-OV will be disabled

- If there are VFs assigned while the driver of the PF is unloaded, SR-IOV will not be disabled. This means that VFs will be visible on the VM. However, they will not be operational. This is applicable to OSs with kernels that use \texttt{pci_stub} and not \texttt{vfio}.

- The VF driver will discover this situation and will close its resources

- When the driver on the PF is reloaded, the VF becomes operational. The administrator of the VF will need to restart the driver in order to resume working with the VF.

5. Load the driver. To verify that the VFs were created. Run:

\begin{verbatim}
  echo [num_vfs] > /sys/class/infiniband/mlx5_0/device/sriov_numvfs
  echo [num_vfs] > /sys/class/infiniband/mlx5_0/device/mlx5_num_vfs
\end{verbatim}
6. Configure the VFs.

After VFs are created, 3 sysfs entries per VF are available under 
/sys/class/infiniband/mlx5_<PF INDEX>/device/sriov (shown below for VFs 0 to 2):

```
|-- 0
 |  |-- node
 |  |-- policy
 |  |-- port
|-- 1
 |  |-- node
 |  |-- policy
 |  |-- port
|-- 2
 |  |-- node
 |  |-- policy
 |  |-- port
```

For each Virtual Function, the following files are available:

- **Node** - Node’s GUID:

The user can set the node GUID by writing to the 
/sys/class/infiniband/<PF>/device/sriov/<index>/node file. The example below, shows how to set the node GUID for VF 0 of mlx5_0.
- Port - Port’s GUID:

The user can set the port GUID by writing to the
/sys/class/infiniband/<PF>/device/sriov/<index>/port file. The example below, shows
how to set the port GUID for VF 0 of mlx5_0.

```
    echo 00:11:22:33:44:55:1:0 > /sys/class/infiniband/mlx5_0/device/sriov/0/node
```

- Policy - The vport’s policy. The user can set the port GUID by writing to the
/sys/class/infiniband/<PF>/device/sriov/<index>/port file. The policy can be one of:

- Down - the VPort PortState remains 'Down'
- Up - if the current VPort PortState is 'Down', it is modified to 'Initialize'. In all other
  states, it is unmodified. The result is that the SM may bring the VPort up.
- Follow - follows the PortState of the physical port. If the PortState of the physical
  port is 'Active', then the VPort implements the 'Up' policy. Otherwise, the VPort
  PortState is 'Down'.

**Notes:**

- The policy of all the vports is initialized to “Down” after the PF driver is restarted
  except for VPort0 for which the policy is modified to 'Follow' by the PF driver.
- To see the VFs configuration, you must unbind and bind them or reboot the VMs if
  the VFs were assigned.

7. Make sure that OpenSM supports Virtualization (Virtualization must be enabled).

The /etc/opensm/opensm.conf file should contain the following line:

```
virt_enabled 2
```
**Note:** OpenSM and any other utility that uses SMP MADs (ibnetdiscover, sminfo, iblink-info, smpdump, ibqueryerr, ibdiagnet and smpquery) should run on the PF and not on the VFs. In case of multi PFs (multi-host), OpenSM should run on Host0.

**VF Initialization Note**

Since the same mlx5_core driver supports both Physical and Virtual Functions, once the Virtual Functions are created, the driver of the PF will attempt to initialize them so they will be available to the OS owning the PF. If you want to assign a Virtual Function to a VM, you need to make sure the VF is not used by the PF driver. If a VF is used, you should first unbind it before assigning to a VM.

➤ **To unbind a device use the following command:**

1. Get the full PCI address of the device.

   ```
   lspci -D
   ```

   Example:

   ```
   0000:09:00.2
   ```

2. Unbind the device.

   ```
   echo 0000:09:00.2 > /sys/bus/pci/drivers/mlx5_core/unbind
   ```

3. Bind the unbound VF.

   ```
   echo 0000:09:00.2 > /sys/bus/pci/drivers/mlx5_core/bind
   ```
PCI BDF Mapping of PFs and VF

PCI addresses are sequential for both of the PF and their VF. Assuming the card's PCI slot is 05:00 and it has 2 ports, the PFs PCI address will be 05:00.0 and 05:00.1.

Given 3 VF per PF, the VF PCI addresses will be:

- 05:00.2-4 for VF 0-2 of PF 0 (mlx5_0)
- 05:00.5-7 for VF 0-2 of PF 1 (mlx5_1)

Additional SR-IOV Configurations

Assigning a Virtual Function to a Virtual Machine

This section describes a mechanism for adding a SR-IOV VF to a Virtual Machine.

Assigning the SR-IOV Virtual Function to the Red Hat KVM VM Server

1. Run the virt-manager.
2. Double click on the virtual machine and open its Properties.
3. Go to Details  Add hardware  PCI host device.
4. Choose a NVIDIA virtual function according to its PCI device (e.g., 00:03.1)

5. If the Virtual Machine is up reboot it, otherwise start it.

6. Log into the virtual machine and verify that it recognizes the NVIDIA card. Run:

```
lspci | grep Mellanox
```

**Example:**

```
lspci | grep Mellanox
01:00.0 Infiniband controller: Mellanox Technologies MT28800 Family [ConnectX-5 Ex]
```

7. Add the device to the `/etc/sysconfig/network-scripts/ifcfg-ethX` configuration file. The MAC address for every virtual function is configured randomly, therefore it is not necessary to add it.
Ethernet Virtual Function Configuration when Running SR-IOV

SR-IOV Virtual function configuration can be done through Hypervisor iproute2/netlink tool, if present. Otherwise, it can be done via sysfs.

```
ip link set { dev DEVICE | group DEVGROUP } [ { up | down } ]
...
[ vf NUM [ mac LLADDR ] [ vlan VLANID [ qos VLAN-QOS ] ]
...
[ spoofchk { on | off } ] ]
...
```

sysfs configuration (ConnectX-4):

```
/sys/class/net/enp8s0f0/device/sriov/[VF]
```

```
+- [VF]
 | +- config
 | +- link_state
 | +- mac
 | +- mac_list
 | +- max_tx_rate
 | +- min_tx_rate
 | +- spoofcheck
 | +- stats
 | +- trunk
 | +- trust
 | +- vlan
```

VLAN Guest Tagging (VGT) and VLAN Switch Tagging (VST)

When running ETH ports on VGT, the ports may be configured to simply pass through packets as is from VFs (VLAN Guest Tagging), or the administrator may configure the Hypervisor to silently force packets to be associated with a VLAN/Qos (VLAN Switch Tagging).
In the latter case, untagged or priority-tagged outgoing packets from the guest will have the VLAN tag inserted, and incoming packets will have the VLAN tag removed.

The default behavior is VGT.

To configure VF VST mode, run:

```bash
ip link set dev <PF device> vf <NUM> vlan <vlan_id> [qos <qos>]
```

where:

- `NUM = 0..max-vf-num`
- `vlan_id = 0..4095`
- `qos = 0..7`

For example:

- `ip link set dev eth2 vf 2 vlan 10 qos 3` - sets VST mode for VF #2 belonging to PF eth2, with `vlan_id = 10` and `qos = 3`
- `ip link set dev eth2 vf 2 vlan 0` - sets mode for VF 2 back to VGT

### Additional Ethernet VF Configuration Options

- **Guest MAC configuration** - by default, guest MAC addresses are configured to be all zeroes. If the administrator wishes the guest to always start up with the same MAC, he/she should configure guest MACs before the guest driver comes up. The guest MAC may be configured by using:

  ```bash
  ip link set dev <PF device> vf <NUM> mac <LLADDR>
  ```

  For legacy and ConnectX-4 guests, which do not generate random MACs, the administrator should always configure their MAC addresses via IP link, as above.
- **Spoof checking** - Spoof checking is currently available only on upstream kernels newer than 3.1.

```
ip link set dev <PF device> vf <NUM> spoofchk [on | off]
```

- **Guest Link State**

```
ip link set dev <PF device> vf <UM> state [enable | disable | auto]
```

**Virtual Function Statistics**

Virtual function statistics can be queried via sysfs:

```
cat /sys/class/infiniband/mlx5_2/device/sriov/2/stats
```

```
tx_packets : 5011
tx_bytes : 4450870
tx_dropped : 0
rx_packets : 5003
rx_bytes : 4450222
rx_broadcast : 0
rx_multicast : 0
tax_broadcast : 0
tax_multicast : 8
rx_dropped : 0
```

**Mapping VFs to Ports**

- To view the VFs mapping to ports:

Use the ip link tool v2.6.34~3 and above.

```
ip link
```
Output:

```
61: p1p1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default
    qlen 1000
    link/ether 00:02:c9:02:72:e0 brd ff:ff:ff:ff:ff:ff
    vf 0 MAC 00:00:00:00:00:00, vlan 4095, spoof checking off, link-state auto
    vf 37 MAC 00:00:00:00:00:00, vlan 4095, spoof checking off, link-state auto
    vf 38 MAC ff:ff:ff:ff:ff:ff, vlan 65535, spoof checking off, link-state disable
    vf 39 MAC ff:ff:ff:ff:ff:ff, vlan 65535, spoof checking off, link-state disable
```

When a MAC is ff:ff:ff:ff:ff:ff, the VF is not assigned to the port of the net device it is listed under. In the example above, vf38 is not assigned to the same port as p1p1, in contrast to vf0.

However, even VFs that are not assigned to the net device, could be used to set and change its settings. For example, the following is a valid command to change the spoof check:

```
ip link set dev p1p1 vf 38 spoofchk on
```

This command will affect only the vf38. The changes can be seen in ip link on the net device that this device is assigned to.

**RoCE Support**

RoCE is supported on Virtual Functions and VLANs may be used with it. For RoCE, the hypervisor GID table size is of 16 entries while the VFs share the remaining 112 entries. When the number of VFs is larger than 56 entries, some of them will have GID table with only a single entry which is inadequate if VF's Ethernet device is assigned with an IP address.

**Virtual Guest Tagging (VGT+)**
VGT+ is an advanced mode of Virtual Guest Tagging (VGT), in which a VF is allowed to tag its own packets as in VGT, but is still subject to an administrative VLAN trunk policy. The policy determines which VLAN IDs are allowed to be transmitted or received. The policy does not determine the user priority, which is left unchanged.

Packets can be sent in one of the following modes: when the VF is allowed to send/receive untagged and priority tagged traffic and when it is not. No default VLAN is defined for VGT+ port. The send packets are passed to the eSwitch only if they match the set, and the received packets are forwarded to the VF only if they match the set.

**Configuration**

![Note]

When working in SR-IOV, the default operating mode is VGT.

➢ *To enable VGT+ mode:*

Set the corresponding port/VF (in the example below port eth5, VF0) range of allowed VLANs.

```bash
echo "<add> <start_vid> <end_vid>" > /sys/class/net/eth5/device/sriov/0/trunk
```

**Examples:**

- Adding VLAN ID range (4-15) to trunk:

```bash
echo add 4 15 > /sys/class/net/eth5/device/sriov/0/trunk
```

- Adding a single VLAN ID to trunk:
Note: When VLAN ID = 0, it indicates that untagged and priority-tagged traffics are allowed.

➢ To disable VGT+ mode, make sure to remove all VLANs.

`echo add 17 17 > /sys/class/net/eth5/device/sriov/0/trunk`

`echo rem 0 4095 > /sys/class/net/eth5/device/sriov/0/trunk`

➢ To remove selected VLANs.

- Remove VLAN ID range (4-15) from trunk:

  `echo rem 4 15 > /sys/class/net/eth5/device/sriov/0/trunk`

- Remove a single VLAN ID from trunk:

  `echo rem 17 17 > /sys/class/net/eth5/device/sriov/0/trunk`

**SR-IOV Advanced Security Features**

**SR-IOV MAC Anti-Spoofing**

Normally, MAC addresses are unique identifiers assigned to network interfaces, and they are fixed addresses that cannot be changed. MAC address spoofing is a technique for altering the MAC address to serve different purposes. Some of the cases in which a MAC address is altered can be legal, while others can be illegal and abuse security mechanisms or disguises a possible attacker.
The SR-IOV MAC address anti-spoofing feature, also known as MAC Spoof Check provides protection against malicious VM MAC address forging. If the network administrator assigns a MAC address to a VF (through the hypervisor) and enables spoof check on it, this will limit the end user to send traffic only from the assigned MAC address of that VF.

**MAC Anti-Spoofing Configuration**

1. **Note**
   
   MAC anti-spoofing is disabled by default.

In the configuration example below, the VM is located on VF-0 and has the following MAC address: 11:22:33:44:55:66.

There are two ways to enable or disable MAC anti-spoofing:

1. Use the standard IP link commands - available from Kernel 3.10 and above.
   
   1. To enable MAC anti-spoofing, run:

      ```bash
      ip link set ens785f1 vf 0 spoofchk on
      ```

   2. To disable MAC anti-spoofing, run:

      ```bash
      ip link set ens785f1 vf 0 spoofchk off
      ```

2. Specify echo "ON" or "OFF" to the file located under /sys/class/net/<ifname>/device/sriov/<VF index>/spoofcheck.

   1. To enable MAC anti-spoofing, run:
2. To disable MAC anti-spoofing, run:

```bash
echo "ON" > /sys/class/net/ens785f1/vf/0/spoofchk
```

```bash
echo "OFF" > /sys/class/net/ens785f1/vf/0/spoofchk
```

**Note**

This configuration is non-persistent and does not survive driver restart.

**Limit and Bandwidth Share Per VF**

This feature enables rate limiting traffic per VF in SR-IOV mode. For details on how to configure rate limit per VF for ConnectX-4 and above adapter cards, please refer to [HowTo Configure Rate Limit per VF for ConnectX-4/ConnectX-5/ConnectX-6 Community post.](#)

**Limit Bandwidth per Group of VFs**

VFs Rate Limit for vSwitch (OVS) feature allows users to join available VFs into groups and set a rate limitation on each group. Rate limitation on a VF group ensures that the total Tx bandwidth that the VFs in this group get (altogether combined) will not exceed the given value.

With this feature, a VF can still be configured with an individual rate limit as in the past (under `/sys/class/net/device/sriov/max_tx_rate`). However, the actual bandwidth limit on the VF will eventually be determined considering the VF group limitation and how many
VFs are in the same group.

For example: 2 VFs (0 and 1) are attached to group 3.

**Case 1:** The rate limitation on the group is set to 20G. Rate limit of each VF is 15G

**Result:** Each VF will have a rate limit of 10G

**Case 2:** Group’s max rate limitation is still set to 20G. VF 0 is configured to 30G limit, while VF 1 is configured to 5G rate limit

**Result:** VF 0 will have 15G de-facto. VF 1 will have 5G

The rule of thumb is that the group’s bandwidth is distributed evenly between the number of VFs in the group. If there are leftovers, they will be assigned to VFs whose individual rate limit has not been met yet.

**VFs Rate Limit Feature Configuration**

1. When VF rate group is supported by FW, the driver will create a new hierarchy in the SRI-OV sysfs named “groups” (/sys/class/net/<ifname>/device/sriov/groups/). It will contain all the info and the configurations allowed for VF groups.

2. All VFs are placed in group 0 by default since it is the only existing group following the initial driver start. It would be the only group available under /sys/class/net/<ifname>/device/sriov/groups/

3. The VF can be moved to a different group by writing to the group file -> echo $GROUP_ID > /sys/class/net/<ifname>/device/sriov/<vf_id>/group

4. The group IDs allowed are 0-255

5. Only when there is at least 1 VF in a group, there will be a group configuration available under /sys/class/net/<ifname>/device/sriov/groups/ (Except for group 0, which is always available even when it’s empty).

6. Once the group is created (by moving at least 1 VF to that group), users can configure the group’s rate limit. For example:

   1. echo 10000 > /sys/class/net/<ifname>/device/sriov/5/max_tx_rate – setting individual rate limitation of VF 5 to 10G (Optional)
2. echo 7 > /sys/class/net/<ifname>/device/sriov/5/group – moving VF 5 to group 7

3. echo 5000 > /sys/class/net/<ifname>/device/sriov/groups/7/max_tx_rate – setting group 7 with rate limitation of 5G

4. When running traffic via VF 5 now, it will be limited to 5G because of the group rate limit even though the VF itself is limited to 10G

5. echo 3 > /sys/class/net/<ifname>/device/sriov/5/group – moving VF 5 to group 3

6. Group 7 will now disappear from /sys/class/net/<ifname>/device/sriov/groups since there are 0 VFs in it. Group 3 will now appear. Since there’s no rate limit on group 3, VF 5 can transmit at 10G (thanks to its individual configuration)

**Notes:**

- You can see to which group the VF belongs to in the ‘stats’ sysfs (cat /sys/class/net/<ifname>/device/sriov/<vf_num>/stats)

- You can see the current rate limit and number of attached VFs to a group in the group’s ‘config’ sysfs (cat /sys/class/net/<ifname>/device/sriov/groups/<group_id>/config)

**Bandwidth Guarantee per Group of VFs**

Bandwidth guarantee (minimum BW) can be set on a group of VFs to ensure this group is able to transmit at least the amount of bandwidth specified on the wire.

Note the following:

- The minimum BW settings on VF groups determine how the groups share the total BW between themselves. It does not impact an individual VF's rate settings.

- The total minimum BW that is set on the VF groups should not exceed the total line rate. Otherwise, results are unexpected.
It is still possible to set minimum BW on the individual VFs inside the group. This will determine how the VFs share the group's minimum BW between themselves. The total minimum BW of the VF member should not exceed the minimum BW of the group.

For instruction on how to create groups of VFs, see Limit Bandwidth per Group of VFs above.

**Example**

With a 40Gb link speed, assuming 4 groups and default group 0 have been created:

```
echo 20000 > /sys/class/net/<ifname>/device/sriov/group/1/min_tx_rate
echo 5000 > /sys/class/net/<ifname>/device/sriov/group/2/min_tx_rate
echo 15000 > /sys/class/net/<ifname>/device/sriov/group/3/min_tx_rate
```

- **Group 0** (default) : 0 - No BW guarantee is configured.
- **Group 1** : 20000 - This is the maximum min rate among groups
- **Group 2** : 5000 which is 25% of the maximum min rate
- **Group 3** : 15000 which is 75% of the maximum min rate
- **Group 4** : 0 - No BW guarantee is configured.

Assuming there are VFs attempting to transmit in full line rate in all groups, the results would look like: In which case, the minimum BW allocation would be:

Group0 – Will have no BW to use since no BW guarantee was set on it while other groups do have such settings.
Group1 – Will transmit at 20Gb/s
Group2 – Will transmit at 5Gb/s
Group3 – Will transmit at 15Gb/s
Group4 - Will have no BW to use since no BW guarantee was set on it while other groups do have such settings.

**Privileged VFs**
In case a malicious driver is running over one of the VFs, and in case that VF's permissions are not restricted, this may open security holes. However, VFs can be marked as trusted and can thus receive an exclusive subset of physical function privileges or permissions. For example, in case of allowing all VFs, rather than specific VFs, to enter a promiscuous mode as a privilege, this will enable malicious users to sniff and monitor the entire physical port for incoming traffic, including traffic targeting other VFs, which is considered a severe security hole.

**Privileged VFs Configuration**

In the configuration example below, the VM is located on VF-0 and has the following MAC address: 11:22:33:44:55:66.

There are two ways to enable or disable trust:

1. Use the standard IP link commands - available from Kernel 4.5 and above.
   1. To enable trust for a specific VF, run:
      ```bash
      ip link set ens785f1 vf 0 trust on
      ```
   2. To disable trust for a specific VF, run:
      ```bash
      ip link set ens785f1 vf 0 trust off
      ```

2. Specify echo "ON" or "OFF" to the file located under `/sys/class/net/<ETH_IF_NAME>/device/sriov/<VF index>/trust.

   1. To enable trust for a specific VF, run:
      ```bash
      echo "ON" > /sys/class/net/ens785f1/device/sriov/0/trust
      ```
   2. To disable trust for a specific VF, run:
Probed VFs

Probing Virtual Functions (VFs) after SR-IOV is enabled might consume the adapter cards' resources. Therefore, it is recommended not to enable probing of VFs when no monitoring of the VM is needed.

VF probing can be disabled in two ways, depending on the kernel version installed on your server:

1. If the kernel version installed is v4.12 or above, it is recommended to use the PCI sysfs interface sriov_drivers_autoprobe. For more information, see [linux-next branch](https://github.com/torvalds/linux-next).

2. If the kernel version installed is older than v4.12, it is recommended to use the mlx5_core module parameter probe_vf with driver version 4.1 or above.

Example:

```
echo "OFF" > /sys/class/net/ens785f1/device/sriov/0/trust
```

For more information on how to probe VFs, see [HowTo Configure and Probe VFs on mlx5 Drivers](https://community.post) Community post.

VF Promiscuous Rx Modes

**VF Promiscuous Mode**

VFs can enter a promiscuous mode that enables receiving the unmatched traffic and all the multicast traffic that reaches the physical port in addition to the traffic originally
targeted to the VF. The unmatched traffic is any traffic's DMAC that does not match any of the VFs' or PFs' MAC addresses.

**Note:** Only privileged/trusted VFs can enter the VF promiscuous mode.

*To set the promiscuous mode on for a VF, run:*

```
ifconfig eth2 promisc
```

*To exit the promiscuous mode, run:*

```
ifconfig eth2 –promisc
```

**VF All-Multi Mode**

VFAs can enter an all-multi mode that enables receiving all the multicast traffic sent from/to the other functions on the same physical port in addition to the traffic originally targeted to the VF.

**Note:** Only privileged/trusted VFs can enter the all-multi RX mode.

*To set the all-multi mode on for a VF, run:*

```
ifconfig eth2 allmulti
```

*To exit the all-multi mode, run:*

```
ifconfig eth2 allmulti
```
Uninstalling the SR-IOV Driver

To uninstall SR-IOV driver, perform the following:

1. For Hypervisors, detach all the Virtual Functions (VF) from all the Virtual Machines (VM) or stop the Virtual Machines that use the Virtual Functions.

   **Please be aware** that stopping the driver when there are VMs that use the VFs, will cause machine to hang.

2. Run the script below. Please be aware, uninstalling the driver deletes the entire driver's file, but does not unload the driver.

   ```
   [root@swl022 ~]# /usr/sbin/ofed_uninstall.sh
   This program will uninstall all OFED packages on your machine.
   Do you want to continue?[y/N]:y
   Running /usr/sbin/vendor_pre_uninstall.sh
   Removing OFED Software installations
   warning: /etc/infiniband/openib.conf saved as /etc/infiniband/openib.conf.rpmsave
   Running /tmp/2818-ofed_vendor_post_uninstall.sh
   ```

3. Restart the server.

SR-IOV Live Migration
Live migration refers to the process of moving a guest virtual machine (VM) running on one physical host to another host without disrupting normal operations or causing other adverse effects for the end user.

Using the Migration process is useful for:

- load balancing
- hardware independence
- energy saving
- geographic migration
- fault tolerance

Migration works by sending the state of the guest virtual machine's memory and any virtualized devices to a destination host physical machine. Migrations can be performed live or not, in the live case, the migration will not disrupt the user operations and it will be transparent to it as explained in the sections below.

### Non-Live Migration

When using the non-live migration process, the Hypervisor suspends the guest virtual machine, then moves an image of the guest virtual machine's memory to the destination host physical machine. The guest virtual machine is then resumed on the destination host physical machine, and the memory the guest virtual machine used on the source host physical machine is freed. The time it takes to complete such a migration depends on the network bandwidth and latency. If the network is experiencing heavy use or low bandwidth, the migration will take longer than desired.

### Live Migration

\[\text{Note}\]

This feature is supported in Ethernet mode only.
When using the Live Migration process, the guest virtual machine continues to run on the source host physical machine while its memory pages are transferred to the destination host physical machine. During migration, the Hypervisor monitors the source for any changes in the pages it has already transferred and begins to transfer these changes when all of the initial pages have been transferred.

It also estimates transfer speed during migration, so when the remaining amount of data to transfer will take a certain configurable period of time, it will suspend the original guest virtual machine, transfer the remaining data, and resume the same guest virtual machine on the destination host physical machine.

**MLX5 VF Live Migration**

The purpose of this section is to demonstrate how to perform basic live migration of a QEMU VM with an MLX5 VF assigned to it. This section does not explain how to create VMs either using libvirt or directly via QEMU.

**Requirements**

The below are the requirements for working with MLX5 VF Live Migration.

<table>
<thead>
<tr>
<th>Components</th>
<th>Description</th>
</tr>
</thead>
</table>
| Adapte Cards | • ConnectX-7 ETH  
• BlueField-3 ETH |
| Firmware | • 28.41.1000  
• 32.41.1000 |
| Kernel | Linux v6.7 or newer |

**Note**

The same PSID must be used on both the source and the target hosts (identical cards, same CAPs and features are needed), and have the same firmware version.
<table>
<thead>
<tr>
<th>Components</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Space Tools</td>
<td>iproute2 version 6.2 or newer</td>
</tr>
<tr>
<td>QEMU</td>
<td>QEMU 8.1 or newer</td>
</tr>
<tr>
<td>Libvirt</td>
<td>Libvirt 8.6 or newer</td>
</tr>
</tbody>
</table>

Setup

**NVCONFIG**

SR-IOV should be enabled and be configured to support the required number of VFs as of enabling live migration. This can be achieved by the below command:

```
mlxconfig -d *<PF_BDF>* s SRIOV_EN=1 NUM_OF_VFS=4 VF_MIGRATION_MODE=2
```

where:

<table>
<thead>
<tr>
<th>SRIOV_EN</th>
<th>Enable Single-Root I/O Virtualization (SR-IOV)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM_OF_VFS</td>
<td>The total number of Virtual Functions (VFs) that can be supported, for each PF.</td>
</tr>
<tr>
<td>VF_MIGRATION_MODE</td>
<td>Defines support for VF migration.</td>
</tr>
<tr>
<td></td>
<td>• 0x0: DEVICE_DEFAULT</td>
</tr>
<tr>
<td></td>
<td>• 0x1: MIGRATION_DISABLED</td>
</tr>
<tr>
<td></td>
<td>• 0x2: MIGRATION_ENABLED</td>
</tr>
</tbody>
</table>

**Kernel Configuration**
Needs to be compiled with driver MLX5_VFIO_PCI enabled. (i.e. CONFIG_MLX5_VFIO_PCI).

To load the driver, run:

```bash
modprobe mlx5_vfio_pci
```

**QEMU**

Needs to be compiled with VFIO_PCI enabled (this is enabled by default).

**Host Preparation**

As stated earlier, creating the VMs is beyond the scope of this guide and we assume that they are already created. However, the VM configuration should be a migratable configuration, similarly to how it is done without SRIOV VFs.

**Note**

The below steps should be done before running the VMs.

**Over libvirt**

1. Set the PF in the "switchdev" mode.

   ```bash
devlink dev eswitch set pci/<PF_BDF> mode switchdev
   ```

2. Create the VFs that will be assigned to the VMs.
3. Set the VFs as migration capable.

   1. See the name of the VFs, run:

      ```bash
devlink port show
      ```

   2. Unbind the VFs from mlx5_core, run:

      ```bash
echo '<VF_BDF>' > /sys/bus/pci/drivers/mlx5_core/unbind
      ```

   3. Use devlink to set each VF as migration capable, run:

      ```bash
devlink port function set pci/<PF_BDF>/1 migratable enable
      ```

4. Assign the VFs to the VMs.

   1. Edit the VMs XML file, run:

      ```bash
virsh edit <VM_NAME>
      ```

   2. Assign the VFs to the VM by adding the following under the "devices" tag:

      ```xml
<hostdev mode='subsystem' type='pci' managed='no'>
  <driver name='vfio'/>
  <source>
    <address domain='0x0000' bus='0x08' slot='0x00' function='0x2'/>
  </source>
  <address type='pci' domain='0x0000' bus='0x09' slot='0x00' function='0x0'/>
</hostdev>
```
5. Set the destination VM in incoming mode.

1. Edit the destination VM XML file, run:

```bash
virsh edit <VM_NAME>
```

2. Set the destination VM in migration incoming mode by adding the following under "domain" tag:

```xml
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
  [...]  
  <qemu:commandline>
    <qemu:arg value='--incoming'/>
    <qemu:arg value='tcp:<DEST_IP>:<DEST_PORT>'/>
  </qemu:commandline>
</domain>
```

- **Note**
  The domain, bus, slot and function values above are dummy values, replace them with your VFs values.

- **Note**
  To be able to save the file, the above "xmlns:qemu" attribute of the "domain" tag must be added as well.
6. Bind the VFs to mlx5_vfio_pci driver.

1. Detach the VFs from libvirt management, run:

   ```bash
   virsh nodedev-detach pci_<VF_BDF>
   ```

2. Unbind the VFs from vfio-pci driver (the VFs are automatically bound to it after running "virsh nodedev-detach"), run:

   ```bash
   echo '<VF_BDF>' > /sys/bus/pci/drivers/vfio-pci/unbind
   ```

3. Set driver override, run:

   ```bash
   echo 'mlx5_vfio_pci' > /sys/bus/pci/devices/<VF_BDF>/driver_override
   ```

4. Bind the VFs to mlx5_vfio_pci driver, run:

   ```bash
   echo '<VF_BDF>' > /sys/bus/pci/drivers/mlx5_vfio_pci/bind
   ```

**Directly over QEMU**

1. Set the PF in "switchdev" mode.

   ```bash
   devlink dev eswitch set pci/<PF_BDF> mode switchdev
   ```

2. Create the VFs that will be assigned to the VMs.
3. Set the VFs as migration capable.

   1. See the name of the VFs, run:

   ```
   devlink port show
   ```

   2. Unbind the VFs from mlx5_core, run:

   ```
   echo '<VF_BDF>' > /sys/bus/pci/drivers/mlx5_core/unbind
   ```

   3. Use devlink to set each VF as migration capable, run:

   ```
   devlink port function set pci/<PF_BDF>/1 migratable enable
   ```

4. Bind the VFs to mlx5_vfio_pci driver:

   1. Set driver override, run:

   ```
   echo 'mlx5_vfio_pci' > /sys/bus/pci/devices/<VF_BDF>/driver_override
   ```

   2. Bind the VFs to mlx5_vfio_pci driver, run:

   ```
   echo '<VF_BDF>' > /sys/bus/pci/drivers/mlx5_vfio_pci/bind
   ```

**Running the Migration**
Over libvirt

1. Start the VMs in source and in destination, run:

```bash
virsh start <VM_NAME>
```

2. Enable switchover-ack QEMU migration capability. Run the following commands both in source and destination:

```bash
virsh qemu-monitor-command <VM_NAME> --hmp "migrate_set_capability return-path on"

virsh qemu-monitor-command <VM_NAME> --hmp "migrate_set_capability switchover-ack on"
```

3. [Optional] Configure the migration bandwidth and downtime limit in source side:

```bash
virsh qemu-monitor-command <VM_NAME> --hmp "migrate_set_parameter max-bandwidth <VALUE>"

virsh qemu-monitor-command <VM_NAME> --hmp "migrate_set_parameter downtime-limit <VALUE>"
```

4. Start migration by running the migration command in source side:

```bash
virsh qemu-monitor-command <VM_NAME> --hmp "migrate -d tcp:<DEST_IP>:<DEST_PORT>"
```

5. Check the migration status by running the info command in source side:

```bash
virsh qemu-monitor-command <VM_NAME> --hmp "info migrate"
```
Directly over QEMU

1. Start the VM in source with the VF assigned to it:

```bash
qemu-system-x86_64 [...] -device vfio-pci,host=<VF_BDF>,id=mlx5_1
```

2. Start the VM in destination with the VF assigned to it and with the "incoming" parameter:

```bash
qemu-system-x86_64 [...] -device vfio-pci,host=<VF_BDF>,id=mlx5_1 -incoming tcp:<DEST_IP>:<DEST_PORT>
```

3. Enable switchover-ack QEMU migration capability. Run the following commands in QEMU monitor, both in source and destination:

```bash
migrate_set_capability return-path on

migrate_set_capability switchover-ack on
```

4. [Optional] Configure the migration bandwidth and downtime limit in source side:

```bash
migrate_set_parameter max-bandwidth <VALUE>
```

Note

When the migration status is "completed" it means the migration has finished successfully.
5. Start migration by running the migration command in QEMU monitor in source side:

```
migrate -d tcp:<DEST_IP>:<DEST_PORT>
```

6. Check the migration status by running the info command in QEMU monitor in source side:

```
info migrate
```

<i>Note</i>

When the migration status is "completed" it means the migration has finished successfully.

**Migration with MultiPort vHCA**

Enables the usage of a dual port Virtual HCA (vHCA) to share RDMA resources (e.g., MR, CQ, SRQ, PDs) across the two Ethernet (RoCE) NIC network ports and display the NIC as a dual port device.

MultiPort vHCA (MPV) VF is made of 2 "regular" VFs, one VF of each port. Creating a migratable MPV VF requires the same steps as regular VF (see steps in section Over libvirt). The steps should be performed on each of the NIC ports. MPV VFs traffic cannot be configured with OVS. TC rules must be defined to configure the MPV VFs traffic.

**Notes**
Enabling Paravirtualization

To enable Paravirtualization:

1. Create a bridge.

```
vim /etc/sysconfig/network-scripts/ifcfg-bridge0
DEVICE=bridge0
TYPE=Bridge
IPADDR=12.195.15.1
NETMASK=255.255.0.0
BOOTPROTO=static
ONBOOT=yes
```
2. Change the related interface (in the example below bridge0 is created over eth5).

```bash
DEVICE=eth5
BOOTPROTO=none
STARTMODE=on
HWADDR=00:02:c9:2e:66:52
TYPE=Ethernet
NM_CONTROLLED=no
ONBOOT=yes
BRIDGE=bridge0
```

3. Restart the service network.

4. Attach a bridge to VM.

```bash
ifconfig -a
...
eth6 Link encap:Ethernet HWaddr 52:54:00:E7:77:99
    inet addr:13.195.15.5 Bcast:13.195.255.255 Mask:255.255.0.0
    inet6 addr: fe80::5054:ff:fee7:7799/64 Scope:Link
    UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
    RX packets:481 errors:0 dropped:0 overruns:0 frame:0
    TX packets:450 errors:0 dropped:0 overruns:0 carrier:0
    collisions:0 txqueuelen:1000
    RX bytes:22440 (21.9 KiB) TX bytes:19232 (18.7 KiB)
    Interrupt:10 Base address:0xa000
...
```

**VXLAN Hardware Stateless Offloads**

VXLAN technology provides scalability and security challenges solutions. It requires extension of the traditional stateless offloads to avoid performance drop. ConnectX family cards offer the following stateless offloads for a VXLAN packet, similar to the ones
offered to non-encapsulated packets. VXLAN protocol encapsulates its packets using outer UDP header.

**Available hardware stateless offloads:**

- Checksum generation (Inner IP and Inner TCP/UDP)
- Checksum validation (Inner IP and Inner TCP/UDP)
- TSO support for inner TCP packets
- RSS distribution according to inner packets attributes
- Receive queue selection - inner frames may be steered to specific QPs

**Enabling VXLAN Hardware Stateless Offloads**

VXLAN offload is enabled by default for ConnectX-4 family devices running the minimum required firmware version and a kernel version that includes VXLAN support.

➢ **To confirm if the current setup supports VXLAN, run:**

```bash
ethtool -k $DEV | grep udp_tnl
```

**Example:**

```bash
ethtool -k ens1f0 | grep udp_tnl
tx-udp_tnl-segmentation: on
```

ConnectX-4 family devices support configuring multiple UDP ports for VXLAN offload. Ports can be added to the device by configuring a VXLAN device from the OS command line using the "ip" command.

**Note:** If you configure multiple UDP ports for offload and exceed the total number of ports supported by hardware, then those additional ports will still function properly, but will not benefit from any of the stateless offloads.
Example:

```
ip link add vxlan0 type vxlan id 10 group 239.0.0.10 ttl 10 dev ens1f0 dstport 4789
ip addr add 192.168.4.7/24 dev vxlan0
ip link set up vxlan0
```

**Note:** `dstport` parameters are not supported in Ubuntu 14.4.

The VXLAN ports can be removed by deleting the VXLAN interfaces.

**Example:**

```
ip link delete vxlan0
```

**Important Note**

VXLAN tunneling adds 50 bytes (14-eth + 20-ip + 8-udp + 8-vxlan) to the VM Ethernet frame. Please verify that either the MTU of the NIC who sends the packets, e.g. the VM virtio-net NIC or the host side veth device or the uplink takes into account the tunneling overhead. Meaning, the MTU of the sending NIC has to be decremented by 50 bytes (e.g 1450 instead of 1500), or the uplink NIC MTU has to be incremented by 50 bytes (e.g 1550 instead of 1500).

**Q-in-Q Encapsulation per VF in Linux (VST)**

**Note**

This feature is supported on ConnectX-5 and ConnectX-6 adapter cards only.
This section describes the configuration of IEEE 802.1ad QinQ VLAN tag (S-VLAN) to the hypervisor per Virtual Function (VF). The Virtual Machine (VM) attached to the VF (via SR-IOV) can send traffic with or without C-VLAN. Once a VF is configured to VST QinQ encapsulation (VST QinQ), the adapter's hardware will insert S-VLAN to any packet from the VF to the physical port. On the receive side, the adapter hardware will strip the S-VLAN from any packet coming from the wire to that VF.

**Setup**

The setup assumes there are two servers equipped with ConnectX-5/ConnectX-6 adapter cards.

**Prerequisites**

- Kernel must be of v3.10 or higher, or custom/inbox kernel must support vlan-stag
- Firmware version 16/20.21.0458 or higher must be installed for ConnectX-5/ConnectX-6 HCAs
- The server should be enabled in SR-IOV and the VF should be attached to a VM on the hypervisor.
In order to configure SR-IOV in Ethernet mode for ConnectX-5/ConnectX-6 adapter cards, please refer to "Configuring SR-IOV for ConnectX-4/ConnectX-5 (Ethernet)" section. In the following configuration example, the VM is attached to VF0.

- Network Considerations - the network switches may require increasing the MTU (to support 1522 MTU size) on the relevant switch ports.

### Configuring Q-in-Q Encapsulation per Virtual Function for ConnectX-5/ConnectX-6

1. Add the required S-VLAN (QinQ) tag (on the hypervisor) per port per VF. There are two ways to add the S-VLAN:

   1. By using sysfs:

   ```bash
   echo '100:0:802.1ad' > /sys/class/net/ens1f0/device/sriov/0/vlan
   ```

   2. By using the `ip link` command (available only when using the latest Kernel version):

   ```bash
   ip link set dev ens1f0 vf 0 vlan 100 proto 802.1ad
   ```

   Check the configuration using the `ip link show` command:

   ```bash
   # ip link show ens1f0
   ens1f0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT qlen 1000
   link/ether ec:0d:9a:44:37:84 brd ff:ff:ff:ff:ff:ff
   vf 0 MAC 00:00:00:00:00:00, vlan 100, vlan protocol 802.1ad, spoof checking off, link-state auto, trust off
   vf 1 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
   vf 2 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
   vf 3 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
   vf 4 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
   ```
2. **Optional:** Add S-VLAN priority. Use the qos parameter in the ip link command (or sysfs):

```
ip link set dev ens1f0 vf 0 vlan 100 qos 3 proto 802.1ad
```

Check the configuration using the ip link show command:

```
# ip link show ens1f0
ens1f0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT qlen 1000
  link/ether ec:0d:9a:44:37:84 brd ff:ff:ff:ff:ff:ff
  vf 0 MAC 00:00:00:00:00:00, vlan 100, qos 3, vlan protocol 802.1ad, spoof checking off, link-state auto, trust off
  vf 1 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
  vf 2 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
  vf 3 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
  vf 4 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
```

3. Create a VLAN interface on the VM and add an IP address.

```
ip link add link ens5 ens5.40 type vlan protocol 802.1q id 40
ip addr add 42.134.135.7/16 brd 42.134.255.255 dev ens5.40
ip link set dev ens5.40 up
```

4. To verify the setup, run ping between the two VMs and open Wireshark or tcpdump to capture the packet.

**802.1Q Double-Tagging**

This section describes the configuration of 802.1Q double-tagging support to the hypervisor per Virtual Function (VF). The Virtual Machine (VM) attached to the VF (via SR-IOV) can send traffic with or without C-VLAN. Once a VF is configured to VST encapsulation, the adapter's hardware will insert C-VLAN to any packet from the VF to the
physical port. On the receive side, the adapter hardware will strip the C-VLAN from any packet coming from the wire to that VF.

**Configuring 802.1Q Double-Tagging per Virtual Function**

1. Add the required C-VLAN tag (on the hypervisor) per port per VF. There are two ways to add the C-VLAN:

   1. By using sysfs:

   ```
   echo '100:0:80.1q' > /sys/class/net/ens1f0/device/sriov/0/vlan
   ```

   2. By using the `ip link` command (available only when using the latest Kernel version):

   ```
   ip link set dev ens1f0 vf 0 vlan 100
   ```

   Check the configuration using the `ip link show` command:

   ```
   # ip link show ens1f0
   ens1f0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT qlen 1000
   link/ether ec:0d:9a:44:37:84 brd ff:ff:ff:ff:ff:ff
   vf 0 MAC 00:00:00:00:00:00, vlan 100, spoof checking off, link-state auto, trust off
   vf 1 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
   vf 2 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
   vf 3 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
   vf 4 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
   ```

2. Create a VLAN interface on the VM and add an IP address.

   ```
   # ip link add link ens5 ens5.40 type vlan protocol 802.1q id 40
   # ip addr add 42.134.135.7/16 brd 42.134.255.255 dev ens5.40
   ```
3. To verify the setup, run ping between the two VMs and open Wireshark or tcpdump to capture the packet.

Scalable Functions

Scalable function is a lightweight function that has a parent PCI function on which it is deployed. Scalable functions are useful for containers where netdevice and RDMA devices of a scalable function can be assigned to a container. This way, the container can get complete offload capabilities of an eswitch, isolation and dedicated accelerated network device. For Step-by-Step Configuration instructions, follow the User Guide here.

Resiliency

The chapter contains the following sections:

- Reset Flow

Reset Flow

Reset Flow is activated by default. Once a "fatal device" error is recognized, both the HCA and the software are reset, the ULPs and user application are notified about it, and a recovery process is performed once the event is raised.

Currently, a reset flow can be triggered by a firmware assert with Recover Flow Request (RFR) only. Firmware RFR support should be enabled explicitly using mlxconfig commands.

To query the current value, run:

```
mlxconfig -d /dev/mst/mt4115_pciconf0 query | grep SW_RECOVERY_ON_ERRORS
```
To enable RFR bit support, run:

```bash
mlxconfig -d /dev/mst/mt4115_pciconf0 set SW_RECOVERY_ON_ERRORS=true
```

## Kernel ULPs

Once a "fatal device" error is recognized, an IB_EVENT_DEVICE_FATAL event is created, ULPs are notified about the incident, and outstanding WQEs are simulated to be returned with "flush in error" message to enable each ULP to close its resources and not get stuck via calling its "remove_one" callback as part of "Reset Flow".

Once the unload part is terminated, each ULP is called with its "add_one" callback, its resources are re-initialized and it is re-activated.

## User Space Applications (IB/RoCE)

Once a "fatal device" error is recognized an IB_EVENTDEVICE_FATAL event is created, applications are notified about the incident and relevant recovery actions are taken.

Applications that ignore this event enter a zombie state, where each command sent to the kernel is returned with an error, and no completion on outstanding WQEs is expected.

The expected behavior from the applications is to register to receive such events and recover once the above event is raised. Same behavior is expected in case the NIC is unbounded from the PCI and later is rebounded. Applications running over RDMA CM should behave in the same manner once the RDMA_CM_EVENT_DEVICE_REMOVAL event is raised.

The below is an example of using the unbind/bind for NIC defined by "0000:04:00.0"

```bash
echo 0000:04:00.0 > /sys/bus/pci/drivers/mlx5_core/unbind
echo 0000:04:00.0 > /sys/bus/pci/drivers/mlx5_core/bind
```

## SR-IOV
If the Physical Function recognizes the error, it notifies all the VFs about it by marking their communication channel with that information, consequently, all the VFs and the PF are reset.

If the VF encounters an error, only that VF is reset, whereas the PF and other VFs continue to work unaffected.

**Forcing the VF to Reset**

If an outside "reset" is forced by using the PCI sysfs entry for a VF, a reset is executed on that VF once it runs any command over its communication channel.

For example, the below command can be used on a hypervisor to reset a VF defined by 0000:04:00.1:

```
echo 1 >/sys/bus/pci/devices/0000:04:00.1/reset
```

**Extended Error Handling (EEH)**

Extended Error Handling (EEH) is a PowerPC mechanism that encapsulates AER, thus exposing AER events to the operating system as EEH events.

The behavior of ULPs and user space applications is identical to the behavior of AER.

**CRDUMP**

CRDUMP feature allows for taking an automatic snapshot of the device CR-Space in case the device's FW/HW fails to function properly.

**Snapshots Triggers:**

The snapshot is triggered after firmware detects a critical issue, requiring a recovery flow.

This snapshot can later be investigated and analyzed to track the root cause of the failure.

Currently, only the first snapshot is stored, and is exposed using a temporary virtual file. The virtual file is cleared upon driver reset.
When a critical event is detected, a message indicating CRDUMP collection will be printed to the Linux log. User should then back up the file pointed to in the printed message. The file location format is: /proc/driver/mlx5_core/crdump/<pci address>

Snapshot should be copied by Linux standard tool for future investigation.

**Firmware Tracer**

This mechanism allows for the device's FW/HW to log important events into the event tracing system (/sys/kernel/debug/tracing) without requiring any NVIDIA tool.

*Note*

To be able to use this feature, trace points must be enabled in the kernel.

This feature is enabled by default, and can be controlled using sysfs commands.

*To disable the feature:*

```bash
echo 0 > /sys/kernel/debug/tracing/events/mlx5/fw_tracer/enable
```

*To enable the feature:*

```bash
echo 1 > /sys/kernel/debug/tracing/events/mlx5/fw_tracer/enable
```

*To view FW traces using vim text editor:*

```bash
vim /sys/kernel/debug/tracing/trace
```
Docker Containers

On Linux, Docker uses resource isolation of the Linux kernel, to allow independent "containers" to run within a single Linux kernel instance.

Docker containers are supported on MLNX_OFED using Docker runtime. Virtual RoCE and InfiniBand devices are supported using SR-IOV mode.

Currently, RDMA/RoCE devices are supported in the modes listed in the following table.

**Linux Containers Networking Modes**

<table>
<thead>
<tr>
<th>Orchestration and Clustering Tool</th>
<th>Version</th>
<th>Networking Mode</th>
<th>Link Layer</th>
<th>Virtualization Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Docker</td>
<td>Docker Engine 17.03 or higher</td>
<td>SR-IOV using sriov-plugin along with docker run wrapper tool</td>
<td>InfiniBand and Ethernet</td>
<td>SR-IOV</td>
</tr>
<tr>
<td>Kubernetes</td>
<td>Kubernetes 1.10.3 or higher</td>
<td>SR-IOV using device plugin, and using SR-IOV CNI plugin</td>
<td>InfiniBand and Ethernet</td>
<td>SR-IOV</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VXLAN using IPoIB bridge</td>
<td>InfiniBand</td>
<td>Shared HCA</td>
</tr>
</tbody>
</table>

**Docker Using SR-IOV**

In this mode, Docker engine is used to run containers along with SR-IOV networking plugin. To isolate the virtual devices, docker_rdma_sriov tool should be used. This mode is applicable to both InfiniBand and Ethernet link layers.

To obtain the plugin, visit: [hub.docker.com/r/rdma/sriov-plugin](https://hub.docker.com/r/rdma/sriov-plugin)

To install the docker_rdma_sriov tool, use the container tools installer available via [hub.docker.com/r/rdma/container_tools_installer](https://hub.docker.com/r/rdma/container_tools_installer)
For instructions on how to use Docker with SR-IOV, refer to Docker RDMA SRIOV Networking with ConnectX4/ConnectX5/ConnectX6 Community post.

**Kubernetes Using SR-IOV**

In order to use RDMA in Kubernetes environment with SR-IOV networking mode, two main components are required:

1. RDMA device plugin - this plugin allows for exposing RDMA devices in a Pod
2. SR-IOV CNI plugin - this plugin provisions VF net device in a Pod

When used in SR-IOV mode, this plugin enables SR-IOV and performs necessary configuration including setting GUID, MAC, privilege mode, and Trust mode.

The plugin also allocates the VF devices when Pods are scheduled and requested by Kubernetes framework.

**Kubernetes with Shared HCA**

One RDMA device (HCA) can be shared among multiple Pods running in a Kubernetes worker nodes. User defined networks are created using VXLAN or VETH networking devices. RDMA device (HCA) can be shared among multiple Pods running in a Kubernetes worker nodes.

**HPC-X**

For information on HPC-X®, please refer to HPC-X User Manual at developer.nvidia.com/networking/hpc-x.

**Fast Driver Unload**

This feature enables optimizing mlx5 driver teardown time in shutdown and kexec flows.

The fast driver unload is disabled by default. To enable it, the `prof_sel` module parameter of mlx5_core module should be set to 3.