Lecture 11

Gidon Rosalki

2026-01-04

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

1 TCP Round Trip Time / Timeout

As we have seen, a message can be lost when being sent to the recipient, or the ACK can be lost on its way to the sender. The sender cannot differentiate between these 2 scenarios, so it is very important that the sender waits long enough to receive the ACK *before* resending the message, so that it does not pointlessly transmit the same message to the recipient when there is no need.
It is important that messages are often sent before their ACK arrives, in order to save time. However, TCP uses *cumulative ACK*, meaning that if ACK(1) was lost, but ACK(2) was received, since ACK(2) implies ACK(1), it will be assumed that message 1 was in fact received as well, even without the ACK. Since the ACK values are comprised of the cumulative sum of the number of received bytes, should message 1 not arrive, but message 2 does arrive, then the ACK value for message 2 will not agree with the expected ACK value, and the sender will know that there was a missed message further back that should be resent.

We need to find a method to set the timeout value. It needs to be longer than the round trip time (RTT), but this RTT value varies. If it is too short then there will be a premature timeout, and we will have unnecessary retransmissions. However, if it is too long, then there will be a slow reaction to segment loss. As we see, we need a good estimate of the RTT. A method for estimating this is **SampleRTT**, where we measure the time from the segment transmission until the ACK receipt, ignoring retransmissions. However, this will vary, and we want the estimated RTT to be more consistent, so we take the exponentially weighted average of the most recent n measurements, rather than just the last one. This exponential weighting ensures that the older history decays over time, and a much larger focus is put on recent measurements.
To calculate the actual EstimatedRTT, we

$$\text{EstimatedRTT} = (1-\alpha)\cdot\text{EstimatedRTT} + \alpha\cdot\text{SampleRTT}$$

This gives an exponential weighted moving average, where the influence of past sample decreases exponentially. A typical value for α is 0.125.

So, now that we have the EstimatedRTT, we may set the timeout. To do this, we take the EstimatedRTT, plus some safety margin. Large variations in EstimatedRTT will result in us needing a larger safety margin. We will call this margin DevRTT, derived from the deviation of the RTT:

$$\text{DevRTT} = (1-\beta)\cdot\text{DevRTT} + \beta\cdot|\text{SampleRTT} - \text{EstimatedRTT}|$$

Where typically $\beta = 0.25$. We may then set the timeout interval:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4\cdot\text{DevRTT}$$

1.1 Fast retransmit

The timeout period is often relatively long, resulting in a long delay before resending the lost packet. Since the sender often sends many segments back to back, it may detect lost segments from duplicate ACKs, since if a segment is lost, there will likely be many duplicate ACKs for the preceding. If sender receives 3 ACKs for same data, it assumes that segment after ACKed data was lost, and so retransmits the segment before the timer expires. The resultant ACK for this retransmitted segment, will be the ACK of the final received segment. For example, if the sender sends packets 1, 2, 3, 4, and 5, but 2 does not arrive, then it will receive ACK(1), since that arrived, nothing for 2, and then ACK(1) 3 times in a row, for 3, 4, and 5. This tells it that 2 did not arrive, so it may retransmit 2 before the timeout expires. Upon receiving 2, the receiver will then have also received 3 - 5, and respond with ACK(5), since it is cumulative.

2 Congestion Control

We may informally define congestion as too many sources sending too much data for the network to handle. Note that this is distinct and *different* from flow control. It manifests as lost packets, from buffer overflows in routers, and long delays, since there is a large queue of messages to be sent in the router buffer.

Congestion control is important. In October 1986, the internet had its first congestion based collapse, where the link from LBL to UC Berkely, which is 365 metres, with 3 hops, and a throughput of 32Kbps, had its throughput drop to 40bps. This happened since the flow control mechanism focused only on receiver congestion, not network congestion. The large number of hosts sharing a slow and small network, resulted in the network becoming the bottleneck, rather than the receivers, but the TCP flow control *only* prevented overwhelming the receivers.

Let us take a moment to be precise about the differences between flow control, and congestion control. **Flow control** prevents the source from sending data that the recipient will not be able to handle because its buffer space is full. This is fairly easy, and TCP handles this by having the recipient advertise its free buffer space. **Congestion control** prevents the source from sending data that will be dropped by a router, because the router's buffer is full. This is more complicated, since many different sources packets' can pass through the same router. Congestion control is handled through a combination of 2 mechanisms, protocols at the end hosts (TCP), and queuing at the routers (Droptail, RED, and so on).

There are also 2 main approaches to congestions control, **end to end**, which has no explicit feedback from the network. Congestion is inferred from the end system observed loss, and delay. This is the approach taken by TCP. There is also **network-assisted congestion control**. Here, routers provide feedback to end systems, in the form of a single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM). They also send the explicit rate at which the senders should transmit.

2.1 End to End protocols and queuing mechanisms

As we have stated, there are two families of transport layer protocols. The protocols that do not congestion control, like UDP, primarily used for time sensitive applications (VoIP, IPTV, online gaming), and servers answering small queries from many clients (DNS), and window based congestion control protocols, like TCP.

TCP sends a window of packets, in every RTT. This is to say, that it sends W packets, where W is the window size. This way, when the RTT ends, the sender receives the ACK of the first packet in the window. In order to use this, one needs to find the size of W. Now, if we limit the number of packets in the network to the window W, then the source rate is

$$\text{Source rate} = W \cdot \frac{MSS}{RTT}$$

Where

$$MSS = \text{Maximum Segment Size}$$

If W is too small, then the rate is less than the capacity, but if the rate is greater than the capacity, then that will cause congestion.

TCP congestion control is split into two main parts, Slow Start, and Congestion Avoidance. The overall concept is that we start very slowly, to prevent beginning with congestion, and slowly increase the amount of sent data, until we pass the slow start threshold (ssthresh), at which point we transition from slow start, to congestion avoidance.

2.1.1 Slow Start

Here, one starts with $W = 1$, and on each successful ACK, we increment W by 1. Whenever we reach the end of an RTT, we use exponential growth, and set $W = 2W$. When $W \geq ssthresh$, then we switch to CA.

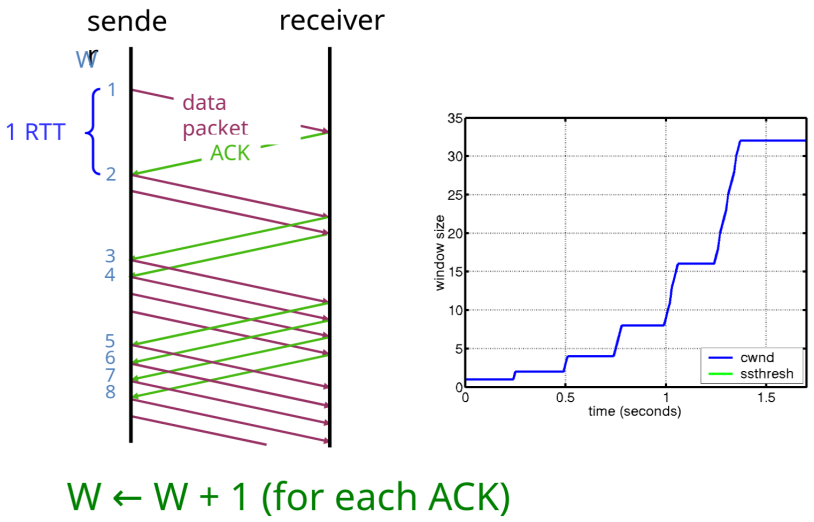

Figure 1: Slow start

2.1.2 Congestion Avoidance

On each successful ACK, set $W = W + \frac{1}{W}$. After each RTT, set $W = W + 1$.

2.1.3 TCP Tahoe

The basic idea is to gently probe the network for spare capacity, and drastically reduce the rate on congestion. Packet loss indicates congestion, and is detected by retransmission timeouts, or receipt of at least 3 duplicate ACKs. When packet loss is detected, then the slow start threshold set to $ssthresh \leftarrow \frac{W}{2}$, and then the algorithm resets to starting slow start from the beginning by setting $W = 1$.

2.1.4 TCP Reno

This avoids slow start, to prevent the network links from emptying after encountering packet loss. The basic idea is to treat timeout, and 3 duplicate ACKs differently, since every duplicate ACK indicates a successful packet transmission. It uses AIMD (Additive Increase Multiplicative Decrease). It probes the available bandwidth, and has a fast recovery to avoid a slow start. Duplicate ACKs result in a fast recovery, with a fast retransmission, but a timeout results in a retransmission, and a reset to slow start.

The basic algorithm is as follows:

Reno 1

1: **for** ACK in ACKs **do**
2: $W+ = \frac{1}{W}$
3: **end for**
4: **for** Every 3 duplicate ACKs **do**
5: ssthreshold $= \frac{W}{2}$
6: $W = \frac{W}{2}$
7: **end for**
8: **for** Every timeout **do**
9: ssthreshold $= \frac{W}{2}$
10: Enter slow start (with $W = 1$)
11: **end for**

The first for loop is the additive increase stage, and the second multiplicative decrease.