Dissecting a Small InfiniBand Application Using the Verbs API

Gregory Kerr*
College of Computer and Information Science
Northeastern University
Boston, MA
kerrg@ccs.neu.edu

Abstract

InfiniBand is a switched fabric interconnect. The InfiniBand specification does not define an API. However the OFED package, libibverbs, has become the default API on Linux and Solaris systems. Sparse documentation exists for the verbs API. The simplest InfiniBand program provided by OFED, `ibv_rc_pingpong`, is about 800 lines long. The semantics of using the verbs API for this program is not obvious to the first time reader. This paper will dissect the `ibv_rc_pingpong` program in an attempt to make clear to users how to interact with verbs. This work was motivated by an ongoing project to include direct InfiniBand support for the DMTCP checkpointing package [1].

1 Introduction

The program `ibv_rc_pingpong` can be found at openfabrics.org[1], under the `''examples/''` directory of the OFED tarball. The source code used for this document is from version 1.1.4. The `ibv_rc_pingpong` program sets up a connection between two nodes running InfiniBand adapters and transfers data. Let's begin by looking at the program in action. In this paper, I will refer to two nodes: `client` and `server`. There are various command line flags that may be set when running the program. It is important to note that the information contained within this document is based on the assumption that the program has been run with no

*This work was partially supported by the National Science Foundation under Grant OCI-0960978.
[1]http://www.openfabrics.org/downloads/OFED/

```
[user@server]$ ibv_rc_pingpong
  local address:  LID 0x0008, QPN 0x580048, PSN 0x2a166f, GID ::
  remote address: LID 0x0003, QPN 0x580048, PSN 0x5c3f21, GID ::
8192000 bytes in 0.01 seconds = 5167.64 Mbit/sec
1000 iters in 0.01 seconds = 12.68 usec/iter

[user@client]$ ibv_rc_pingpong server
  local address:  LID 0x0003, QPN 0x580048, PSN 0x5c3f21, GID ::
  remote address: LID 0x0008, QPN 0x580048, PSN 0x2a166f, GID ::
8192000 bytes in 0.01 seconds = 5217.83 Mbit/sec
1000 iters in 0.01 seconds = 12.56 usec/iter
```

Figure 1: manpage entries for the verbs API

```
ibv_get_device_list (3), ibv_open_device (3), ibv_alloc_pd (3),
ibv_reg_mr (3), ibv_create_cq (3), ibv_create_qp (3), ibv_modify_qp (3),
ibv_post_recv (3), ibv_post_send (3), ibv_ack_cq_events (3)
```

command line flags configured. Configuring these flags will alter much of the program's behavior.

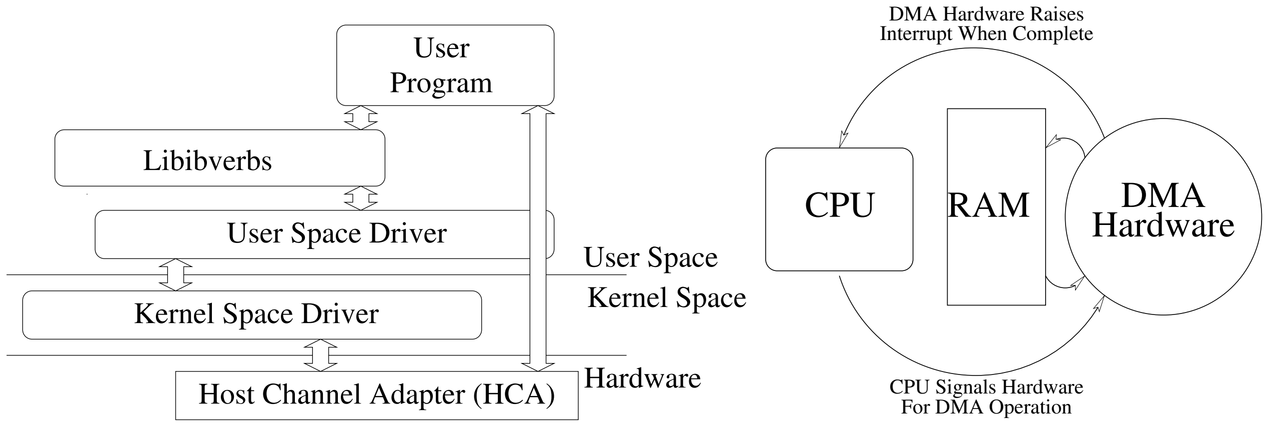
Since both nodes run the same executable, the "client" is the instance that is launched with a hostname as an argument. The LID, QPN, and PSN will be explained later.

Before we delve into the actual code, please look at a list of all verbs API functions which will be used for our purposes. I encourage the reader to pause and read the man page for each of these.

2 Layers

There are multiple drivers, existing in kernel and userspace, involved in a connection. See Figure 2a. To explain it simply, much of the connection setup work goes through the kernel driver, as speed is not a critical concern in that area.

The user space drivers are involved in function calls such as `ibv_post_send` and `ibv_post_recv`. Instead of going through kernel space, they interact directly with the hardware by writing to a segment of mapped memory. Avoiding kernel traps is one way to decrease the overall latency of each operation.



(a) Software Stack Layers (b) Direct Memory Access

Figure 2: Layers and DMA

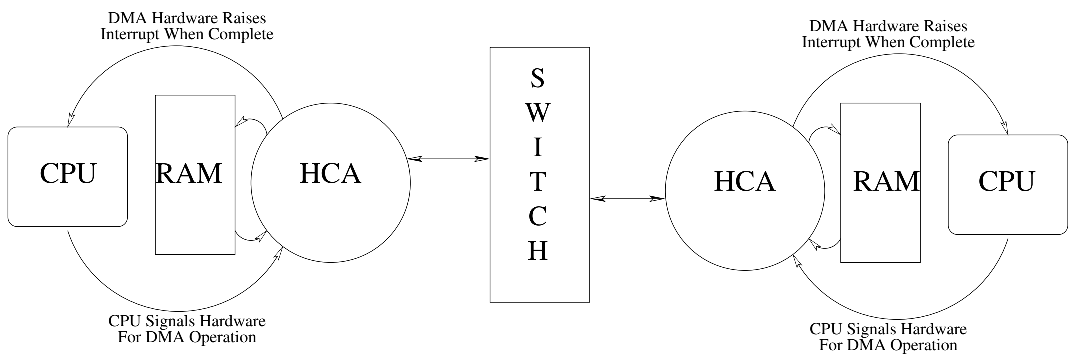


Figure 3: InfiniBand Remote Direct Memory Acess

3 Remote Direct Memory Access

One of the key concepts in InfiniBand is Remote Direct Memory Access (RDMA). This allows a node to directly access the memory of another node on the subnet, without involving the remote CPU or software layers.

Remember the key concepts of Direct Memory Access (DMA) as illustrated by Figure 2b.

In the DMA, the CPU sends a command to the hardware to begin a DMA operation. When the operation finishes, the DMA hardware raises an interrupt with the CPU, signaling completion. The RDMA concept used in InfiniBand is similar to DMA, except with two nodes accessing each other's memory; one node is the sender and one is the receiver.

Figure 2 illustrates an InfiniBand connection. In this case the DMA Hardware is the Host Channel Adapter (HCA), and the two HCAs are connected, through a switch, to each other. The HCA is InfiniBand's version of a network card; it is the hardware local to each node that facilitates communications. This allows an HCA on one node to use another HCA to perform DMA operations on a remote node.

4 Overview

The `ibv_rc_pingpong` program does the following.

1. Reserves memory from the operating system for sending and receiving data
2. Allocates resources from the verbs API
3. Uses a TCP socket to exchange InfiniBand connection information
4. Creates a connection between two InfiniBand ports
5. Transfers data over the connection
6. Acknowledges the successful completion of the transfer

5 Data Transfer Modes

The InfiniBand specification states four different connection types: Reliable Connection (RC), Unreliable Connection (UC), Reliable Datagram (RD), Unreliable Datagram (UD). This program, `ibv_rc_pingpong` uses a simple RC model. RD is not supported by current hardware.

The difference between reliable and unreliable is predictable – in a reliable connection data is transferred in order and guaranteed to arrive. In an unreliable connection neither of those guarantees is made.

A connection type is an association strictly between two hosts. In a datagram, a host is free to communicate with any other host on the subnet.

6 Queue Based Model

The InfiniBand hardware processes requests from the client software through requests, which are placed into queues. To send messages between nodes, each node must have at minimum three queues: a Send Queue (SQ), Receive Queue (RQ), and Completion Queue (CQ).

In a reliable connection, used in the `ibv_rc_pingpong` program, queue pairs on two distinct hosts compromise an end-to-end context. They send messages to each other, and only each other. This paper restricts itself to this mode.

The queues themselves exist on the HCA. However the libibverbs will return to the user a data structure which corresponds with the QP. While the library will create the QP, the user assumes the responsibility of "connecting" the QP with the remote node. This is generally done by opening an out-of-band socket connection, trading the identification numbers for the queues, and then updating the hardware with the information.

More recently, `librdma_cm` (an OFED library for connection management) allows a user to create and connect QPs through library calls reminiscent of POSIX sockets. Those calls are outside the scope of this document.

6.1 Posting Work Requests to Queues

To send and receive data in the InfiniBand connection (end-to-end context), work requests, which become Work Queue Entries (WQE, pronounced "wookie") are posted to the appropriate queue. These work requests point to lists of scatter/gather elements (each element has an address and size associated with it). This is a means of writing to and reading from buffers which are non-contiguous in memory.

The memory buffers must be registered with the hardware; that process is explained later. **Memory buffers must be posted to the receive queue before the remote host can post any sends**. The `ibv_rc_pingpong` program posts numerous buffers to the receive queue at the beginning of execution, and then repopulates the queue as necessary. A receive queue entry is processed when the remote host posts a send operation.

When the hardware processes the work request, a Completion Queue Entry (CQE, pronounced "cookie") is placed on the CQ. There is a sample of code showing how to handle completion events in `ibv_ack_cq_events (3)`.

7 Connecting the Calls

The table below which the function calls used in `ibv_rc_pingpong` to create a connection, and the order in which they are called.

struct ibv_device **	ibv_get_device_list(int * num_devices);
struct ibv_context *	ibv_open_device(struct ibv_device * device);
struct ibv_pd *	/* protection domain */ ibv_alloc_pd(struct ibv_context * ctx);
struct ibv_mr *	/* memory region */ ibv_reg_mr(struct ibv_pd * pd, void * addr, size_t length, enum ibv_access_flags access);
struct ibv_cq *	/* completion queue */ ibv_create_cq(struct ibv_context * context, int cqe, void * cq_context, struct ibv_comp_channel channel, int comp_vector);
struct ibv_qp *	/* queue pair */ ibv_create_qp(struct ibv_pd * pd, struct ibv_qp_init_attr * qp_init_attr);
int	ibv_modify_qp(struct ibv_qp * qp, struct ibv_qp_attr * attr, int attr_mask);

This table introduces the resources which are allocated in the process of creating a connection. These resources will be explained in detail later.

8 Allocating Resources

8.1 Creating a Context

The first function call to the verbs API made by the `ibv_rc_pingpong` source code is here:

```
    dev_list = ibv_get_device_list(NULL);
```

As the man page states, this function returns a list of available HCAs.

The argument to the function is an optional pointer to an `int`, which the library uses to specify the size of the list.

Next it populates the `pingpong_context` structure with the function `pp_init_ctx`.

The `pingpong_context` structure wraps all the resources associated with a connection into one unit.

Listing 1: struct pingpong_context

```
struct pingpong_context {
    struct ibv_context  *context;
    struct ibv_comp_channel *channel;
```

```
    struct ibv_pd        *pd;
    struct ibv_mr        *mr;
    struct ibv_cq        *cq;
    struct ibv_qp        *qp;
    void                 *buf;
    int                  size;
    int                  rx_depth;
    int                  pending;
    struct ibv_port_attr     portinfo;
};
```

Listing 2: Initializing the struct pingpong_context

```
ctx = pp_init_ctx(ib_dev, size, rx_depth, ib_port,
                            use_event, !servername);
```

The `ib_dev` argument is a `struct device *` and comes from `dev_list`. The argument `size` specifies the size of the message to be sent (4096 bytes by default), `rx_depth` sets the number of receives to post at a time, `ib_port` is the port of the HCA and `use_event` specifies whether to sleep on CQ events or poll for them.

The function `pp_init_ctx` first allocates a buffer of memory, which will be used to send and receive data. Note that the buffer is `memalign`-ed to a page, since it is pinned (see section 8.3 for a definition of pinning).

Listing 3: Allocating a Buffer

```
    ctx->buf = memalign(page_size, size);
    if (!ctx->buf) {
        fprintf(stderr, "Couldn't allocate work buf.\n");
        return NULL;
    }

    memset(ctx->buf, 0x7b + is_server, size);
```

Next the `ibv_context` pointer is populated with a call to `ibv_open_device`. The `ibv_context` is a structure which encapsulates information about the device opened for the connection.

Listing 4: Opening a Context

```
    ctx->context = ibv_open_device(ib_dev);
    if (!ctx->context) {
        fprintf(stderr, "Couldn't get context for %s\n",
            ibv_get_device_name(ib_dev));
        return NULL;
    }
```

From the ``infiniband/verbs.h'' header, the `struct ibv_context` is as follows:

Listing 5: struct ibv_context

```
struct ibv_context {
    struct ibv_device      *device;
    struct ibv_context_ops  ops;
    int         cmd_fd;
    int         async_fd;
    int         num_comp_vectors;
    pthread_mutex_t     mutex;
    void            *abi_compat;
    struct ibv_more_ops     *more_ops;
};
```

The `struct ibv_device *` is a pointer to the device opened for this connection. The `struct ibv_context_ops ops` field contains function pointers to driver specific functions, which the user need not access directly.

8.2 Protection Domain

After the device is opened and the context is created, the program allocates a protection domain.

Listing 6: Opening a Protection Domain

```
    ctx->pd = ibv_alloc_pd(ctx->context);
    if (!ctx->pd) {
        fprintf(stderr, "Couldn't allocate PD\n");
        return NULL;
    }
```

A protection domain, according to the InfiniBand specification [2, page 107], allows the client to control which remote computers can access its memory regions during InfiniBand sends and receives.

The protection domain mostly exists on the hardware itself. Its user-space data structure is sparse:

Listing 7: struct ibv_pd from ``infiniband/verbs.h''

```
struct ibv_pd {
    struct ibv_context     *context;
    uint32_t        handle;
};
```

8.3 Memory Region

The `ibv_rc_pingpong` program next registers one memory region with the hardware.

When the memory region is registered, two things happen. The memory is pinned by the kernel, which prevents the physical address from being swapped to disk. On Linux operating systems, a call to `mlock` is used to perform this operation. In addition, a translation of the virtual address to the physical address is given to the HCA.

Listing 8: Registering a Memory Region

```
    ctx->mr = ibv_reg_mr(ctx->pd, ctx->buf, size,
                         IBV_ACCESS_LOCAL_WRITE);
    if (!ctx->mr) {
        fprintf(stderr, "Couldn't register MR\n");
        return NULL;
    }
```

The arguments are the protection domain with which to associate the memory region, the address of the region itself, the size, and the flags. The options for the flags are defined in ``infiniband/verbs.h''.

Listing 9: Access Flags

```
enum ibv_access_flags {
    IBV_ACCESS_LOCAL_WRITE      = 1,
    IBV_ACCESS_REMOTE_WRITE     = (1<<1),
    IBV_ACCESS_REMOTE_READ      = (1<<2),
    IBV_ACCESS_REMOTE_ATOMIC    = (1<<3),
    IBV_ACCESS_MW_BIND      = (1<<4)
};
```

When the memory registration is complete, an `lkey` field or Local Key is created. According to the InfiniBand Technical Specification [2, Page 76] the `lkey` is used to identify the appropriate memory addresses and provide authorization to access them.

8.4 Completion Queue

The next part of the connection is the completion queue (CQ), where work completion queue entries are posted. Please note that you must create the CQ before the QP. As stated previously, `ibv_ack_cq_events (3)` has helpful examples of how to manage completion events.

Listing 10: Creating a CQ

```
    ctx->cq = ibv_create_cq(ctx->context, rx_depth + 1, NULL,
                ctx->channel, 0);
    if (!ctx->cq) {
        fprintf(stderr, "Couldn't create CQ\n");
        return NULL;
    }
```

8.5 Queue Pairs

Communication in InfiniBand is based on the concept of queue pairs. Each queue pair contains a send queue and a receive queue, and must be associated with at least one completion queue. The queues themselves exist on the HCA. A data structure containing a reference to the hardware queue pair resources is returned to the user.

First, look at the code to create a QP.

Listing 11: Creating a QP

```
struct ibv_qp_init_attr attr = {
    .send_cq = ctx->cq,
    .recv_cq = ctx->cq,
    .cap     = {
        .max_send_wr  = 1,
        .max_recv_wr  = rx_depth,
        .max_send_sge = 1,
        .max_recv_sge = 1
    },
    .qp_type = IBV_QPT_RC
};

ctx->qp = ibv_create_qp(ctx->pd, &attr);
if (!ctx->qp) {
    fprintf(stderr, "Couldn't create QP\n");
    return NULL;
}
```

Notice that a data structure which defines the initial attributes of the QP must be given as an argument. There are a few other elements in the data structure, which are optional to define.

The first two elements, `send_cq` and `recv_cq`, associate the QP with a CQ as stated earlier. The send and receive queue may be associated with the same completion queue.

The cap field points to a `struct ibv_qp_cap` and specifies how many send and receive work requests the queues can hold. The `max_{send,recv}_sge` field specifies the maximum number of scatter/gather elements that each work request will be able to hold. A scatter gather element is used in a direct memory access (DMA) operation, and each SGE points to a buffer in memory to be used in the read or write. In this case, the attributes state that only one buffer may be pointed to at any given time.

The `qp_type` field specifies what type of connection is to be used, in this case a reliable connection

Now the queue pair has been created. It must be moved into the initialized state, which involves a library call. In the initialized state, the QP will silently drop [2, Page 460] any incoming packets and no work requests can be posted to the send queue.

Listing 12: Setting QP to INIT

```
        struct ibv_qp_attr attr = {
            .qp_state        = IBV_QPS_INIT,
            .pkey_index      = 0,
            .port_num        = port,
            .qp_access_flags = 0
        };

        if (ibv_modify_qp(ctx->qp, &attr,
                  IBV_QP_STATE              |
                  IBV_QP_PKEY_INDEX         |
                  IBV_QP_PORT               |
                  IBV_QP_ACCESS_FLAGS)) {
            fprintf(stderr, "Failed to modify QP to INIT\n");
            return NULL;
        }
```

The third argument to **`ibv_modify_qp`** is a bitmask stating which options should be configured. The flags are specified in `enum ibv_qp_attr_mask` in `infiniband/verbs.h`.

At this point the **`ibv_rc_pingpong`** program posts a receive work request to the QP.

```
    routs = pp_post_recv(ctx, ctx->rx_depth);
```

Look at the definition of **`pp_post_recv`**.

Listing 13: Posting Recv Requests

```
static int pp_post_recv(struct pingpong_context *ctx, int n)
{
    struct ibv_sge list = {
        .addr   = (uintptr_t) ctx->buf,
        .length = ctx->size,
        .lkey   = ctx->mr->lkey
    };
    struct ibv_recv_wr wr = {
        .wr_id      = PINGPONG_RECV_WRID,
        .sg_list    = &list,
        .num_sge    = 1,
    };
    struct ibv_recv_wr *bad_wr;
    int i;

    for (i = 0; i < n; ++i)
        if (ibv_post_recv(ctx->qp, &wr, &bad_wr))
            break;

    return i;
}
```

The ibv_sge list is the list pointing to the scatter/gather elements (in this case, a list of size 1). To review, the SGE is a pointer to a memory region which the HCA can read to or write from.

Next is the ibv_recv_wr structure. The first field, wr_id, is a field set by the program to identify the work request. This is needed when checking the completion queue elements; it specifies which work request completed.

The work request given to ibv_post_recv is actually a linked list, of length 1.

Listing 14: Linked List

```
    struct ibv_recv_wr wr = {
        .wr_id      = PINGPONG_RECV_WRID,
        .sg_list    = &list,
        .num_sge    = 1,
    };
```

If one of the work requests fails, the library will set the bad_wr pointer to the failed wr in the linked list.

Receive buffers must be posted before any sends. It is common practice to loop over the ibv_post_recv call to post numerous buffers at the beginning of execution. Eventually these buffers will be used up; internal flow control must be implemented by the applications to ensure that sends are not posted without corresponding receives.

8.6 Connecting

The next step occurs in pp_client_exch_dest and pp_server_exch_dest. The QPs need to be configured to point to a matching QP on a remote node. However, the QPs currently have no means of locating each other. The processes open an out-of-band TCP socket and transmit the needed information. That information, once manually communicated, is given to the driver and then each side's QP is configured to point at the other. (The OFED librdma_cm library is an alternative to explicit out-of-band TCP.)

So what information needs to be exchanged/configured? Mainly the LID, QPN, and PSN. The LID is the "Local Identifier" and it is a unique number given to each port when it becomes active. The QPN is the Queue Pair Number, and it is the identifier assigned to each queue on the HCA. This is used to specify to what queue messages should be sent. Finally, the destinations must share their PSNs.

The PSN stands for Packet Sequence Number. In a reliable connection it is used by the HCA to verify that packets are coming in order and that packets are not missing. The initial PSN, for the first packet, must be specified by the user code. If it is too similar to a recently used PSN, the hardware will assume that the incoming packets are stale packets from an old connection and reject them.

The GID, seen in the code sample below, is a 128-bit unicast or multicast identifier used to identify an endport [2, page 74]. The link layer specifies which interconnect the software is running on; there are other interconnects that OFED supports, though that is not within the scope of this paper.

Within `pp_connect_ctx` the information, once transmitted, is used to connect the QPs into an end-to-end context.

Listing 15: Setting Up Destination Information

```
    my_dest.lid = ctx->portinfo.lid;
    if (ctx->portinfo.link_layer == IBV_LINK_LAYER_INFINIBAND &&
                                                    !my_dest.lid) {
        fprintf(stderr, "Couldn't get local LID\n");
        return 1;
    }

    if (gidx >= 0) {
        if (ibv_query_gid(ctx->context, ib_port, gidx, &my_dest.gid)) {
            fprintf(stderr, "Could not get local gid for gid index "
                                                        "%d\n", gidx);
            return 1;
        }
    } else
        memset(&my_dest.gid, 0, sizeof my_dest.gid);

    my_dest.qpn = ctx->qp->qp_num;
    my_dest.psn = lrand48() & 0xffffff;
```

The `my_dest` data structure is filled and then transmitted via TCP. Figure 4 illustrates this data transfer.

8.6.1 Modifying QPs

Look at the attributes given to the `ibv_modify_qp` call.

Listing 16: Moving QP to Ready to Recv

```
    struct ibv_qp_attr attr = {
        .qp_state        = IBV_QPS_RTR,
        .path_mtu        = mtu,
        .dest_qp_num         = dest->qpn,
        .rq_psn          = dest->psn,
        .max_dest_rd_atomic = 1,
        .min_rnr_timer       = 12,
        .ah_attr         = {
```

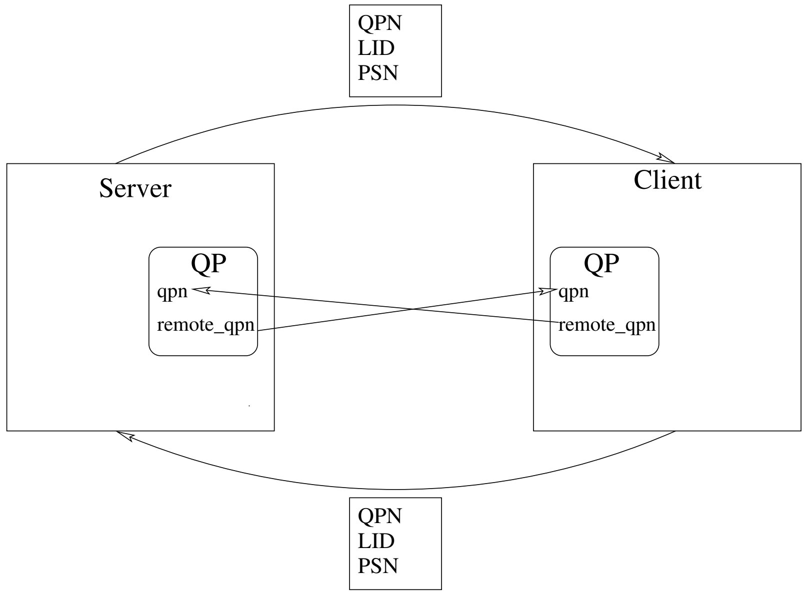


Figure 4: Exchanging QPNs via TCP

```
                .is_global    = 0,
                .dlid         = dest->lid,
                .sl           = sl,
                .src_path_bits = 0,
                .port_num     = port
            }
        };

    if (ibv_modify_qp(ctx->qp, &attr,
                IBV_QP_STATE              |
                IBV_QP_AV                 |
                IBV_QP_PATH_MTU           |
                IBV_QP_DEST_QPN           |
                IBV_QP_RQ_PSN             |
                IBV_QP_MAX_DEST_RD_ATOMIC |
                IBV_QP_MIN_RNR_TIMER)) {
        fprintf(stderr, "Failed to modify QP to RTR\n");
        return 1;
    }
```

As you can see `.qp_state` is set to `IBV_QPS_RTR`, or Ready-To-Receive. The three fields swapped over TCP, the PSN, QPN, and LID, are now given to the hardware. With this information, the QPs are registered with each other by the hardware, but are not ready to begin exchanging messages. The `min_rnr_timer` is the time, in seconds, between retries before a timeout occurs.

The QP must be moved into the Ready-To-Send state before the "connection" process is complete.

Listing 17: Moving QP to Ready to Send

```
    attr.qp_state        = IBV_QPS_RTS;
    attr.timeout         = 14;
    attr.retry_cnt       = 7;
    attr.rnr_retry       = 7;
    attr.sq_psn      = my_psn;
    attr.max_rd_atomic   = 1;
    if (ibv_modify_qp(ctx->qp, &attr,
                IBV_QP_STATE              |
                IBV_QP_TIMEOUT            |
                IBV_QP_RETRY_CNT          |
                IBV_QP_RNR_RETRY          |
                IBV_QP_SQ_PSN             |
                IBV_QP_MAX_QP_RD_ATOMIC)) {
        fprintf(stderr, "Failed to modify QP to RTS\n");
        return 1;
    }
```

The `attr` used to move the QP into `IBV_QPS_RTS` is the same `attr` used in the previous call. There is no need to zero out the structure because the bitmask, given as the third argument, specifies which fields should be set.

After the QP is moved into the Ready-To-Send state, the connection (end-to-end context) is ready.

8.7 Sending Data

Since the server already posted receive buffers, the client will now post a "send" work request.

Listing 18: Client Posting Send

```
    struct ibv_sge list = {
        .addr   = (uintptr_t) ctx->buf,
        .length = ctx->size,
        .lkey   = ctx->mr->lkey
    };
    struct ibv_send_wr wr = {
        .wr_id      = PINGPONG_SEND_WRID,
        .sg_list    = &list,
        .num_sge    = 1,
        .opcode     = IBV_WR_SEND,
        .send_flags = IBV_SEND_SIGNALED,
    };
    struct ibv_send_wr *bad_wr;

    int rslt = ibv_post_send(ctx->qp, &wr, &bad_wr);
```

The `wr_id` is an ID specified by the programmer to identify the completion notification corresponding with this work request. In addition, the flag `IBV_SEND_SIGNALED` sets the completion notification indicator. According to `ibv_post_send (3)`, it is only relevant if the QP is created with `sq_sig_all = 0`.

8.8 Flow Control

Programmers must implement their own flow control when working with the verbs API. Let us examine the flow control used in `ibv_rc_pingpong`. Remember from earlier that a client cannot post a send if its remote node does not have a buffer waiting to receive the data.

Flow control must be used to ensure that receivers do not exhaust their supply of posted receives. Furthermore, the CQ must not overflow. If the client does not pull CQEs off the queue fast enough, the CQ is thrown into an error state, and can no longer be used.

You can see at the top of the loop, which will send/recv the data, that `ibv_rc_pingpong` tracks the send and recv count.

Listing 19: Flow Control

```
	rcnt = scnt = 0;
	while (rcnt < iters || scnt < iters) {
```

Now the code will poll the CQ for two completions; a send completion and a receive completion.

Listing 20: Polling the CQ

```
			do {
				ne = ibv_poll_cq(ctx->cq, 2, wc);
				if (ne < 0) {
					fprintf(stderr, "poll CQ failed %d\n", ne);
					return 1;
				}

			} while (!use_event && ne < 1);
```

The `use_event` variable specifies whether or not the program should sleep on CQ events. By default, `ibv_rc_pingpong` will poll. Hence the while-loop. On success, `ibv_poll_cq` returns the number of completions found.

Next, the program must account for how many sends and receives have been posted.

Listing 21: Flow Control Accounting

```
			switch ((int) wc[i].wr_id) {
			case PINGPONG_SEND_WRID:
```

```
                ++scnt;
                break;

            case PINGPONG_RECV_WRID:
                if (--routs <= 1) {
                    routs += pp_post_recv(ctx, ctx->rx_depth - routs);
                    if (routs < ctx->rx_depth) {
                        fprintf(stderr,
                            "Couldn't post receive (%d)\n",
                            routs);
                        return 1;
                    }
                }

                ++rcnt;
                break;

            default:
                fprintf(stderr, "Completion for unknown wr_id %d\n",
                        (int) wc[i].wr_id);
                return 1;
            }
```

The ID given to the work request is also given to its associated work completion, so that the client knows what WQE the CQE is associated with. In this case, if it finds a completion for a send event, it increments the send counter and moves on.

The case for `PINGPONG_RECV_WRID` is more interesting, because it must make sure that receive buffers are always available. In this case the `routs` variable indicates how many recv buffers are available. So if only one buffer remains available, `ibv_rc_pingpong` will post more recv buffers. In this case, it calls `pp_post_recv` again, which will post another 500 (by default). After that it increments the recv counter.

Finally, if more sends need to be posted, the program will post another send before continuing the loop.

Listing 22: Posting Another Send

```
                    ctx->pending &= ~(int) wc[i].wr_id;
                    if (scnt < iters && !ctx->pending) {
                        if (pp_post_send(ctx)) {
                            fprintf(stderr, "Couldn't post send\n");
                            return 1;
                        }
                        ctx->pending = PINGPONG_RECV_WRID |
                                       PINGPONG_SEND_WRID;
                    }
```

8.9 ACK

The `ibv_rc_pingpong` program will now ack the completion events with a call to `ibv_ack_cq_events`. To avoid races, the CQ destroy operation will wait for all completion events returned by `ibv_get_cq_event` to be acknowledged. The call to `ibv_ack_cq_events` must take a mutex internally, so it is best to ack multiple events at once.

```
        ibv_ack_cq_events(ctx->cq, num_cq_events);
```

As a reminder, `ibv_ack_cq_events (3)` has helpful sample code.

9 Conclusion

InfiniBand is the growing standard for supercomputer interconnects, even appearing in departmental clusters. The API is complicated and sparsely documented, and the sample program provided by OFED, `ibv_rc_pingpong`, does not fully explain the functionality of the verbs. This paper will hopefully enable the reader to better understand the verbs interface.

10 Acknowledgements

Gene Cooperman (Northeastern University) and Jeff Squyres (Cisco) contributed substantially to the organization, structure, and content of this document. Jeff also took the time to discuss the details of InfiniBand with me. Josh Hursey (Oak Ridge National Laboratory) shared his knowledge of InfiniBand with me along the way. Roland Dreier (PureStorage) pointed out, and corrected, a mistake in my explanation of acks.

References

[1] Jason Ansel, Gene Cooperman, and Kapil Arya. DMTCP: Scalable user-level transparent checkpointing for cluster computations and the desktop. In Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS-09, systems track). IEEE Press, 2009. published on CD; software available at `http://dmtcp.sourceforge.net`.

[2] InfiniBand Trade Assocation. InfiniBand Architecture Specification Volume 1, Release 1.2.1, November 2007. `http://www.infinibandta.org/content/pages.php?pg=technology_download`.