INFINIBAND, VERBS, RDMA

Sample Code Repository (https://thegeekinthecorner.wordpress.com/2013/10/04/sample-code-repository/)

Updated code for the RDMA tutorials is now hosted at GitHub:

https://github.com/tarickb/the-geek-in-the-corner (https://github.com/tarickb/the-geek-in-the-corner)

*Special thanks to **SeongJae P. (https://thegeekinthecorner.wordpress.com/about/#comment-263)** for the Makefile fixes.*

October 4, 2013 | Categories: **InfiniBand, Verbs, RDMA (https://thegeekinthecorner.wordpress.com/category/infiniband-verbs-rdma/)** | **6 Comments (https://thegeekinthecorner.wordpress.com/2013/10/04/sample-code-repository/#comments)**

RDMA tutorial PDFs (https://thegeekinthecorner.wordpress.com/2013/02/02/rdma-tutorial-pdfs/)

In cooperation with the **HPC Advisory Council (http://www.hpcadvisorycouncil.com/)**, I've reformatted three of my RDMA tutorials for easier offline reading. You can find them, along with several papers on InfiniBand, GPUs, and other interesting topics, at the **HPC Training (http://www.hpcadvisorycouncil.com/network_training.php)** page. For easier access, here are my three papers:

- **Building an RDMA-Capable Application with IB Verbs (http://www.hpcadvisorycouncil.com/pdf/building-an-rdma-capable-application-with-ib-verbs.pdf)**
- **RDMA Read and Write with IB Verbs (http://www.hpcadvisorycouncil.com/pdf/rdma-read-and-write-with-ib-verbs.pdf)**
- **Basic Flow Control for RDMA Transfers (http://www.hpcadvisorycouncil.com/pdf/vendor_content/basic-flow-control-for-rdma-transfers.pdf)**

February 2, 2013 | Categories: **InfiniBand, Verbs, RDMA (https://thegeekinthecorner.wordpress.com/category/infiniband-verbs-rdma/)** | **8 Comments (https://thegeekinthecorner.wordpress.com/2013/02/02/rdma-tutorial-pdfs/#comments)**

Basic flow control for RDMA transfers (https://thegeekinthecorner.wordpress.com/2012/12/19/basic-flow-control-for-rdma-transfers/)

Commenter **Matt** **recently asked (https://thegeekinthecorner.wordpress.com/2010/09/28/rdma-read-and-write-with-ib-verbs/#comment-188)** about sending large(r) amounts of data:

> *… I'm wondering if you would be able to provide some pointers or even examples that send very large amounts of data. e.g. sending files up to or > 2GB. Your examples use 1024 byte buffers. I suspect there is an efficient way of doing this given that there is a 2**31 limit for the message size.*
>
> *I should point out that I don't have lots of memory available as it's used for other things.*

There are many ways to do this, but since I've already covered **using send/receive operations (https://thegeekinthecorner.wordpress.com/2010/08/14/building-an-rdma-capable-application-with-ib-verbs-part-3-the-client/)** and **using RDMA read/write (https://thegeekinthecorner.wordpress.com/2010/09/28/rdma-read-and-write-with-ib-verbs/)**, this would be a good occasion to combine elements of both and talk about how to handle flow control in general. I'll also talk a bit about the RDMA-write-with-immediate-data (`IBV_WR_RDMA_WRITE_WITH_IMM`) operation, and I'll illustrate these methods with a **sample (https://github.com/tarickb/the-geek-in-the-corner/tree/master/03_file-transfer)** that transfers, using RDMA, a file specified on the command line.

As in previous posts, our sample consists of a server and a client. The server waits for connections from the client. The client does essentially two things after connecting to the server: it sends the name of the file it's transferring, and then sends the contents of the file. We won't concern ourselves with the nuts and bolts of establishing a connection; that's been covered in **previous (https://thegeekinthecorner.wordpress.com/2010/08/14/building-an-rdma-capable-application-with-ib-verbs-part-2-the-server/) posts (https://thegeekinthecorner.wordpress.com/2010/08/14/building-an-rdma-capable-application-with-ib-verbs-part-3-the-client/)**. Instead, we'll focus on synchronization and the flow control. What I've done with the code for this post though is invert the structure I built for my **RDMA read/write post (https://thegeekinthecorner.wordpress.com/2010/09/28/rdma-read-and-write-with-ib-verbs/)** — there, I had the connection management code separated into `client.c` and `server.c` with the completion-processing code in `common.c` whereas here I've centralized the connection management in `common.c` and divided the completion processing between `client.c` and `server.c`.

Back to our example. There are many ways we could orchestrate the transfer of an entire file from client to server. For instance:

- Load the entire file into client memory, connect to the server, wait for the server to post a set of receives, then issue a send operation (on the client side) to copy the contents to the server.

- Load the entire file into client memory, register the memory, pass the region details to the server, let it issue an RDMA read to copy the entire file into its memory, then write the contents to disk.
- As above, but issue an RDMA write to copy the file contents into server memory, then signal it to write to disk.
- Open the file on the client, read one chunk, wait for the server to post a receive, then post a send operation on the client side, and loop until the entire file is sent.
- As above, but use RDMA reads.
- As above, but use RDMA writes.

Loading the entire file into memory can be impractical for large files, so we'll skip the first three options. Of the remaining three, I'll focus on using RDMA writes so that I can illustrate the use of the RDMA-write-with-immediate-data operation, something I've been meaning to talk about for a while. This operation is similar to a regular RDMA write except that the initiator can "attach" a 32-bit value to the write operation. Unlike regular RDMA writes, RDMA writes with immediate data require that a receive operation be posted on the target's receive queue. The 32-bit value will be available when the completion is pulled from the target's queue.

Update, Dec. 26: *Roland D. rather helpfully pointed out that RDMA write with immediate data isn't supported by iWARP adapters. We could rewrite to use an RDMA write (without immediate data) followed by a send, but this is left as an exercise for the reader.*

Now that we've decided we're going to break up the file into chunks, and write the chunks one at a time into the server's memory, we need to find a way to ensure we don't write chunks faster than the server can process them. We'll do this by having the server send explicit messages to the client when it's ready to receive data. The client, on the other hand, will use writes with immediate data to signal the server. The sequence looks something like this:

1. Server starts listening for connections.
2. Client posts a receive operation for a flow-control message and initiates a connection to the server.
3. Server posts a receive operation for an RDMA write with immediate data and accepts the connection from the client.
4. Server sends the client its target memory region details.
5. Client re-posts a receive operation then responds by writing the name of the file to the server's memory region. The immediate data field contains the length of the file name.
6. Server opens a file descriptor, re-posts a receive operation, then responds with a message indicating it is ready to receive data.
7. Client re-posts a receive operation, reads a chunk from the input file, then writes the chunk to the server's memory region. The immediate data field contains the size of the chunk in bytes.
8. Server writes the chunk to disk, re-posts a receive operation, then responds with a message indicating it is ready to receive data.
9. Repeat steps 7, 8 until there is no data left to send.
10. Client re-posts a receive operation, then initiates a zero-byte write to the server's memory. The immediate data field is set to zero.
11. Server responds with a message indicating it is done.
12. Client closes the connection.
13. Server closes the file descriptor.

A diagram may be helpful:

Looking at this sequence we see that the server only ever sends small messages to the client and only ever receives RDMA writes from the client. The client only ever executes RDMA writes and only ever receives small messages from the server.

Let's start by looking at the server. The connection-establishment details are now hidden behind `rc_init()`, which sets various callback functions, and `rc_server_loop()`, which runs an event loop:

```
int main(int argc, char **argv)
{
  rc_init(
    on_pre_conn,
    on_connection,
    on_completion,
    on_disconnect);

  printf("waiting for connections. interrupt (^C) to exit.\n");

  rc_server_loop(DEFAULT_PORT);

  return 0;
}
```

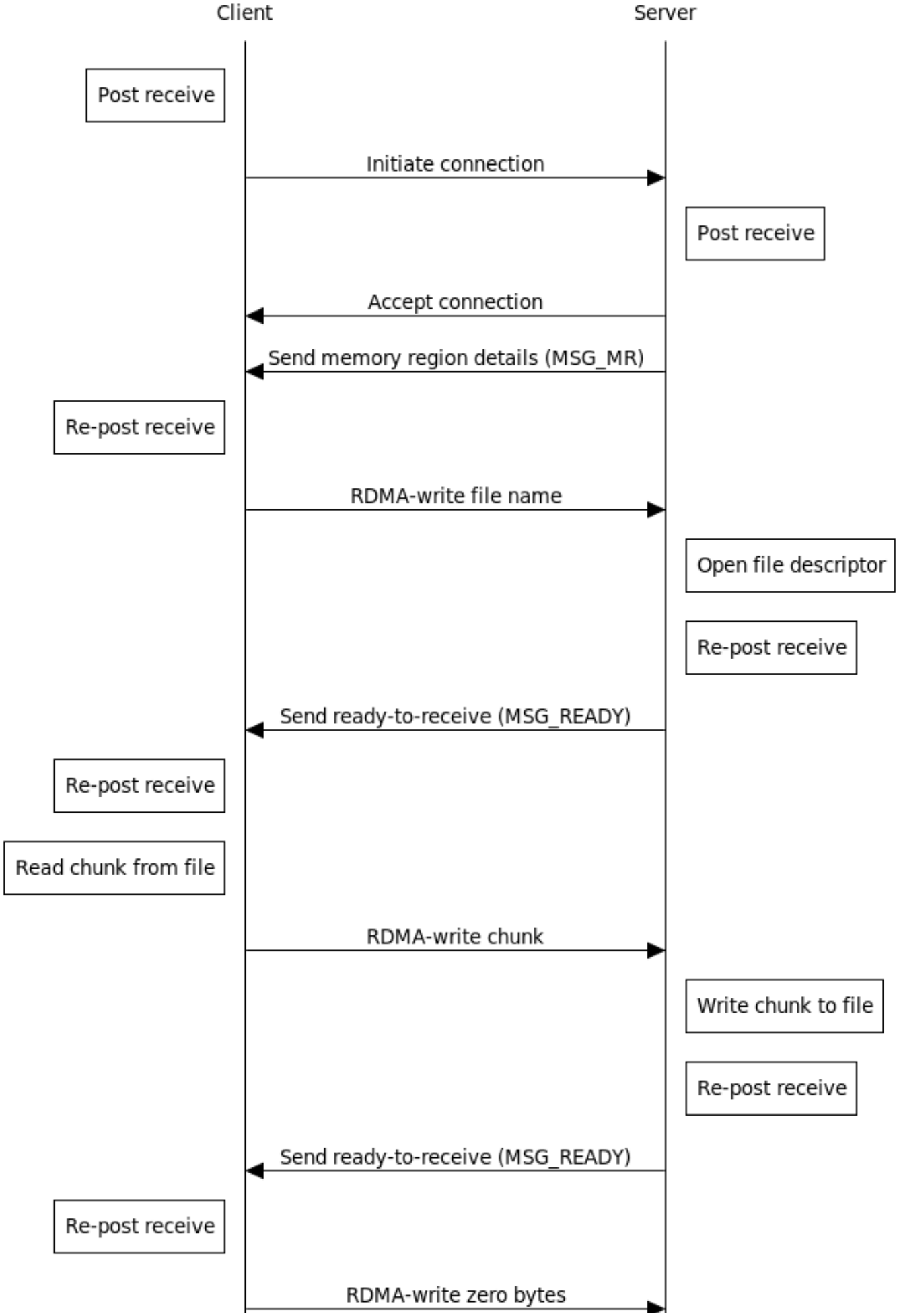
The callback names are fairly obvious: `on_pre_conn()` is called when a connection request is received but before it is accepted, `on_connection()` is called when a connection is established, `on_completion()` is called when an entry is pulled from the completion queue, and `on_disconnect()` is called upon disconnection.

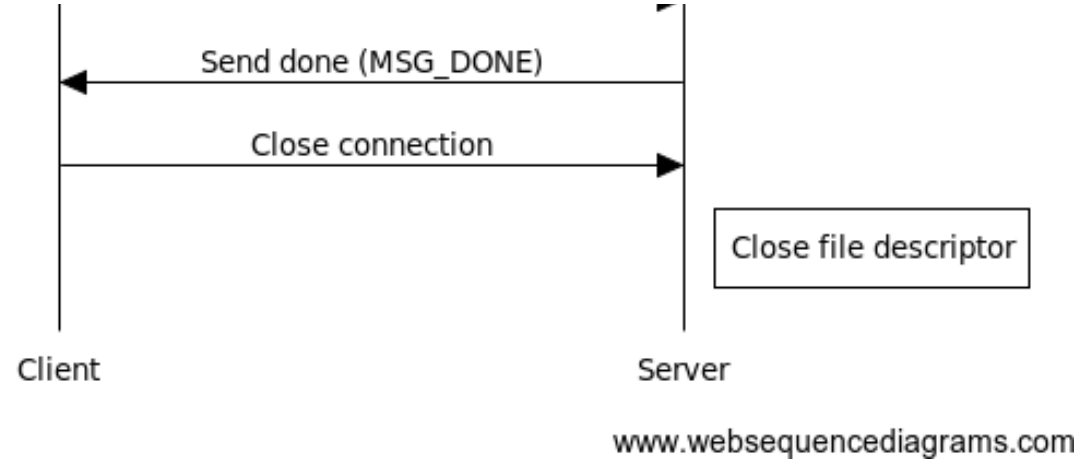
In `on_pre_conn()`, we allocate a structure to contain various connection context fields (a buffer to contain data from the client, a buffer from which to send messages to the client, etc.) and post a receive work request for the client's RDMA writes:

```
static void post_receive(struct rdma_cm_id *id)
{
  struct ibv_recv_wr wr, *bad_wr = NULL;

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

  wr.wr_id = (uintptr_t)id;
  wr.sg_list = NULL;
  wr.num_sge = 0;

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

What's interesting here is that we're setting `sg_list = NULL` and `num_sge = 0`. Incoming RDMA write requests will specify a target memory address, and since this work request is only ever going to match incoming RDMA writes, we don't need to use `sg_list` and `num_sge` to specify a location in memory for the receive. After the connection is established, `on_connection()` sends the memory region details to the client:

File Transfer






```
static void on_connection(struct rdma_cm_id *id)
{
  struct conn_context *ctx = (struct conn_context *)id->context;

  ctx->msg->id = MSG_MR;
  ctx->msg->data.mr.addr = (uintptr_t)ctx->buffer_mr->addr;
  ctx->msg->data.mr.rkey = ctx->buffer_mr->rkey;

  send_message(id);
}
```

This prompts the client to begin issuing RDMA writes, which trigger the `on_completion()` callback:

```
 1  static void on_completion(struct ibv_wc *wc)
 2  {
 3    struct rdma_cm_id *id = (struct rdma_cm_id *)(uintptr_t)wc->wr_id;
 4    struct conn_context *ctx = (struct conn_context *)id->context;
 5
 6    if (wc->opcode == IBV_WC_RECV_RDMA_WITH_IMM) {
 7      uint32_t size = ntohl(wc->imm_data);
 8
 9      if (size == 0) {
10        ctx->msg->id = MSG_DONE;
11        send_message(id);
12
13        // don't need post_receive() since we're done with this connection
14
15      } else if (ctx->file_name[0]) {
16        ssize_t ret;
17
18        printf("received %i bytes.\n", size);
19
20        ret = write(ctx->fd, ctx->buffer, size);
21
22        if (ret != size)
23          rc_die("write() failed");
24
25        post_receive(id);
26
27        ctx->msg->id = MSG_READY;
28        send_message(id);
29
30      } else {
31        memcpy(ctx->file_name, ctx->buffer, (size > MAX_FILE_NAME) ? MAX_FILE_NAME : size);
32        ctx->file_name[size - 1] = '\0';
33
34        printf("opening file %s\n", ctx->file_name);
35
36        ctx->fd = open(ctx->file_name, O_WRONLY | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
37
38        if (ctx->fd == -1)
39          rc_die("open() failed");
40
41        post_receive(id);
42
43        ctx->msg->id = MSG_READY;
44        send_message(id);
45      }
46    }
47  }
```

We retrieve the immediate data field in line 7 and convert it from network byte order to host byte order. We then test three possible conditions:

1. If `size == 0`, the client has finished writing data (lines 9–14). We acknowledge this with `MSG_DONE`.
2. If the first byte of `ctx->file_name` is set, we already have the file name and have an open file descriptor (lines 15–30). We call `write()` to append the client's data to our open file then reply with `MSG_READY`, indicating we're ready to accept more data.
3. Otherwise, we have yet to receive the file name (lines 30–45). We copy it from the incoming buffer, open a file descriptor, then reply with `MSG_READY` to indicate we're ready to receive data.

Upon disconnection, in `on_disconnect()`, we close the open file descriptor and tidy up memory registrations, etc. And that's it for the server!

On the client side, `main()` is a little more complex in that we need to pass the server host name and port into `rc_client_loop()`:

```
int main(int argc, char **argv)
{
  struct client_context ctx;

  if (argc != 3) {
    fprintf(stderr, "usage: %s <server-address> <file-name>\n", argv[0]);
    return 1;
  }

  ctx.file_name = basename(argv[2]);
  ctx.fd = open(argv[2], O_RDONLY);

  if (ctx.fd == -1) {
    fprintf(stderr, "unable to open input file \"%s\"\n", ctx.file_name);
    return 1;
  }

  rc_init(
    on_pre_conn,
    NULL, // on connect
    on_completion,
    NULL); // on disconnect

  rc_client_loop(argv[1], DEFAULT_PORT, &ctx);

  close(ctx.fd);

  return 0;
}
```

We don't provide on-connection or on-disconnect callbacks because these events aren't especially relevant to the client. The `on_pre_conn()` callback is fairly similar to the server's, except that the connection context structure is pre-allocated, and the receive work request we post (in `post_receive()`) requires a memory region:

```
static void post_receive(struct rdma_cm_id *id)
{
  struct client_context *ctx = (struct client_context *)id->context;

  struct ibv_recv_wr wr, *bad_wr = NULL;
  struct ibv_sge sge;

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

  wr.wr_id = (uintptr_t)id;
  wr.sg_list = &sge;
  wr.num_sge = 1;

  sge.addr = (uintptr_t)ctx->msg;
  sge.length = sizeof(*ctx->msg);
  sge.lkey = ctx->msg_mr->lkey;

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

We point `sg_list` to a buffer large enough to hold a `struct message`. The server will use this to pass along flow control messages. Each message will trigger a call to `on_completion()`, which is where the client does the bulk of its work:

```
static void on_completion(struct ibv_wc *wc)
{
  struct rdma_cm_id *id = (struct rdma_cm_id *)(uintptr_t)(wc->wr_id);
  struct client_context *ctx = (struct client_context *)id->context;

  if (wc->opcode & IBV_WC_RECV) {
    if (ctx->msg->id == MSG_MR) {
      ctx->peer_addr = ctx->msg->data.mr.addr;
      ctx->peer_rkey = ctx->msg->data.mr.rkey;

      printf("received MR, sending file name\n");
      send_file_name(id);
    } else if (ctx->msg->id == MSG_READY) {
      printf("received READY, sending chunk\n");
      send_next_chunk(id);
    } else if (ctx->msg->id == MSG_DONE) {
      printf("received DONE, disconnecting\n");
      rc_disconnect(id);
      return;
    }

    post_receive(id);
  }
}
```

This matches the sequence described above. Both `send_file_name()` and `send_next_chunk()` ultimately call `write_remote()`:

```
static void write_remote(struct rdma_cm_id *id, uint32_t len)
{
  struct client_context *ctx = (struct client_context *)id->context;

  struct ibv_send_wr wr, *bad_wr = NULL;
  struct ibv_sge sge;

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

  wr.wr_id = (uintptr_t)id;
  wr.opcode = IBV_WR_RDMA_WRITE_WITH_IMM;
  wr.send_flags = IBV_SEND_SIGNALED;
  wr.imm_data = htonl(len);
```

```
  wr.wr.rdma.remote_addr = ctx->peer_addr;
  wr.wr.rdma.rkey = ctx->peer_rkey;

  if (len) {
    wr.sg_list = &sge;
    wr.num_sge = 1;

    sge.addr = (uintptr_t)ctx->buffer;
    sge.length = len;
    sge.lkey = ctx->buffer_mr->lkey;
  }

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

This RDMA request differs from those used in earlier posts in two ways: we set `opcode` to `IBV_WR_RDMA_WRITE_WITH_IMM`, and we set `imm_data` to the length of our buffer.

That wasn't too bad, was it? If everything's working as expected, you should see the following:

```
ib-host-1$ ./server
waiting for connections. interrupt (^C) to exit.
opening file test-file
received 10485760 bytes.
received 10485760 bytes.
received 5242880 bytes.
finished transferring test-file
^C

ib-host-1$ md5sum test-file
5815ed31a65c5da9745764c887f5f777  test-file
```

```
ib-host-2$ dd if=/dev/urandom of=test-file bs=1048576 count=25
25+0 records in
25+0 records out
26214400 bytes (26 MB) copied, 3.11979 seconds, 8.4 MB/s

ib-host-2$ md5sum test-file
5815ed31a65c5da9745764c887f5f777  test-file

ib-host-2$ ./client ib-host-1 test-file
received MR, sending file name
received READY, sending chunk
received READY, sending chunk
received READY, sending chunk
received READY, sending chunk
received DONE, disconnecting
```

If instead you see an error during memory registration, such as the following, you may need to increase your locked memory resource limits:

```
error: ctx->buffer_mr = ibv_reg_mr(rc_get_pd(), ctx->buffer, BUFFER_SIZE, IBV_ACCESS_LOCAL_WRITE) failed (returned zero/null).
```

The **OpenMPI FAQ** (http://www.open-mpi.org/faq/?category=openfabrics#ib-locked-pages-user) has a good explanation of how to do this.

Once more, the sample code is **available here** (https://github.com/tarickb/the-geek-in-the-corner/tree/master/03_file-transfer) .

Updated, Dec. 21: *Updated post to describe locked memory limit errors, and updated sample code to: check for* `ibv_reg_mr()` *errors; use* `basename()` *of file path rather than full path; add missing* `mode` *parameter to* `open()` *call; add missing library reference to Makefile. Thanks Matt.*

Updated, Oct. 4: *Sample code is now at* ***https://github.com/tarickb/the-geek-in-the-corner/tree/master/03_file-transfer*** *(https://github.com/tarickb/the-geek-in-the-corner/tree/master/03_file-transfer) .*

December 19, 2012 | Categories: **InfiniBand, Verbs, RDMA** (https://thegeekinthecorner.wordpress.com/category/infiniband-verbs-rdma/) | **32 Comments** (https://thegeekinthecorner.wordpress.com/2012/12/19/basic-flow-control-for-rdma-transfers/#comments)

RDMA read and write with IB verbs (https://thegeekinthecorner.wordpress.com/2010/09/28/rdma-read-and-write-with-ib-verbs/)

In my **last** (https://thegeekinthecorner.wordpress.com/2010/08/13/building-an-rdma-capable-application-with-ib-verbs-part-1-basics/) **few** (https://thegeekinthecorner.wordpress.com/2010/08/14/building-an-rdma-capable-application-with-ib-verbs-part-2-the-server/) **posts** (https://thegeekinthecorner.wordpress.com/2010/08/14/building-an-rdma-capable-application-with-ib-verbs-part-3-the-client/) I wrote about building basic verbs applications that exchange data by posting sends and receives. In this post I'll describe the construction of applications that use remote direct memory access, or **RDMA** (http://en.wikipedia.org/wiki/RDMA) .

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** (http://en.wikipedia.org/wiki/ISCSI_Extensions_for_RDMA)). 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:

```
/* PEER 1 */

const size_t SIZE = 1024;

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

```
/* 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;
```

```
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 */
```

```
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.

The **code for this sample (https://github.com/tarickb/the-geek-in-the-corner/tree/master/02_read-write)** 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 (http://linux.die.net/man/3/rdma_accept)** 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->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->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);
```

And that's it. 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.
```

One again, the sample code is **available here (https://github.com/tarickb/the-geek-in-the-corner/tree/master/02_read-write)** .

Updated, Oct. 4: *Sample code is now at* ***https://github.com/tarickb/the-geek-in-the-corner/tree/master/02_read-write (https://github.com/tarickb/the-geek-in-the-corner/tree/master/02_read-write)*** .

September 28, 2010 | Categories: **InfiniBand, Verbs, RDMA (https://thegeekinthecorner.wordpress.com/category/infiniband-verbs-rdma/)** | **76 Comments (https://thegeekinthecorner.wordpress.com/2010/09/28/rdma-read-and-write-with-ib-verbs/#comments)**

On DAPL (https://thegeekinthecorner.wordpress.com/2010/08/14/on-dapl/)

One of InfiniBand's upper-layer protocols is DAPL (for Direct Access Programming Library), available both in user mode (as uDAPL) and in kernel mode (as kDAPL). DAPL is the result of an attempt early on to provide a common API for both InfiniBand and iWARP (Ethernet) devices. For whatever reason, DAPL never really seems to have caught on. Intel MPI uses (used?) it, as does OpenMPI (though OpenMPI also uses verbs natively through the openib BTL).

Having written and (successfully, albeit certainly not easily) deployed an application written against the uDAPL API, I feel confident in saying that it is to be avoided at all costs. It may appear, initially, to be easier to use than verbs, but it isn't. The documentation isn't always internally consistent, and I've had to work around several painfully-debugged issues with the uDAPL library. Given that verbs works with both InfiniBand and iWARP adapters, I can't see a compelling reason to use DAPL.

To paraphrase someone much better-informed on the matter than I, "who the **** still uses DAPL?"

Enough said.

August 14, 2010 | Categories: InfiniBand, Verbs, RDMA (https://thegeekinthecorner.wordpress.com/category/infiniband-verbs-rdma/) , Rants (https://thegeekinthecorner.wordpress.com/category/rants/) | 2 Comments (https://thegeekinthecorner.wordpress.com/2010/08/14/on-dapl/#comments)

Building an RDMA-capable application with IB verbs, part 3: the client (https://thegeekinthecorner.wordpress.com/2010/08/14/building-an-rdma-capable-application-with-ib-verbs-part-3-the-client/)

In my last post (https://thegeekinthecorner.wordpress.com/2010/08/13/building-an-rdma-capable-application-with-ib-verbs-part-1-basics/) , I covered the steps involved in building the passive/server side of our basic verbs application. In this post I'll discuss the active/client side. Since the code is very similar, I'll focus on the differences. To recap, the steps involved in connecting to the passive/server side are:

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

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 by calling `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 */
}
```

And that's it. Once again, the source code for both the client and the server **is available here (https://github.com/tarickb/the-geek-in-the-corner/tree/master/01_basic-client-server)** . If you've managed to build everything properly, your output should look like the following:

On the server side:

```
$ /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 post in this series will look at reading and writing directly from/to remote memory.

Updated, Oct. 4: *Sample code is now at* ***https://github.com/tarickb/the-geek-in-the-corner/tree/master/01_basic-client-server (https://github.com/tarickb/the-geek-in-the-corner/tree/master/01_basic-client-server)*** .

August 14, 2010 | Categories: **InfiniBand, Verbs, RDMA (https://thegeekinthecorner.wordpress.com/category/infiniband-verbs-rdma/)** | **14 Comments (https://thegeekinthecorner.wordpress.com/2010/08/14/building-an-rdma-capable-application-with-ib-verbs-part-3-the-client/#comments)**

Building an RDMA-capable application with IB verbs, part 2: the server (https://thegeekinthecorner.wordpress.com/2010/08/14/building-an-rdma-capable-application-with-ib-verbs-part-2-the-server/)

In my last post (https://thegeekinthecorner.wordpress.com/2010/08/13/building-an-rdma-capable-application-with-ib-verbs-part-1-basics/) , I covered some basics and described the steps involved in setting up a connection from both the passive/server and active/client sides. In this post I'll describe the passive side. To recap, the steps involved are:

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.

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:

```
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 post, 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`. Complete code, for both the passive side/server and the active side/client, **is available here (https://github.com/tarickb/the-geek-in-the-corner/tree/master/01_basic-client-server)** . It's far from optimal, but I'll talk more about optimization in later posts.

In my next post I'll describe the implementation of the active side.

Updated, Oct. 4: *Sample code is now at* ***https://github.com/tarickb/the-geek-in-the-corner/tree/master/01_basic-client-server (https://github.com/tarickb/the-geek-in-the-corner/tree/master/01_basic-client-server)*** .

August 14, 2010 | Categories: InfiniBand, Verbs, RDMA (https://thegeekinthecorner.wordpress.com/category/infiniband-verbs-rdma/) | 12 Comments (https://thegeekinthecorner.wordpress.com/2010/08/14/building-an-rdma-capable-application-with-ib-verbs-part-2-the-server/#comments)

Building an RDMA-capable application with IB verbs, part 1: basics (https://thegeekinthecorner.wordpress.com/2010/08/13/building-an-rdma-capable-application-with-ib-verbs-part-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 (https://cw.infinibandta.org/document/dl/7268)** — 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:

- You're not limited to `send()` and `recv()`. Reading and writing directly from/to remote memory (i.e., RDMA) is enormously useful.
- Everything is asynchronous. Requests are made and notification is received at some point in the future that they have (or have not) completed.
- 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.
- 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 later posts 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 (https://wiki.openfabrics.org/tiki-index.php?page=IB+and+RDMA+Communication+Managers)**) 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. 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 posts 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 (http://http://www.openfabrics.org/)** . You'll need it to build the sample code provided in the next posts.

Updated, Nov. 6: *Fixed link to IBTA introduction.*

August 13, 2010 | Categories: **InfiniBand, Verbs, RDMA (https://thegeekinthecorner.wordpress.com/category/infiniband-verbs-rdma/)** | **39 Comments (https://thegeekinthecorner.wordpress.com/2010/08/13/building-an-rdma-capable-application-with-ib-verbs-part-1-basics/#comments)**