RDMA Tutorial

Netdev 0x16

Roland Dreier (Enfabrica)
Jason Gunthorpe (NVIDIA)

What is RDMA?

Technology for high-performance communication, defined by three major attributes:

1. HW data path via asynchronous submission and completion queues
2. Direct HW access via kernel bypass
3. One-sided operations: remote direct memory access (RDMA)

Asynchronous queues

- Data path is driven by submitting work requests to send and receive queues and collecting completions from completion queues.
- Allows for efficient overlapping of computation and communication.
- Also supports efficient parallel applications via per-CPU or per-thread queues.

Kernel bypass

- Transport is offloaded from application CPUs, including handling of packetization / reliability / retransmission.
- RDMA HW and driver stack is designed so that data path can safely go directly from a userspace application to HW by mapping a subset of doorbells etc. into userspace processes.
- Supports architectures such as polling loops with minimal jitter.
- Control path, including resource management and cleanup remains in the kernel.

Choice of one-sided and two-sided operations

- One-sided operations: one host moves data directly to or from memory of its communication peer without involving or even notifying its peer’s application CPU.
- Key assumption: memory is pre-registered and physically pinned (although on-demand paging is supported by some HW platforms).
- Two-sided operations: one host posts a receive work request and its peer then posts a send work request that consumes the receive work request to deliver data.
- Can mix and match one-sided (RDMA) and two-sided (send/receive) ops on a connection.

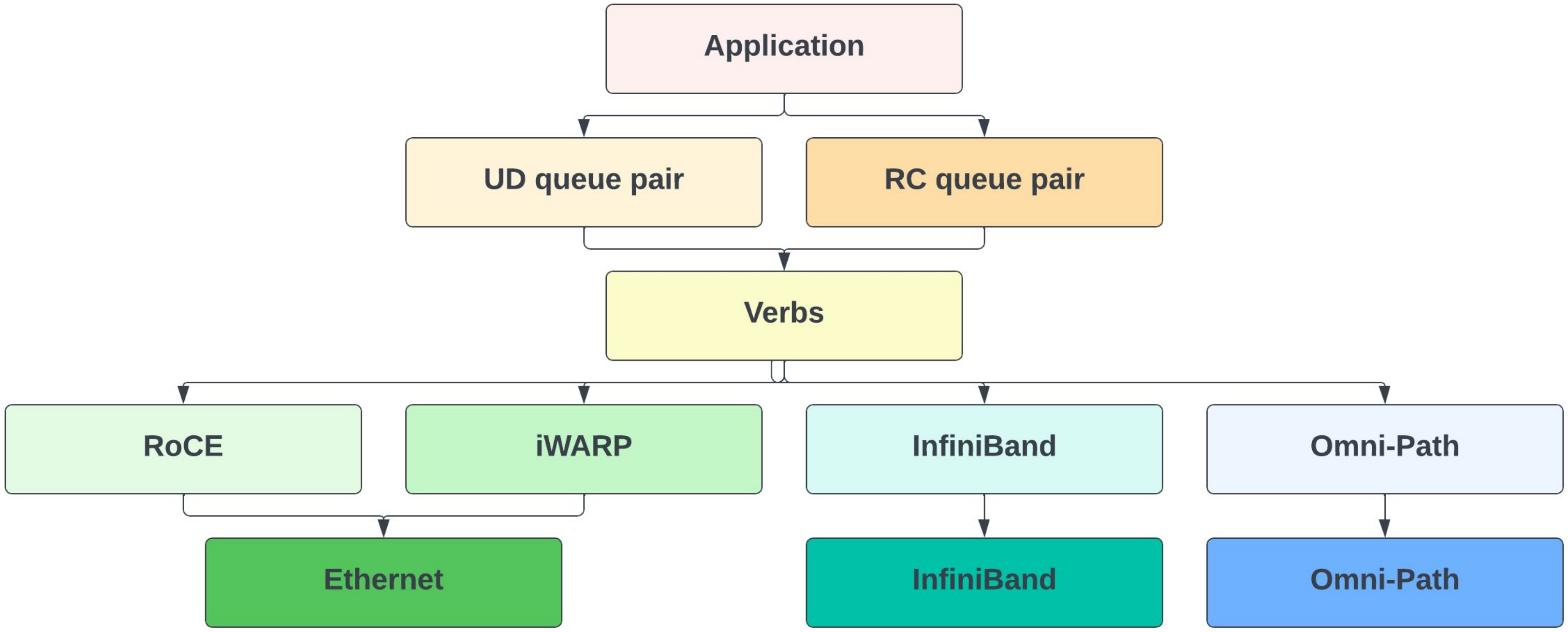
Transports

Multiple implementations of RDMA:

- Physical / link layers: InfiniBand and Ethernet
- Transports: InfiniBand / RoCE (aka IB-over-Ethernet), iWARP, proprietary (eg Intel Omni-Path or AWS EFA)
- Transport properties: reliable vs. unreliable, connected vs. datagram

Can experiment with a software implementation on any Ethernet network via `rxe` (soft RoCE) driver.

Transport layering



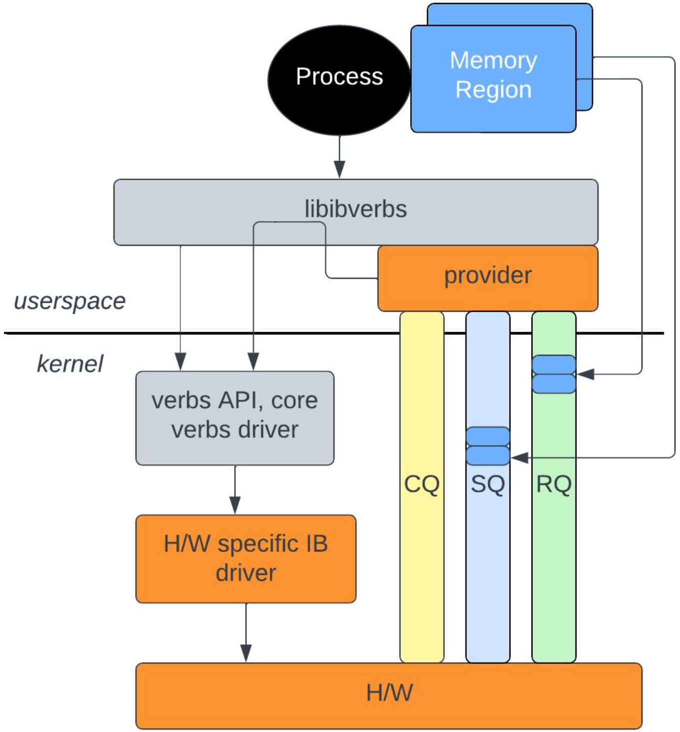
RDMA programming on Linux

RDMA apps on Linux are developed using `libibverbs` and `librdmacm` from rdma-core (https://github.com/linux-rdma/rdma-core), which provide C/C++ and Python bindings.

- Librdmacm provides connection establishment with IP addressing
- Libibverbs provides an API for other control and data path operations, realizing the abstract "verbs" interface as defined by the IBTA.

RDMA stack summary

Libibverbs, kernel driver stack, and userspace “provider” libraries work together to support RDMA programming.



Major object types

- `ibv_pd`: “Protection Domain” - high-level container for other objects
- `ibv_qp_ex`: “Queue pair” – encapsulates a queue for posting receive work requests and a queue for posting send work requests
- `ibv_cq_ex`: “Completion queue” – queue that receives completion notifications for receive and send work requests; may be attached to one or more work queues
- `ibv_mr`: “Memory region” – represents a memory buffer than can be targeted by work requests; has a local key (`L_Key`) for use in local work requests and a remote key (`R_Key`) that can be shared with a peer for use in remote one-sided operations.

Exchanging data via a Reliable Connected (RC) QP

Key steps in communicating via RC QP:

1. Register buffer(s) that will be used for communication
2. Create and connect a QP via librdmacm
3. Post receive work request(s)
4. Post send work request(s)
5. Poll for completion of work requests

Working source in examples directories of

`https://github.com/linux-rdma/rdma-core`

Initial set up

Create required objects including PD and CQ

```
struct ibv_pd *pd = ibv_alloc_pd(verbs_context);
if (!pd) {
    /* error handling… */

struct ibv_cq_init_attr_ex cq_attr = {
    .cqe = num_entries, cq_context = my_context, … };
struct ibv_cq_ex *cq = ibv_create_cq_ex(verbs_context, &cq_attr);
if (!cq) { /*...*/
```

Register memory

Allocate a buffer to hold data and register it with libibverbs:

```
void *buf = malloc(BUF_SIZE);
struct ibv_mr *mr = ibv_reg_mr(pd, buf, BUF_SIZE,
                               IBV_ACCESS_LOCAL_WRITE);
```

Connection establishment with librdmacm

Sockets-like, with an asynchronous event-driven interface

(Not strictly required, but provides an abstraction that covers multiple transports)

Connection establishment with librdmacm

Sockets-like, with an asynchronous event-driven interface

First create an “event channel”:

```
struct rdma_event_channel *channel;
channel = rdma_create_event_channel();
if (!channel) {
    /* error handling... */
}
```

Connection establishment with librdmacm

Both sides resolve server address:

```
struct rdma_addrinfo hints, *rai;

memset(&hints, 0, sizeof hints);
hints.ai_flags = RAI_PASSIVE;
hints.ai_port_space = RDMA_PS_TCP;
err = rdma_getaddrinfo(server_addr, port, &hints, &rai);
```

Connection establishment with librdmacm

Passive side creates and binds a listen “ID” and listens:

```
struct rdma_cm_id *listen_id;
err = rdma_create_id(channel, &listen_id, myctx, RDMA_PS_TCP);
err = rdma_bind_addr(listen_id, rai->ai_src_addr);
err = rdma_listen(listen_id, 0);
/* events will be generated for incoming connection requests */
```

Connection establishment with librdmacm

Active side creates ID and resolves address of server:

```
struct rdma_cm_id *cma_id;
err = rdma_create_id(channel, &cma_id, myctx, RDMA_PS_TCP);
err = rdma_resolve_addr(cma_id, rai->ai_src_addr,
                       rai->ai_dst_addr, 2000);
/* rdma_resolve_addr will generate an event on completion */
```

Connection establishment with librdmacm

Event loop for handling connection events:

```
struct rdma_cm_event *event;
while (true) {
    err = rdma_get_cm_event(test.channel, &event);
    switch (event->event) {
    case RDMA_CM_EVENT_ADDR_RESOLVED: /* etc */
    }
    rdma_ack_cm_event(event);
}
```

Connection establishment with librdmacm

Notable events to handle:

```
case RDMA_CM_EVENT_ADDR_RESOLVED: /* call rdma_resolve_route() */
case RDMA_CM_EVENT_ROUTE_RESOLVED: /* call rdma_create_qp()
                                      and rdma_connect() */
case RDMA_CM_EVENT_CONNECT_REQUEST: /* call rdma_accept() */
case RDMA_CM_EVENT_ESTABLISHED: /* start communication */
case RDMA_CM_EVENT_UNREACHABLE:
case RDMA_CM_EVENT_REJECTED: /* handle these and other errors */
case RDMA_CM_EVENT_DISCONNECTED: /* handle disconnection */
```

Post receive work request

Fill in a scatter list and queue work request to receive queue:

```
struct ibv_recv_wr wr, *bad_wr;
struct ibv_sge sge;
wr.sg_list = &sge;
wr.num_sge = 1;
wr.wr_id = (uint64_t) my_id;
sge.addr = (uintptr_t) buf;
sge.length = BUF_SIZE;
sge.lkey = mr->lkey;
err = ibv_post_recv(qp, wr, &bad_wr);
```

Post send work request

Fill in an gather list and queue work request to send queue:

```
ibv_wr_start(qp);
qp->wr_id = MY_WR_ID;
qp->wr_flags = 0; /* ordering/fencing etc */
ibv_wr_set_sge(qp, mr->lkey, (uintptr_t) buf, BUF_SIZE);
/* ibv_wr_set_sge_list() for multiple buffers */
err = ibv_wr_complete(qp);
```

Poll for completion

Non-blocking check for completion queue entries

```
struct ibv_poll_cq_attr attr = {};
err = ibv_start_poll(cq, &attr);
while (!err) {
     end_flag = true;
     /* consume cq->status, cq->wr_id, etc */
     err = ibv_next_poll(cq);
}
if (end_flag)
     ibv_end_poll(cq);
```

Differences for Unreliable Datagram (UD)

UD QPs are not connected and can receive messages from multiple sources

- Message size limited to one packet (path MTU)
- Must specify queue key (Q_Key) for destination
- Source information delivered with each message