RDMA Read and Write with IB Verbs[*]

Tarick Bedeir
Schlumberger
tbedeir@slb.com

October 22, 2010

Abstract

This paper explains the operations required to use the remote direct memory access read and write features exposed by the InfiniBand verbs library. Sample code illustrates connection setup and data transfer using read and write (or get/put) semantics.

1 Introduction

In my last paper [1], I described basic verbs applications that exchange data by posting sends and receives. In this paper I'll describe the construction of applications that use remote direct memory access, or RDMA [5].

Why would we want to use RDMA? Because it can provide lower latency and allow for zero-copy transfers (i.e., place data at the desired target location without buffering). Consider the iSCSI Extensions for RDMA, iSER [4]. The initiator, or client, issues a read request that includes a destination memory address in its local memory. The target, or server, responds by writing the desired data directly into the initiator's memory at the requested location. No buffering, minimal operating system involvement (since data is copied by the network adapters), and low latency – generally a winning formula.

Using RDMA with verbs is fairly straightforward: first register blocks of memory, then exchange memory descriptors, then post read/write operations. Registration is accomplished with a call to `ibv_reg_mr()`, which pins the block of memory in place (thus preventing it from being swapped out) and returns a `struct ibv_mr *` containing a `uint32_t` key allowing remote access to the registered memory. This key, along with the block's address, must then be exchanged with peers through some out-of-band mechanism. Peers can then use the key and address in calls to `ibv_post_send()` to post RDMA read and write requests. Some code might be instructive:

[*]Adapted from a blog post at http://thegeekinthecorner.wordpress.com/

```
/* PEER 1 */

const size_t SIZE = 1024;

char *buffer = malloc(SIZE);
struct ibv_mr *mr;
uint32_t my_key;
uint64_t my_addr;

mr = ibv_reg_mr(
  pd,
  buffer,
  SIZE,
  IBV_ACCESS_REMOTE_WRITE);

my_key = mr->rkey;
my_addr = (uint64_t)mr->addr;

/* exchange my_key and
my_addr with peer 2 */
```

```
/* PEER 2 */

const size_t SIZE = 1024;

char *buffer = malloc(SIZE);
struct ibv_mr *mr;
struct ibv_sge sge;
struct ibv_send_wr wr, *bad_wr;
uint32_t peer_key;
uint64_t peer_addr;

mr = ibv_reg_mr(
  pd,
  buffer,
  SIZE,
  IBV_ACCESS_LOCAL_WRITE);

/* get peer_key and
peer_addr from peer 1 */

strcpy(buffer, "Hello!");

memset(&wr, 0, sizeof(wr));

sge.addr = (uint64_t)buffer;
sge.length = SIZE;
sge.lkey = mr->lkey;

wr.sg_list = &sge;
wr.num_sge = 1;
wr.opcode = IBV_WR_RDMA_WRITE;

wr.wr.rdma.remote_addr = peer_addr;
wr.wr.rdma.rkey = peer_key;

ibv_post_send(qp, &wr, &bad_wr);
```

The last parameter to `ibv_reg_mr()` for peer 1, `IBV_ACCESS_REMOTE_WRITE`, specifies that we want peer 2 to have write access to the block of memory located at `buffer`.

Using this in practice is more complicated. The sample code that accompanies this post connects two hosts, exchanges memory region keys, reads from or writes to remote memory, then disconnects. The sequence is as follows:

1. Initialize context and register memory regions.
2. Establish connection.
3. Use send/receive model described in previous posts to exchange memory region keys between peers.
4. Post read/write operations.
5. Disconnect.

Each side of the connection will have two threads: the main thread, which processes connection events, and the thread polling the completion queue. In order to avoid deadlocks and race conditions, we arrange our operations so that only one thread at a time is posting work requests. To elaborate on the sequence above, after establishing the connection the client will:

1. Send its RDMA memory region key in a `MSG_MR` message.
2. Wait for the server's `MSG_MR` message containing its RDMA key.
3. Post an RDMA operation.
4. Signal to the server that it is ready to disconnect by sending a `MSG_DONE` message.

5. Wait for a `MSG_DONE` message from the server.
6. Disconnect.

Step one happens in the context of the RDMA connection event handler thread, but steps two through six are in the context of the verbs CQ polling thread. The sequence of operations for the server is similar:

1. Wait for the client's `MSG_MR` message with its RDMA key.
2. Send its RDMA key in a `MSG_MR` message.
3. Post an RDMA operation.
4. Signal to the client that it is ready to disconnect by sending a `MSG_DONE` message.
5. Wait for a `MSG_DONE` message from the client.
6. Disconnect.

Here all six steps happen in the context of the verbs CQ polling thread. Waiting for `MSG_DONE` is necessary otherwise we might close the connection before the peer's RDMA operation has completed. Note that we don't have to wait for the RDMA operation to complete before sending `MSG_DONE` – the InfiniBand specification requires that requests will be initiated in the order in which they're posted. This means that the peer won't receive `MSG_DONE` until the RDMA operation has completed.

2 Read/Write Demonstrations

The code for this sample [2] merges a lot of the client and server code from the previous set of posts for the sake of brevity (and to illustrate that they're nearly identical). Both the client (`rdma-client`) and the server (`rdma-server`) continue to operate different RDMA connection manager event loops, but they now share common verbs code – polling the CQ, sending messages, posting RDMA operations, etc. We also use the same code for both RDMA read and write operations since they're very similar. `rdma-server` and `rdma-client` take either "read" or "write" as their first command-line argument.

Let's start from the top of `rdma-common.c`, which contains verbs code common to both the client and the server. We first define our message structure. We'll use this to pass RDMA memory region (MR) keys between nodes and to signal that we're done.

```
struct message {
  enum {
    MSG_MR,
    MSG_DONE
  } type;

  union {
    struct ibv_mr mr;
  } data;
};
```

Our connection structure has been expanded to include memory regions for RDMA operations as well as the peer's MR structure and two state variables:

```
struct connection {
  struct rdma_cm_id *id;
  struct ibv_qp *qp;

  int connected;

  struct ibv_mr *recv_mr;
  struct ibv_mr *send_mr;
  struct ibv_mr *rdma_local_mr;
  struct ibv_mr *rdma_remote_mr;

  struct ibv_mr peer_mr;
  struct message *recv_msg;
  struct message *send_msg;

  char *rdma_local_region;
  char *rdma_remote_region;

  enum {
    SS_INIT,
    SS_MR_SENT,
    SS_RDMA_SENT,
    SS_DONE_SENT
  } send_state;

  enum {
    RS_INIT,
    RS_MR_RECV,
    RS_DONE_RECV
  } recv_state;
};
```

`send_state` and `recv_state` are used by the completion handler to properly sequence messages and RDMA operations between peers. This structure is initialized by `build_connection()`:

```
void build_connection(struct rdma_cm_id *id)
{
  struct connection *conn;
  struct ibv_qp_init_attr qp_attr;

  build_context(id->verbs);
  build_qp_attr(&qp_attr);

  TEST_NZ(rdma_create_qp(id, s_ctx->pd, &qp_attr));

  id->context = conn = (struct connection *)malloc(sizeof(struct connection));

  conn->id = id;
  conn->qp = id->qp;

  conn->send_state = SS_INIT;
  conn->recv_state = RS_INIT;

  conn->connected = 0;

  register_memory(conn);
  post_receives(conn);
}
```

Since we're using RDMA read operations, we have to set `initiator_depth` and `responder_resources` in `struct rdma_conn_param`. These control [3] the number of simultaneous outstanding RDMA read requests:

```
void build_params(struct rdma_conn_param *params)
{
  memset(params, 0, sizeof(*params));

  params->initiator_depth = params->responder_resources = 1;
  params->rnr_retry_count = 7; /* infinite retry */
}
```

Setting `rnr_retry_count` to 7 indicates that we want the adapter to resend indefinitely if the peer responds with a receiver-not-ready (RNR) error. RNRs happen when a send request is posted before a corresponding receive request is posted on the peer. Sends are posted with the `send_message()` function:

```
void send_message(struct connection *conn)
{
  struct ibv_send_wr wr, *bad_wr = NULL;
  struct ibv_sge sge;

  memset(&wr, 0, sizeof(wr));

  wr.wr_id = (uintptr_t)conn;
  wr.opcode = IBV_WR_SEND;
  wr.sg_list = &sge;
  wr.num_sge = 1;
  wr.send_flags = IBV_SEND_SIGNALED;

  sge.addr = (uintptr_t)conn->send_msg;
  sge.length = sizeof(struct message);
  sge.lkey = conn->send_mr->lkey;

  while (!conn->connected);

  TEST_NZ(ibv_post_send(conn->qp, &wr, &bad_wr));
}
```

`send_mr()` wraps this function and is used by `rdma-client` to send its MR to the server, which then prompts the server to send its MR in response, thereby kicking off the RDMA operations:

```
void send_mr(void *context)
{
  struct connection *conn = (struct connection *)context;

  conn->send_msg->type = MSG_MR;
  memcpy(&conn->send_msg->data.mr, conn->rdma_remote_mr, sizeof(struct ibv_mr));

  send_message(conn);
}
```

The completion handler does the bulk of the work. It maintains `send_state` and `recv_state`, replying to messages and posting RDMA operations as appropriate:

```
void on_completion(struct ibv_wc *wc)
{
  struct connection *conn = (struct connection *)(uintptr_t)wc->wr_id;

  if (wc->status != IBV_WC_SUCCESS)
    die("on_completion: status is not IBV_WC_SUCCESS.");

  if (wc->opcode & IBV_WC_RECV) {
    conn->recv_state++;

    if (conn->recv_msg->type == MSG_MR) {
      memcpy(&conn->peer_mr, &conn->recv_msg->data.mr, sizeof(conn->peer_mr));
      post_receives(conn); /* only rearm for MSG_MR */

      if (conn->send_state == SS_INIT) /* received peer's MR before sending ours, so send ours back */
        send_mr(conn);
    }

  } else {
    conn->send_state++;
    printf("send completed successfully.\n");
  }

  if (conn->send_state == SS_MR_SENT && conn->recv_state == RS_MR_RECV) {
    struct ibv_send_wr wr, *bad_wr = NULL;
    struct ibv_sge sge;

    if (s_mode == M_WRITE)
      printf("received MSG_MR. writing message to remote memory...\n");
    else
      printf("received MSG_MR. reading message from remote memory...\n");

    memset(&wr, 0, sizeof(wr));

    wr.wr_id = (uintptr_t)conn;
    wr.opcode = (s_mode == M_WRITE) ? IBV_WR_RDMA_WRITE : IBV_WR_RDMA_READ;
    wr.sg_list = &sge;
    wr.num_sge = 1;
    wr.send_flags = IBV_SEND_SIGNALED;
    wr.wr.rdma.remote_addr = (uintptr_t)conn->peer_mr.addr;
    wr.wr.rdma.rkey = conn->peer_mr.rkey;

    sge.addr = (uintptr_t)conn->rdma_local_region;
    sge.length = RDMA_BUFFER_SIZE;
    sge.lkey = conn->rdma_local_mr->lkey;

    TEST_NZ(ibv_post_send(conn->qp, &wr, &bad_wr));

    conn->send_msg->type = MSG_DONE;
    send_message(conn);

  } else if (conn->send_state == SS_DONE_SENT && conn->recv_state == RS_DONE_RECV) {
    printf("remote buffer: %s\n", get_peer_message_region(conn));
    rdma_disconnect(conn->id);
  }
}
```

Let's examine `on_completion()` in parts. First, the state update:

```
if (wc->opcode & IBV_WC_RECV) {
  conn->recv_state++;

  if (conn->recv_msg->type == MSG_MR) {
    memcpy(&conn->peer_mr, &conn->recv_msg->data.mr, sizeof(conn->peer_mr));
    post_receives(conn); /* only rearm for MSG_MR */

    if (conn->send_state == SS_INIT) /* received peer's MR before sending ours, so send ours back */
      send_mr(conn);
  }

} else {
  conn->send_state++;
  printf("send completed successfully.\n");
}
```

If the completed operation is a receive operation (i.e., if `wc-&gt;opcode` has `IBV_WC_RECV` set), then `recv_state` is incremented. If the received message is `MSG_MR`, we copy the received MR into our connection structure's `peer_mr` member, and rearm the receive slot. This is necessary to ensure that we receive the `MSG_DONE` message that follows the completion of the peer's RDMA operation. If we've received the peer's MR but haven't sent ours (as is the case for the server), we send our MR back by calling `send_mr()`. Updating `send_state` is uncomplicated.

Next we check for two particular combinations of `send_state` and `recv_state`:

```
if (conn->send_state == SS_MR_SENT && conn->recv_state == RS_MR_RECV) {
  struct ibv_send_wr wr, *bad_wr = NULL;
  struct ibv_sge sge;

  if (s_mode == M_WRITE)
    printf("received MSG_MR. writing message to remote memory...\n");
  else
    printf("received MSG_MR. reading message from remote memory...\n");

  memset(&wr, 0, sizeof(wr));

  wr.wr_id = (uintptr_t)conn;
  wr.opcode = (s_mode == M_WRITE) ? IBV_WR_RDMA_WRITE : IBV_WR_RDMA_READ;
  wr.sg_list = &sge;
  wr.num_sge = 1;
  wr.send_flags = IBV_SEND_SIGNALED;
  wr.wr.rdma.remote_addr = (uintptr_t)conn->peer_mr.addr;
  wr.wr.rdma.rkey = conn->peer_mr.rkey;

  sge.addr = (uintptr_t)conn->rdma_local_region;
  sge.length = RDMA_BUFFER_SIZE;
  sge.lkey = conn->rdma_local_mr->lkey;

  TEST_NZ(ibv_post_send(conn->qp, &wr, &bad_wr));

  conn->send_msg->type = MSG_DONE;
  send_message(conn);

} else if (conn->send_state == SS_DONE_SENT && conn->recv_state == RS_DONE_RECV) {
  printf("remote buffer: %s\n", get_peer_message_region(conn));
  rdma_disconnect(conn->id);
}
```

The first of these combinations is when we've both sent our MR and received the peer's MR. This indicates that we're ready to post an RDMA operation and post `MSG_DONE`. Posting an RDMA operation means building an RDMA work request. This is similar to a send work request, except that we specify an RDMA opcode and pass the peer's RDMA address/key:

```
wr.opcode = (s_mode == M_WRITE) ? IBV_WR_RDMA_WRITE : IBV_WR_RDMA_READ;

wr.wr.rdma.remote_addr = (uintptr_t)conn->peer_mr.addr;
wr.wr.rdma.rkey = conn->peer_mr.rkey;
```

Note that we're not required to use `conn-&gt;peer_mr.addr` for `remote_addr` – we could, if we wanted to, use any address falling within the bounds of the memory region registered with `ibv_reg_mr()`.

The second combination of states is `SS_DONE_SENT` and `RS_DONE_RECV`, indicating that we've sent `MSG_DONE` and received `MSG_DONE` from the peer. This means it is safe to print the message buffer and disconnect:

```
printf("remote buffer: %s\n", get_peer_message_region(conn));
rdma_disconnect(conn->id);
```

3 Conclusion

If everything's working properly, you should see the following when using RDMA writes:

```
$ ./rdma-server write
listening on port 47881.
received connection request.
send completed successfully.
received MSG_MR. writing message to remote memory...
send completed successfully.
send completed successfully.
remote buffer: message from active/client side with pid 20692
peer disconnected.
```

```
$ ./rdma-client write 192.168.0.1 47881
address resolved.
route resolved.
send completed successfully.
received MSG_MR. writing message to remote memory...
send completed successfully.
send completed successfully.
remote buffer: message from passive/server side with pid 26515
disconnected.
```

And when using RDMA reads:

```
$ ./rdma-server read
listening on port 47882.
received connection request.
send completed successfully.
received MSG_MR. reading message from remote memory...
send completed successfully.
send completed successfully.
remote buffer: message from active/client side with pid 20916
peer disconnected.
```

```
$ ./rdma-client read 192.168.0.1 47882
address resolved.
route resolved.
send completed successfully.
received MSG_MR. reading message from remote memory...
send completed successfully.
send completed successfully.
remote buffer: message from passive/server side with pid 26725
disconnected.
```

References

[1] T. Bedeir. Building an RDMA-Capable Application with IB Verbs. Technical report, HPC Advisory Council, 2010. Available from: http://www.hpcadvisorycouncil.com/pdf/building-an-rdma-capable-application-with-ib-verbs.pdf.

[2] T. Bedeir. RDMA Read/Write Sample Code [online]. 2010. Available from: https://sites.google.com/a/bedeir.com/home/rdma-read-write.tar.gz?attredirects=0&amp;d=1.

[3] rdma_accept(3) – linux man page [online]. Available from: http://linux.die.net/man/3/rdma_accept.

[4] Wikipedia. iSCSI Extensions for RDMA [online]. 2010. Available from: http://en.wikipedia.org/wiki/ISCSI_Extensions_for_RDMA.

[5] Wikipedia. Remote direct memory access [online]. 2010. Available from: http://en.wikipedia.org/wiki/RDMA.