Congestion Control

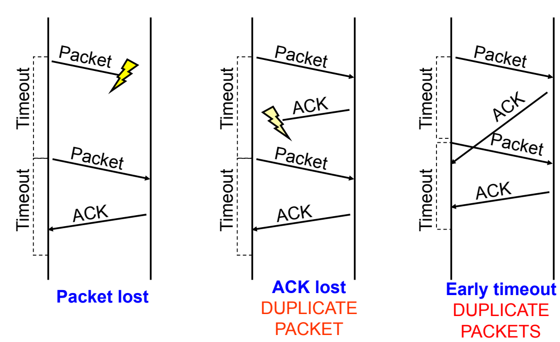
Michael Freedman
COS 461: Computer Networks

http://www.cs.princeton.edu/courses/archive/spr14/cos461/

(First, remainder of slides from Monday's lecture on Transport Layer)

Optimizing Retransmissions

Reasons for Retransmission



How Long Should Sender Wait?

- Sender sets a timeout to wait for an ACK
  - Too short: wasted retransmissions
  - Too long: excessive delays when packet lost
- TCP sets timeout as a function of the RTT
  - Expect ACK to arrive after an "round-trip time"
  - … plus a fudge factor to account for queuing
- But, how does the sender know the RTT?
  - Running average of delay to receive an ACK

5

Still, timeouts are slow (≈RTT)

- When packet n is lost…
  - … packets n+1, n+2, and so on may get through
- Exploit the ACKs of these packets
  - ACK says receiver is still awaiting nth packet
  - Duplicate ACKs suggest later packets arrived
  - Sender uses "duplicate ACKs" as a hint
- Fast retransmission
  - Retransmit after "triple duplicate ACK"

6

When is Fast Retransmit effective?

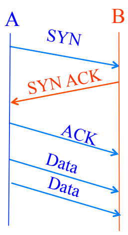
- High likelihood of many packets in flight
- Long data transfers, large window size, …
- Implications for Web traffic
  - Most Web transfers are short (e.g., 10 packets)
    - So, often there aren't many packets in flight
  - Making fast retransmit is less likely to "kick in"
    - Forcing users to click "reload" more often…

7

Starting and Ending a Connection: TCP Handshakes

8

Establishing a TCP Connection

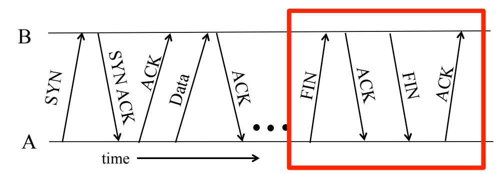


Each host tells its ISN to the other host.

- Three-way handshake to establish connection
  - Host A sends a **SYN** (open) to the host B
  - Host B returns a SYN acknowledgment (**SYN ACK**)
  - Host A sends an **ACK** to acknowledge the SYN ACK

9

Tearing Down the Connection



- Closing (each end of) the connection
  - Finish (FIN) to close and receive remaining bytes
  - And other host sends a FIN ACK to acknowledge
  - Reset (RST) to close and not receive remaining bytes

10

Sending/Receiving the FIN Packet

- Sending a FIN: close()
  - Process is done sending data via socket
  - Process invokes "close()"
  - Once TCP has sent all the outstanding bytes…
  - … then TCP sends a FIN
- Receiving a FIN: EOF
  - Process is reading data from socket
  - Eventually, read call returns an EOF

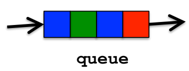
11

Congestion Control

Distributed Resource Sharing

Congestion

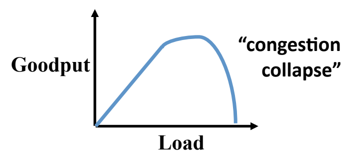
- Best-effort network does not "block" calls
  - So, they can easily become overloaded
  - Congestion == "Load higher than capacity"
- Examples of congestion
  - Link layer: Ethernet frame collisions
  - Network layer: full IP packet buffers



- Excess packets are simply dropped
  - And the sender can simply retransmit

Congestion Collapse

- Easily leads to *congestion collapse*
  - Senders retransmit the lost packets
  - Leading to even *greater* load
  - … and even *more* packet loss



Increase in load that results in a *decrease* in useful work done.

Detect and Respond to Congestion

- What does the end host see?
- What can the end host change?

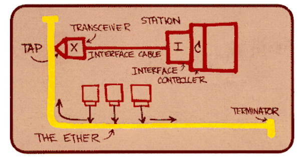
Detecting Congestion

- Link layer
  - Carrier sense multiple access
  - Seeing your own frame collide with others
- Network layer
  - Observing end-to-end performance
  - Packet delay or loss over the path

Responding to Congestion

- Upon detecting congestion
  - Decrease the sending rate
- But, what if conditions change?
  - If more bandwidth becomes available,
  - … unfortunate to keep sending at a low rate
- Upon *not* detecting congestion
  - Increase sending rate, a little at a time
  - See if packets get through

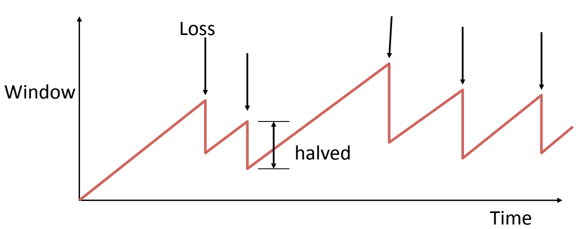
Ethernet Back-off Mechanism



- **Carrier sense:**
  - Wait for link to be idle
  - If idle, start sending
  - If not, wait until idle
- **Collision detection:** listen while transmitting
  - If collision: abort transmission, and send jam signal
- **Exponential back-off:** wait before retransmitting
  - Wait random time, exponentially larger per retry

TCP Congestion Control

- Additive increase, multiplicative decrease
  - On packet loss, divide congestion window in half
  - On success for last window, increase window linearly



Why Exponential?

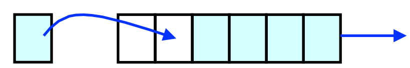
- Respond aggressively to bad news
  - Congestion is (very) bad for everyone
  - Need to react aggressively
- Examples:
  - Ethernet: *double* retransmission timer
  - TCP: divide sending rate in *half*
- Nice theoretical properties
  - Makes efficient use of network resources

TCP Congestion Control

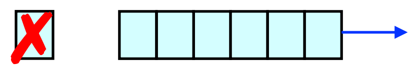
21

Congestion in a Drop-Tail FIFO Queue

- Access to the bandwidth: first-in first-out queue
  - Packets transmitted in the order they arrive



- Access to the buffer space: drop-tail queuing
  - If the queue is full, drop the incoming packet

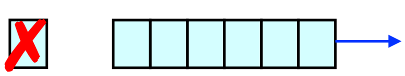


22

How it Looks to the End Host

- Delay: Packet experiences high delay
- Loss: Packet gets dropped along path

- How does TCP sender learn this?
  - Delay: Round-trip time estimate
  - Loss: Timeout and/or duplicate acknowledgments



23

TCP Congestion Window

- Each TCP sender maintains a congestion window
  - Max number of bytes to have in transit (not yet ACK'd)

- Adapting the congestion window
  - Decrease upon losing a packet: backing off
  - Increase upon success: optimistically exploring
  - Always struggling to find right transfer rate

- Tradeoff
  - Pro: avoids needing explicit network feedback
  - Con: continually under- and over-shoots "right" rate

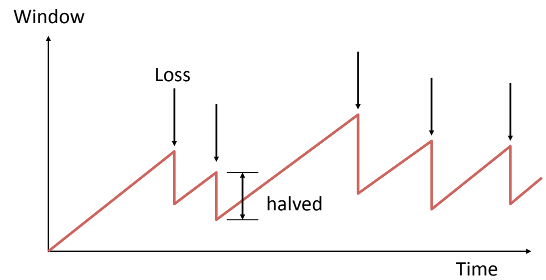
24

Additive Increase, Multiplicative Decrease

- How much to adapt?
  - Additive increase: On success of last window of data, increase window by 1 Max Segment Size (MSS)
  - Multiplicative decrease: On loss of packet, divide congestion window in half
- Much quicker to slow down than speed up!
  - Over-sized windows (causing loss) are much worse than under-sized windows (causing lower thruput)
  - AIMD: A necessary condition for stability of TCP

25

Leads to the TCP "Sawtooth"



26

Receiver Window vs. Congestion Window

- Flow control
  - Keep a *fast sender* from overwhelming *a slow receiver*
- Congestion control
  - Keep a *set of senders* from overloading the *network*
- Different concepts, but similar mechanisms
  - TCP flow control: receiver window
  - TCP congestion control: congestion window
  - Sender TCP window =

    min { congestion window, receiver window }

27

Sources of poor TCP performance

- The below conditions *may* primarily result in:

  (A) Higher pkt latency (B) Greater loss (C) Lower thruput

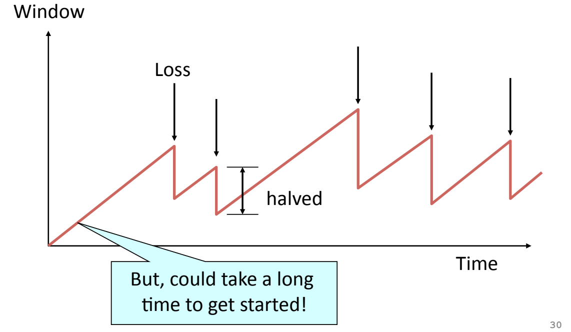
1. Larger buffers in routers
2. Smaller buffers in routers
3. Smaller buffers on end-hosts
4. Slow application receivers

28

Starting a New Flow

How Should a New Flow Start?

Start slow (a small CWND) to avoid overloading network

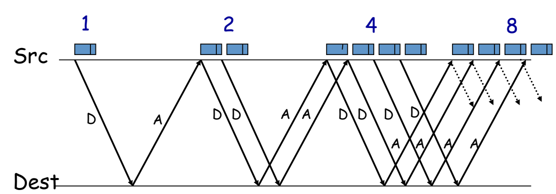


"Slow Start" Phase

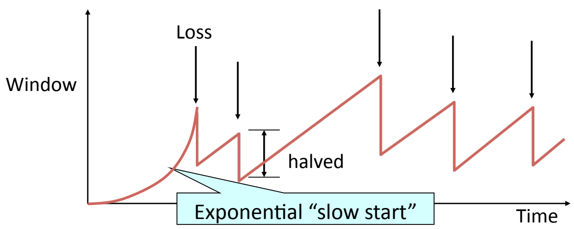
- Start with a small congestion window
  - Initially, CWND is 1 MSS
  - So, initial sending rate is MSS / RTT
- Could be pretty wasteful
  - Might be much less than actual bandwidth
  - Linear increase takes a long time to accelerate
- Slow-start phase (really "fast start")
  - Sender starts at a slow rate (hence the name)
  - … but increases rate exponentially until the first loss

Slow Start in Action

Double CWND per round-trip time



Slow Start and the TCP Sawtooth



- TCP originally had *no* congestion control
  - Source would start by sending entire receiver window
  - Led to congestion collapse!
  - "Slow start" is, comparatively, slower

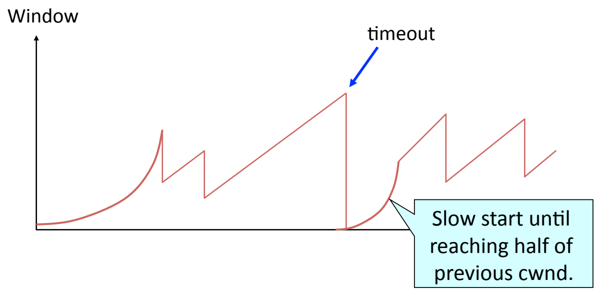
33

Two Kinds of Loss in TCP

- Timeout vs. Triple Duplicate ACK
  - Which suggests network is in worse shape?
- Timeout
  - If entire window was lost, buffers may be full
  - ...blasting entire CWND would cause another burst
  - ...be aggressive: start over with a low CWND
- Triple duplicate ACK
  - Might be do to bit errors, or "micro" congestion
  - ...react less aggressively (halve CWND)

34

Repeating Slow Start After Timeout



Slow-start restart: Go back to CWND of 1, but take advantage of knowing the previous value of CWND.

35

Repeating Slow Start After Idle Period

- Suppose a TCP connection goes idle for a while
- Eventually, the network conditions change
  - Maybe many more flows are traversing the link
- Dangerous to start transmitting at the old rate
  - Previously-idle TCP sender might blast network
  - ... causing excessive congestion and packet loss
- So, some TCP implementations repeat slow start
  - Slow-start restart after an idle period

36

TCP Problem

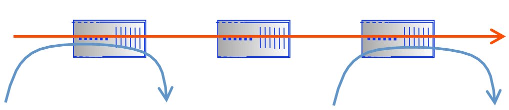
- 1 MSS = 1KB
- Max capacity of link: 200 KBps
- RTT = 100ms
- New TCP flow starting, no other traffic in network, assume no queues in network

1. About what is cwnd at time of first packet loss?
   (A) 16 pkts (B) 32 KB (C) 100 KB (D) 200 KB
2. About how long until sender discovers first loss?
   (A) 400 ms (B) 600 ms (C) 1s (D) 1.6s

Fairness

TCP Achieves a Notion of Fairness

- Effective utilization is not only goal
  – We also want to be *fair* to various flows
- Simple definition: equal bandwidth shares
  – N flows that each get 1/N of the bandwidth?
- But, what if flows traverse different paths?
  – Result: bandwidth shared in proportion to RTT



What About Cheating?

- Some folks are more fair than others
  – Using multiple TCP connections in parallel (BitTorrent)
  – Modifying the TCP implementation in the OS
    - Some cloud services start TCP at > 1 MSS
  – Use the User Datagram Protocol
- What is the impact
  – Good guys slow down to make room for you
  – You get an unfair share of the bandwidth

Preventing Cheating

- Possible solutions?
  - Routers detect cheating and drop excess packets?
  - Per user/customer failness?
  - Peer pressure?

41

Conclusions

- Congestion is inevitable
  - Internet does not reserve resources in advance
  - TCP actively tries to push the envelope
- Congestion can be handled
  - Additive increase, multiplicative decrease
  - Slow start and slow-start restart
- Fundamental tensions
  - Feedback from the network?
  - Enforcement of "TCP friendly" behavior?

42