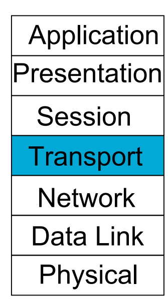
CSE/EE 461

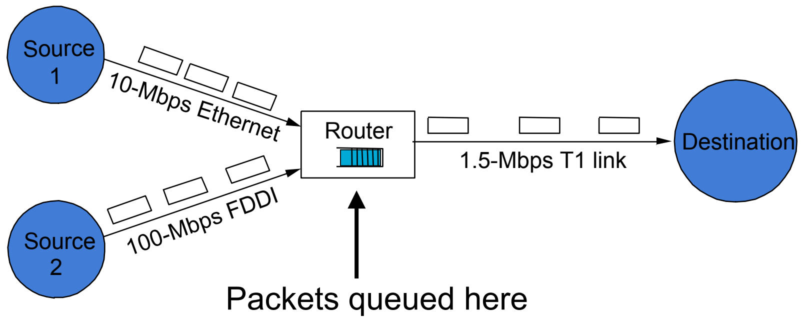
TCP and network congestion

This Lecture

- Focus
  - How should senders pace themselves to avoid stressing the network?
- Topics
  - congestion collapse
  - congestion control



Congestion from in the network



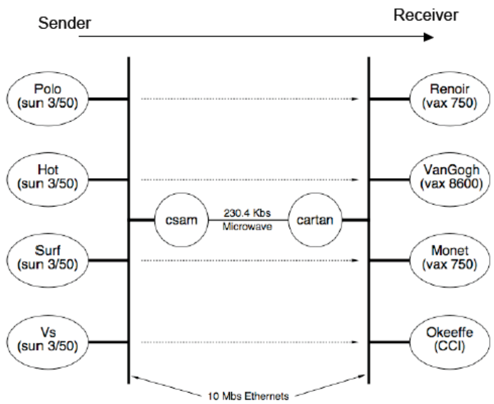
- Buffers at routers used to absorb bursts when input rate > output
- Loss (drops) occur when sending rate is persistently > drain rate

Congestion Collapse

- In the limit, premature retransmissions lead to <u>congestion collapse</u>
  - e.g., 1000x drop in effective bandwidth of network
  - sending more packets into the network when it is overloaded exacerbates the problem of congestion (overflow router queues)
  - network stays busy but very little useful work is being done

- This happened in real life ~1987
  - Led to Van Jacobson's TCP algorithms
    - these form the basis of congestion control in the Internet today
  - Researchers asked two questions:
    - Was TCP misbehaving?
    - Could TCP be "trained" to work better under 'absymal network conditions?'

A Scenario

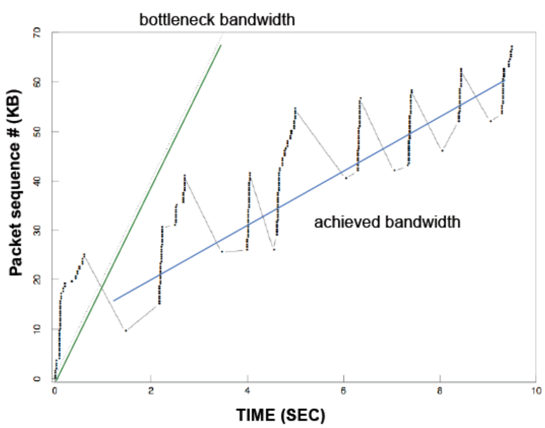


Receiver window size is 16KB.

Bottleneck router buffer size is 15 KB.

Data bandwidth is about 20KB/s

Effects of early retransmission



Slope is bandwidth.

Steep smooth upward slope == means good bandwidth.

Downward slope means retransmissions (*bad).*

If only…

- We knew RTT and Current Router Queue Size,
  - then we would send:

    MIN(Router Queue Size, Effective Window Size)

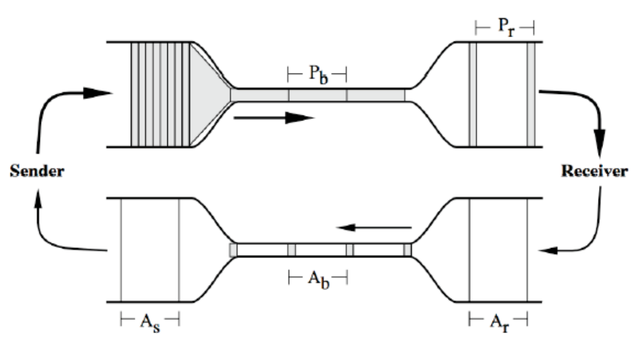
  - and not retransmit a packet until it had been sent RTT ago.

- But we don't know these things
  - so we have to estimate them

- They change over time because of other data sources
  - so we have to continually adapt them

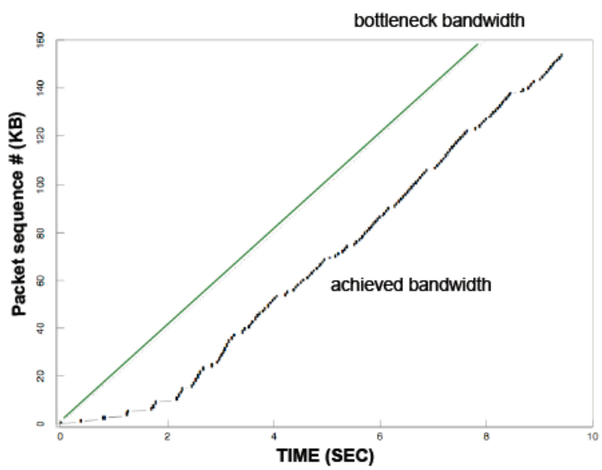
Ideal packet flow: stable equilibrium

Pr = Interpacket spacing --> mirrors that of slowest link



As = Inter-ACK spacing --> mirrors that of slowest downstream link

Modern TCP in previous scenario



Notice:

• no retransmissions, (and thus no packet loss)

• achieved BW = bottleneck BW

1988 Observations on Congestion Collapse

- Implementation, not the protocol, leads to collapse
  - choices about when to retransmit, when to "back off" because of losses
- "Obvious" ways of doing things lead to non-obvious and undesirable results
  - "send effective-window-size # packets, wait RTT, try again"
- Remedial algorithms achieve network stability by forcing the transport connection to obey a 'packet conservation' principle.
  - for connection in equilibrium (stable with full window in transit), packet flow is conservative
    - a new packet not put in network until an old packet leaves

Resulting TCP/IP Improvements

- *Slow-start*
- *Round-trip time variance estimation*
- *Exponential retransmit timer backoff*
- *More aggressive receiver ack policy*
- *Dynamic window sizing on congestion*
- Clamped retransmit backoff (Karn)
- Fast Retransmit

Packet Conservation Principle

Congestion control means: "Finding places that violate the conservation of packets principle and then fixing them."

Key ideas

- Routers queue packets
  - if queue overflows, packet loss occurs
  - happens when network is "congested"
- Retransmissions deal with loss
  - need to retransmit sensibly
    - too early: needless retransmission
    - too late: lost bandwidth
- Senders must control their transmission pace
  - flow control: send no more than receiver can handle
  - congestion control: send no more than network can handle

Basic rules of TCP congestion control

1. The connection must reach equilibrium.
   - hurry up and stabilize!
   - when things get wobbly, put on the brakes and reconsider
2. Sender must not inject a new packet before an old packet has left
   - a packet leaves when the receiver picks it up,
   - or if it gets lost.
     - damaged in transit or dropped at congested point
     - (far fewer than 1% of packets get damaged in practice)
   - ACK or packet timeout signals that a packet has "exited."
     - ACK are easy to detect.
     - appropriate timeouts are harder…. all about estimating RTT.
3. Equilibrium is lost because of resource contention along the way.
   - new competing stream appears, must restabilize

1. The connection must reach equilibrium.

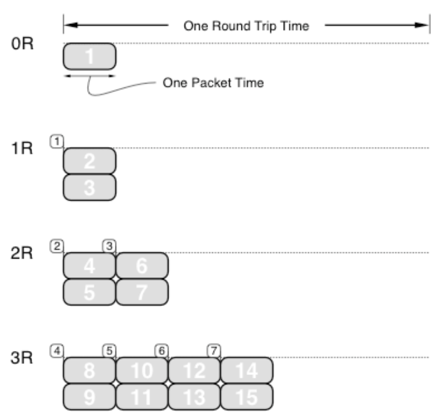
1. Getting to Equilibrium -- Slow Start

- Goal
  - Quickly determine the appropriate window size
    - Basically, we're trying to sense the bottleneck bandwidth
- Strategy
  - Introduce *congestion_window (cwnd)*
  - When starting off, set cwnd to 1
  - For each ACK received, add 1 to cwnd
  - When sending, send the minimum of receiver's advertised window and cwnd
- Guaranteed to not transmit at more than twice the max BW, and for no more than RTT.
  - (bw delay product)

Cwnd doubles every RTT;

Opening a window of size W takes time (RTT)$\log_2$W.

Figure 2: The Chronology of a Slow-start



The horizontal direction is time. The continuous time line has been chopped into one-round-trip-time pieces stacked vertically with increasing time going down the page. The grey, numbered boxes are packets. The white numbered boxes are the corresponding acks. As each ack arrives, two packets are generated: one for the ack (the ack says a packet has left the system so a new packet is added to take its place) and one because an ack opens the congestion window by one packet. It may be clear from the figure why an add-one-packet-to-window policy opens the window exponentially in time.

Slow Start

- Note that the effect is to double transmission rate every RTT
  - This is ‘slow’?
- Basically an effective way to probe for the bottleneck bandwidth, using packet losses as the feedback
  - No change in protocol/header was required to implement
- When do you need to do this kind of probing?

2. A sender must not inject a new packet before an old packet has exited.

2. Packet Injection. Estimating RTTs

- Do not inject a new packet until an old packet has left.
  - 1. ACK tells us that an old packet has left.
  - 2. Timeout expiration tells us as well.
    - *We must estimate RTT properly.*
- Strategy 1: pick some constant RTT.
  - simple, but probably wrong. (certainly not adaptive)
- Strategy 2: Estimate based on past behavior.

Tactic 0: Mean
Tactic 1: Mean with something??

Original TCP (RFC793) retransmission timeout algorithm

- Use EWMA to estimate RTT:

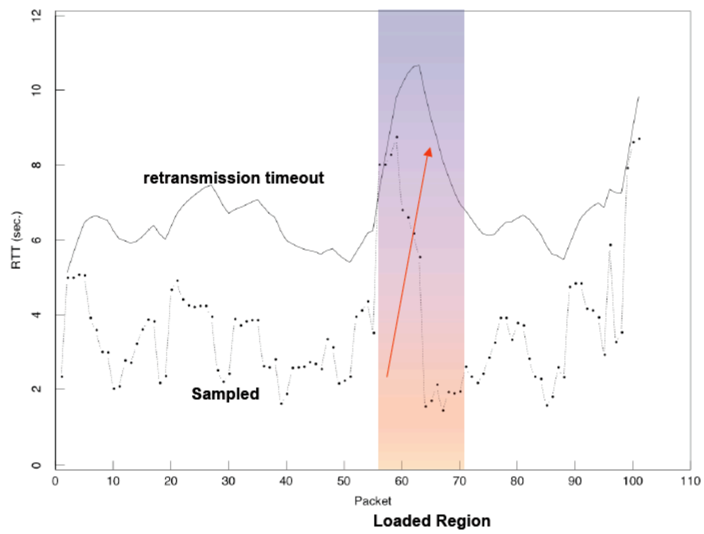
$$EstimatedRTT = (1-g)(EstimatedRTT) + g(SampleRTT)$$

$$0 \leq g \leq 1, \text{ usually } g = .1 \text{ or } .2$$

- Conservatively set timeout to small multiple (2x) of the estimate

$$Retransmission\ Timeout = 2 \times EstimatedRTT$$

Figure 5: Performance of an RFC793 retransmit timer



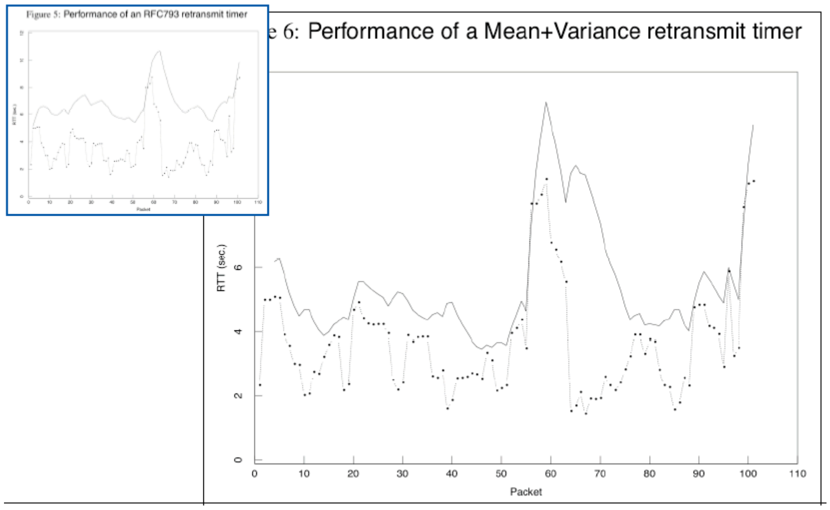
Jacobson/Karels Algorithm

1. DevRTT = (1-b) * DevRTT + b * |SampledRTT - EstimatedRTT|
   - typically, b = .25

2. Retransmission timeout = EstimatedRTT + k * DevRTT
   - k is generally set to 4

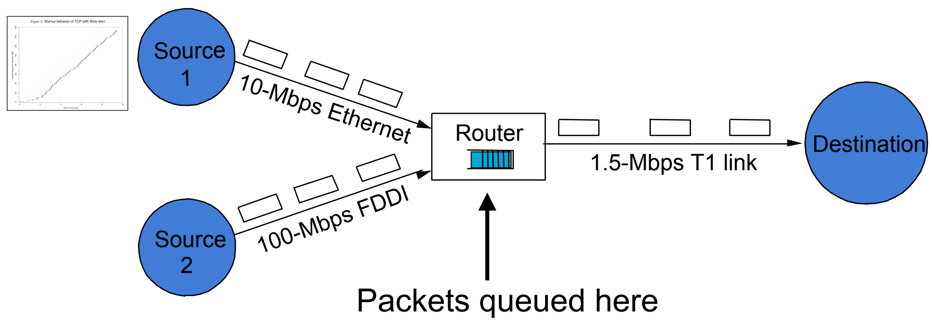
   - timeout =~ EstimatedRTT when variance is low (estimate is good)

Estimate with Mean + Variance



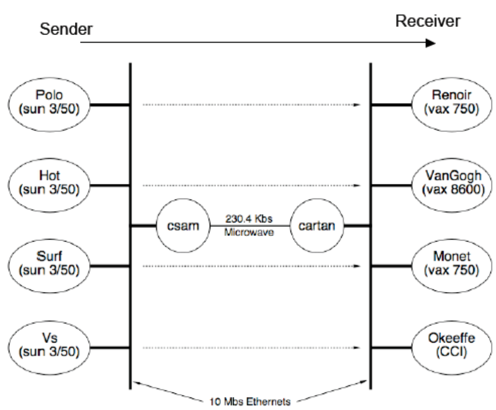
3. Equilibrium is lost because of resource contention along the way.

Congestion from Multiple Sources



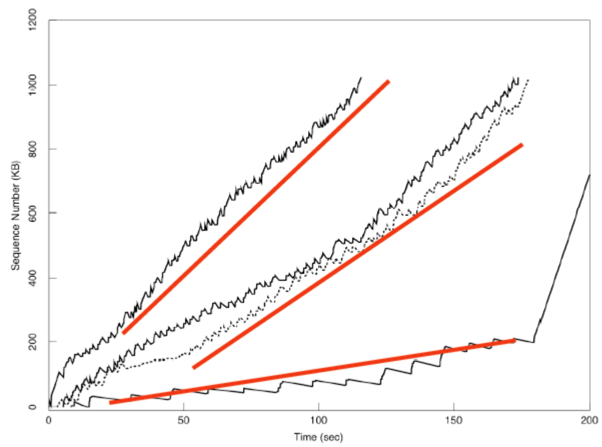
Packets Lost Here

In Real Life

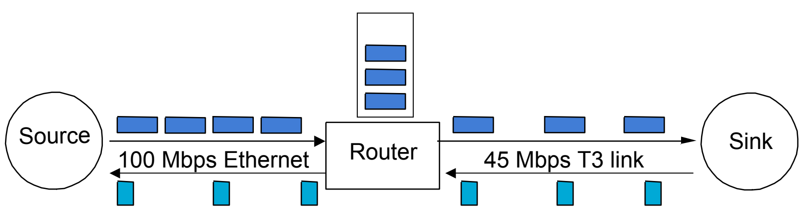


Four Simultaneous Streams

Figure 8: Multiple, simultaneous TCPs with no congestion avoidance



TCP is "Self-Clocking"



- ACKs pace transmissions at approximately the botteneck rate
  - So just by sending packets we can discern the "right" sending rate (called the packet-pair technique)

Congestion Control Relies on Signals from the Network

- The network is not saturated: *Send even more*
- The network is saturated: *Send less*

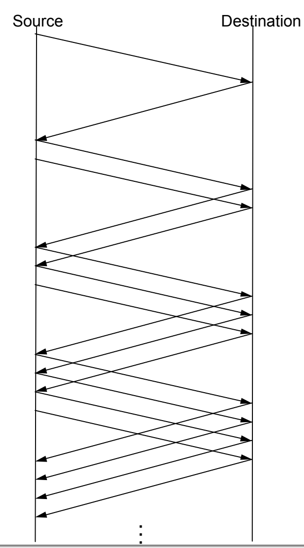
- *ACK* signals that the network is not saturated.
- A lost packet (no ACK) signals that the network is saturated
- Leads to a simple strategy:
  - On each ack, increase *congestion window* (***additive increase)***
  - On each lost packet, decrease congestion window (***multiplicative decrease)***
- Why increase slowly and decrease quickly?
  - *Respond to good news conservatively, but bad news aggressively*

AIMD (Additive Increase/Multiplicative Decrease)

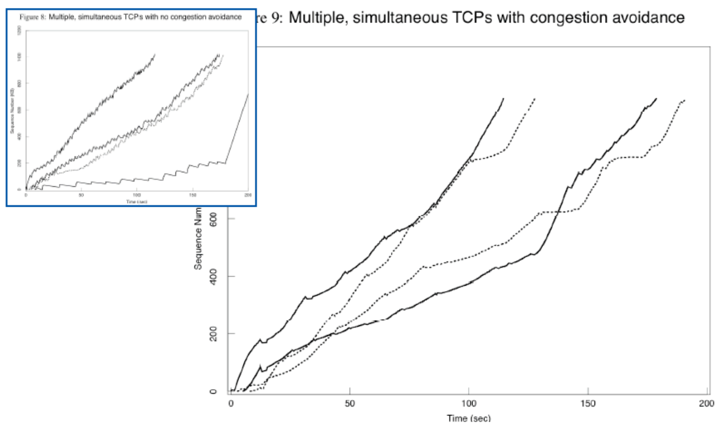
- How to adjust probe rate?

- Increase slowly while we believe there is bandwidth
  - Additive increase per RTT
  - Cwnd += 1 packet / RTT

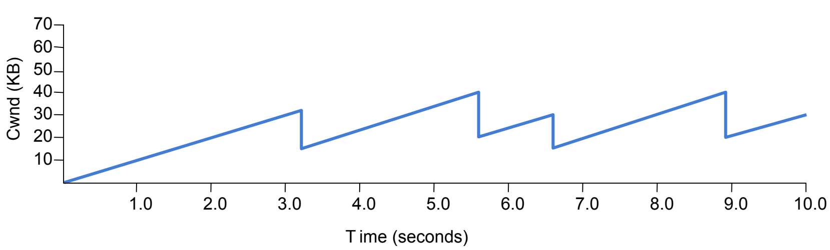
- Decrease quickly when there is loss (went too far!)
  - Multiplicative decrease
  - Cwnd /= 2



With Additive Increase/Multiplicative Decrease

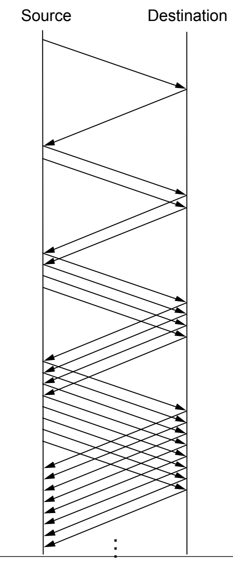


TCP Sawtooth Pattern

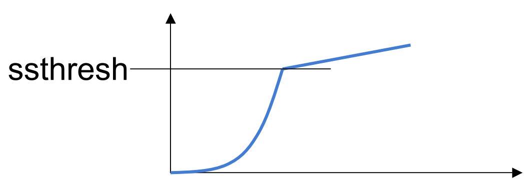


Comparing to "Slow Start"

- Q: What is the ideal value of cwnd? How long will AIMD take to get there?
- Use a different strategy to get close to ideal value
  - Slow start:
    - Double cwnd every RTT
      - cwnd *= 2 per RTT
      - i.e., cwnd += 1 per ACK
  - AIMD:
    - add one to cwnd per RTT
      - cwnd +=1 per RTT
      - i.e., cwnd += (1/cwnd) per ACK

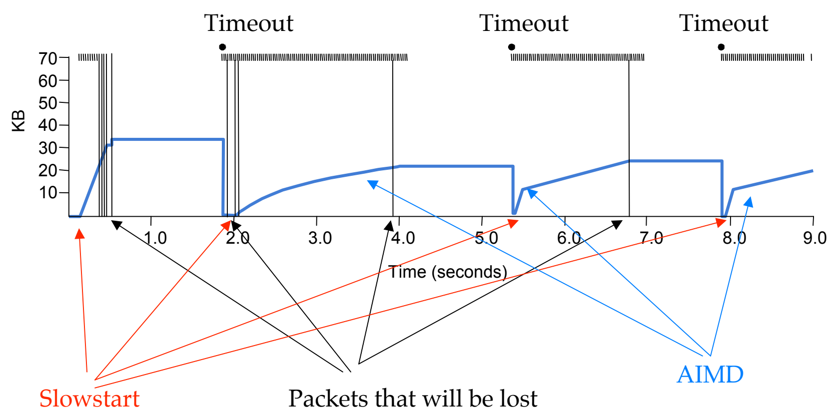


Combining Slow Start and AIMD



- Slow start is used whenever the connection is not running with packets outstanding
  - initially, and after timeouts indicating that there's no data on the wire
- But we don't want to overshoot our ideal cwnd on next slow start, so remember the last cwnd that worked with no loss
  - ssthresh = cwnd after cwnd /= 2 on loss
  - switch to AIMD once cwnd passes ssthresh

Example (Slow Start +AIMD)

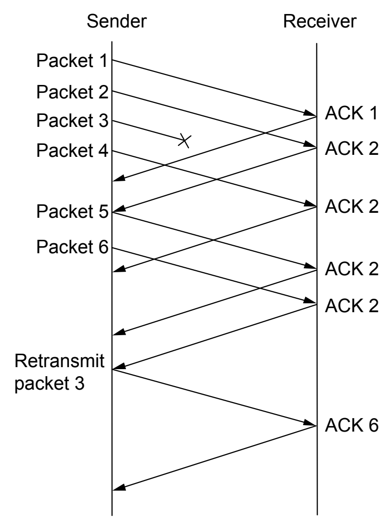


The Long Timeout Problem

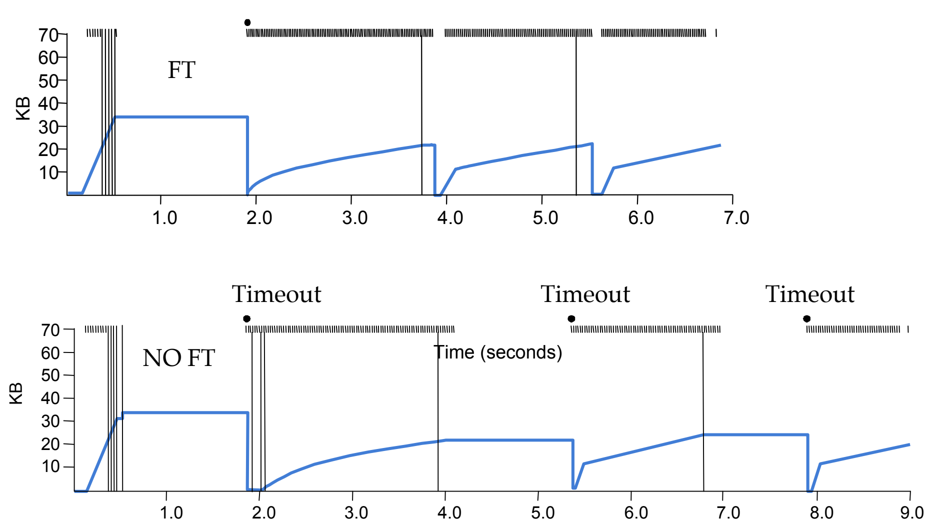
- Would like to "signal" a lost packet earlier than timeout
  - enable retransmit sooner
- Can we infer that a packet has been lost?
  - Receiver receives an "out of order packet"
  - Good indicator that the one(s) before have been misplaced
- Receiver generates a duplicate ack on receipt of a misordered packet
- Sender interprets sequence of duplicate acks as a signal that the as-yet-unacked packet has not arrived

Fast Retransmit

- TCP uses cumulative acks, so duplicate acks start arriving after a packet is lost.
- We can use this fact to infer which packet was lost, instead of waiting for a timeout.
- 3 duplicate acks are used in practice

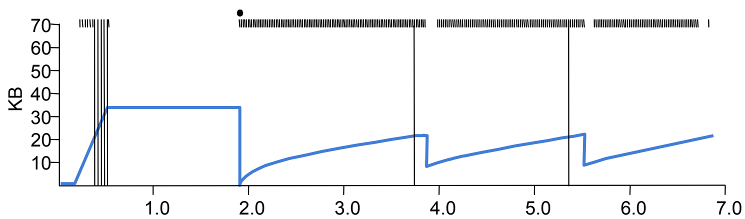


Example (with Fast Retransmit)



Fast Recovery

- After Fast Retransmit, use further duplicate acks to grow cwnd and clock out new packets, since these acks represent packets that have left the network.
- End result: Can achieve AIMD when there are single packet losses. Only slow start the first time and on a real timeout.



Key Concepts

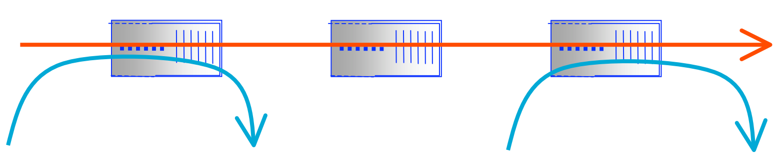
- Packet conservation is a fundamental concept in TCP's congestion management
  - Get to equilibrium
    - *Slow Start*
  - Do nothing to get out of equilibrium
    - *Good RTT Estimate*
  - Adapt when equilibrium has been lost due to other's attempts to get to/stay in equilibrium
    - *Additive Increase/Multiplicative Decrease*
- The network reveals its own behavior

Repeating Slow Start After Idle Period

- Suppose a TCP connection goes idle for a while
  - E.g., Telnet session where you don't type for an hour
- Eventually, the network conditions change
  - Maybe many more flows are traversing the link
  - E.g., maybe everybody has come back from lunch!
- Dangerous to start transmitting at the old rate
  - Previously-idle TCP sender might blast the network
  - … causing excessive congestion and packet loss
- So, some TCP implementations repeat slow start
  - Slow-start restart after an idle period

TCP Achieves Some Notion of Fairness

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



What About Cheating?

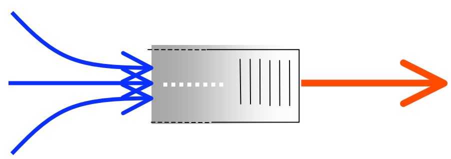
- Some folks are more fair than others
  - Running multiple TCP connections in parallel
  - Modifying the TCP implementation in the OS
  - Use the User Datagram Protocol
- What is the impact
  - Good guys slow down to make room for you
  - You get an unfair share of the bandwidth
- Possible solutions?
  - Routers detect cheating and drop excess packets?
  - Peer pressure?
  - ???

Queuing Mechanisms

Random Early Detection (RED)
Explicit Congestion Notification (ECN)

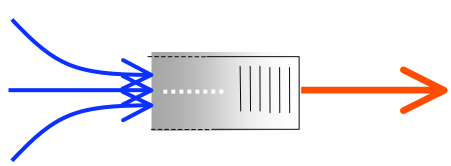
Bursty Loss From Drop-Tail Queuing

- TCP depends on packet loss
  - Packet loss is the indication of congestion
  - In fact, TCP *drives* the network into packet loss
  - … by continuing to increase the sending rate
- Drop-tail queuing leads to *bursty* loss
  - When a link becomes congested…
  - … many arriving packets encounter a full queue
  - And, as a result, many flows divide sending rate in half
  - … and, many individual flows lose multiple packets



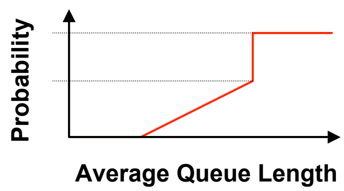
Slow Feedback from Drop Tail

- Feedback comes when buffer is completely full
  - … even though the buffer has been filling for a while
- Plus, the filling buffer is increasing RTT
  - … and the variance in the RTT
- Might be better to give early feedback
  - Get one or two connections to slow down, not all of them
  - Get these connections to slow down before it is too late



Random Early Detection (RED)

- Basic idea of RED
  - Router notices that the queue is getting backlogged
  - … and randomly drops packets to signal congestion
- Packet drop probability
  - Drop probability increases as queue length increases
  - If buffer is below some level, don't drop anything
  - … otherwise, set drop probability as function of queue



Properties of RED

- Drops packets before queue is full
  - In the hope of reducing the rates of some flows
- Drops packet in proportion to each flow's rate
  - High-rate flows have more packets
  - … and, hence, a higher chance of being selected
- Drops are spaced out in time
  - Which should help desynchronize the TCP senders
- Tolerant of burstiness in the traffic
  - By basing the decisions on *average* queue length

Problems With RED

- Hard to get the tunable parameters just right
  - How early to start dropping packets?
  - What slope for the increase in drop probability?
  - What time scale for averaging the queue length?
- Sometimes RED helps but sometimes not
  - If the parameters aren't set right, RED doesn't help
  - And it is hard to know how to set the parameters
- RED is implemented in practice
  - But, often not used due to the challenges of tuning right
- Many variations in the research community
  - With cute names like "Blue" and "FRED"… ☺

Explicit Congestion Notification

- Early dropping of packets
  - Good: gives early feedback
  - Bad: has to drop the packet to give the feedback
- Explicit Congestion Notification
  - Router marks the packet with an ECN bit
  - … and sending host interprets as a sign of congestion
- Surmounting the challenges
  - Must be supported by the end hosts and the routers
  - Requires two bits in the IP header (one for the ECN mark, and one to indicate the ECN capability)
  - Solution: borrow two of the Type-Of-Service bits in the IPv4 packet header

Other TCP Mechanisms

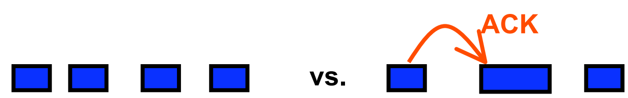
Nagle's Algorithm and Delayed ACK

Motivation for Nagle's Algorithm

- Interactive applications
  - Telnet and rlogin
  - Generate many small packets (e.g., keystrokes)
- Small packets are wasteful
  - Mostly header (e.g., 40 bytes of header, 1 of data)
- Appealing to reduce the number of packets
  - Could force every packet to have some minimum size
  - … but, what if the person doesn't type more characters?
- Need to balance competing trade-offs
  - Send larger packets
  - … but don't introduce much delay by waiting

Nagle's Algorithm

- Wait if the amount of data is small
  - Smaller than Maximum Segment Size (MSS)
- And some other packet is already in flight
  - I.e., still awaiting the ACKs for previous packets
- That is, send at most one small packet per RTT
  - … by waiting until all outstanding ACKs have arrived



- Influence on performance
  - Interactive applications: enables batching of bytes
  - Bulk transfer: transmits in MSS-sized packets anyway

Motivation for Delayed ACK

- TCP traffic is often bidirectional
  - Data traveling in both directions
  - ACKs traveling in both directions
- ACK packets have high overhead
  - 40 bytes for the IP header and TCP header
  - … and zero data traffic
- Piggybacking is appealing
  - Host B can send an ACK to host A
  - … as part of a data packet from B to A