Basic Flow Control for RDMA Transfers*

Tarick Bedeir
Schlumberger
tbedeir@slb.com

January 16, 2013

Abstract

This paper describes a basic flow-control protocol for the transfer of large messages between hosts using the InfiniBand verbs library. The protocol breaks messages into segments and then uses the remote direct memory access (RDMA) write with immediate data operation to transfer these segments. To illustrate, an example is provided that copies a file between two hosts.

1 Motivation

A commenter at *The Geek in the Corner* asked 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 > 2 GB. 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. I described send/receive operations and the fundamentals of InfiniBand verbs in "Building an RDMA-Capable Application with IB Verbs" [2], and the use of RDMA read/write operations in "RDMA Read and Write with IB Verbs" [3]. For this problem we will combine elements of both approaches and discuss how to handle flow control in general. I will also briefly discuss the RDMA-write-with-immediate-data (`IBV_WR_RDMA_WRITE_WITH_IMM`) operation, and I will illustrate these methods with a sample [1] that transfers, using RDMA, a file specified on the command line.

As in previous papers, 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 is transferring, and then sends the contents of the file. We will not concern ourselves with the nuts and bolts of establishing a connection; that was covered in a previous paper [2]. Instead, we will focus on developing a protocol for synchronization and flow control.

2 Protocol

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.

*Adapted from a blog post at *The Geek in the Corner*, http://thegeekinthecorner.wordpress.com/.

- 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 will skip the first three options. Of the remaining three, I will focus on using RDMA writes so that I can illustrate the use of the RDMA-write-with-immediate-data operation, something that I have been intending to discuss for some time. 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.

Now that we decided that we are going to break up the file into chunks, and write the chunks one at a time into the server's memory, we must find a way to ensure that we do not write chunks faster than the server can process them. We will do this by instructing the server to send explicit messages to the client when it is 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 (see Figure 1 for a graphical illustration):

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 that 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 that it is ready to receive data.
9. Repeat steps 7 and 8 until there are 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 that it is finished.
12. Client closes the connection.
13. Server closes the file descriptor.

This sequence lends itself well to an event-driven implementation—the client acts in response to a message from the server, and vice versa. Because the client only processes three types of messages from the server, and the server only processes the immediate data from incoming RDMA writes, our completion processing functions, where we do the bulk of our work, will be relatively simple.

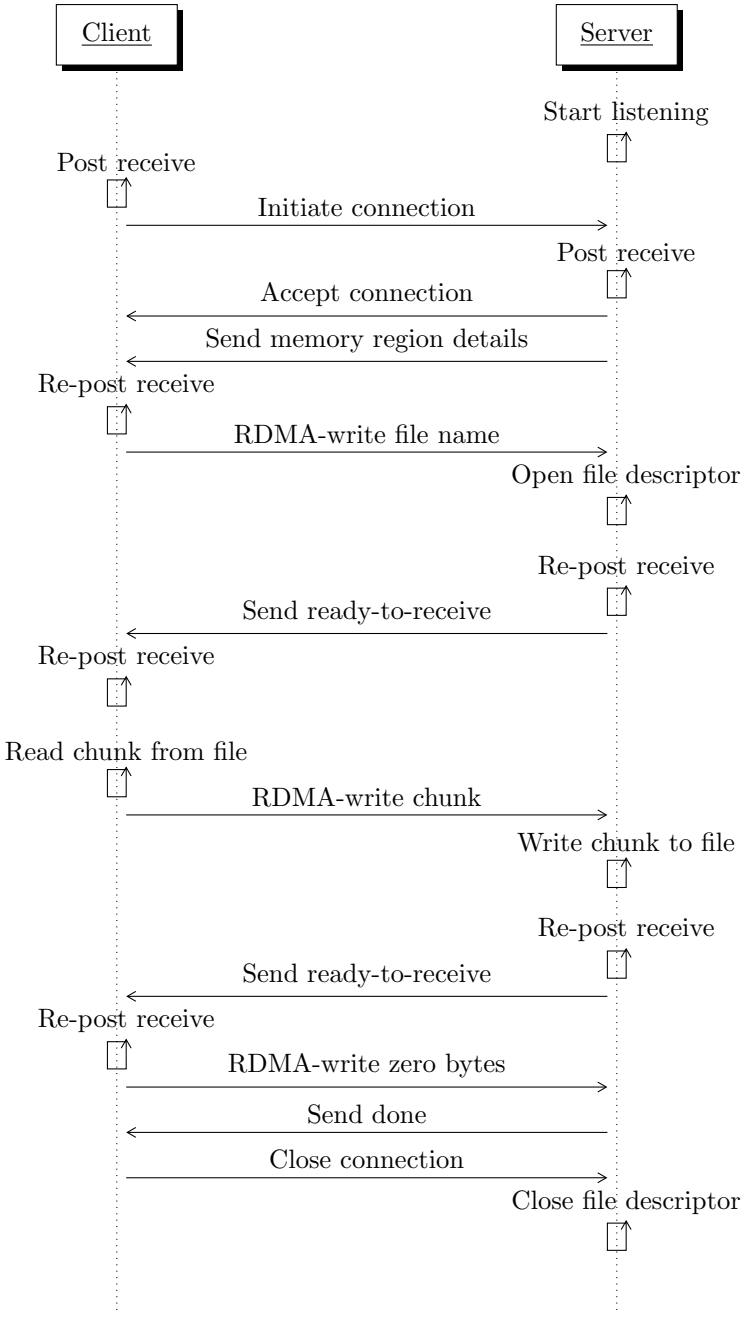

Figure 1: Sequence diagram for client-server file transfer

3 Implementation

Our implementation [1] is divided into two pieces: the client, in `client.c`, which is responsible for initiating a connection to the server and transferring the user-specified file, and the server, in `server.c`, which accepts connections and writes received files. Both the client and the server share some common connection-setup code, which we have in `common.c`.

For this example I have inverted the structure from my RDMA read/write example [3]; 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 have centralized the connection management in `common.c` and divided the completion processing between `client.c` and `server.c`. There are six public connection management functions, declared in `common.h`:

- `rc_init()`: Initializes the connection management code and sets the callback functions that will be used by event handlers.
- `rc_client_loop()`: Initiates a connection to the specified server address (and port), then waits in a loop, processing events, until the connection is closed.
- `rc_disconnect()`: Disconnects the current connection.
- `rc_die()`: Print a message and abort the program.
- `rc_get_pd()`: Gets the active InfiniBand protection domain (required to register memory with `ibv_reg_mr()`).
- `rc_server_loop()`: Starts listening on the specified port and waits in a loop, processing events.

3.1 Server

We start by examining the server. Here we see the server-side use of the common connection management functions `rc_init()` and `rc_server_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 is interesting here is that we are setting `sg_list = NULL` and `num_sge = 0`. Incoming RDMA write requests will specify a target memory address, and because this work request will only match incoming RDMA writes, we do not 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:

```
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 15–29). We call `write()` to append the client's data to our open file then reply with `MSG_READY`, indicating that we are 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 that we are ready to receive data.

Upon disconnection, in `on_disconnect()`, we close the open file descriptor and tidy up memory registrations.

3.2 Client

On the client side, `main()` is a little more complex in that we must 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 do not provide on-connection or on-disconnection callbacks because these events are not especially relevant to the client. The `on_pre_conn()` callback is 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 papers in two ways: we set `opcode` to `IBV_WR_RDMA_WRITE_WITH_IMM`, and we set `imm_data` to the length of our buffer.

4 Caveats

4.1 iWARP

Ethernet (iWARP) adapters do not support `IBV_WR_RDMA_WRITE_WITH_IMM`. The protocol we developed in this paper will therefore only work with InfiniBand host channel adapters. Building a protocol that works with iWARP would require only minor modifications—instead of `IBV_WR_RDMA_WRITE_WITH_IMM`, the client would first post an `IBV_WR_RDMA_WRITE` request, and follow that by posting an `IBV_WR_SEND` to send the size of the buffer. The server would be modified such that the receive operation points to a registered buffer that would contain the size of the buffer sent by the client. This modification is left as an exercise for the reader.

4.2 Performance

The protocol we developed is simple and easy to explain, but far from the fastest way to transfer large amounts of data using InfiniBand verbs. A faster protocol would, at a minimum, overlap disk reads with RDMA writes. This would require that the server make several buffers available and that the client write to them in a round-robin fashion. A faster, multi-buffer protocol will be described in a future paper.

5 Conclusion

If the code compiles and runs as expected, we 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 we see an error during memory registration, such as the following, we may need to increase the locked memory resource limit:

```
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 [4] explains how to do this.

References

[1] RDMA Flow Control Basics Sample Code [online]. 2013. URL: https://sites.google.com/a/bedeir.com/home/rdma-file-transfer.tar.gz.

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

[3] T. Bedeir. RDMA Read and Write with IB Verbs. Technical report, HPC Advisory Council, 2010. URL: http://www.hpcadvisorycouncil.com/pdf/rdma-read-and-write-with-ib-verbs.pdf.

[4] OpenMPI Frequently Asked Questions [online]. URL: http://www.open-mpi.org/faq/?category=openfabrics#ib-locked-pages-user.