Programming

Table of contents

Note

This chapter is aimed for application developers and expert users that wish to develop applications over MLNX_OFED.

Raw Ethernet Programming

Raw Ethernet programming enables writing an application that bypasses the kernel stack. To achieve this, packet headers and offload options need to be provided by the application.
For a basic example on how to use Raw Ethernet programming, refer to the Raw Ethernet Programming: Basic Introduction—Code Example Community post.

Packet Pacing

Packet pacing is a raw Ethernet sender feature that enables controlling the rate of each QP, per send queue.
For a basic example on how to use packet pacing per flow over libibverbs, refer to Raw Ethernet Programming: Packet Pacing—Code Example Community post.

TCP Segmentation Offload (TSO)

TCP Segmentation Offload (TSO) enables the adapter cards to accept a large amount of data with a size greater than the MTU size. The TSO engine splits the data into separate packets and inserts the user-specified L2/L3/L4 headers automatically per packet. With the usage of TSO, CPU is offloaded from dealing with a large throughput of data.
To be able to program that on the sender side, refer to the Raw Ethernet Programming: TSO—Code Example Community post.

ToS Based Steering

ToS/DSCP is an 8-bit field in the IP packet that enables different service levels to be assigned to network traffic. This is achieved by marking each packet in the network with a DSCP code and appropriating the corresponding level of service to it.
To be able to steer packets according to the ToS field on the receiver side, refer to the Raw Ethernet Programming: ToS—Code Example Community post.

Flow ID Based Steering

Flow ID based steering enables developing a code that will steer packets using flow ID when developing Raw Ethernet over verbs. For more information on flow ID based steering, refer to the Raw Ethernet Programming: Flow ID Steering—Code Example Community post.

VXLAN Based Steering

VXLAN based steering enables developing a code that will steer packets using the VXLAN tunnel ID when developing Raw Ethernet over verbs. For more information on VXLAN based steering, refer to the Raw Ethernet Programming: VXLAN Steering—Code Example Community post.

Device Memory Programming

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

Device Memory is an API that allows using on-chip memory located on the device as a data buffer for send/receive and RDMA operations. The device memory can be mapped and accessed directly by user and kernel applications, and can be allocated in various

sizes, registered as memory regions with local and remote access keys for performing the send/receive and RDMA operations.
Using the device memory to store packets for transmission can significantly reduce transmission latency compared to the host memory.

Device Memory Programming Model

The new API introduces a similar procedure to the host memory for sending packets from the buffer:

- ibv_alloc_dm()/ibv_free_dm() - to allocate/free device memory
- ibv_reg_dm_mr - to register the allocated device memory buffer as a memory region and get a memory key for local/remote access by the device
- ibv_memcpy_to_dm - to copy data to a device memory buffer
- ibv_memcpy_from_dm - to copy data from a device memory buffer
- ibv_post_send/ibv_post_receive - to request the device to perform a send/receive operation using the memory key

For examples, see Device Memory.

RDMA-CM QP Timeout Control

RDMA-CM QP Timeout Control feature enables users to control the QP timeout for QPs created with RDMA-CM.

A new option in 'rdma_set_option' function has been added to enable overriding calculated QP timeout, in order to provide QP attributes for QP modification. To achieve that, rdma_set_option() should be called with the new flag RDMA_OPTION_ID_ACK_TIMEOUT. Example:

```
rdma_set_option(cma_id, RDMA_OPTION_ID, RDMA_OPTION_ID_ACK_TIMEOUT,
&timeout, sizeof(timeout));
```

RDMA-CM Application Managed QP

Applications which do not create a QP through rdma_create_qp() may want to postpone the ESTABLISHED event on the passive side, to let the active side complete an application-specific connection establishment phase. For example, modifying the init state of the QP created by the application to RTR state, or make some preparations for receiving messages from the passive side. The feature returns a new event on the active side: CONNECT_RESPONSE, instead of ESTABLISHED, if id->qp==NULL. This gives the application a chance to perform the extra connection setup. Afterwards, the new rdma_establish() API should be called to complete the connection and generate an ESTABLISHED event on the passive side.

In addition, this feature exposes the 'rdma_init_qp_attr' function in librdmacm API, which enables applications to get the parameters for creating Address Handler (AH) or control QP attributes after its creation.