Lecture 3

Gidon Rosalki
2026-05-05

Notice: If you find any mistakes, please open an issue in the github repository

1. Reminder

So far, we have mostly spoken about how programs communicate with network cards. We began with ideas of them wanting to communicate, and they call specific functions, which cause context switching, and the OS then copies the memory, and sends the data, which is overall quite slow. From here we reached a different model, in which the user level processes may communicate directly with the hardware / NIC, and discussed the idea of the message queue model. We have the work queue for sending messages, so the process puts data into the work queue, and the NIC takes data / jobs out of the work queue. When it has done with this, the NIC puts messages into the completion queue, and the process removes messages from the completion queue.

2. Verbs essentials

A Berkeley socket is an API for sending messages across the network. Originally created in Berkeley (shocking, I know), and became a standard. When creating user level networking, there were a few different methods, that were all vying for being the standard. In the end, it was established that having a standard was less important than standard concepts. What we did instead was define something very general, and everyone may extend it as they desire. After this was done, a number of different people made their own APIs, and one of these did in fact manage to become the "standard" of user level networking.

2.1. The "Verbs" library

Verbs is an abstract description of the functionality that is provided for applications using RDMA, and *not* an API, for there are several different implementations for it. Verbs may be divided into two major groups:

- Control path - manage the resources, and usually requires context switch
  - Create
  - Destroy
  - Modify
  - Query
  - Work with events
- Data path - Use the resources to send / receive data, does not require context switch
  - Post Send
  - Post Receive
  - Poll CQ
  - Request for completion event

Data path is the "important" part here, where it must be incredibly efficient, since it is primarily what we want in RDMA. When we want to receive data, we post Receive, and then poll CQ until there is a corresponding completion for the receive we wanted. Remember that there is a great risk here for data races, when given a buffer to the NIC, you must *not* change it until you have received word that the NIC is done with said buffer, otherwise **undefined behaviour** will occur.

2.2. Why use verbs

Verbs is a low level description for RDMA programming. It is very close to the "bare metal", and provides the best performance. It may be used as building blocks for many applications, such as sockets, storage, and parallel computing. Any other level of abstraction over verbs may harm the performance. Unless you wind up in this type of world for programming super high performance networking, this course will probably be the last time you work with verbs.

2.3. libibverbs

This is an Infiniband verbs library, developed and maintained by Roland Dreier since 2006, and are the de-factor verbs API standard in *nix. It is FOSS, and the kernel part of the verbs is integrated in the Linux kernel since 2005, version 2.6.11.

- Source code that uses libibverbs should include the header `#include <infiniband/verbs.h>`
- Executables / libraries that work with libibverbs should be linked with `-libverbs`
- All input structures should be zeroes, using `memset()` or structure initialisation
- Most resource handles are pointers, so using bad handles may cause segfault
- Verbs that return a pointer will return a valid value in case of success, and NULL for failure
- Verbs that return an integer return 0 for success, and -1 or errno for failure

To save us time, when doing exercises with verbs and the like, we will be given nonempty files to help us get started.

3. Memory Region (MR)

Memory Region is a virtual contiguous memory block, that was registered. I.e. prepared for work with RDMA. The OS is responsible for ensuring that the memory pages are swapped in and out, and does this without the user process knowing. This can be problematic with RDMA, since I can plan to use a particular buffer, on a certain page, and then have that page swapped out without me knowing, causing slowdowns. We may prevent the OS from swapping out pages, by *pinning* them. This is dangerous, since you could pin too much (memory leak), and freeze the entire system by stealing all the RAM. As a result, the amount of memory you can pin is limited. One way to pin pages is by registering them. Any memory buffer in the process' virtual space may be registered. It has a few available permissions:

- Local operations (local read is always supported)
  - IBV_ACCESS_LOCAL_WRITE
  - IBV_ACCESS_MV_BIND
- Remote operations
  - IBV_ACCESS_REMOTE_WRITE
  - IBV_ACCESS_REMOTE_READ
  - IBV_ACCESS_REMOTE_ATOMIC

If remote write, or remote atomic is enabled, then local write should be enabled too. The same memory buffer can be registered multiple times, even with different permissions. After a successful memory registration, 2 keys are generated:

- Local key (lkey)
- Remote key (rkey)

These keys are used when referring to this MR in a work request.

4. Queue Pair (QP)

For our purposes, we will not be creating a separate Send Queue, and Receive Queue, but create them together as a Queue Pair. The queues within the pair are still independent within, but we store them together as a single object.

When creating a socket, we may describe it with a few different features. The first is *reliability*, which is either true or false. Reliable transport ensure that

1. The data will arrive
2. In the order in which it was sent
3. Once

TCP is a reliable protocol, UDP, is unreliable.
The second is *connectivity*. TCP is connected, where we directly connect the sockets together, when I send data from a TCP socket, I know to which socket it will arrive, and this will not change. So, TCP is connected transport. Put these together, and we have a matrix:

con\rel	R(eliable)	U(nreliable)
C(onnected)	RC (Reliable Connected - TCP)	UC (Unreliable Connected)
D(isconneccted)	RD (Reliable Disconnected)	UD (UDP)

Table 1: Connected / Reliability table

UC, and RD are interesting concepts. In UC, we are connected, and maintain our knowledge of each connection, but do not send reliable data. RD is a fascinating concept, where we reliably send data, to disconnected recipients. This is exceedingly difficult to create. The RDMA standard did create a standard for Reliable Datagram, but it is incredibly difficult to implement, and so nobody uses it in actuality. Essentially all NICs only support RC, UC, and UD, but do not bother with RD, and instead use alternatives.

Metric	UD	UC	RC
Reliability			✓
Send (with immediate)	✓	✓	✓
RDMA Write (with immediage)		✓	✓
RDMA Read			✓
Atomic operations			✓
Multicast	✓		
Max message size	MTU	2 GB	2 GB
CRC	✓	✓	✓

Let us take a moment to consider an HPC system from around 10 years ago, where we buy 10 servers, with say 100 cores each, total of 1000 cores across 10 computers. Each has a network card, and is connected, and so on. We may run a separate process on each core. At the beginning, we have 10×100 processes, and each may speak with another 1000 processes. Now, 1000 sockets does not like many, but when you consider that you have 100 processes on a given computer, each with 1000 sockets, we now have too many sockets being handled by a single NIC. If we did not have this issue of sockets, and the cost of RC, we would always use RC, since it is truly wonderful. There's no need to worry about order, or whether or not it will arrive, but we cannot, since there are just too many connections. It simply becomes a massive waste of memory, and it reaches a point where for a computer, we are using almost all the resources *purely* to communicate with the neighbours.

Wasting memory on communication becomes a very large problem in connected systems. People dream of reaching 90% efficiency, lots of servers in the wild run at 70% efficiency, meaning that 30% is wasted purely on the fact that you are working over a distributed system.

5. APIs

5.1. Memory Region (MR)

```
struct ibv_mr *ibv_reg_mr (
  struct ibv_pd *pd,
  void *addr,
  size_t length,
  enum ibv_access_flags access
)
```

This registers a memory buffer with specific permissions. Note the following fields in the `ibv_mr` struct:

- `lkey` - The local key of this MR
- `rkey` - The remote key of this MR
- `addr` - The start address of the memory buffer that this MR registered
- `length` - The size of the memory buffer that was registered

`int ibv_dereg_mr (struct ibv_mr *mr)`: This de-registers an MR. This verb should be called if there is no outstanding Send Request, or Receive Request that points to it.

We will note that in contrast to above, we can in fact pin all the memory, since we know what is in fact important here, and the OS only sort of does. This works because in HPC, when using a server, only *one* person is using the server, and all others leave it alone, since a single core experiencing a 5% slowdown causes the entire distributed computation to experience said slowdown, as they all wait for said core. Gil said that this reaches the level of seriousness that with him at Nvidia, if he's using a server, and someone else logs in, they are fired.

5.2. Security (tangent)

We love security, it is very important to us. However, it is hard in HPC, since it comes at a severe performance cost. As a result, we make it as secure as we can, as long as it does not affect the results. Ignoring this though, let us consider when we have two competing companies using the same hardware, they will naturally encrypt all the disk information. This can then be extended into not trusting the data centre itself. The people that built it may be trying to steal your data when it is decrypted into the memory / CPU. This brings us to the idea of *confidential computing*. Here, we want the memory, the information inside the CPU, everything, to be encrypted at all times. This is a hot topic in cryptography, and discussed more extensively in my notes there. There are CPUs that one can buy today which are guaranteed to have confidential computing, and this is the "best" that we have today.

5.3. Connecting QPs

Communication should be established between the connected QPs, where each side needs:

- To know who is the other side
- To have information about the other side, and the path there
- To configure attributes that describe the send attributes

Thing is, we need to now how to connect QP X to QP Y, since we cannot transfer the needed information to open the connection, before the connection is open. There are 2 solutions to this problem:

1. Exchange information Out of Band, such as over sockets
2. Use Communication Manager (CM) **WHICH IS THE CORRECT, AND USED WAY TO CONNECT QPs**