13. TCP Flow Control and Congestion Control

- TCP Flow Control
- Congestion control – general principles
- TCP congestion control

Jon Turner – slides adapted from Kurose and Ross
TCP Flow Control

- Receive side of TCP connection has a receive buffer
- If receiver’s application does not read data fast enough, the buffer may fill up
- The TCP flow control mechanism prevents the sender from sending more data than receiver can store
  - essentially a speed-matching service that prevents the sender from going too fast for the receiver
How TCP Flow Control Works

- To simplify discussion, pretend TCP receiver discards out-of-order segments
- Spare room in buffer
  \[ \text{RcvWindow} = \text{RcvBuffer} - (\text{LastByteRcvd} - \text{LastByteRead}) \]
- Receiver sends the value of RcvWindow in the value of RcvWindow field of each TCP segment
- Sender never allows the number of unacknowledged bytes to exceed RcvWindow
  - guarantees receive buffer doesn't overflow
- To avoid deadlock, sender continues to send one byte segments when RcvWindow=0
  - this ensures that sender is informed when RcvWindow>0
Exercises

1. Suppose host A is sending data to host B using TCP. Assume that the DSL link leading to B has a rate of 2 Mb/s and that this is the “bottleneck” in the connection. Assume that A has 500 KB to send and that B’s receive buffer can only hold 20 KB. Also, assume that for the first three seconds, the application at B does not read any data from the connection, but that after that it reads data at the rate of 100 KB/s. Draw a chart showing the amount of data in the receive buffer at B as a function of time. Label the points on the curve where either A’s sending rate or B’s reading rate changes.

2. In the scenario from the previous question, how many single byte segments does A send, if the RTT is 100 ms.
Principles of Congestion Control

- What is meant by congestion?
  - this is when the amount of traffic arriving at a network link exceeds the link rate for an “excessive time period”
  - caused by sources sending too much data for the link to handle
    - note that it’s the network that is limiting the traffic flow in this case, not the receiver
  - can cause router queues to fill up and overflow

- Consequences
  - packets get lost at routers
  - network delays get large
  - network throughput can actually drop as load increases
    - this happens because packets may be dropped after passing through several routers, wasting the capacity of “upstream” links
    - since some network effort is wasted, throughput drops below peak
Congestion Scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission

- large delays when congested
- achieve max possible throughput
Congestion Scenario 2

- One router, *finite* buffers
- Sender retransmission of timed-out packet
  - application-layer input = application-layer output: \( \lambda_{in} = \lambda_{out} \)
  - transport-layer input includes retransmissions: \( \lambda_{in} \geq \lambda_{in} \)
Congestion Scenario 2a

- Sender sends only when router buffers available
Congestion Scenario 2b

- Packets may get dropped at router due to full buffers
- Sender only resends if packet *known* to be lost
  - no lost acks
  - no premature timeouts

when sending at $R/2$, some packets are retransmissions but asymptotic goodput is still $R/2$ (why?)
Congestion Scenario 2c

- Packets may get dropped at router due to full buffers
- Sender times out prematurely, sending two copies, both of which are delivered

when sending at R/2, some packets are retransmissions including duplicated that are delivered!
Congestion Scenario 3

- Four senders
- Multihop paths
- Timeout/retransmit

Q: What happens as $\lambda_\text{in}'$ grows and exceeds $R/2$?
Congestion Control Options

- Two major approaches to congestion control
- End-to-end congestion control – used by TCP
  - no explicit feedback from network
  - congestion inferred from loss, delay observed by hosts
  - relies on cooperation of end hosts
- Network-assisted congestion control
  - routers provide feedback to end systems
    - more rapid response to traffic changes than end-to-end approach
  - simplest approach – single bit congestion indication
    - TCP and IP support this, but capability is usually disabled
  - explicit rate control
    - senders request a sending rate, routers decide allowable rates
    - can prevent “greedy hosts” from hogging network capacity
    - used in Asynchronous Transfer Mode (ATM) networks
Exercises

1. When TCP detects that a packet has been lost, it assumes that the loss was caused by congestion. Under what circumstances is this a reasonable assumption? In what common situation is it not reasonable? How does TCP's assumption affect performance when packets may be lost for other reasons?
TCP Congestion Control Big Picture

- Objective: send as fast as possible, but not too fast
  - when lost segments are detected, sender reduces its rate
  - when there are no lost segments, sender increases its rate

- Key questions
  - when it’s time to cut rate, by how much should it be cut?
    - TCP cuts sending rate in half in order to reduce congestion quickly
  - when it’s time to increase rate, by how much should it increase?
    - TCP makes small incremental increases to avoid going right back into congestion
    - but TCP allows a "new sender" to increase its rate more quickly

- Additive increase/multiplicative decrease + "slow-start"
  - during stable traffic periods, rates oscillate around ideal rates
  - different end-to-end flows get roughly “fair shares” of capacity
  - can be slow to respond to traffic changes
Basic AIMD Behavior

- Sawtooth pattern
  - sending rate oscillates around “ideal rate”
  - when ideal rate is large, the “cycle time” can also be large
  - implies slow response time

- Trends towards “fairness”
  - when several TCP connections share a common “bottleneck” link, it’s desirable that they each receive a roughly equal share
  - AIMD makes things approximately equal in the long run
    - connections tend to oscillate in sync with each other, so when total rate is too large for link, all halve their rates together
    - this has bigger impact on higher rate senders
    - caveat: connections with short RTTs get more capacity
Simulation for Large and Small Buffers

- Top graph: $B=20,000 \times \text{MSS}, N=100$, RTT=100ms, $R=500 \text{ Mb/s}$
  - with large buffers get large delay variance

- Bottom graph: $B=1,000 \times \text{MSS}, N=100$, RTT=100ms, $R=500 \text{ Mb/s}$
  - with small buffers get underflow and low throughput
Exercises

1. Suppose that $N$ TCP connections pass through a “bottleneck link” that has a rate of 1 Gb/s and a buffer capacity of 25 MB. Assume that for all connections, the RTT is 100 ms. Suppose that just before the buffer fills, the input rate is 1.2 Gb/s. Assuming that this causes all of the TCP senders to halve their sending rate, how much will the buffer level drop during the next 2 RTTs?
TCP Congestion Control Variants

- A series of congestion control algorithms have been developed and used for TCP
  - the differences affect only the sender-side of a TCP connection, so hosts running different versions of TCP can still communicate
- TCP Tahoe
  - the original approach developed in the late 1980s
  - basic AIMD + slow-start strategy
- TCP Reno and New Reno
  - New Reno is now most widely deployed approach
  - added a transient “fast recovery“ operating mode to TCP
- BIC and CUBIC
  - provides faster congestion response in high speed networks
  - CUBIC is now the default choice in Linux
TCP Tahoe Overview

- TCP sender has two primary operating “states”
  - congestion avoidance
    - increase sending rate in small increments
  - slow start
    - allows more rapid increase in rates for new senders
    - also entered after a packet loss is detected

- Sender maintains two variables to control congestion
  - the \textit{congestion window} variable (\textit{cwnd}) limits number of unacknowledged bytes
  - the \textit{slow start threshold} (\textit{ssthresh}) controls when sender leaves the slow-start state
  - variables are updated in response to lost packets and reception of ACKs
TCP Tahoe Details

- **Updating cwnd**
  - in slow start, cwnd is effectively doubled each RTT (if no loss)
  - in congestion avoidance, cwnd grows by about 1 MSS per RTT

- **After transition from congestion avoidance to slow start**
  - it takes about 1 RTT for a new ACK to arrive and nothing much happens during this period
  - after ACK arrives, # of unACK-ed bytes becomes 0, sender can resume sending, and cwnd grows as ACKs arrive
Understanding Slow Start

- A “new” source starts with a small window, but is allowed to increase it quickly
  - initially cwnd = 1 MSS
  - cwnd is effectively doubled for every RTT with no packet loss
  - after first packet loss, sender halves cwnd and reverts to additive increase

- “Slow-start” something of a misnomer, since allows fast increase in rate
Exercises

1. Suppose that a TCP Tahoe connection in the congestion avoidance state has a $cwnd$ value of 50 KB, an MSS of 1 KB and an RTT of 100 ms. Suppose that at this point, it detects a lost packet. How does this change the value of $cwnd$ and $ssthresh$? Approximately how much time passes before the sender goes back into the congestion avoidance state? Assuming that no more packets are lost until $cwnd$ exceeds 50 KB again, approximately how much time is spent in the congestion avoidance state? For this connection, does slow-start have a big impact on the throughput achieved?
TCP Reno

- Two ways to detect packet loss
  - timeout or three duplicate ACKs

- TCP Reno treats these cases differently
  - because timeout is an indication of more severe congestion
    - typically, many packets must be lost to trigger timeout
  - on timeout, Reno behaves like Tahoe (goes to slow start)
  - on triple-dup-ACK, it goes to a new fast recovery state
    - sets $ssthresh = \frac{cwnd}{2}$ and $cwnd = ssthresh + 3*MSS$

- Fast recovery is a special transient state that is typically active during the first RTT after a triple-dup-ACK
  - when the lost packet is ACK-ed (after about one RTT), sender goes back to congestion avoidance with $cwnd = ssthresh$
  - before then, each dup-ACK increases cwnd by MSS
    - this allows sender to send ($cwnd/2$ bytes during this RTT)
Putting it All Together

**slow start**
- duplicate ACK
- cwnd = cwnd + MSS
- dupACKcount = 0
- transmit new segment(s), as allowed
- cwnd = ssthresh
- do nothing
- ssthresh = ssthresh = cwnd/2
- cwnd = ssthresh = 1 MSS
- dupACKcount = 0
- retransmit missing segment
- dupACKcount = 3
- ssthresh = cwnd/2
- cwnd = ssthresh = 3 MSS
- retransmit missing segment
- cwnd = cwnd + MSS
- transmit new segment(s), as allowed

**congestion avoidance**
- new ACK
- cwnd = cwnd + MSS (cwnd/MSS)
- dupACKcount = 0
- transmit new segment(s), as allowed
- cwnd = ssthresh
- do nothing
- ssthresh = cwnd/2
- cwnd = ssthresh = 1 MSS
- dupACKcount = 0
- retransmit missing segment
- dupACKcount = 3
- ssthresh = cwnd/2
- cwnd = ssthresh = 3 MSS
- retransmit missing segment
- cwnd = cwnd + MSS
- transmit new segment(s), as allowed

**fast recovery**
- duplicate ACK
- cwnd = cwnd + MSS
- transmit new segment(s), as allowed
Understanding TCP Performance

- TCP seeks to keep the link busy while limiting congestion
  - if link queue is large enough, per host throughput $T=R/N$
  - for small queues, $T \approx 0.75 \frac{R}{N}$

- The “cycle time” of TCP’s control algorithm is approximately $(1 + \frac{R}{1.5 \cdot N} \cdot \frac{RTT}{8 \cdot MSS}) \cdot RTT$
  - where $R$ is the link rate and $N$ is the number of flows
  - note, the cycle time scales up with link rate
    - so, as links get faster, TCP reacts more slowly to changes in traffic
    - example: $R=1\text{Gb/s}$, $N=10$, $RTT=0.1s$, $MSS=10^4$, cycle time=8.5s
  - also note that 1 packet is lost per cycle and number sent per cycle is $(cycle\ time) \cdot (R/N \cdot 8MSS) = (cycle\ time) \cdot T/(8\cdot MSS)$
    - so losses occur less often as cycle time increases
TCP Throughput Approximation

- The throughput of a TCP connection can be approximated by
  \[ T \approx \frac{1.22 \cdot MSS}{RTT \sqrt{L}} \]
  or equivalently
  \[ L \approx \left( \frac{1.22 \cdot MSS}{T \cdot RTT} \right)^2 \]

  > where \( L \) is the fraction of packets that are lost in transit

- If packet losses only due to TCP-induced buffer overflow
  - can derive expression using fact that loss rate is
    \( 1/(\# \text{ of packets sent per cycle}) \)

- If only losses are due to bit errors
  - can derive expression using fact that TCP goes through one
    cycle every \( 1/L \) packets, halves its rate at start of each cycle
    plus fact that average \( \# \) of packets sent per \( RTT \) is \( RTT \cdot (T/MSS) \)
Fairness in the Internet

- TCP attempts to share available bandwidth “fairly”
  - operates at the level of TCP connections or “flows”, not at the level of application sessions or users
- But easy for “greedy” applications/users to get an “unfair share”
  - use multiple TCP connections for a given application session
    - web servers commonly do this
  - use UDP, which has no congestion control
    - many multimedia applications do this
- No clear solution
  - host-based mechanisms must rely on well-behaved users
  - internet lacks mechanisms for enforcement of fair usage
  - potential solutions involve usage-based charging which is unpopular
Exercises

1. Suppose that a TCP Reno connection in the congestion avoidance state has a \textit{cwnd} value of 50 KB, an MSS of 1 KB and an RTT of 100 ms. Suppose that at this point, it detects a lost packet (by duplicate \textit{ack}). How does this change the value of \textit{cwnd} and \textit{ssthresh}? Approximately how much time passes before the sender goes back into the congestion avoidance state? Assuming that no more packets are lost until \textit{cwnd} exceeds 50 KB again, approximately how much time is spent in the congestion avoidance state?

2. Consider a TCP Reno connection that is achieving a throughput of 40 Mb/s. Assume that the MSS is 1 KB and the RTT is 100 ms. Estimate the loss rate for this connection.

3. Consider a TCP Reno connection that is experiencing a packet loss rate of 4%. Assume that the MSS is 1 KB and the RTT is 100 ms. Estimate the throughput of this connection.