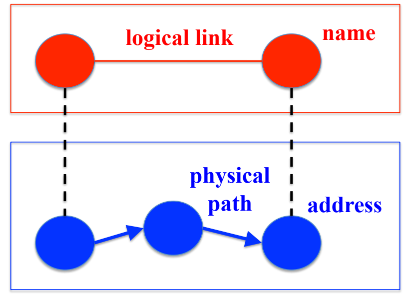
Congestion

Michael Freedman
COS 461: Computer Networks
Lectures: MW 10-10:50am in Architecture N101

http://www.cs.princeton.edu/courses/archive/spr13/cos461/

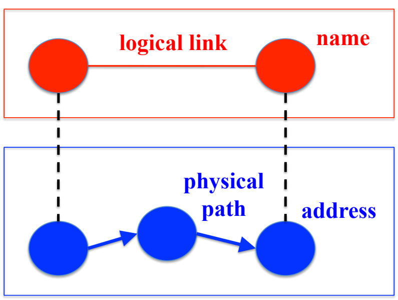
Last Week: Discovery and Routing

Provides end-to-end *connectivity*, but not necessarily good *performance*



Today: Congestion Control

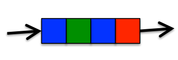
What can the *end-points* do to collectively to make good use of shared underlying resources?



Distributed Resource Sharing

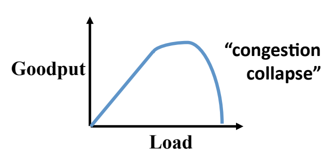
Congestion

- 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?

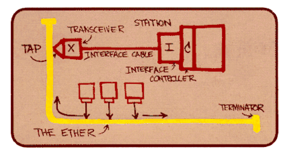
Detecting Congestion

- 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

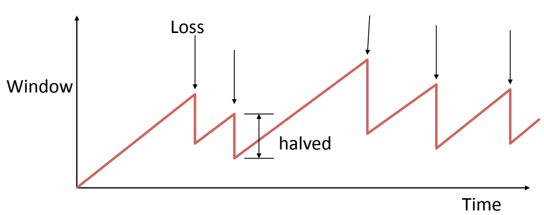
Ethernet Back-off Mechanism



- **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



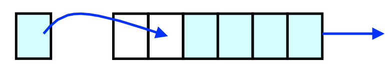
Why Exponential?

- 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

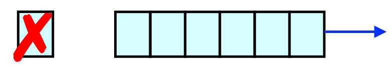
TCP Congestion Control

Congestion in a Drop-Tail FIFO Queue

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



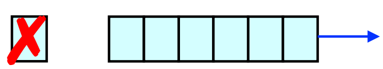
- Access to the buffer space: drop-tail queuing
  - If the queue is full, drop the incoming packet



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



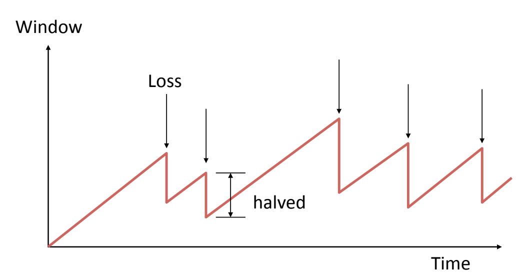
TCP Congestion Window

- 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

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 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

Leads to the TCP "Sawtooth"



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 }

19

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

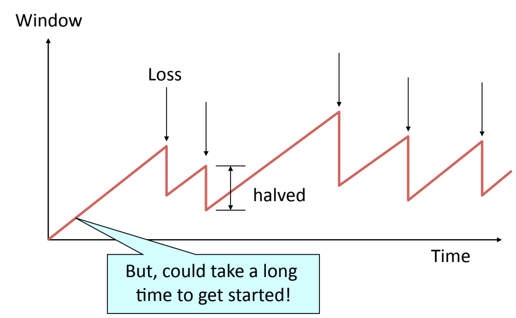
20

Starting a New Flow

21

How Should a New Flow Start?

Start slow (a small CWND) to avoid overloading network



22

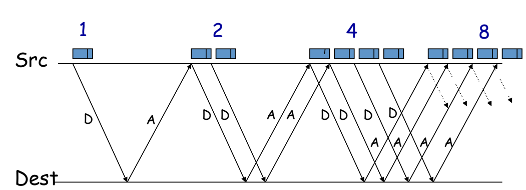
"Slow Start" Phase

- 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

23

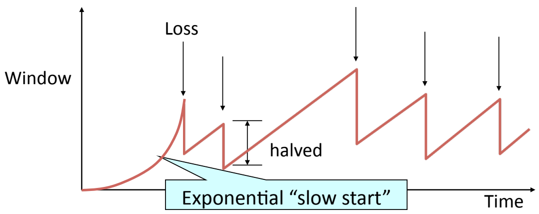
Slow Start in Action

Double CWND per round-trip time



24

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

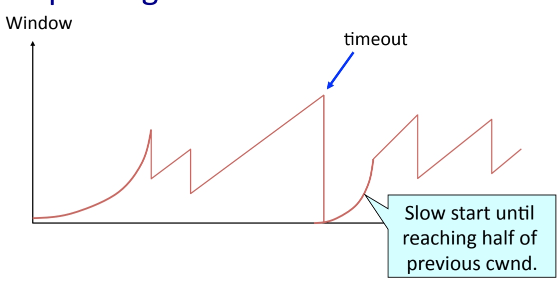
25

Two Kinds of Loss in TCP

- Timeout
  - Packet n is lost and detected via a timeout
  - Blasting entire CWND would cause another burst
  - Better to start over with a low CWND
- Triple duplicate ACK
  - Packet n is lost, but packets n+1, n+2, etc. arrive
  - Then, sender quickly resends packet n
  - Do a multiplicative decrease and keep going

26

Repeating Slow Start After Timeout



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

27

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

28

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) 8 pkts (B) 16 pkts (C) 32 KB (D) 100 KB (E) 200 KB
2. About how long until sender discovers first loss?
   (A) 200 ms (B) 400 ms (C) 600 ms (D) 1s (E) 1.6s

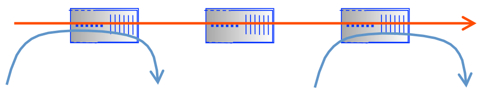
29

Fairness

30

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



31

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

32

Preventing Cheating

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

33

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?

34