Lecture-26: Congestion Control

1 TCP congestion control

TCP provides a reliable transport service between two processes running on different hosts. A key component of TCP is its congestion-control mechanism. The IP layer provides no explicit feedback to the end systems regarding network congestion, and hence TCP must use end-to-end congestion control rather than network-assisted congestion control.

The approach taken by TCP is to have each sender limit the rate at which it sends traffic into its connection as a function of perceived network congestion. If a TCP sender perceives no congestion on the path between itself and the destination, then the TCP sender increases its send rate. If the sender perceives congestion along the path, the sender reduces its send rate. This approach raises the following three questions for a TCP sender.

1. How to limit the rate at which the traffic is sent into the connection?
2. How to perceive the end-to-end congestion on the path to the destination?
3. What algorithm should be used to change the sending rate as a function of perceived congestion?

1.1 TCP variables

Suppose host A sends a large file to host B over a TCP connection.

1.1.1 Receiver TCP variables

Host B allocates a receive buffer to this connection; denote its size by `RcvBuffer`. From time to time, the application process in host B reads from the buffer. Define the following variables at the receiver.

(a) `RcvBuffer`: the buffer size for the data stream at B.
(b) `LastByteRead`: the number of the last byte in the data stream read from the buffer by the application process in B.
(c) `LastByteRcvd`: the number of the last byte in the data stream that has arrived from the network and has been placed in the receive buffer at B.
(d) `rwnd`: the size of receive window at B.

Because TCP is not permitted to overflow the allocated buffer, we must have

$$\texttt{LastByteRcvd} - \texttt{LastByteRead} \leqslant \texttt{RcvBuffer}. \tag{1}$$

The receive window, denoted `rwnd` is set to the amount of spare room in the buffer. That is,

$$\texttt{rwnd} = \texttt{RcvBuffer} - (\texttt{LastByteRcvd} - \texttt{LastByteRead}). \tag{2}$$

Because the spare room changes with time, `rwnd` is dynamic. How does the connection use the variable `rwnd` to provide the flow-control service? host B tells host A how much spare room it has in the connection buffer by placing its current value of `rwnd` in the receive window field of every segment it sends to A. Initially, host B sets $\texttt{rwnd} = \texttt{RcvBuffer}$. Note that to ensure that the receiver buffer does not overflow, host B must keep track of several connection-specific variables.

1.1.2 Sender TCP variables

Host A in turn keeps track of the following two variables,

(a) `LastByteSent`: the number of the last byte in the data stream that was sent from host A to the network and has been placed in the transmit buffer at A.
(b) `LastByteAcked`: the number of bytes that have been acknowledged to be received by the receiver B, and hence can be removed from the transmit buffer at A.

Note that the difference between these two variables, $\texttt{LastByteSent} - \texttt{LastByteAcked}$, is the amount of unacknowledged data that A has sent into the connection. By keeping the amount of unacknowledged data less than the value of `rwnd`, host A is assured that it is not overflowing the receive buffer at host B. Thus, host A makes sure throughout the connection's life that

$$\texttt{LastByteSent} - \texttt{LastByteAcked} \leqslant \texttt{rwnd}. \tag{3}$$

There is one minor technical problem with this scheme. To see this, suppose host B's receive buffer becomes full so that $\texttt{rwnd} = 0$. After advertising $\texttt{rwnd} = 0$ to host A, also suppose that B has nothing to send to A. Now consider what happens. As the application process at B empties the buffer, TCP does not send new segments with new `rwnd` values to host A. Indeed, TCP sends a segment to host A only if it has data to send or if it has an acknowledgment to send. Therefore, host A is never informed that some space has opened up in host B's receive buffer. Hence, host A is blocked and can transmit no more data! To solve this problem, the TCP specification requires host A to continue to send segments with one data byte when B's receive window is zero. These segments will be acknowledged by the receiver. Eventually the buffer will begin to empty and the acknowledgments will contain a nonzero `rwnd` value.

1.2 Congestion window

The TCP congestion-control mechanism operating at the sender keeps track of an additional variable, the *congestion window* denoted `cwnd`. The variable `cwnd` imposes a constraint on the rate at which a TCP sender can send traffic into the network. Specifically, the amount of unacknowledged data at a sender may not exceed the minimum of `cwnd` and `rwnd`. That is,

$$\texttt{LastByteSent} - \texttt{LastByteAcked} \leqslant \min\{\texttt{cwnd}, \texttt{rwnd}\}. \tag{4}$$

In order to focus on congestion control (as opposed to flow control), let us henceforth assume that the TCP receive buffer is so large that the receive-window constraint can be ignored. Thus, the amount of unacknowledged data at the sender is solely limited by `cwnd`. We will also assume that the sender always has data to send, i.e., that all segments in the congestion window are sent.

1.3 Transmit rate

The constraint in (4) limits the amount of unacknowledged data at the sender and therefore indirectly limits the sender's send rate. To see this, consider a connection for which loss and packet transmission delays are negligible. Let $T \triangleq \texttt{RTT}$ and consider the time instant sequence $(T_0, T_1, \ldots,)$ where $T_n = nT$ is the beginning of nth round trip duration. We denote the size of `cwnd` at this instant T_n by W_n. Hence, at instant T_n, the constraint (4) permits the sender to send `cwnd` bytes of data into the connection. At time T_{n+1}, the sender receives acknowledgements for the data. Thus the sender's send rate is roughly $\frac{W_n}{T}$ bytes per second in the nth round trip duration $[T_n, T_{n+1})$. By adjusting the value of W_n, the sender can adjust the rate at which it sends data into its connection.

1.4 Loss event

Let's next consider how a TCP sender perceives that there is congestion on the path between itself and the destination. Let us define a *loss event* at a TCP sender as the occurrence of either (a) a timeout or (b) the receipt of three duplicate ACKs from the receiver. When there is excessive congestion, then one or more router buffers along the path overflows, causing a datagram containing a TCP segment to be dropped. The dropped datagram, in turn, results in a loss event at the sender. That is, either a timeout or the receipt of three duplicate ACKs, which is taken by the sender to be an indication of congestion on the sender-to-receiver path.

Having considered how congestion is detected, let's next consider the more optimistic case when the network is congestion-free, that is, when a loss event doesn't occur. In this case, acknowledgements for

previously unacknowledged segments will be received at the TCP sender. As we'll see, TCP will take the arrival of these acknowledgements as an indication that all is well. That is, the segments being transmitted into the network are being successfully delivered to the destination. Hence, TCP will use acknowledgements to increase its congestion window size and hence its transmission rate. Note that if acknowledgements arrive at a relatively slow rate e.g., if the end-end path has high delay or contains a low-bandwidth link, then the congestion window will be increased at a relatively slow rate. On the other hand, if acknowledgements arrive at a high rate, then the congestion window will be increased more quickly. Because TCP uses acknowledgments to trigger or clock its increase in congestion window size, TCP is said to be *self-clocking*.

Given the *mechanism* of adjusting the value of `cwnd` to control the sending rate, the critical question remains: How should a TCP sender determine the rate at which it should send? If TCP senders collectively send too fast, they can congest the network, leading to the type of congestion collapse that we saw in Figure 3.48. Indeed, the version of TCP that we'll study shortly was developed in response to observed Internet congestion collapse [Jacobson 1988] under earlier versions of TCP. However, if TCP senders are too cautious and send too slowly, they could under utilize the bandwidth in the network; that is, the TCP senders could send at a higher rate without congesting the network. How then do the TCP senders determine their sending rates such that they don't congest the network but at the same time make use of all the available bandwidth? Are TCP senders explicitly coordinated, or is there a distributed approach in which the TCP senders can set their sending rates based only on local information? TCP answers these questions using the following guiding principles.

- A lost segment implies congestion, and hence, the TCP sender's rate should be decreased when a segment is lost. Recall from our discussion in Section 3.5.4, that a timeout event or the receipt of four acknowledgements for a given segment (one original ACK and then three duplicate ACKs) is interpreted as an implicit *loss event* indication of the segment following the quadruply ACKed segment, triggering a retransmission of the lost segment. From a congestion-control standpoint, the question is how the TCP sender should decrease its congestion window size, and hence its sending rate, in response to this inferred loss event.

- An acknowledged segment indicates that the network is delivering the sender's segments to the receiver, and hence, the sender's rate can be increased when an ACK arrives for a previously unacknowledged segment. The arrival of acknowledgements is taken as an implicit indication that all is well. The segments are being successfully delivered from sender to receiver, and the network is thus not congested. The congestion window size can thus be increased.

- Bandwidth probing. Given ACKs indicating a congestion-free source-to-destination path and loss events indicating a congested path, TCP's strategy for adjusting its transmission rate is to increase its rate in response to arriving ACKs until a loss event occurs, at which point, the transmission rate is decreased. The TCP sender thus increases its transmission rate to probe for the rate that at which congestion onset begins, backs off from that rate, and then to begins probing again to see if the congestion onset rate has changed. The TCP sender's behavior is perhaps analogous to the child who requests (and gets) more and more goodies until finally he/she is finally told *No!*, backs off a bit, but then begins making requests again shortly afterwards. Note that there is no explicit signalling of congestion state by the network. ACKs and loss events serve as implicit signals, and that each TCP sender acts on local information asynchronously from other TCP senders.

2 TCP congestion control algorithm

Given this overview of TCP congestion control, we're now in a position to consider the details of the celebrated TCP congestion-control algorithm, which was first described in [Jacobson 1988] and is standardized in [RFC 5681]. The algorithm has three major components: (1) slow start, (2) congestion avoidance, and (3) fast recovery. Slow start and congestion avoidance are mandatory components of TCP, differing in how they increase the size of `cwnd` in response to received ACKs. We'll see shortly that slow start increases the size of `cwnd` more rapidly (despite its name!) than congestion avoidance. Fast recovery is recommended, but not required, for TCP senders.

2.1 Slow start

When a TCP connection begins, the value of cwnd is typically initialized to a small value of 1 MSS, resulting in an initial sending rate of roughly $\frac{\text{MSS}}{\text{RTT}}$. For example, if MSS $= 500$ bytes and RTT $= 200$ milli seconds, the resulting initial sending rate is only about 20 kbps. Since the available bandwidth to the TCP sender may be much larger than $\frac{\text{MSS}}{\text{RTT}}$, the TCP sender would like to find the amount of available bandwidth quickly. Thus, in the slow-start state, the value of cwnd begins at 1 MSS and increases by 1 MSS every time a transmitted segment is first acknowledged. In one RTT the number of MSS sent by cwnd, and if each of them is received successfully then the cwnd is doubled. That is, we can write the evolution of cwnd value at each RTT, when there are no loss events, as $W_n = 2W_{n-1}$. This process results in a doubling of the sending rate every RTT. Thus, the TCP send rate starts slow but grows exponentially during the slow start phase.

First, if there is a loss event i.e., congestion indicated by a timeout, the TCP sender sets the value of cwnd to 1 and begins the slow start process anew. It also sets the value of a second state variable *slow start threshold* denoted by ssthresh to $\frac{\text{cwnd}}{2}$ to half of the value of the congestion window value when congestion was detected. The second way in which slow start may end is directly tied to the value of ssthresh. Since ssthresh is half the value of cwnd when congestion was last detected, it might be a bit reckless to keep doubling cwnd when it reaches or surpasses the value of ssthresh. Thus, when the value of cwnd equals ssthresh, slow start ends and TCP transitions into congestion avoidance mode. TCP increases cwnd more cautiously when in congestion-avoidance mode. The final way in which slow start can end is if three duplicate ACKs are detected, in which case TCP performs a fast retransmit and enters the fast recovery state.

2.2 Congestion avoidance

On entry to the congestion-avoidance state, the value of cwnd is approximately half its value when congestion was last encountered. In this state, congestion could be just around the corner, and thus, rather than doubling the value of cwnd every RTT, TCP adopts a more conservative approach and increases the value of cwnd by just a single MSS every RTT. This can be accomplished in several ways. A common approach is for the TCP sender to increase cwnd by MSS bytes ($\frac{\text{MSS}}{\text{cwnd}}$) whenever a new acknowledgment arrives. For example, if MSS is 1,460 bytes and cwnd is 14,600 bytes, then 10 segments are being sent within an RTT. Each arriving ACK (assuming one ACK per segment) increases the congestion window size by $\frac{1}{10}$MSS, and thus, the value of the congestion window will have increased by one MSS after ACKs when all 10 segments have been received. That is, when there are no loss events, we can write the evolution of cwnd for each RTT under congestion avoidance as $W_n = W_{n-1} + 1$.

TCP's congestion-avoidance algorithm behaves the same when a timeout occurs. As in the case of slow start, the value of cwnd is set to 1 MSS, and the value of ssthresh is updated to half the value of cwnd when the loss event occurred. Recall, however, that a loss event also can be triggered by a triple duplicate ACK event. In this case, the network is continuing to deliver segments from sender to receiver (as indicated by the receipt of duplicate ACKs). So TCP's behavior to this type of loss event should be less drastic than with a timeout-indicated loss. TCP halves the value of cwnd (adding in 3 MSS for good measure to account for the triple duplicate ACKs received) and records the value of ssthresh to be half the value of cwnd when the triple duplicate ACKs were received. The fast-recovery state is then entered.

2.3 Fast recovery

In fast recovery, the value of cwnd is increased by 1 MSS for every duplicate ACK received for the missing segment that caused TCP to enter the fast-recovery state. Eventually, when an ACK arrives for the missing segment, TCP enters the congestion-avoidance state after deflating cwnd. If a timeout event occurs, fast recovery transitions to the slow-start state after performing the same actions as in slow start and congestion avoidance: The value of cwnd is set to 1 MSS, and the value of ssthresh is set to half the value of cwnd when the loss event occurred. Fast recovery is a recommended, but not required, component of TCP.

An early version of TCP, known as TCP Tahoe, unconditionally cut its congestion window to 1MSS and entered the slow-start phase after either a timeout-indicated or triple-duplicate-ACK-indicated loss event. The newer version of TCP, TCP Reno, incorporated fast recovery. Figure 3.53 illustrates the evolution of TCP's congestion window for both Reno and Tahoe. In this figure, the threshold is initially equal to 8MSS. For the first eight transmission rounds, Tahoe and Reno take identical actions. The congestion window

climbs exponentially fast during slow start and hits the threshold at the fourth round of transmission. The congestion window then climbs linearly until a triple duplicate- ACK event occurs, just after transmission round 8. Note that the congestion window is 12MSS when this loss event occurs. The value of ssthresh is then set to 0.5cwnd = 6MSS. Under TCP Reno, the congestion window is set to cwnd = 6MSS and then grows linearly. Under TCP Tahoe, the congestion window is set to 1MSS and grows exponentially until it reaches the value of ssthresh, at which point it grows linearly. Figure 3.52 presents the complete FSM description of TCP's congestion-control algorithms: slow start, congestion avoidance, and fast recovery. The figure also indicates where transmission of new segments or retransmitted segments can occur. Although it is important to distinguish between TCP error control/retransmission and TCP congestion control, it's also important to appreciate how these two aspects of TCP are inextricably linked.

2.4 Retrospective

Ignoring the initial slow-start period when a connection begins and assuming that losses are indicated by triple duplicate ACKs rather than timeouts, TCP's congestion control consists of linear (additive) increase in cwnd of 1 MSS per RTT and then a halving (multiplicative decrease) of cwnd on a triple duplicate-ACK event. For this reason, TCP congestion control is often referred to as an additive-increase, multiplicative- decrease (AIMD) form of congestion control. AIMD congestion control gives rise to the *saw tooth* behavior shown in Figure 3.54, which also nicely illustrates our earlier intuition of TCP *probing* for bandwidth. TCP linearly increases its congestion window size (and hence its transmission rate) until a triple duplicate-ACK event occurs. It then decreases its congestion window size by a factor of two but then again begins increasing it linearly, probing to see if there is additional available bandwidth.

TCP splitting: Optimizing the performance of cloud services

For cloud services such as search, e-mail, and social networks, it is desirable to provide a high-level of responsiveness, ideally giving users the illusion that the services are running within their own end systems (including their smartphones). This can be a major challenge, as users are often located far away from the data centers that are responsible for serving the dynamic content associated with the cloud services. Indeed, if the end system is far from a data center, then the RTT will be large, potentially leading to poor response time performance due to TCP slow start.

As a case study, consider the delay in receiving a response for a search query. Typically, the server requires three TCP windows during slow start to deliver the response. Thus the time from when an end system initiates a TCP connection until the time when it receives the last packet of the response is roughly 4RTT (one RTT to set up the TCP connection plus three RTTs for the three windows of data) plus the processing time in the data center. These RTT delays can lead to a noticeable delay in returning search results for a significant fraction of queries. Moreover, there can be significant packet loss in access networks, leading to TCP retransmissions and even larger delays.

One way to mitigate this problem and improve user-perceived performance is to (1) deploy front-end servers closer to the users, and (2) utilize TCP splitting by breaking the TCP connection at the front-end server. With TCP splitting, the client establishes a TCP connection to the nearby front-end, and the front-end maintains a persistent TCP connection to the data center with a very large TCP congestion window. With this approach, the response time roughly becomes $4\text{RTT}_{\text{FE}} + \text{RTT}_{\text{BE}}$ + processing time, where RTT_{FE} is the round-trip time between client and front-end server, and RTT_{BE} is the round-trip time between the front-end server and the data center (back-end server). If the front-end server is close to client, then this response time approximately becomes RTT plus processing time, since RTT_{FE} is negligibly small and RTT_{BE} is approximately RTT.

In summary, TCP splitting can reduce the networking delay roughly from 4RTT to RTT, significantly improving user-perceived performance, particularly for users who are far from the nearest data center. TCP splitting also helps reduce TCP retransmission delays caused by losses in access networks. Today, Google and Akamai make extensive use of their CDN servers in access networks to perform TCP splitting for the cloud services they support.

Algorithm 1 TCP congestion control algorithm

Slow start: $\texttt{cwnd} = \texttt{1MSS}, \texttt{ssthresh} = \texttt{64KB}, \texttt{dupAckCount} = 0$
if $\texttt{cwnd} < \texttt{ssthresh}$ **and** $\texttt{dupAckCount} < 3$ **then**
    if newAck **then**
        $\texttt{cwnd} = \texttt{cwnd} + \texttt{MSS}, \texttt{dupAckCount} = 0$
    else if dupAck **then**
        $\texttt{dupAckCount}++$
    else if timeout **then**
        $\texttt{ssthresh} = \texttt{cwnd}/2, \texttt{cwnd} = \texttt{1MSS}, \texttt{dupAckCount} = 0$
    end if
else if $\texttt{dupAckCount} == 3$ **then**
    Fast Recovery: $\texttt{ssthresh} = \texttt{cwnd}/2, \texttt{cwnd} = \texttt{ssthresh} + \texttt{3MSS}$
    if dupAck **then**
        Fast Recovery: $\texttt{cwnd} = \texttt{cwnd} + \texttt{MSS}$
    else if newAck **then**
        Congestion Avoidance: $\texttt{cwnd} = \texttt{ssthresh}, \texttt{dupAckCount} = 0$
    else if timeout **then**
        Slow start: $\texttt{ssthresh} = \texttt{cwnd}/2, \texttt{cwnd} = \texttt{1MSS}, \texttt{dupAckCount} = 0$
    end if
else if $\texttt{cwnd} \geqslant \texttt{ssthresh}$ **then**
    Congestion Avoidance
    if $\texttt{dupAckCount} < 3$ **then**
        if newAck **then**
            $\texttt{cwnd} = \texttt{cwnd} + \frac{\texttt{MSS}}{\texttt{cwnd}}, \texttt{dupAckCount} = 0$
        else if dupAck **then**
            $\texttt{dupAckCount}++$
        else if timeout **then**
            Slow start: $\texttt{ssthresh} = \texttt{cwnd}/2, \texttt{cwnd} = \texttt{1MSS}, \texttt{dupAckCount} = 0$
        end if
    else if $\texttt{dupAckCount} == 3$ **then**
        Fast Recovery: $\texttt{ssthresh} = \texttt{cwnd}/2, \texttt{cwnd} = \texttt{ssthresh} + \texttt{3MSS}$
    end if
end if