Lecture 4

Gidon Rosalki
2026-05-26

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

We discussed verbs last week, and things like queue pairs, scatter gather, and so on. Today we will discuss memory, and once we have these building blocks, we are going to try and make RDMA practical, such as when, and how we use this, when can one actually access memory. There will be two real world examples for this. Further on in the semester, we will move on to things like AI, training and inference, and how RDMA has an impact on this. For example, we will discuss DeepSeek, which is remarkable open about their training processes, where whenever they do something, they publish a paper on it. Naturally, they also screw up, but the publicity of their work is nice.

1. Reminder

1.1. OS Memory Access

The CPU uses the Main Memory Unit in order to access the main memory (RAM). Other devices use the IOMMU. Memory is kept in *pages*, usually of size 4KB. One cannot access less memory than that at once. Memory is swapped in and out of RAM according to how much space there is in the RAM, and then information from the RAM is swapped in and out of the L caches of the CPU for the CPU to access quickly. Direct Memory Access is a device to access the memory directly, rather than requesting from the CPU. This access is over a "bus", such as PCI(e). The OS manages a virtual memory space for the process, where the process thinks that it is the only process on the machine, such that it cannot try and infer information about other processes, or change their memory. This makes the mapping to the swap happen transparently to the program, since it just requests a page, and the OS handles returning it to the program.

1.2. Memory Registration

This is a mechanism that allows an application to describe a set of virtual or physical contiguous memory location to the NIC as as virtually contiguous buffer using a Virtual Address. This registration process pins the memory pages, to prevent the pages from being swapped out, and keep the virtual to physical mapping.
During the registration, the OS checks the permissions, and the registration writes to the NIC translation table from virtual addresses to physical addresses.

This results in a Memory Region (MR), where every MR has a remote, and a local key (r_key, l_key), which are used in the WR. The same memory buffer can be registered several times, with different access permissions. Every registration will result in different keys.

This is an *expensive* operation, as we discussed previously. As a result, one should minimise how frequently one calls it.

1.2.1. Region Registration

There is a flow to carry out this memory access:

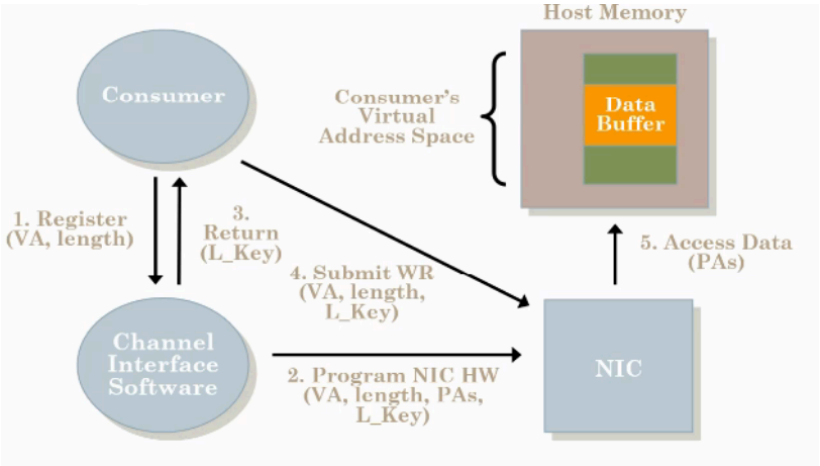

Figure 2: Region Registration and Local Access

When the consumer wants to send data, it registers the memory. The channel interface software then pins the memory with the NIC, and returns the l_key, and r_key to the consumer. Once this is done, the consumer submits a WR to the NIC, which then sends the access data to the remote.

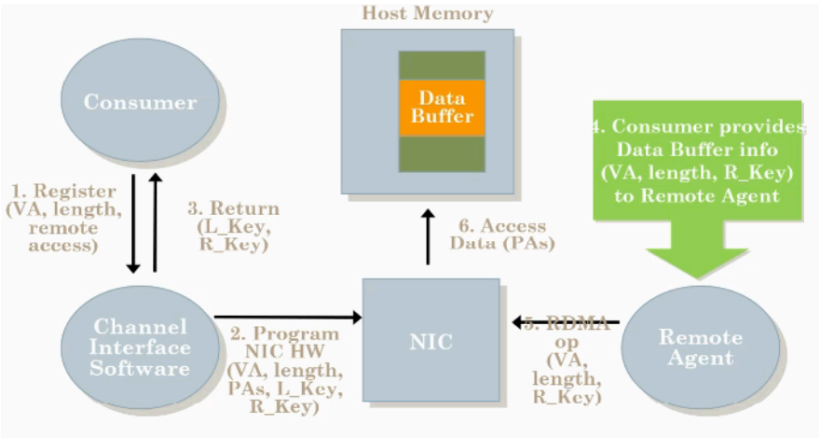

Figure 3: Region Registration and Remote Access

On the remote, it is a bit more complicated, so we will continue to pretend bits of it do not exist for a bit. The remote agent has received the data buffer info (VA, length, r_key) to the remote agent, which then carries out an RDMA operation, where it sends a WR to the NIC, which checks if it is legal, carries out the translation, and accesses the memory.

```c
int foo (struct ibv_ *pd, struct ibv_qp *qp)
{
  struct ibv_mr *mr;
  struct ibv_sge *sge;
  struct ibv_recv_wr wr, *bar_wr;
  int length = 16384;
  char *buf;

  buf = malloc(length);

  mr = ibv_reg_mr(pd, buf, length, IBV_ACCESS_LOCAL_WRITE);

  sge.addr = (uintptr_t)buf;
  sge.length = length;
  sge.lkey = mr->lkey;

  wr->wr_id = (uintptr_t)buf;
  wr->sg_list = &sge;
  wr->num_sge = 1;
  wr->opcode = IBV_WR_SEND;
  wr->send_flags = IBV_SEND_SIGNALED;

  return ibv_post_send(qp, &wr, &bad_wr);
}
```

The above code is the standard, and now given that, we need to somewhat discuss the protection domain. Let us assume that we have some sort of server, say a university server, where not all users are allowed to access the entire machine at once. Let us assume that Gil wants to give us all access to *read* our grades, but naturally does not want to give us write access. Naturally, Gil needs both read, *and* write access. However, we now have a problem with this model of returning a pointer, since once we have done memory registration, we only have a single access key, with a set of permissions. We are doing something that is a security nightmare in memory registration, where we can directly access memory. What we will do is something called a **protection domain**. So now, we have a buffer that is part of a protection domain, and instead of people accessing the memory directly, they access the protection domain. Blocks of memory are assigned to protection domains, and trying to access memory in a different protection domain is an error.

On the one hand, this is relatively expensive, but it is excellent in terms of security to give everything its own protection domain. So what we can now do is register the memory twice, once to domain A, once to domain B, where A has read and write access, but B only has read access. So, as part of the API, whenever we register memory, we now also need to register it to a protection domain.

1.2.2. On Demand Paging

As we discussed, memory registration is expensive, and simply registering at the beginning is not necessarily a solution, because we may dynamically need more memory. A solution is that the OS already has a disk paging solution, where we bug the OS to return our pages to us from the disk. A similar solution may be provided to the NIC, which requests pages that were swapped out by the OS. This operates almost identically to the CPU, but also requires that the OS informs the NIC when pages are swapped out, so that they may be deleted from its translation table.

This concept is called On Demand Paging, which does in fact work, but interestingly, is not used in the real world. This is because we want predictable performance, we do not want to have to wait for the OS to swap pages back in, since this can cause significant delays to the entire compute cluster.
We instead assume that the users know what they are doing, and write software that handles the memory themselves.

2. IB Ops and Protocols

Recall the RDMA opcodes, such as send, Write, Read, Atomic, and so on.

2.1. The "Eager" Protocol

Let us assume that we want to send a message. We begin with asking how much data we want to send. Beginning with the responder, we open a few 4K buffers, and post them as receive buffers. The requester sends to a remote QP using the send opcodce, which are typically signalled, so we receive a work completion when done.

This protocol has minimal startup overheads, and is used to implement low latency message passing for smaller message. The down side is that is needs `memcpy`, which means that this is not zero copy. If the Responder is not ready, then this will complete with error, i.e., the status is not `WC_SUCCESS`.

2.2. The "Rendezvous" Protocol

Since eager only works for small messages, we needed something new. For those who were lucky enough to not learn French, rendezvous simply means meeting, and is not pronounced how you think.

So, let us suppose that we want to send a large message, say a gigabyte in size. To do this, the requestor will use eager send to send the responder the sixe, and location info. The responder may then wait for the user, or allocate space, and then either

- Call on the remote location on the requester
- Send back local information (address + rkey), and the requester then calls

2.2.1. RDMA_READ vs RDMA_WRITE

This is a common, and standard protocol for RDMA communication. We even build a high abstraction layer over it, called MPI, which makes life easier for physicists, that should not be learning how to use RDMA verbs directly, but rather should focus on their difficult physics questions.

Which is better between read and write? Depends on the situation. Consider for read, we can send the data and have it cost nanoseconds to the sender CPU, since it just informs the receiver, and then can wait for the receiver to read it, while doing other tasks.
For the sender, this is also very cheap, since the NIC is doing the reading, and the CPU continues to compute other tasks while the NIC reads the data across the network. When receiving data, we need to use "progress" every so often, such as `MPI_progress`, to check on the progress of the transfer, but we can choose when to do this, such as when we are not doing expensive operations in memory, since this will cause us to lose the data in the cache. We can instead do it when we are not in the middle of a matrix multiplication (for example), and not lose vital time in our operations.

3. Advanced Offloading

We are moving on to consider everything in the world of HPC (High Performance Computing). We have created our communication methods, with eager which is very good for small messages, and rendezvous for large messages.

3.1. CORE Direct Technology

Let us discuss an idea that Gil had about 20 years ago. We know that when we have two operations in a work queue, then the second will not start before the first, but that there is no ordering across work queues. Perhaps we want to create some sort of dependence between work queues, such as we want to forward information from one host, to another. As we have discussed, this requires progress, where the CPU checks the state, to carry out operations accordingly. However, we do not want to bother the CPU, but would rather have the NIC handle this all.

To do this, we need a language, which we may use to define dependencies. We can use this to create a tree of dependencies between operations, but does require something new that we have not yet had. Let us suppose that we have a WQ, and within a couple of WQEs. Recall the fence concept, which is similar to barrier, where we tell the NIC not to move past this point, before everything else reaches this point. Fences are similar to what we want, but does not quite do everything.

To create these dependencies, we will add a new WQE, with the opcode . This will say to wait until the work request in another WQ is completed.

When we consider this in something like gradient descent, which is not particularly expensive, but requires large amounts of communication, and still takes months, it turns out that this tree of dependencies does not in fact change throughout all this time. So, instead of creating this tree, what we can do is create something called **persistent communication**, where we create a template for communication. Despite us starting from these basic verbs, of read, write, and so on, this is not in fact important or useful. We are not going to be creating more efficient implementations that Intel, Nvidia, or anyone else. When we work with chips, we send information over parallel buses. Whenever we want to send a packet, it is not sent over a wide bus, but rather serially, to another machine. Since physics is moving slower than compsci, we are now able to create information faster than physics allows us to transport it. To demonstrate this, the most powerful computers we can build today are about 40 million times more powerful than they were a decade ago. However, communication speed over a fibreoptic has increased by 4 times. Since our ability to communicate is improving much more slowly, we need to be clever with how we communicate between our processors.

Information passing doubles every 2 or three years (or so), but Nvidia creates a new GPU every year. This means in 3 years, our latest generation will be severely limited by communication speed. To resolve this, we can do something similar to CPUs handling Moore's law limitations, where they split processing into multiple CPUs (or cores). Similarly, in this networking, instead of making a faster cable, we put multiple cables. For example, on a GPU, instead of connecting it to the CPU over a single cable, we use multiple connections, to allow us to communicate faster. This is very standard at this point, and we may even use 4 or 8 connections at once. Nvidia created nvlink for communication between GPUs, which is around 10 times faster, and achieves this essentially by having more cables connection between the GPUs.

Rings are work processes, where we arrange our processors in a circle, and each passes the information to the next process in the circle (say, to the right). This will be studied extensively, since this will be our final project.