

## Section: Sockets, TCP, and Unreliable Networks

### 1 Announcements

- midterms coming up....
- watch for extra office hours and such
- project questions?

Disclaimer: These notes aren't substitutes for attending lecture or reading the book. Those have been created with careful revision and review. These notes are mainly intended to be a more thorough index of the lectures and reading so that you can go off and read the full treatment on the topic and be able to ask the right questions while reading those two sources. If you have comments or suggestions please send me email and I'll fix them up.

### 2 Recap and Intro

Last we talked a great deal about networks and the advantages that networks can offer such as building large distributed systems. However fundamentally those systems relied on the idea that the system and the network are all reliable. However the IP network that provides the framework for the internet is an unreliable network. It can garble packets, reorder packets, and drop packets at random. The problem is further aggravated by the fact that there are malicious hosts and the hosts themselves can go down. In this section we'll try to see how the OS can manage this unreliable but powerful resource into a useable form.

### 3 Understanding Network Terminology

There are two terms that are fundamental when talking about networking and i'm sure you have heard them in many contexts. However i want to make sure to nail them down since they are important to the understanding of networks and the analysis of networks. They are *bandwidth* and *latency*.

- **latency:** this is a measure of how long it take from the time the first byte enters the network to when the first byte leaves the network. The units are time
- **bandwidth:** this is a measure of how many bits per second the network can pump through the network at any given point in time. The units are bytes/time

Lets motivate this with an extreme example before diving into much more realistic situations. Lets say for argument sake that i wanted to loan you my entire collection of James Bond movies so that you could share in the glory that is James Bond. What I could do is take my big stack of 10 DVDs (about 100 Gigabytes of storage) and hop in my car and drive it over. Lets also say that it takes me an hour to drive the movies over. However it takes merely a second to hand the box of 10 DVDs over to you once I get there. In this situation the latency is 1 hour since the time it takes the first byte of data to leave my house and get to your house. However it only takes 1 second for the rest of the 100 Gigabytes of data to flow right behind it. So thus the bandwidth is 100 Gigabytes per second even though the latency is one hour. So in this case it is a very high latency high bandwidth "network." The obvious goal of most network designers is low latency and high bandwidth. The total transfer time is often given by the  $\alpha$  (aka latency),  $\beta$  (aka bandwidth) model.

$$\text{total transfer time} = \alpha + \frac{\text{Message Size}}{\beta}$$

**Question:** Ethernet is known to have a very high latency compared to some of the other networks out there, but has a good bandwidth. Aside from cost considerations, why is this network model ok for most of our applications? When is this not the greatest network to use?

**Question:** Assuming i had a network that had a 1ms latency between here and SF, what bandwidth would i need to achieve to be able to transfer 100 Gigabytes in the same amount of time?

**Question:** Lets say I had two networks. Network A has a latency of 100 ms but a bandwidth of 10 MBps and network B has a latency of 10 ms but a bandwidth of 1MBps. If i wanted to send 10 bytes over the network, which is faster? 100 bytes? 1kB? 1MB? 1GB?

**Question:** Given the networks above, is there a message size such that they would both exhibit the same performance? If so what is it?

## 4 Flow Control and Ordered Delivery

In order to guarantee ordered delivery on an unordered and unreliable network we need to make sure to build a system that allows for self correction and failure detection. This failure detection and self correction is the service that Transmission Control Protocol (TCP) and any flow control protocol will provide. There are a lot of interesting variations on the idea that Kubi went over in lecture. I'll just touch upon the two most important parts of TCP as related to this class and the project.

### 4.1 Ordered Delivery and Sequence Numbers

In order to place a strict ordering on packets the simple solution is to number every packet with an identified that tells you what position the packet is in the stream. That way with the number it is easy to tell what position the data belongs in. In order to guarantee delivery

we take a simple approach, for every packet that comes in we send an acknowledgment to the host that sent us the packet saying that we got the packet. Thus the host waits on a timer and checks if the packets have been acked with in a certain time out period. If they haven't been acked it assumes that it needs to resend the packets since they might have been dropped. Notice here that it is impossible to tell the difference between a dropped data packet or a dropped acknowledgment. If we happen to get duplicate packets we simply throw them away.

## 4.2 Windows

In the simple case we send a packet, wait for the acknowledgment and then send the next packet. If you take a look at the example above, notice that because of the high latency terms the time needed to send larger messages is not directly proportional to the message size. So thus it makes sense for us to send a bunch of packets at a time (in essence flooding the network) and then waiting for those packets to get acknowledged. This "bunch" of packets is called a window in TCP. In real networks the window size is dynamic and adapts to network conditions, however in nachos it is fixed at 16.

Now lets return to the dropped packet cases above. What happens if one of the middle packets in the window gets dropped? How do we signal the retransmission of that packet. The approach that is normally taken is that the acknowledgment contains the sequence number of the next packet that it needs. For example lets say that we send packets with numbers 0 through 7 and packet 4 gets dropped. The remote end then sends acks with sequence numbers 1 (saying packet 0 has arrived) 2 (implying packet 1 has arrived), 3, 4, 4, 4, 4. (no that is not a typo) This says that the receiver has received 7 packets however packet 4 is missing out of the set. So then the sender sees this and retransmits packet 4 and then then an ack for packet 8 comes in saying that 0-7 have been received. Thus the sequence number in the acknowledgment signals which packets were successfully received. In this example we used a window size of 8. Your flow control implementations will need to manage 16 packets in flight at a time.

**Question: Another valid acking technique is to ack the packet we receive. That is if we receive packet 0 send an ack for 0 (instead of 1 as described above). What are the advantages and disadvantages of such a scheme?**

**Question: What is the impact of timeout on retransmission? What are the advantages of a large timeout? What are the advantages of a small timeout?**

In real TCP the window size exponentially grows at the start of a connection until the first dropped packet. From then on it cuts the window size in half and then grows the window size linearly until a packet is dropped.

**Question: Why is the TCP functionality hidden inside the OS? Are there more than performance and coding ease considerations?**

**Question: What are the advantages of large window sizes?**

**Question: What are the advantages of small window sizes?**

## 5 The Socket Abstraction

Now imagine that everytime you wanted to write a network based program you had to manage the window sizes like that and manage different state machines like that for every applications. UGH what a pain. Thus enter sockets. Sockets are a clean wrapper and an abstraction around all that TCP / IP goup to make it cleanly work. If you are doing file I/O you don't know how that file is getting to the right place on the disk, you just know that it works. Sockets provide the same functionality. You put bytes on the network and are sure that they will safely get there on the remote end (if you are using TCP and not UDP). Thus the interface is very similar to read and write I/O calls. The only difference is how connections are established. Every machine has a unique address associated with it (the IP address). Along with the IP address an OS usually provides 65536 ports. These ports are software objects that provide a set of mailboxes within that machine. For example, Soda hall has one street address but the mailroom itself has one slot for every person within soda. Similarly the computer has 65536 slots available for use. A socket is can be uniquely defined by the following tuple (remote IP, remote port, local IP, local port). Thus two processes on different machines can construct a socket between each other and send messages between each other and communicate. The underlying OS and TCP system will take care of all the fragmentation, acknowledgments, window sizes, etc etc and you don't need to worry about it. A socket is thus the wrapper around the state machine. Remember that sockets are purely software objects and an abstraction to multiplex one physical link across multiple processes.

**When you work with the Network layer, REMEMBER TO SYNCHRONIZE the connections since multiple threads could be trying to run the protocols simultaneously.**

**Question: Why are sockets implemented inside the OS? What is dangerous about implementing them at user level?**

**Question: If the the OS provides the allocation of ports and such why do we need encryption?**

### 5.1 connect()

The `connect()` tells the operating system to open a connection to the remote end. As arguments it takes in the remote host and the remote port to connect to. This call will establish the connection in nachos and is a blocking call. That is it doesn't return until the remote end is ready to accept messages.

### 5.2 accept()

The `accept()` call checks if there are any remote machines that want to connect to the local host. When it does it picks up the local port (which is one of the arguments) and establishes a socket on that line. It then tells the remote machine that the socket is ready to go. And then the local machine can do read or writes on that socket.

### 5.3 read() and write()

Once the connection has been established it gets a file descriptor like any other file and we can simply do read and write syscalls to that file descriptor. The way to use read and write is very similar to the console and the socket objects should be extensions to OpenFile.

## 6 Node Failures and Two Phase Commit

So far we have only focused on network failures, however what happens if the hosts go down? In many applications like the web and such its not that big of a deal since all it means is that a website won't be available for a while. It maybe annoying but its not the end of the world. However what about banking? Lets say the banks servers go down while I'm trying to deposit my stipend check and the ATM thinks the transaction has been accepted and sucks up the check while the bank never actually records the transaction. I'd be pissed! So we introduce the idea of two-phase-commit to protect ourselves against node failures of critical servers at critical times. The two phase commit algorithm might seem very inefficient, but in most cases where you need to use this algorithm you are willing to sacrifice performance for correctness. I could care less if the bank takes 10 seconds instead of 1 second to process my transaction as long as it can guarantee that the money has been recorded properly or it gives me back an error message (and the check) saying that the money wasn't deposited and tells me to try again later. Notice that the key for two-phase commit is the idea that the transaction goes all the way to completion (the bank records my deposit) or the transaction never happens at all (the bank returns my check).

Two phase commit is so named because there are two phases in the commit process (the prepare phase and the commit phase) Lets go over the ATM/Bank example Kubi went through in class. Most all two-phase commits schemes work like this:

There is always one party that takes the lead in the communication. Lets say that the ATM is the leader since it is the one initiating the transaction.

1. ATM first writes "BEGIN"
2. ATM sends a message to telling it about the new transaction
3. Bank then either sends an "ABORT" because of lack of enough funds
4. If there are enough funds it firsts notates the possible change and then sends a "OK TO COMMIT" (notice at this time the bank hasn't actually comitted the new amount to the stable storage yet. Just a message saying here is a possible pending transaction).
5. both parties now enter the commit phase and A then commits the transaction
6. A then sends a message to B saying "COMMIT" and waits for an ack
7. B records "GOT COMMIT" to the log, commits the transaction, and then sends the ack

One of the keys to the algorithm is that you write to the stable storage *before* you actually send the messages. This way you always know what messages to resend.

**Question:** What is the point of no return in this algorithm? That is at what point does the transaction have to be recorded?

**Exercise:** Between each step, imagine that *A* goes down and comes back up an hour later. How would *B* respond? At what stages is the system blocked and at what stages can the system abort safely?

**Exercise:** Between each step, imagine that *B* goes down and comes back up an hour later. How would *A* respond?

## 7 Malicious Nodes and the Byzantine Generals

Until now we have only talked about nodes or network links that go down or up. However this is only one style of attack and is not nearly as malicious as other styles of attacks. How do we protect ourselves against malicious nodes in a distributed system?

The problem statement is cast into the byzantine generals problem as follows. Lets say we have  $n$  nodes that all need to agree or disagree to do a certain task (like commit or abort a transaction). One of these  $n$  nodes is decided to be the general and the others are the lieutenants. The byzantine generals problem works under the following conditions:

- All loyal lieutenants obey the same order
- If the general is loyal then all lieutenants must obey that order

Now those might seem like simple statements but there is a bit of subtlety to it. If the general is loyal then all the loyal lieutenants must obey the order. However, if the general is not loyal we still require that all lieutenants obey the same order. Notice that when the general is a traitor, the loyal lieutenants must still arrive at the same decision (and not undefined behavior as we'd normally expect).

Through some fancy math and proofs that the original papers do a wonderful job of proving, we can show that if we have  $n$  nodes then we can only tolerate at most  $\frac{n}{3} - 1$  failures before the conditions specified above can't be implemented. Thus with  $f$  faulty nodes you need at least  $3f + 1$  total nodes or  $2f + 1$  loyal nodes.

If you are at all interested in doing systems or security research you'll have to read the Lamport et. al paper on the fault tolerance. It was actually one of my favorite CS papers of all times and is a very interesting and applicable read.

## 8 Conclusion

We talked a lot about unreliable networks this week. Starting with what happens when network links go faulty to nodes going down to nodes becoming arbitrarily malicious. Dealing with failures at different levels is and the assumptions we can make is a very important thing to take away.

Remember that there is no way we can design our large distributed systems to be perfect. Thus, rather than assuming they are perfect, we can assume that they are imperfect and build in error detection and error recovery schemes that make systems much more robust than trying to design them for perfection.