CS162 – Fall 2011 TA: Angela Juang
Discussion Notes Week #07

# CS162 Discussion #7: Reliability and Naming

**Announcements**

- Project 2 initial design doc due Tuesday, 10/18 (`submit proj2-initial-design`)
    - Give Angela a hardcopy in section or drop off at 346 Soda
    - Sign up for design reviews at http://goo.gl/KmRYu
- Midterm this Thursday, 10/13 @5-6:30pm in 155 Dwinelle
- Office Hours Thursday, 10/13 @10am-12pm in 651 Soda (no office hours on Friday)

BUT WHEN SHE TRACED THE KILLER'S IP ADDRESS... IT WAS IN THE 192.168/16 BLOCK!
GASP!

**Protocols Upon Protocols**

Up until now, we've talked about hosts connecting to each other, but haven't really gone into detail about how they do it. To set up a connection, we need to establish a protocol. A *protocol* is an agreement between the sender and receiver on how the data is going to be transmitted. Here are a few:

- *Internet Protocol (IP).* Defines a method for addressing between source and destination machines, using IP addresses. This protocol delivers messages unreliably.
- *User Datagram Protocol (UDP).* Built on top of IP, unreliably transfers unordered datagrams from a source port to a destination port.
- *Transmission Control Protocol (TCP).* Built on top of IP, reliably transfers an ordered stream of bytes from a source port to a destination port.

IP allows packets to travel between machines with distinct IP addresses; UDP and TCP allows us to send messages to particular processes by using ports. Since each address can have a number of ports, each with its own connection, we define each unique connection by a 5-tuple of source address, source port, destination address, destination port, and protocol.

Note: Subnets are specified by IP address, in a form such as 128.32.131.0/24. What this "/24" means is that the top 24 bits of the address must be fixed, and the rest are variable. (This is called a mask.) Therefore, since the address has 32 bits, there are 8 bits worth of possible addresses in this subnet, and they all have to start with 124.32.131.

**ACK! How Do We Get Reliability?**

Packets being sent across a network can always be garbled or dropped before reaching their destination. How do we make sure that they all get to the receiver in the right order? The first step to achieving this behavior is by assigning each packet a sequence number, in increasing order. This way, we know what order the packets were meant to be in and can reorder them if necessary when we receive them. The next thing we have to do is make sure that if any packets were dropped, they get resent. The only way for the sender to know whether or not it should resend a packet is for the receiver to notify that it has gotten the message. To do this, we have the receiver send an acknowledgement packet (ack) saying the packet was successfully received. Using these two techniques, we can establish a simple way to send messages reliably: The sender will send its message, and continue resending for any packets that have timed out and it hasn't received an ack for yet.

However, since messages can be arbitrarily long, if the sender decides to send the entire message at once before resending anything, the receiver might end up with a lot of queued up packets that it can't do anything with because some packet at the beginning got dropped. At the same time, if we always wait for an ack to arrive before sending the next packet, the message will take a long time to transmit. Therefore, we go with something in the middle by using a *sliding window protocol*.

With a sliding window protocol, the sender and receiver each have a window of some established size. The sender sends all the packets it hasn't yet sent in this window, and waits for acks to come back. Every time it receives an ack, the sender's window slides forward so it can send another untransmitted packet. Meanwhile, the receiver takes in all packets corresponding to the unreceived packets in its window, dropping any packets that have already been received or are outside the range of the window. Every time a packet at the beginning of the window is received, the window slides forward until the first sequence number in the window is one that hasn't been received yet. In addition, the receiver acks all packets that it has either already received or have been stored for the first time; any packets that are beyond the window range are dropped without ack. This way, multiple packets can be sent at a time without having the possibility of building up too many packets on the receiving side blocked by dropped or unordered packets.

Note: Reliable transport sounds great and convenient. So…why would we ever use UDP instead of TCP? UDP is more efficient and has less overhead – we'd like to use it in those cases were we actually don't need reliability. These are situations where timing is more important than reliability. For example, in real-time applications like webcam and voice chat, it's more important to keep up with what's happening right now rather than make sure a packet or two in the middle get filled in (packets that arrive late aren't really useful).

**What's in a Name?**

We've seen just now that the network identifies hosts by IP address. However, people don't typically remember IP addresses – we remember host names. This separation of name and address allows us to make both users (who like names) and the network (which likes addresses) happy, as long as we have a way to translate between name and address. It also allows us the following nice properties:

- The address of a particular host can change, but users don't have to know – they can keep using the same name.
- Multiple names can be mapped to the same address, which is useful if you have several commonly used names.
- Multiple addresses can be associated with the same name, which can help load balance the site by using mirrors.

To resolve names, we go through Domain Name System (DNS) servers. DNS servers work like a distributed database - we have a hierarchy of DNS servers, each of which knows about the location of a particular subset of hosts. When a host needs to look up an address, it contacts its local DNS server, which then asks a root DNS server where to go. The root then tells us where the Top-Level Domain (TLD) server is (each of which knows about one of the .edu / .com / etc. domains), which then tells us where to go next to find the next piece is. We continue doing this recursively until we find the full address.