Lecture 10

Gidon Rosalki

2025-12-28

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_networking`

1 Recap - Traffic management

We covered traffic engineering in the network layer (L3). We can also perform congestion control in the transport layer (L4), through carefully engineering the protocols. It is at this layer that concepts such as TCP, and UDP appear.

2 Transport layer

The purpose of transport services, and protocols is to provide *logical communication* between app processes running on different hosts. Transport protocols run in the end systems, the nodes on the network, rather than purely on the switches, or routers. On the sending side, the node breaks app messages into segments which it passes to the network layer, and the receiver reassembles segments received in the network layer back into messages for the application layer.
There is more than 1 transport protocol available to applications: UDP and TCP.
TCP provides reliable, in order delivery. It includes congestion control, flow control, and setting up / closing connections. It ensures that when a message is sent, all of its packets will arrive at the destination host, in order. In contrast, UDP guarantees nothing, it is a minimal, no frills extension of IP.
Consider UDP to be shouting into the void. The recipient might hear, but it is not guaranteed, and if they do hear, it is possible that they will hear your words out of order, due to strange echoes. In contrast, TCP involves numbering every word you say, verifying that the recipient is ready to receive each word, and that they have heard each word every time. From this, we can intuitively see that TCP is slower than UDP. However, when one wants to receive a webpage, for example, the speed of download with a difference of potentially milliseconds is not relevant, but what is relevant is that the words on the webpage appear in the right order, and are all received.

3 Multiplexing

The transport layer also allows multiplexing messages on a single wire. We may have more than one application communication between two hosts, A and B, over the same wire. To enable this, we have the socket abstraction of the transport layer. Sockets are abstractions of IP addresses, combined with ports. Ports numbered from 0 to 65535 (16 bit number). The socket of the packet is part of the header in TCP / UDP messages. Each application sends / receives messages on a single socket, and this way the OS can separate messages that have been received from a single wire, and a single host to different applications on the computer.

Demultiplexing works as follows. When the host receives IP datagrams, each datagram has in its header the source IP address, and the destination IP address. Each datagram is responsible for 1 transport layer segment, and each segment has a source port number, and a destination port number. The host uses IP addresses, and port number to direct segments to the appropriate sockets.
UDP sockets are identified by the tuple (destination IP address, destination port number). When the host OS receives a UDP segment, it checks the destination port number in the segment, and directs the segment to the socket with that port number. It should be noted that IP datagrams with difference source IP addresses / source sockets, but the same destination port number will be directed to the same socket.
In contrast, TCP sockets are identified by 4-tuples:

- Source IP address
- Source port number
- Destination IP address
- Destination port number

The receiving host OS then directs the segments to the appropriate socket, ensuring that different transmitting hosts do not transmit to the same socket. Note that this means for web servers, they will have a different socket for every connecting client. In fact, if the clients do not use persistent HTTP connections, then there will be a different socket for every request.

4 UDP

This is a bare bones wrapper around IP. The segments sent over UDP may be lost, and delivered out of order. There is no handshake between the sender, and the receiver, and each segment is handled independently. Since TCP seems so much more stable, we are left wondering why UDP exists at all. Well, sometimes it is necessary, you cannot really have a TCP handshake in DHCP. Furthermore, the handshake can induce a (relatively) large delay before the start of transmission, which is not always desirable.
It is also incredibly simple, with no connection state at the sender, or the receiver, it has a very small segment header, and has no congestion control, so can send messages as fast as desired. This is particularly useful in things like video / voice transmission when in a video / phone call. Since each packet contains a small amount of the data stream, losing a single packet here and there will not significantly, or even noticeably impact the experience. However, if the video stream was being sent by TCP, then should a packet be lost, then the call would freeze until that packet had been received, or worse, groups of frames would start to be played out of order, along with their attached audio, making the entire thing completely incomprehensible. Simply losing a couple of frames in the middle is greatly preferable.

5 TCP

5.1 Overview

TCP is point to point, with a single sender / receiver pair, sending a reliable, in order byte stream, and is pipelined, with congestion and flow control, according to a set window size. It sends full duplex, so data can flow in both directions in the same connection, with a defined Maximum Segment Size (MSS).

6 Opening and closing

Since we need a sender, and a receiver to establish a TCP connection before exchanging data segments, it follows that some variables need to be initialised first: The sequence numbers, the buffers into which data may be received, and the flow control information. The client is the connection initiator, sending a request to open a connection on a socket, and the server is contacted by the client, where it will accept a TCP connection on a socket. This process may be summarised by the following three way handshake:

1. The client host sends a TCP SYN segment to the server, which specifies the initial sequence number, but contains no data
2. The server host receives the SYN, and replies with the SYNACK segment. The server allocates buffers, and specifies the server initial sequence number
3. The client receives the SYNACK, to which it responds with an ACK segment, which may contain data

To close a connection, the client closes the socket. This involves two steps.

1. The client sends the TCP FIN control segment to the server
2. The server responds with ACK, and then its own FIN
3. (The client sends an ACK, but keeps the socket open for "timed wait", a period of time where it waits to see if the server will send another ACK, FIN combination, because it did not receive the clients ACK)

After the tied wait is over, the client is confident that the server has received its own ACK, and closes the socket.

6.1 Sequences and ACKs

Each segment begins with a number, known as the sequence number, which indicates the number of this segment in the stream of data. Let us consider the transmitting host A, and receiving host B. Both hosts are maintaining their own sequence numbers, which are mirrored in the other host's ACK numbers. So, for a simplified example, when A sends a segment, then along with the data it sends it with the next sequence number in its own stream, and the ACK that corresponds to the next sequence number that B will transmit. B then responds to this message with an ACK message, containing the next sequence number in its stream, and an ACK that corresponds to A's next sequence number.

Direction	Seq	ACK	Time
A $\rightarrow$ B	42	79	1
B $\rightarrow$ A	79	43	2
A $\rightarrow$ B	43	80	3

Table 1: Simplified example

As we can see, TCP creates reliable service on top of IP's unreliable service. It provides pipelined segments, with cumulative ACKs, and a single retransmission timer. Retransmission is triggered by timeout events (too much time has passed without an ACK), and duplicate ACKs.

6.2 Simplified sender

Let us consider a simplified sender, that ignores duplicate ACKs, flow control, and congestion control. Upon receiving data, create a segment with a sequence number. The sequence number is a byte stream number of the first data byte in the segment. Start the ACK timer, if it is not already running (since if there is already a segment that has not been ACKed, the timer will already be running). If the timer reaches the timeout interval, then retransmit the segment that started the timer, and restart it. When an ACK is received, if it acknowledges previously unACKed segments, then update which segments are known to be ACKed, and start the timer if there remain outstanding segments.