Building an RDMA-Capable Application with IB Verbs*

Tarick Bedeir
Schlumberger
tbedeir@slb.com

August 21, 2010

Abstract

This paper explains the steps required to set up a connection between applications using InfiniBand verbs such that they may exchange data. The RDMA Connection Manager is used to automate and simplify the setup process. Sample code illustrates connection setup as well as data transfer using verbs' send/receive semantics.

1 Basics

If you're looking to build an application that uses InfiniBand natively, now would be a good time to ask yourself if you wouldn't be better off using one of InfiniBand's upper-layer protocols (ULPs), such as IP-over-IB/SDP or RDS, or, most obviously, MPI. Writing programs using the verbs library (*libibverbs*, but I'll refer to it as *ibverbs*) isn't hard, but why reinvent the wheel?

My own reasons for choosing ibverbs rather than MPI or any of the available ULPs had to do with comparative performance advantages over IPoIB and that my target applications are ill-suited to the MPI message-passing model. MPI-2's one-sided communication semantics would probably have worked, but for reasons irrelevant to this discussion MPI is/was a non-starter anyway.

Before looking at the details of programming with ibverbs, we should cover some prerequisites. I strongly recommend reading though the InfiniBand Trade Association's introduction[1] – chapters one and four in particular (only thirteen pages!). I'm also going to assume that you're comfortable programming in C, and have at least passing familiarity with sockets, MPI, and networking in general.

Our goal is to connect two applications such that they can exchange data. With reliable, connection-oriented sockets (i.e., `SOCK_STREAM`) , this involves setting up a listening socket on the server side, and connecting to it from the client side. Once a connection is established, either side can call `send()` and `recv()` to transfer data. This doesn't change much with ibverbs, but things are done in a much more explicit manner. The significant differences are:

1. You're not limited to `send()` and `recv()`. Reading and writing directly from/to remote memory (i.e., RDMA) is enormously useful.

2. Everything is asynchronous. Requests are made and notification is received at some point in the future that they have (or have not) completed.

3. At the application level, nothing is buffered. Receives have to be posted before sends. Memory used for a send request cannot be modified until the request has completed.

*Adapted from a series of blog posts at http://thegeekinthecorner.wordpress.com/
[1]Available at http://members.infinibandta.org/kwspub/Intro_to_IB_for_End_Users.pdf

4. Memory used for send/receive operations has to be registered, which effectively "pins" it such that it isn't swapped out.

So in an InfiniBand world, how do we establish connections between applications? If you've read the IBTA's introduction you'll know that the key components we need to set up are the queue pair (consisting of a send queue and a receive queue on which we post send and receive operations, respectively) and the completion queue, on which we receive notification that our operations have completed. Each side of a connection will have a send-receive queue pair and a completion queue (but note that the mapping between an individual send or receive queue and completion queues within any given application can be many-to-one). I'm going to focus on the reliable, connected service (similar to TCP) for now. In a future paper I'll explore the datagram service.

Building queue pairs and connecting them to each other, such that operations posted on one side are executed on the other, involves the following steps:

1. Create a protection domain (which associates queue pairs, completion queues, memory registrations, etc.), a completion queue, and a send-receive queue pair.
2. Determine the queue pair's address.
3. Communicate the address to the other node (through some out-of-band mechanism).
4. Transition the queue pair to the ready-to-receive (RTR) state and then the ready-to-send (RTS) state.
5. Post send, receive, etc. operations as appropriate.

Step four in particular isn't very pleasant, so we'll use a event-driven connection manager (CM) to connect queue pairs, manage state transitions, and handle errors. We could use the InfiniBand Connection Manager (ib_cm), but the RDMA Connection Manager (available in *librdmacm*, and also known as the connection manager abstraction[2]), uses a higher-level IP address/port number abstraction that should be familiar to anyone who's written a sockets program.

This gives us two distinct procedures, one for the passive (responder) side of the connection, and another for the active (initiator) side:

Passive Side

1. Create an event channel so that we can receive rdmacm events, such as connection-request and connection-established notifications.
2. Bind to an address.
3. Create a listener and return the port/address.
4. Wait for a connection request.
5. Create a protection domain, completion queue, and send-receive queue pair.
6. Accept the connection request.
7. Wait for the connection to be established.
8. Post operations as appropriate.

Active Side

1. Create an event channel so that we can receive rdmacm events, such as address-resolved, route-resolved, and connection-established notifications.

[2] https://wiki.openfabrics.org/tiki-index.php?page=IB+and+RDMA+Communication+Managers

2. Create a connection identifier.
3. Resolve the peer's address, which binds the connection identifier to a local RDMA device.
4. Create a protection domain, completion queue, and send-receive queue pair.
5. Resolve the route to the peer.
6. Connect.
7. Wait for the connection to be established.
8. Post operations as appropriate.

Both sides will share a fair amount of code – steps one, five, seven, and eight on the passive side are roughly equivalent to steps one, four, seven, and eight on the active side. It may or may not be worth pointing out that as with sockets once the connection has been established, both sides are peers. Making use of the connection requires that we post operations on the queue pair. Receive operations are posted (unsurprisingly) on the receive queue. On the send queue, we post send requests, RDMA read/write requests, and atomic operation requests.

The next two sections will describe in detail the construction of two applications: one will act as the passive/server side and the other will act as the active/client side. Once connected, the applications will exchange a simple message and disconnect.

If you haven't already, download and install the OpenFabrics software stack[3]. You'll need it to build the sample code provided in the next sections. Complete sample code, for both the passive side/server and the active side/client, is available online[4]. It's far from optimal, but I'll talk more about optimization in later papers.

2 Passive/Server Side

The previous section established the steps involved in setting up a connection on the passive side. Let's now examine them in detail. Since almost everything is handled asynchronously, we'll structure our code as an event-processing loop and a set of event handlers. First, the fundamentals:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <rdma/rdma_cma.h>

#define TEST_NZ(x) \
  do { if ( (x)) die("error: " #x " failed (returned non-zero)." ); } while (0)

#define TEST_Z(x)  \
  do { if (!(x)) die("error: " #x " failed (returned zero/null)."); } while (0)

static void die(const char *reason);

int main(int argc, char **argv)
{
  return 0;
}

void die(const char *reason)
{
  fprintf(stderr, "%s\n", reason);
  exit(EXIT_FAILURE);
}
```

Next, we set up an event channel, create an rdmacm ID (roughly analogous to a socket), bind it, and wait in a loop for events (namely, connection requests and connection-established notifications). `main()` becomes:

[3] Available at http://http://www.openfabrics.org/

[4] https://sites.google.com/a/bedeir.com/home/basic-rdma-client-server.tar.gz?attredirects=0&d=1

```
static void on_event(struct rdma_cm_event *event);

int main(int argc, char **argv)
{
  struct sockaddr_in addr;
  struct rdma_cm_event *event = NULL;
  struct rdma_cm_id *listener = NULL;
  struct rdma_event_channel *ec = NULL;
  uint16_t port = 0;

  memset(&addr, 0, sizeof(addr));
  addr.sin_family = AF_INET;

  TEST_Z(ec = rdma_create_event_channel());
  TEST_NZ(rdma_create_id(ec, &listener, NULL, RDMA_PS_TCP));
  TEST_NZ(rdma_bind_addr(listener, (struct sockaddr *)&addr));
  TEST_NZ(rdma_listen(listener, 10)); /* backlog=10 is arbitrary */

  port = ntohs(rdma_get_src_port(listener));

  printf("listening on port %d.\n", port);

  while (rdma_get_cm_event(ec, &event) == 0) {
    struct rdma_cm_event event_copy;

    memcpy(&event_copy, event, sizeof(*event));
    rdma_ack_cm_event(event);

    if (on_event(&event_copy))
      break;
  }

  rdma_destroy_id(listener);
  rdma_destroy_event_channel(ec);

  return 0;
}
```

`ec` is a pointer to the rdmacm event channel. `listener` is a pointer to the rdmacm ID for our listener. We specified `RDMA_PS_TCP` when creating it, which indicates that we want a connection-oriented, reliable queue pair. `RDMA_PS_UDP` would indicate a connectionless, unreliable queue pair.

We then bind this ID to a socket address. By setting the port, `addr.sin_port`, to zero, we instruct rdmacm to pick an available port. We've also indicated that we want to listen for connections on any available RDMA interface/device.

Our event loop gets an event from rdmacm, acknowledges the event, then processes it. Failing to acknowledge events will result in `rdma_destroy_id()` blocking. The event handler for the passive side of the connection is only interested in three events:

```
static void on_connect_request(struct rdma_cm_id *id);
static void on_connection(void *context);
static void on_disconnect(struct rdma_cm_id *id);

int on_event(struct rdma_cm_event *event)
{
  int r = 0;

  if (event->event == RDMA_CM_EVENT_CONNECT_REQUEST)
    r = on_connect_request(event->id);
  else if (event->event == RDMA_CM_EVENT_ESTABLISHED)
    r = on_connection(event->id->context);
  else if (event->event == RDMA_CM_EVENT_DISCONNECTED)
    r = on_disconnect(event->id);
  else
    die("on_event: unknown event.");

  return r;
}
```

rdmacm allows us to associate a `void *` context pointer with an ID. We'll use this to attach a connection context structure:

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

```
  struct ibv_mr *recv_mr;
  struct ibv_mr *send_mr;

  char *recv_region;
  char *send_region;
};
```

This contains a pointer to the queue pair (redundant, but simplifies the code slightly), two buffers (one for sends, the other for receives), and two memory regions (memory used for sends/receives has to be "registered" with the verbs library). When we receive a connection request, we first build our verbs context if it hasn't already been built. Then, after building our connection context structure, we pre-post our receives (more on this in a bit), and accept the connection request:

```
static void build_context(struct ibv_context *verbs);
static void build_qp_attr(struct ibv_qp_init_attr *qp_attr);
static void post_receives(struct connection *conn);
static void register_memory(struct connection *conn);

int on_connect_request(struct rdma_cm_id *id)
{
  struct ibv_qp_init_attr qp_attr;
  struct rdma_conn_param cm_params;
  struct connection *conn;

  printf("received connection request.\n");

  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->qp = id->qp;

  register_memory(conn);
  post_receives(conn);

  memset(&cm_params, 0, sizeof(cm_params));
  TEST_NZ(rdma_accept(id, &cm_params));

  return 0;
}
```

We postpone building the verbs context until we receive our first connection request because the rdmacm listener ID isn't necessarily bound to a specific RDMA device (and associated verbs context). However, the first connection request we receive will have a valid verbs context structure at `id->verbs`. Building the verbs context involves setting up a static context structure, creating a protection domain, creating a completion queue, creating a completion channel, and starting a thread to pull completions from the queue:

```
struct context {
  struct ibv_context *ctx;
  struct ibv_pd *pd;
  struct ibv_cq *cq;
  struct ibv_comp_channel *comp_channel;

  pthread_t cq_poller_thread;
};

static void * poll_cq(void *);

static struct context *s_ctx = NULL;

void build_context(struct ibv_context *verbs)
{
  if (s_ctx) {
    if (s_ctx->ctx != verbs)
      die("cannot handle events in more than one context.");

    return;
  }

  s_ctx = (struct context *)malloc(sizeof(struct context));

  s_ctx->ctx = verbs;

  TEST_Z(s_ctx->pd = ibv_alloc_pd(s_ctx->ctx));
```

```
  TEST_Z(s_ctx->comp_channel = ibv_create_comp_channel(s_ctx->ctx));
  TEST_Z(s_ctx->cq = ibv_create_cq(s_ctx->ctx, 10, NULL, s_ctx->comp_channel, 0));
  TEST_NZ(ibv_req_notify_cq(s_ctx->cq, 0));

  TEST_NZ(pthread_create(&s_ctx->cq_poller_thread, NULL, poll_cq, NULL));
}
```

Using a completion channel allows us to block the poller thread waiting for completions. We create the completion queue with `cqe` set to 10, indicating we want room for ten entries on the queue. This number should be set large enough that the queue isn't overrun.The poller waits on the channel, acknowledges the completion, rearms the completion queue (with `ibv_req_notify_cq()`), then pulls events from the queue until none are left:

```
static void on_completion(struct ibv_wc *wc);

void * poll_cq(void *ctx)
{
  struct ibv_cq *cq;
  struct ibv_wc wc;

  while (1) {
    TEST_NZ(ibv_get_cq_event(s_ctx->comp_channel, &cq, &ctx));
    ibv_ack_cq_events(cq, 1);
    TEST_NZ(ibv_req_notify_cq(cq, 0));

    while (ibv_poll_cq(cq, 1, &wc))
      on_completion(&wc);
  }

  return NULL;
}
```

Back to our connection request. After building the verbs context, we have to initialize the queue pair attributes structure:

```
void build_qp_attr(struct ibv_qp_init_attr *qp_attr)
{
  memset(qp_attr, 0, sizeof(*qp_attr));

  qp_attr->send_cq = s_ctx->cq;
  qp_attr->recv_cq = s_ctx->cq;
  qp_attr->qp_type = IBV_QPT_RC;

  qp_attr->cap.max_send_wr = 10;
  qp_attr->cap.max_recv_wr = 10;
  qp_attr->cap.max_send_sge = 1;
  qp_attr->cap.max_recv_sge = 1;
}
```

We first zero out the structure, then set the attributes we care about. `send_cq` and `recv_cq` are the send and receive completion queues, respectively. `qp_type` is set to indicate we want a reliable, connection-oriented queue pair. The queue pair capabilities structure, `qp_attr->cap`, is used to negotiate minimum capabilities with the verbs driver. Here we request ten pending sends and receives (at any one time in their respective queues), and one scatter/gather element (SGE; effectively a memory location/size tuple) per send or receive request. After building the queue pair initialization attributes, we call `rdma_create_qp()` to create the queue pair. We then allocate memory for our connection context structure (`struct connection`), and allocate/register memory for our send and receive operations:

```
const int BUFFER_SIZE = 1024;

void register_memory(struct connection *conn)
{
  conn->send_region = malloc(BUFFER_SIZE);
  conn->recv_region = malloc(BUFFER_SIZE);

  TEST_Z(conn->send_mr = ibv_reg_mr(
    s_ctx->pd,
    conn->send_region,
    BUFFER_SIZE,
    IBV_ACCESS_LOCAL_WRITE | IBV_ACCESS_REMOTE_WRITE));

  TEST_Z(conn->recv_mr = ibv_reg_mr(
    s_ctx->pd,
```

```
    conn->recv_region,
    BUFFER_SIZE,
    IBV_ACCESS_LOCAL_WRITE | IBV_ACCESS_REMOTE_WRITE));
}
```

Here we allocate two buffers, one for sends and the other for receives, then register them with verbs.We specify we want local write and remote write access to these memory regions. The next step in our connection-request event handler (which is getting rather long) is the pre-posting of receives. The reason it is necessary to post receive work requests (WRs) before accepting the connection is that the underlying hardware won't buffer incoming messages – if a receive request has not been posted to the work queue, the incoming message is rejected and the peer will receive a receiver-not-ready (RNR) error. I'll discuss this further in another paper, but for now it suffices to say that receives have to be posted before sends. We'll enforce this by posting receives before accepting the connection, and posting sends after the connection is established. Posting receives requires that we build a receive work-request structure and then post it to the receive queue:

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

  wr.wr_id = (uintptr_t)conn;
  wr.next = NULL;
  wr.sg_list = &sge;
  wr.num_sge = 1;

  sge.addr = (uintptr_t)conn->recv_region;
  sge.length = BUFFER_SIZE;
  sge.lkey = conn->recv_mr->lkey;

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

The (arbitrary) `wr_id` field is used to store a connection context pointer. Finally, having done all this setup, we're ready to accept the connection request. This is accomplished with a call to `rdma_accept()`.

The next event we need to handle is `RDMA_CM_EVENT_ESTABLISHED`, which indicates that a connection has been established. This handler is simple – it merely posts a send work request:

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

  snprintf(
    conn->send_region,
    BUFFER_SIZE,
    "message from passive/server side with pid %d",
    getpid());

  printf("connected. posting send...\n");

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

  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_region;
  sge.length = BUFFER_SIZE;
  sge.lkey = conn->send_mr->lkey;

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

  return 0;
}
```

This isn't radically different from the code we used to post a receive, except that send requests specify an opcode. Here, `IBV_WR_SEND` indicates a send request that must match a corresponding receive request on the peer. Other options include RDMA write, RDMA read, and various atomic operations. Specifying `IBV_SEND_SIGNALED` in `wr.send_flags` indicates that we want completion notification for this send request.

The last rdmacm event we want to handle is `RDMA_CM_EVENT_DISCONNECTED`, where we'll perform some cleanup:

```
int on_disconnect(struct rdma_cm_id *id)
{
  struct connection *conn = (struct connection *)id->context;

  printf("peer disconnected.\n");

  rdma_destroy_qp(id);

  ibv_dereg_mr(conn->send_mr);
  ibv_dereg_mr(conn->recv_mr);

  free(conn->send_region);
  free(conn->recv_region);

  free(conn);

  rdma_destroy_id(id);

  return 0;
}
```

All that's left for us to do is handle completions pulled from the completion queue:

```
void on_completion(struct ibv_wc *wc)
{
  if (wc->status != IBV_WC_SUCCESS)
    die("on_completion: status is not IBV_WC_SUCCESS.");

  if (wc->opcode & IBV_WC_RECV) {
    struct connection *conn = (struct connection *)(uintptr_t)wc->wr_id;

    printf("received message: %s\n", conn->recv_region);

  } else if (wc->opcode == IBV_WC_SEND) {
    printf("send completed successfully.\n");
  }
}
```

Recall that in `post_receives()` we set `wr_id` to the connection context structure. And that's it! Building is straightforward, but don't forget `-lrdmacm`.

3 Active/Client Side

We've looked, in detail, at the passive side. We've also outlined the rough steps involved in setting up a connection on the active side. Let's now examine the active side in detail. Since the code is very similar, I'll focus on the differences.

On the command line, our client takes a server host name or IP address and a port number. We use `getaddrinfo()` to translate these two parameters to `struct sockaddr`. This requires that we include a new header file:

```
#include <netdb.h>
```

We also modify `main()` to determine the server's address (using `getaddrinfo()`):

```
const int TIMEOUT_IN_MS = 500; /* ms */

int main(int argc, char **argv)
{
  struct addrinfo *addr;
  struct rdma_cm_event *event = NULL;
  struct rdma_cm_id *conn= NULL;
  struct rdma_event_channel *ec = NULL;

  if (argc != 3)
    die("usage: client <server-address> <server-port>");

  TEST_NZ(getaddrinfo(argv[1], argv[2], NULL, &addr));

  TEST_Z(ec = rdma_create_event_channel());
```

```
  TEST_NZ(rdma_create_id(ec, &conn, NULL, RDMA_PS_TCP));
  TEST_NZ(rdma_resolve_addr(conn, NULL, addr->ai_addr, TIMEOUT_IN_MS));

  freeaddrinfo(addr);

  while (rdma_get_cm_event(ec, &event) == 0) {
    struct rdma_cm_event event_copy;

    memcpy(&event_copy, event, sizeof(*event));
    rdma_ack_cm_event(event);

    if (on_event(&event_copy))
      break;
  }

  rdma_destroy_event_channel(ec);

  return 0;
}
```

Whereas with sockets we'd establish a connection with a simple call to `connect()`, with rdmacm we have a more elaborate connection process:

1. Create an ID with `rdma_create_id()`.
2. Resolve the server's address with `rdma_resolve_addr()`, passing a pointer to `struct sockaddr`.
3. Wait for the `RDMA_CM_EVENT_ADDR_RESOLVED` event, then call `rdma_resolve_route()` to resolve a route to the server.
4. Wait for the `RDMA_CM_EVENT_ROUTE_RESOLVED` event, then call `rdma_connect()` to connect to the server.
5. Wait for `RDMA_CM_EVENT_ESTABLISHED`, which indicates that the connection has been established.

`main()` starts this off by calling `rdma_resolve_addr()`, and the handlers for the subsequent events complete the process:

```
static int on_addr_resolved(struct rdma_cm_id *id);
static int on_route_resolved(struct rdma_cm_id *id);

int on_event(struct rdma_cm_event *event)
{
  int r = 0;

  if (event->event == RDMA_CM_EVENT_ADDR_RESOLVED)
    r = on_addr_resolved(event->id);
  else if (event->event == RDMA_CM_EVENT_ROUTE_RESOLVED)
    r = on_route_resolved(event->id);
  else if (event->event == RDMA_CM_EVENT_ESTABLISHED)
    r = on_connection(event->id->context);
  else if (event->event == RDMA_CM_EVENT_DISCONNECTED)
    r = on_disconnect(event->id);
  else
    die("on_event: unknown event.");

  return r;
}
```

In our passive side code, `on_connect_request()` initialized `struct connection` and built the verbs context. On the active side, this initialization happens as soon as we have a valid verbs context pointer – in `on_addr_resolved()`:

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

  struct ibv_mr *recv_mr;
  struct ibv_mr *send_mr;

  char *recv_region;
  char *send_region;
```

```
  int num_completions;
};

int on_addr_resolved(struct rdma_cm_id *id)
{
  struct ibv_qp_init_attr qp_attr;
  struct connection *conn;

  printf("address resolved.\n");

  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->num_completions = 0;

  register_memory(conn);
  post_receives(conn);

  TEST_NZ(rdma_resolve_route(id, TIMEOUT_IN_MS));

  return 0;
}
```

Note the `num_completions` field in `struct connection`: we'll use it to keep track of the number of completions we've processed for this connection. The client will disconnect after processing two completions: one send, and one receive. The next event we expect is `RDMA_CM_EVENT_ROUTE_RESOLVED`, where we call `rdma_connect()`:

```
int on_route_resolved(struct rdma_cm_id *id)
{
  struct rdma_conn_param cm_params;

  printf("route resolved.\n");

  memset(&cm_params, 0, sizeof(cm_params));
  TEST_NZ(rdma_connect(id, &cm_params));

  return 0;
}
```

Our `RDMA_CM_EVENT_ESTABLISHED` handler also differs in that we're sending a different message:

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

  snprintf(
    conn->send_region,
    BUFFER_SIZE,
    "message from active/client side with pid %d",
    getpid());

  printf("connected. posting send...\n");

  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_region;
  sge.length = BUFFER_SIZE;
  sge.lkey = conn->send_mr->lkey;

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

  return 0;
}
```

Perhaps most importantly, our completion callback now counts the number of completions and disconnects after two are processed:

```
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)
    printf("received message: %s\n", conn->recv_region);
  else if (wc->opcode == IBV_WC_SEND)
    printf("send completed successfully.\n");
  else
    die("on_completion: completion isn't a send or a receive.");

  if (++conn->num_completions == 2)
    rdma_disconnect(conn->id);
}
```

Lastly, our `RDMA_CM_EVENT_DISCONNECTED` handler is modified to signal to the event loop in `main()` that it should exit:

```
int on_disconnect(struct rdma_cm_id *id)
{
  struct connection *conn = (struct connection *)id->context;

  printf("disconnected.\n");

  rdma_destroy_qp(id);

  ibv_dereg_mr(conn->send_mr);
  ibv_dereg_mr(conn->recv_mr);

  free(conn->send_region);
  free(conn->recv_region);

  free(conn);

  rdma_destroy_id(id);

  return 1; /* exit event loop */
}
```

This completes our active side/client application.

4 Conclusion

If you've managed to build everything properly, your output on the server side should look like the following:

```
$ /sbin/ifconfig ib0 | grep "inet addr"
          inet addr:192.168.0.1  Bcast:192.168.0.255  Mask:255.255.255.0
$ ./server
listening on port 45267.
received connection request.
connected. posting send...
received message: message from active/client side with pid 29717
send completed successfully.
peer disconnected.
```

And on the client side:

```
$ ./client 192.168.0.1 45267
address resolved.
route resolved.
```

```
connected. posting send...
send completed successfully.
received message: message from passive/server side with pid 14943
disconnected.
```

The IP address passed to `client` is the IP address of the IPoIB interface on the server. As far as I can tell it's an rdmacm requirement that the `struct sockaddr` passed to `rdma_resolve_addr()` point to an IPoIB interface.

So we now have a working pair of applications. The next paper in this series will look at reading and writing directly from/to remote memory.