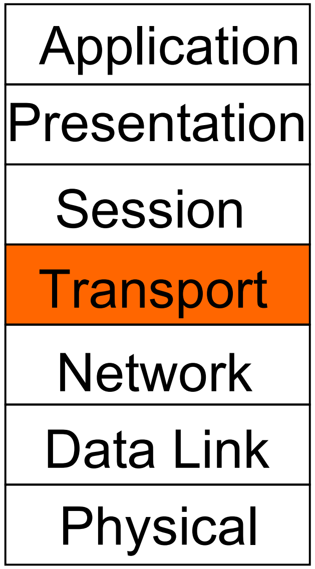
CSE 461

TCP and network congestion

This Lecture

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



Application

Presentation

Session

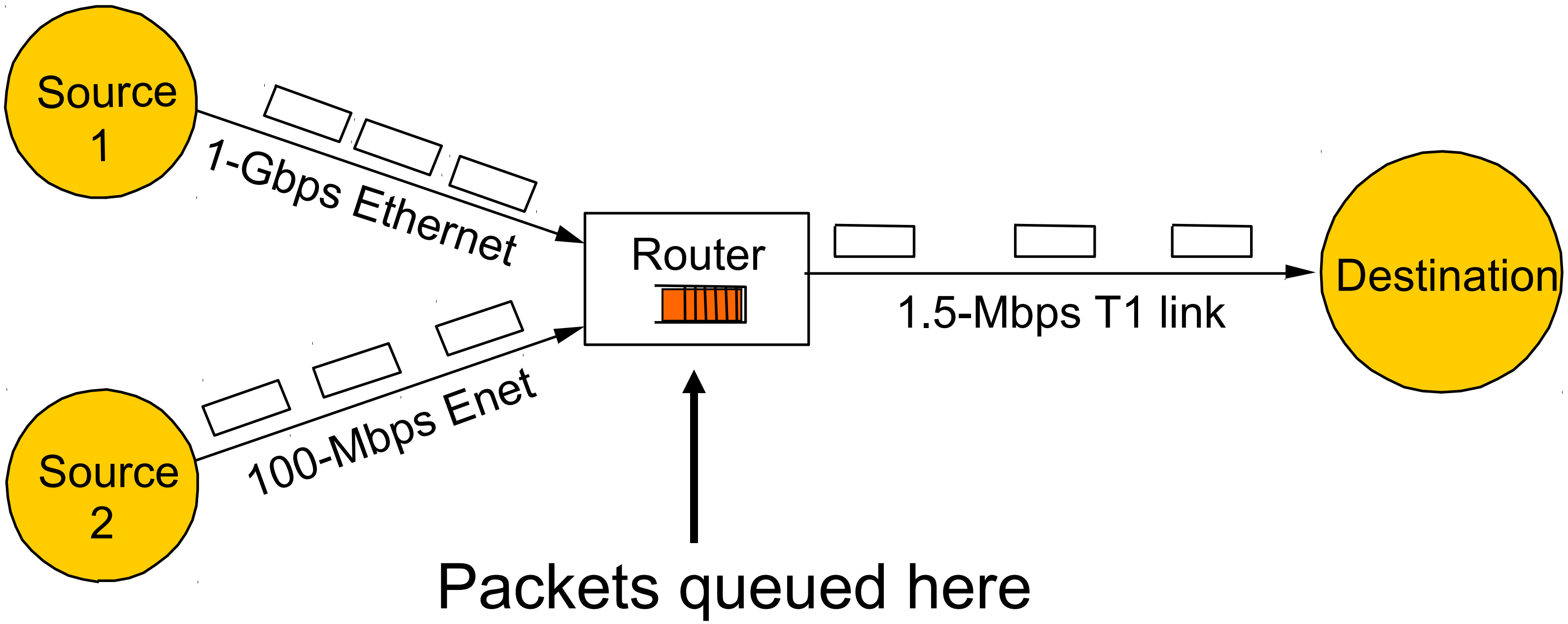
Transport

Network

Data Link

Physical

Congestion from in the network

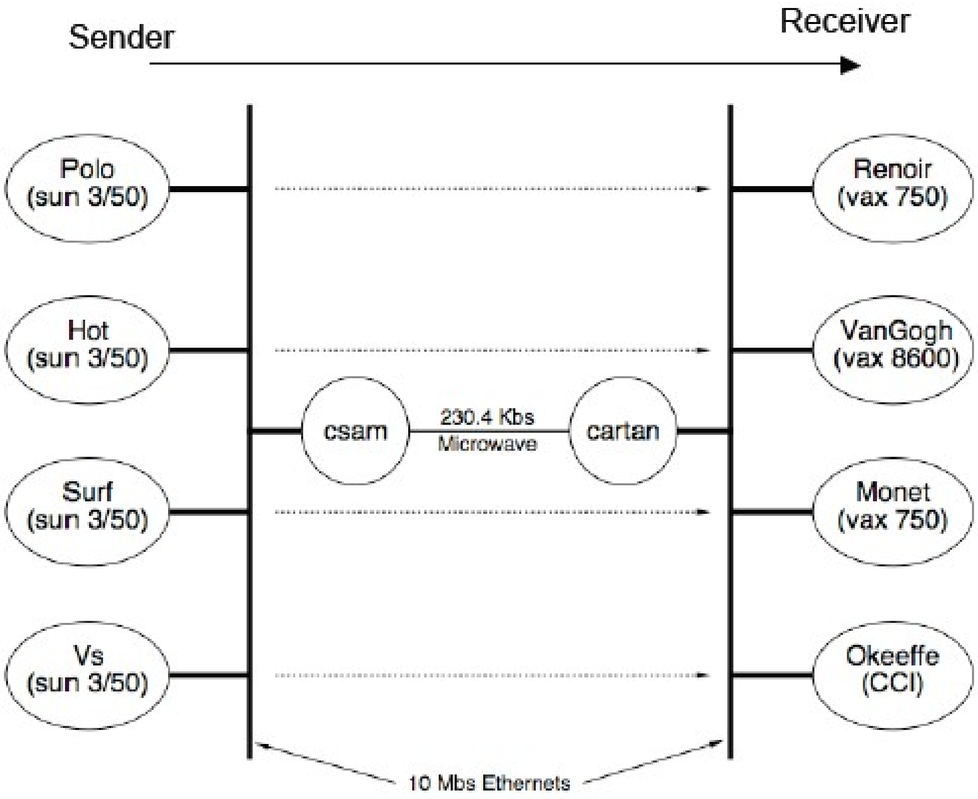


- Buffers at routers used to absorb bursts when input rate > output
- Loss (drops) occur when sending rate is persistently > drain rate

Congestion Collapse

- In the limit, premature retransmissions lead to congestion collapse
  - e.g., 1000x drop in effective bandwidth of network
  - sending more packets into the network when it is overloaded exacerbates the problem (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 'abysmal network conditions?'

A Scenario

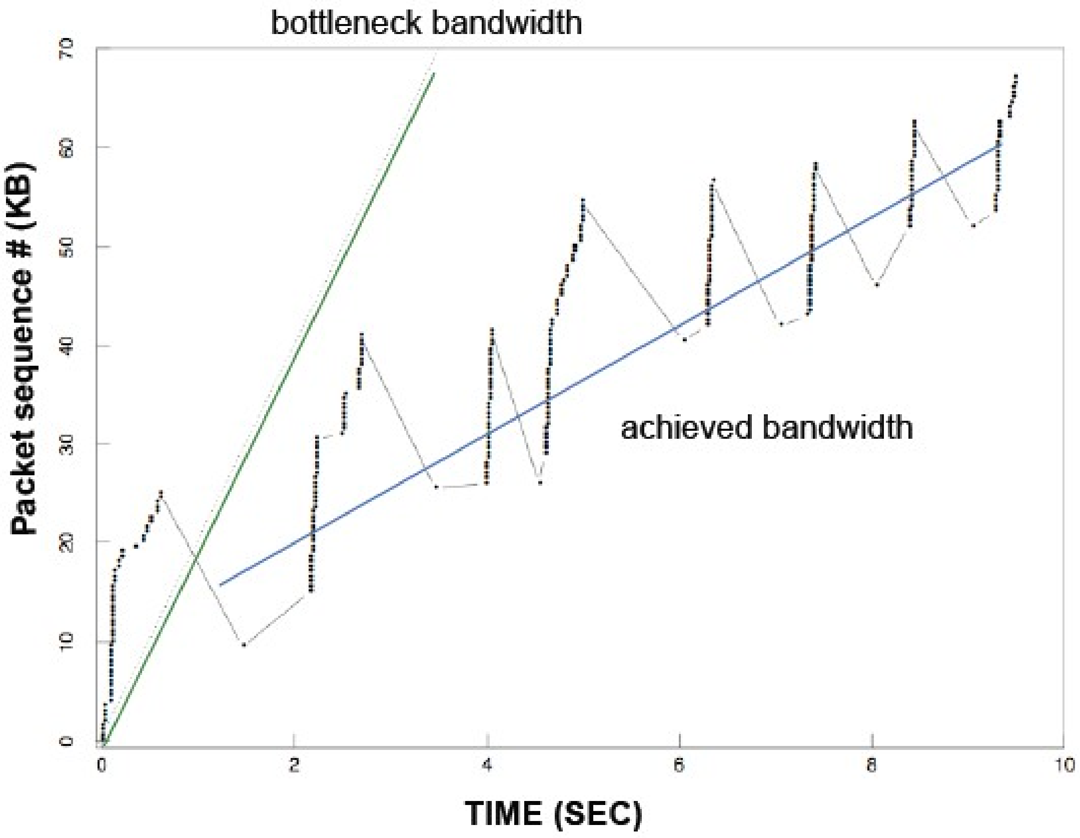


Receiver window size is 16KB.

Bottleneck router buffer size is 15 KB.

Data bandwidth is about 20KB/s

Behavior of Basic Sliding Window

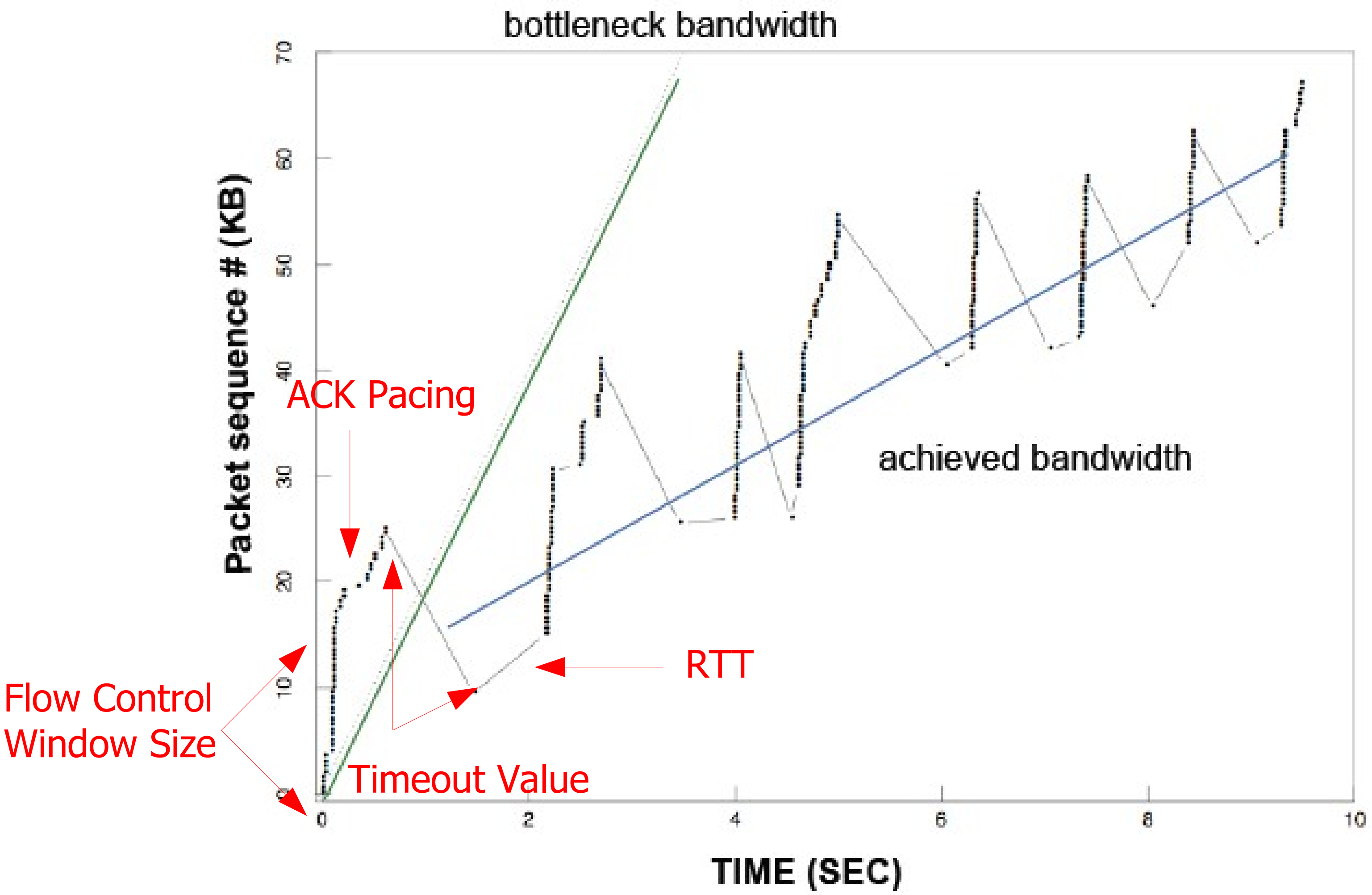


Slope is bandwidth.

Steep smooth upward slope == means good bandwidth.

Downward slope means retransmissions (*bad).*

Behavior of Basic Sliding Window



Let's Fix It

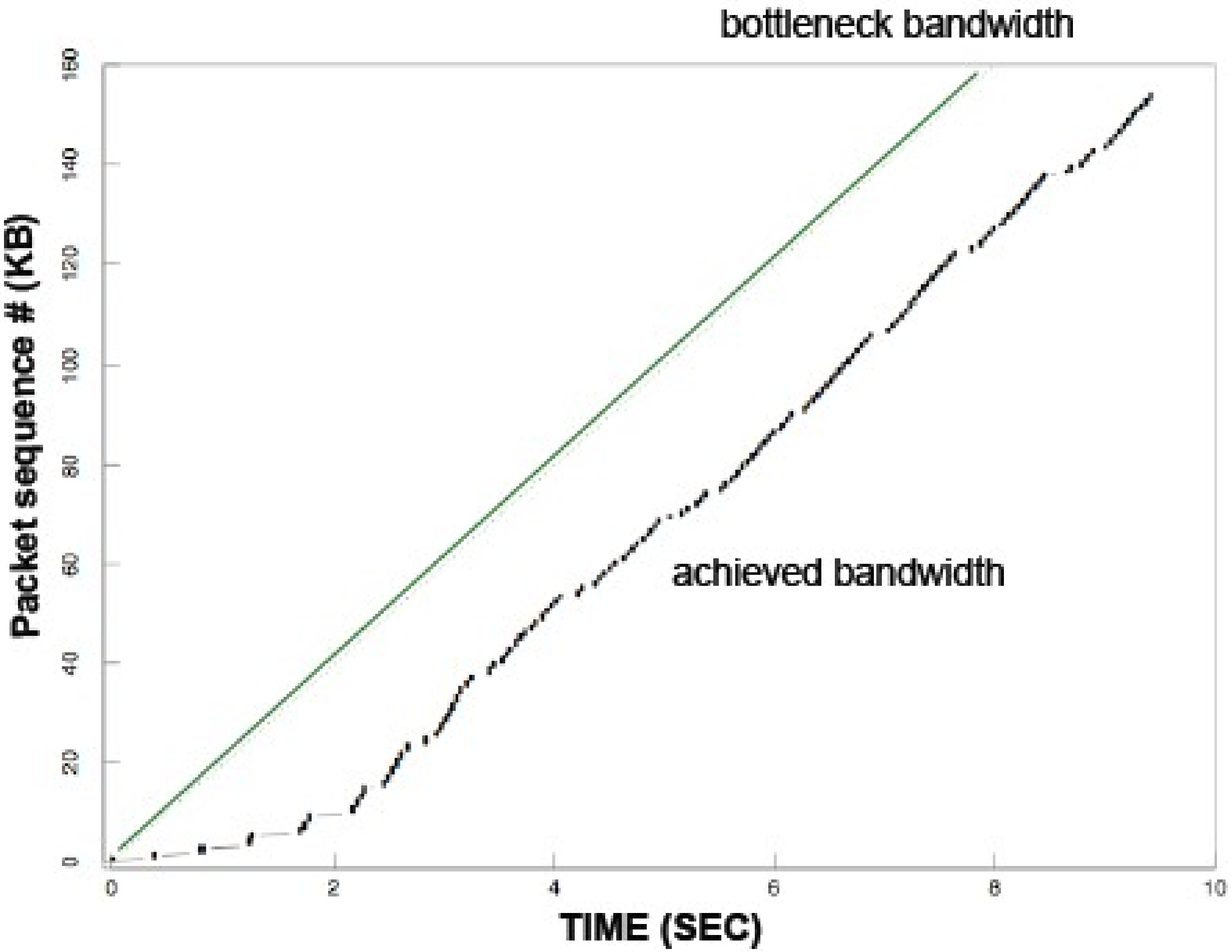
- We want to avoid overflowing the bottleneck router's queue
  - Idea 1: use something like flow-control, but with the router as the remote end point
  - Idea 2: throttle the source send rate so that it's no more than the bottleneck router's available capacity

- We'd like to retransmit instantly when a packet is actually lost, and not at all if it isn't

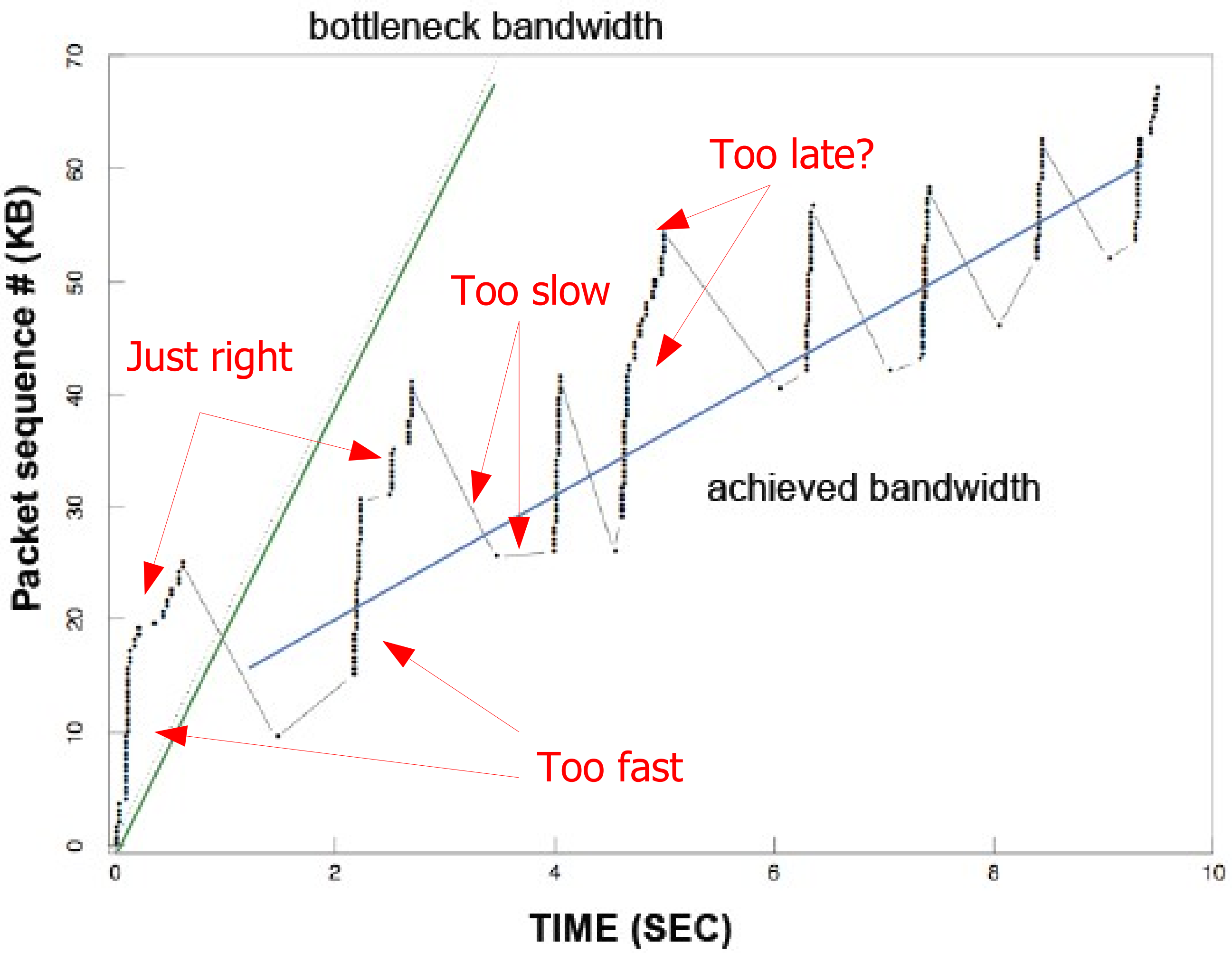
- If we controlled what the routers did, approaches come to mind

- We don't (can't) control what the routers do

Modern TCP in previous scenario



Behavior of Basic Sliding Window



Restriction on Approaches

- Can't ask routers to do anything they don't already do
- Can't radically change TCP (sliding window basis)
  - There's no way to deploy “a new TCP” to every Internet endpoint at once (or maybe ever)
  - You can deploy incrementally
    - One side of a TCP connection can do something new, if
    - It doesn't break the other, unmodified side
- The bottleneck router's capacity may change from time to time
  - (Even which router is the bottleneck may change)

TCP's Solutions

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

    MIN(Router Capacity - 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

1988 Observations on Congestion Collapse

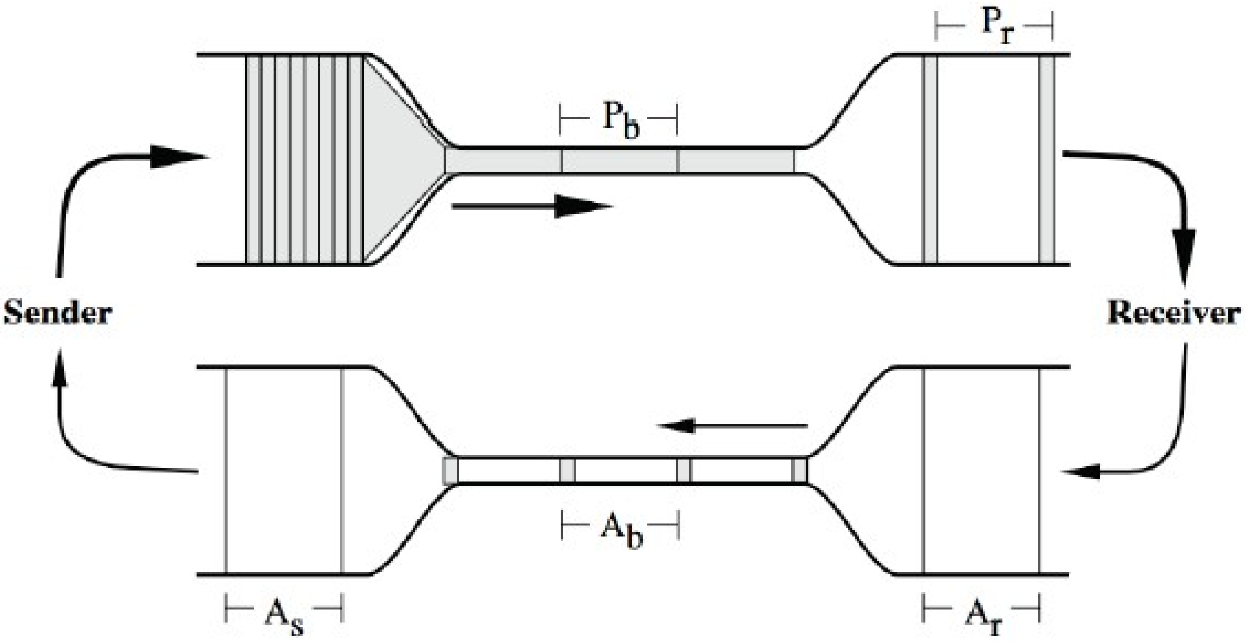
- Implementation, not the protocol, leads to collapse
  - choices about when to retransmit, when to “back off” because of losses

- “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

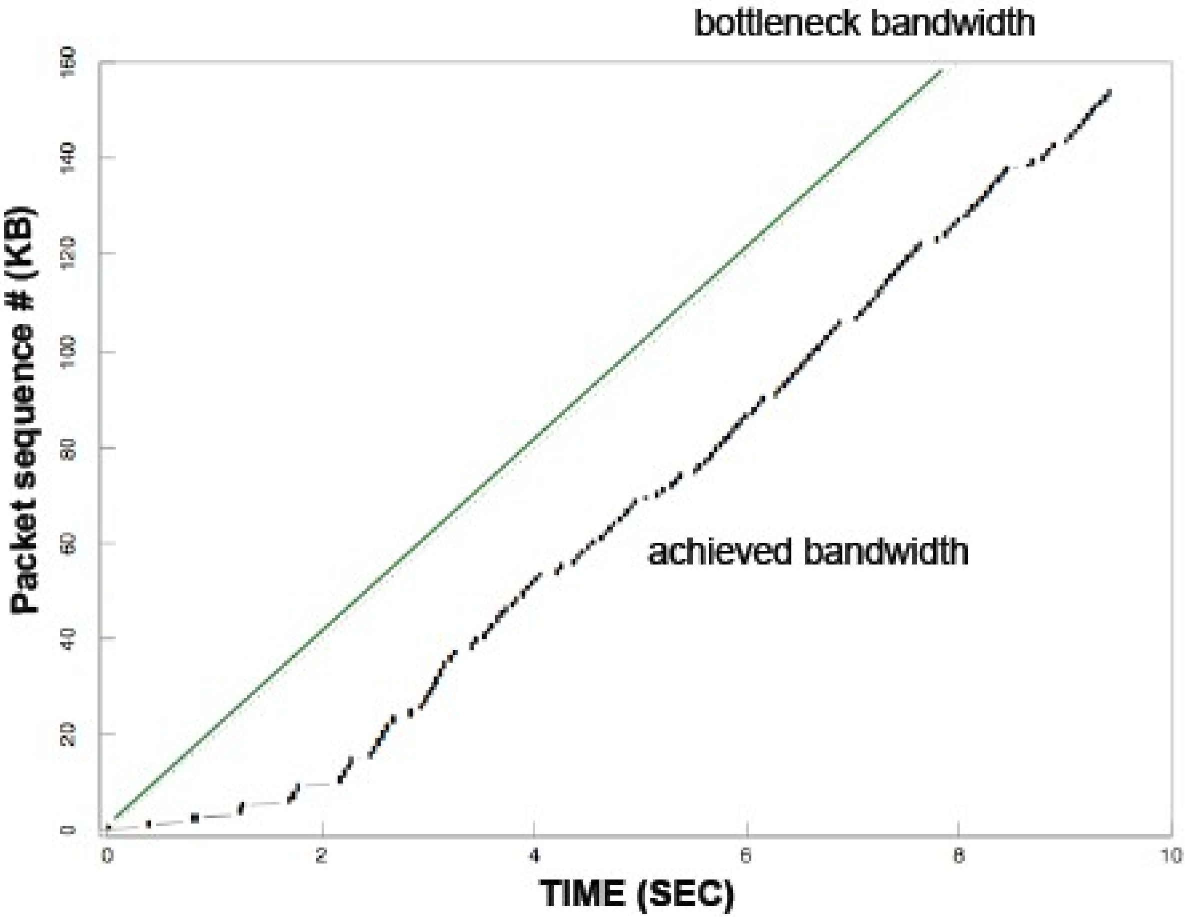
Ideal packet flow: stable equilibrium

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

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

1. 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 re-stabilize

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.”

1. The connection must reach equilibrium.

1. Getting to Equilibrium: Slow Start

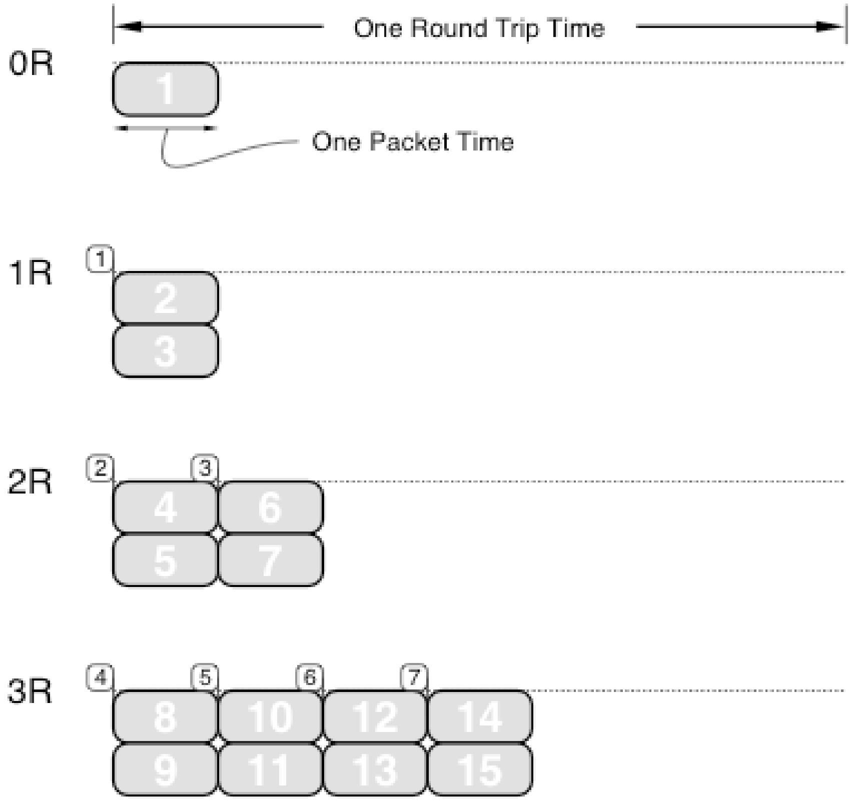
- Goal
  - Quickly determine the appropriate congestion window size
    - Basically, we're trying to sense the bottleneck bandwidth
- Strategy
  - Introduce *congestion_window (cwnd)*
  - When starting off, set cwnd to 1
  - For each ACK received, add 1 to cwnd
  - When sending, send the minimum of receiver's advertised window and cwnd

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
  - Guaranteed to not transmit at more than twice the max BW, and for no more than one RTT.
- 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 exponential decay

Tactic 2: Tactic 1 + *safety margin*

safety margin based on current estimate of error in Tactic 1

Tactic 0: Use the Mean

- Measure the RTT of each packet
  - Time from sending packet until receiving the ACK for it
- `EstimatedRTT = (Sum of SampleRTT’s) / N`
  - Note: requires only constant storage
- Why not do this?

Tactic 1: Original TCP (RFC793) retransmission timeout algorithm

- Use EWMA to estimate RTT:

$$EstimatedRTT = (1\text{-}g)(EstimatedRTT) + g(SampleRTT)$$

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

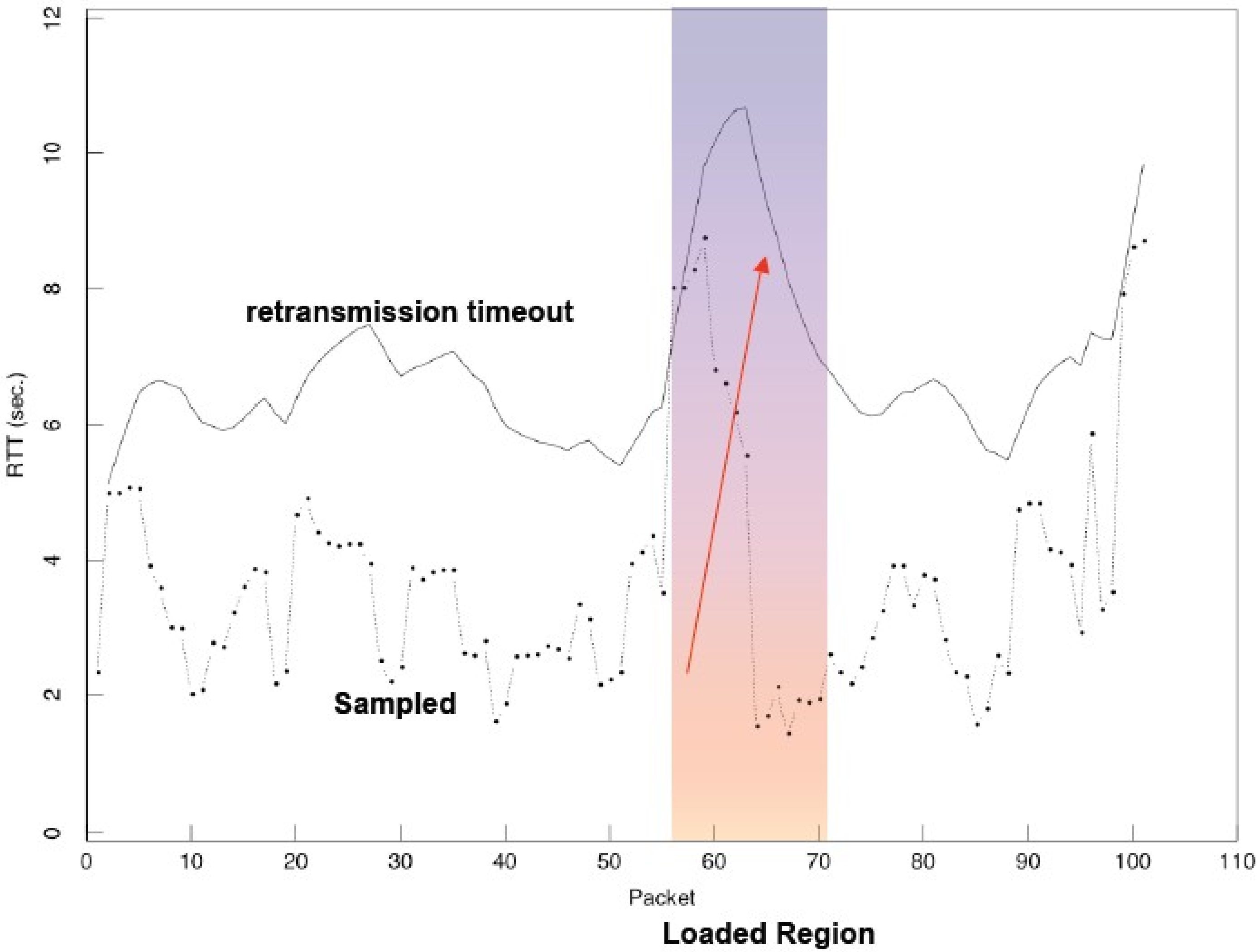
- Conservatively set timeout to small multiple (2x) of the estimate

$$Retransmission\ Timeout = 2\ x\ EstimatedRTT$$

- *(Expressed in manner ofTactic 2)*

$$Retransmission\ Timeout = EstimatedRTT + EstimatedRTT$$

Figure 5: Performance of an RFC793 retransmit timer



Tactic 2: Jacobson/Karels Algorithm

1. Explicitly estimate the deviation in the RTT

   DevRTT = (1-b) * DevRTT + b * |SampledRTT - EstimatedRTT|

   - typically, b = .25

2. Retransmission timeout = 1 x EstimatedRTT + k * DevRTT
   - k is generally set to 4
   - timeout =~ EstimatedRTT when variance is low (estimate is good)

Estimate with Mean + Variance

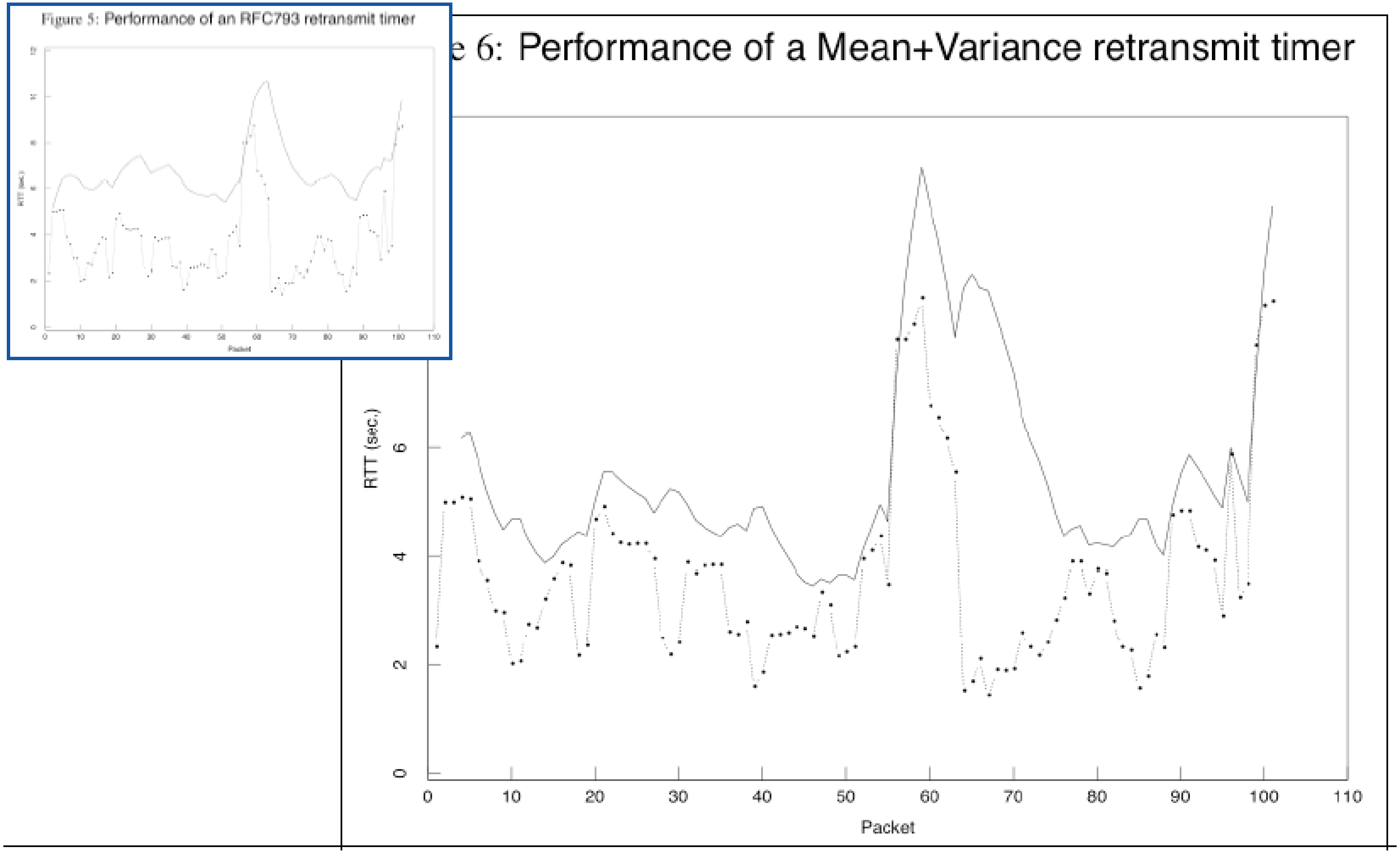
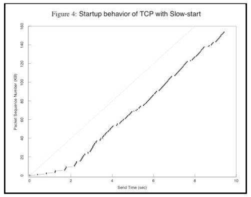


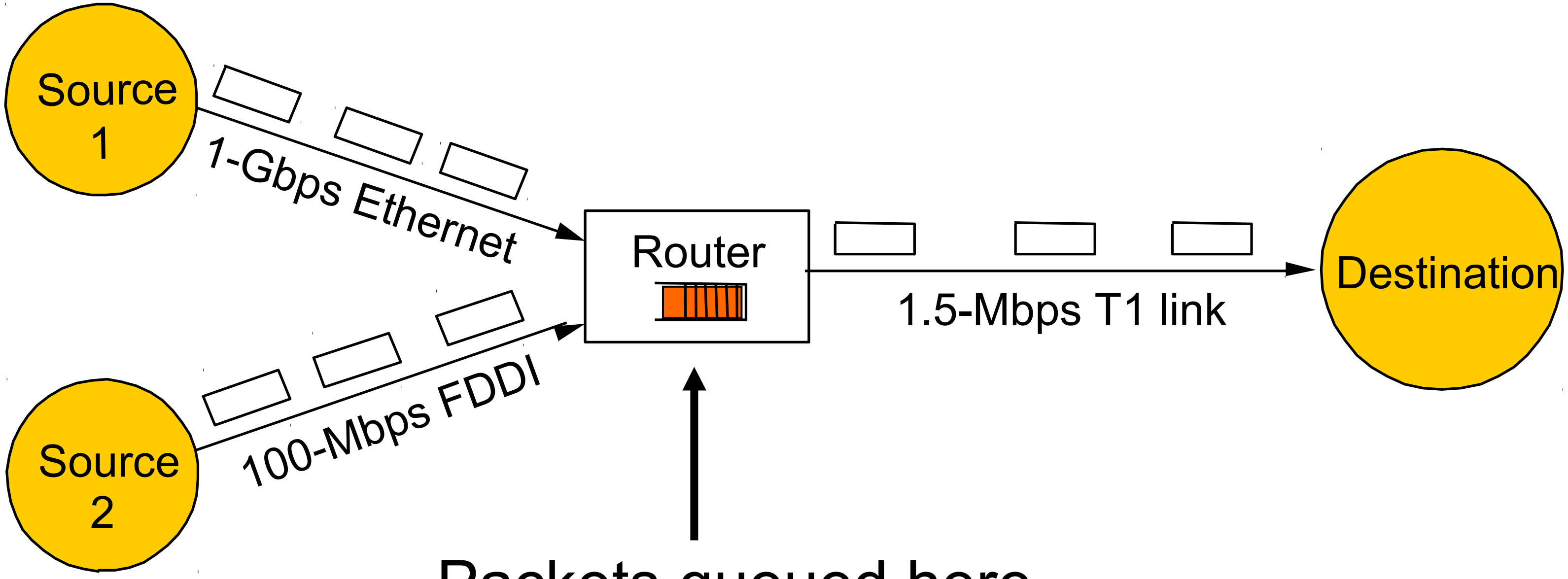
Figure 5: Performance of an RFC793 retransmit timer

e 6: Performance of a Mean+Variance retransmit timer

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

Congestion from Multiple Sources




Source 1

1-Gbps Ethernet

Source 2

100-Mbps FDDI

Router

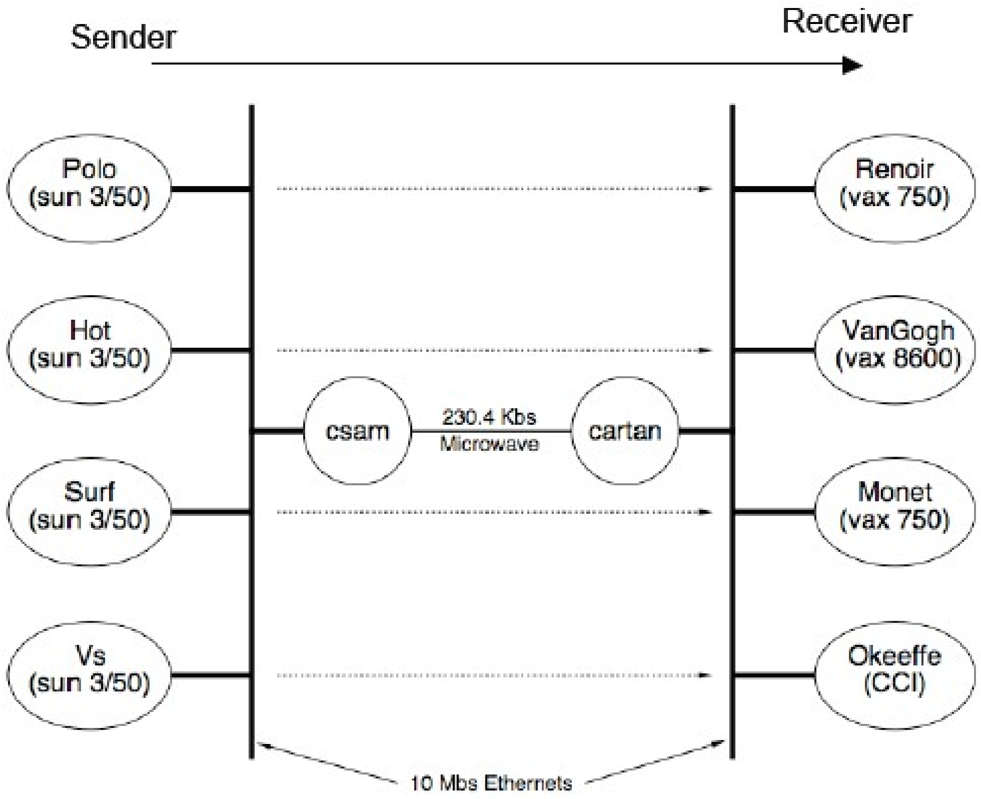
1.5-Mbps T1 link

Destination

Packets queued here

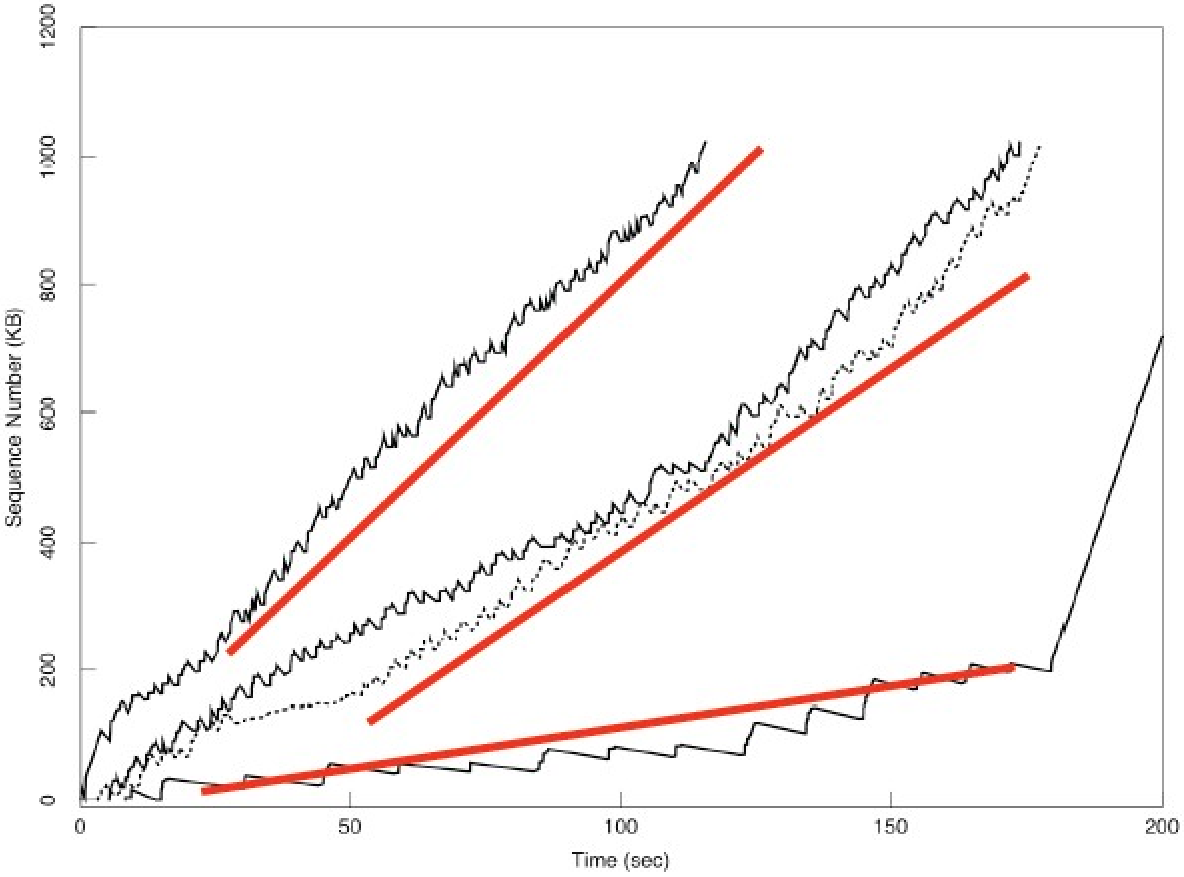
Packets Lost Here

In Real Life

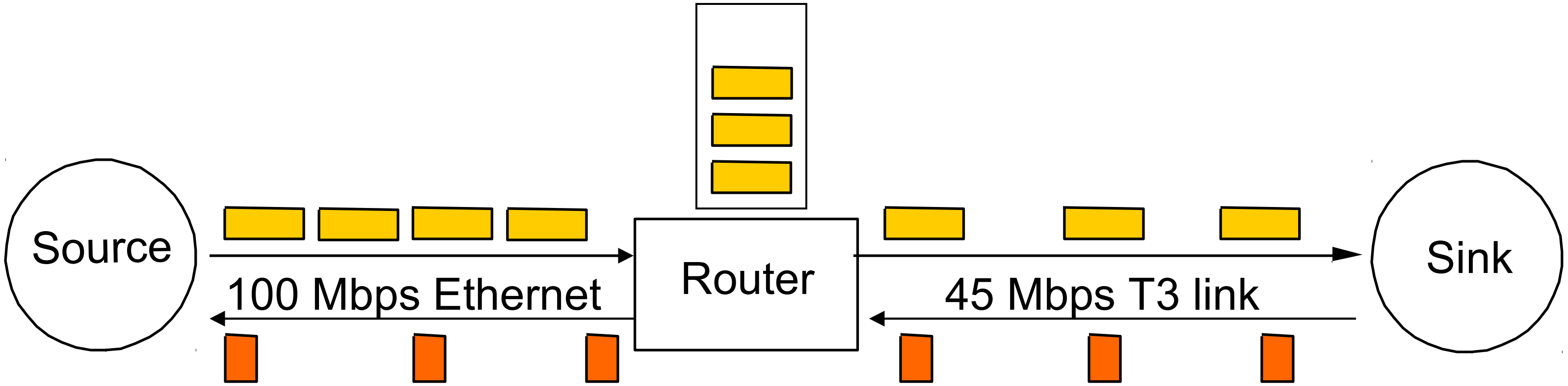

Sender
Receiver
Polo (sun 3/50)
Hot (sun 3/50)
Surf (sun 3/50)
Vs (sun 3/50)
csam
230.4 Kbs Microwave
cartan
Renoir (vax 750)
VanGogh (vax 8600)
Monet (vax 750)
Okeeffe (CCI)
10 Mbs Ethernets

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)

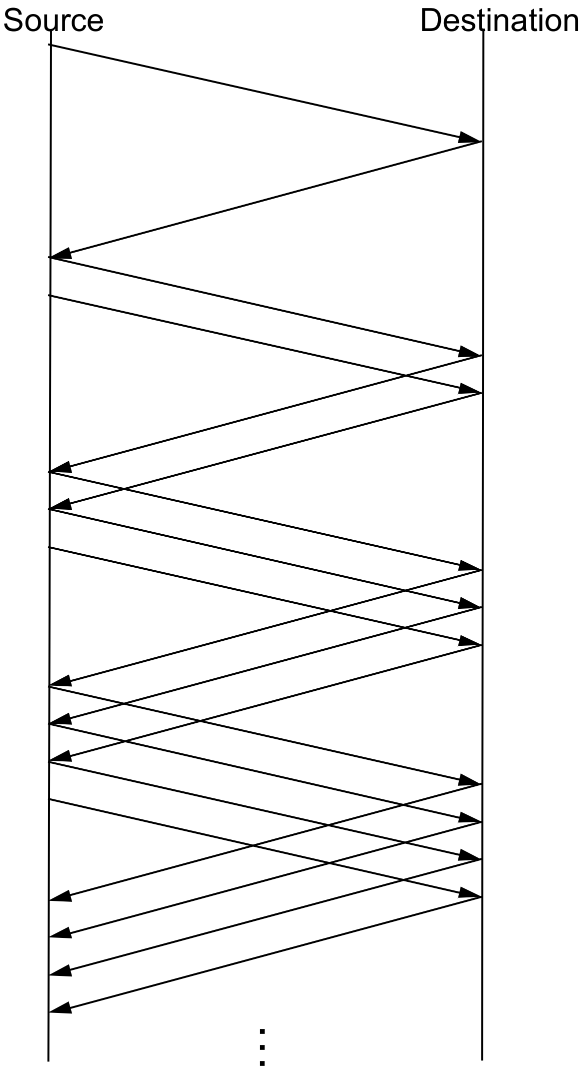
Congestion Control Relies on Signals from the Network

- 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

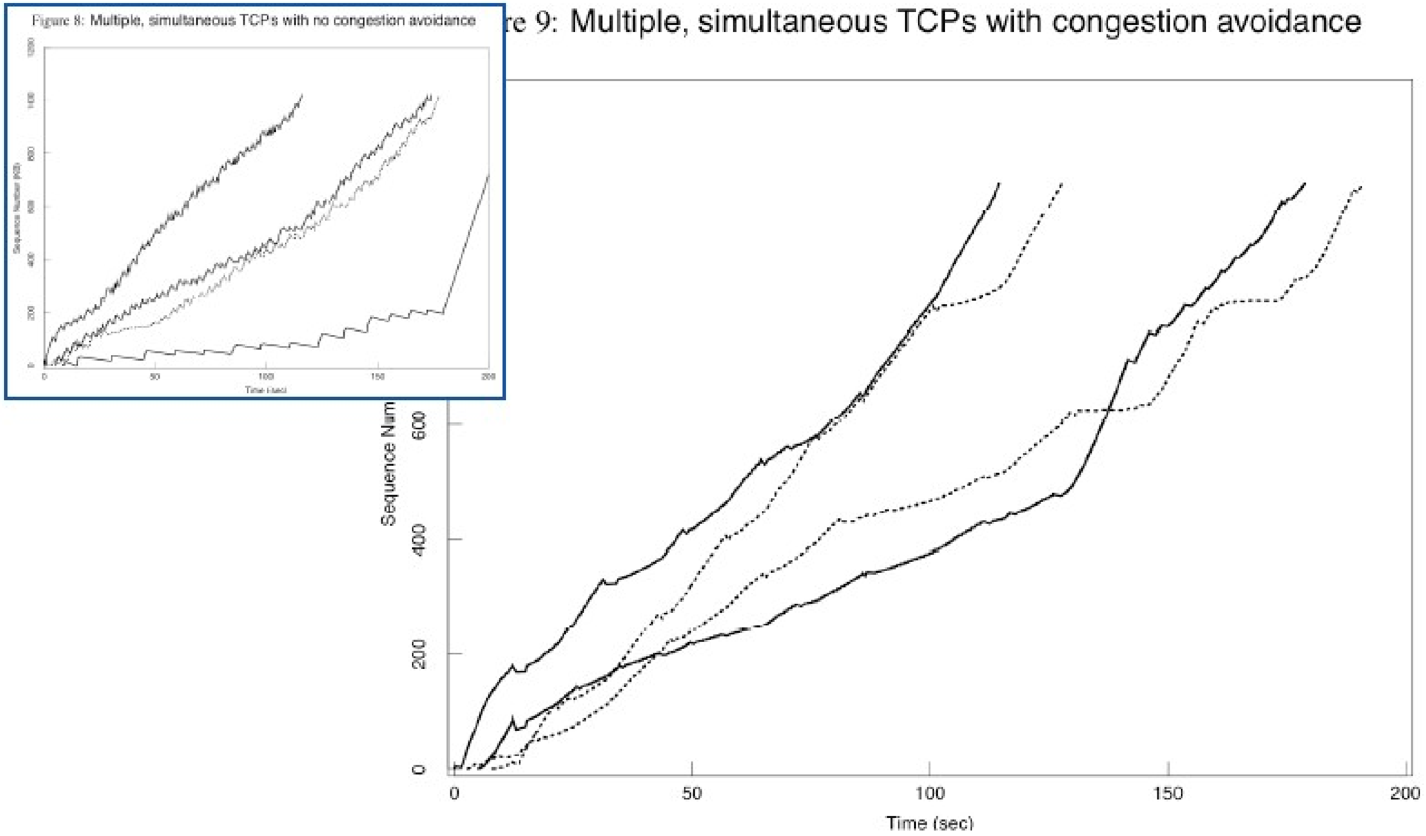
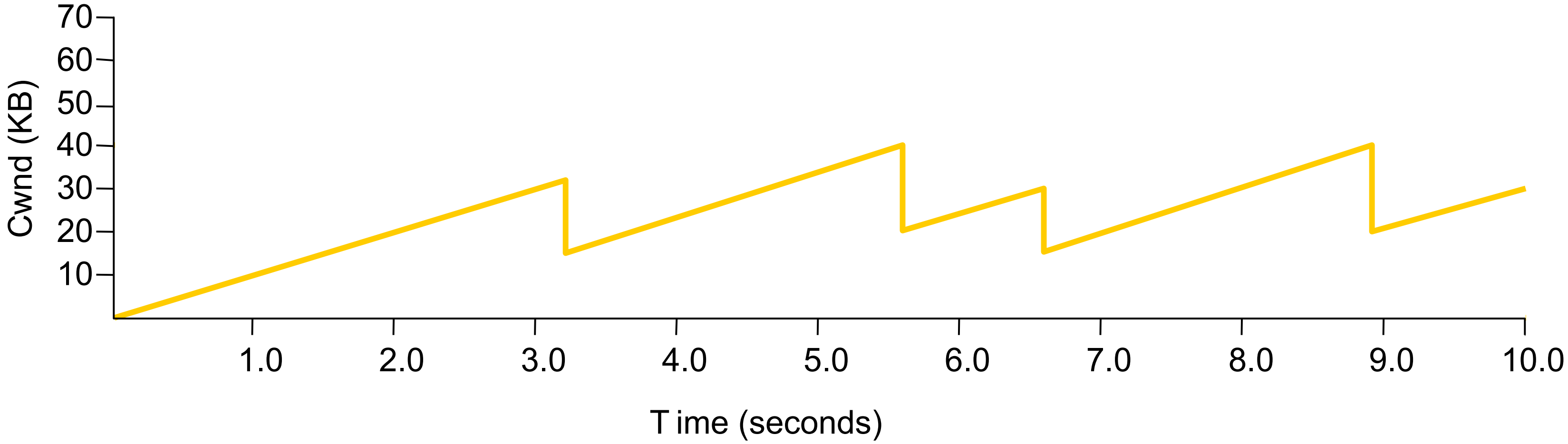


Figure 8: Multiple, simultaneous TCPs with no congestion avoidance

re 9: Multiple, simultaneous TCPs with congestion avoidance

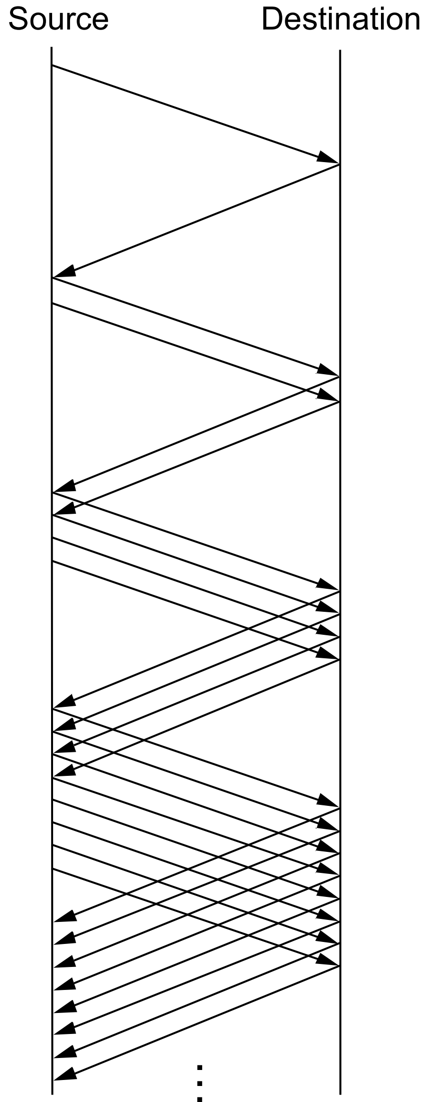
TCP Sawtooth Pattern



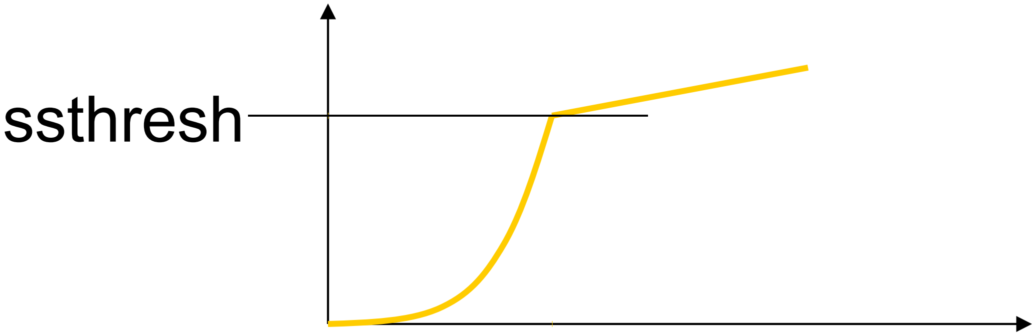
Comparing to "Slow Start"

- 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



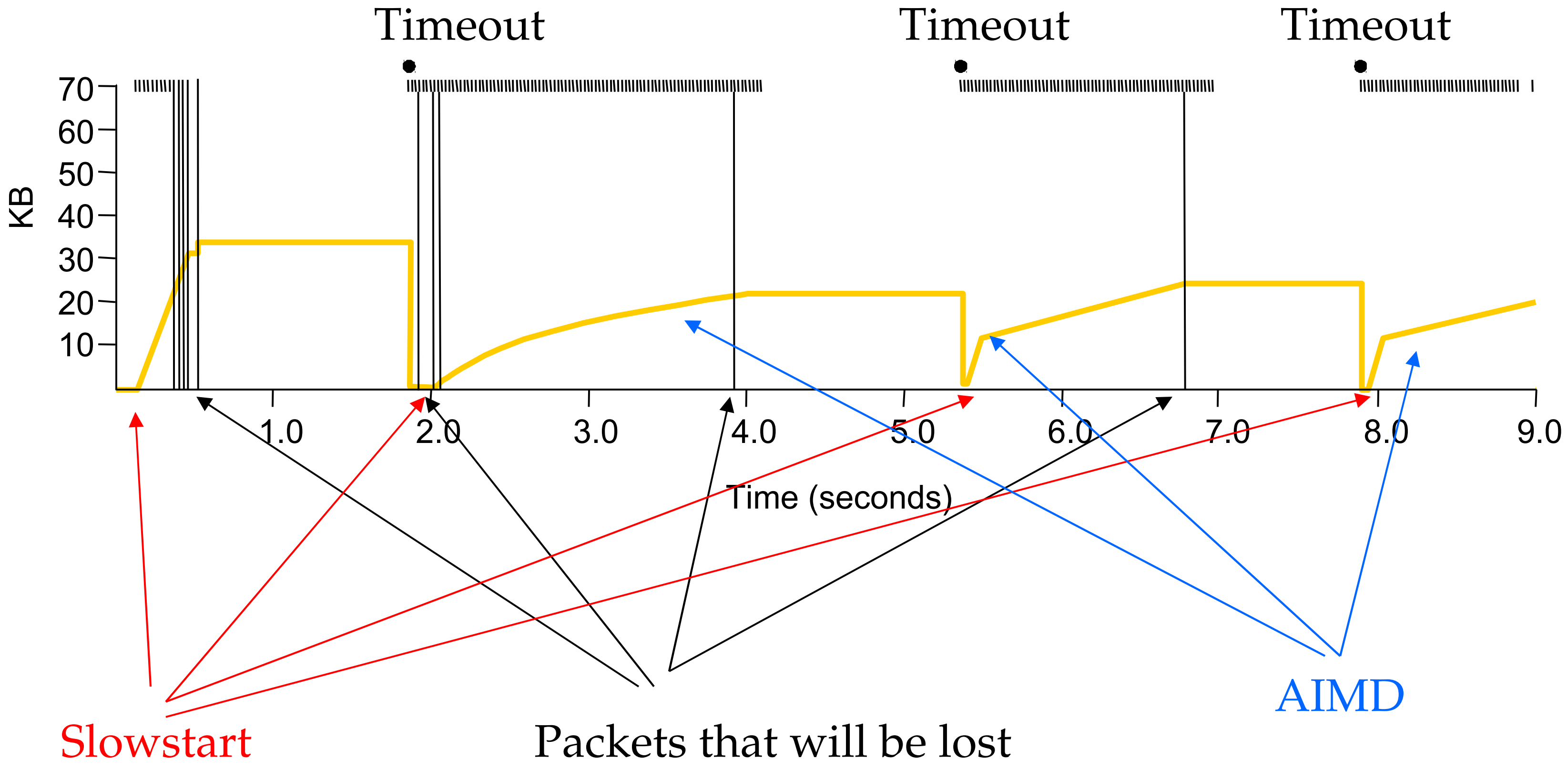
Combining Slow Start and AIMD



- Slow start is used whenever the connection is not running with packets outstanding
  - There won't be any more ACKs until we send again
  - 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)



Slowstart

Packets that will be lost

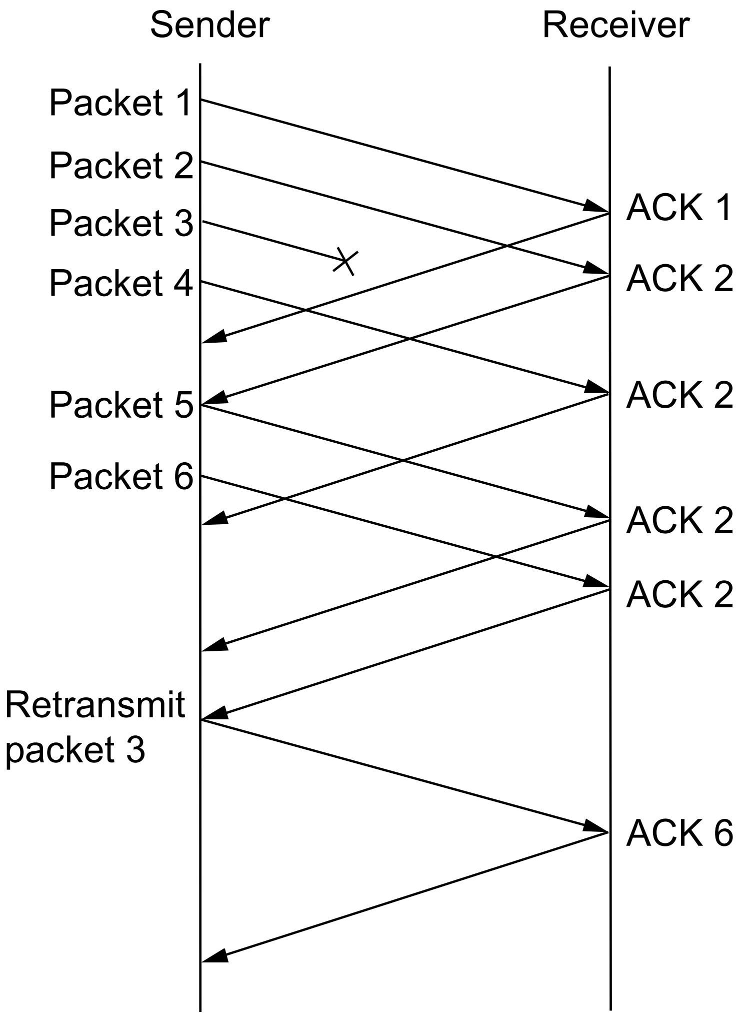
AIMD

The Long Timeout Problem

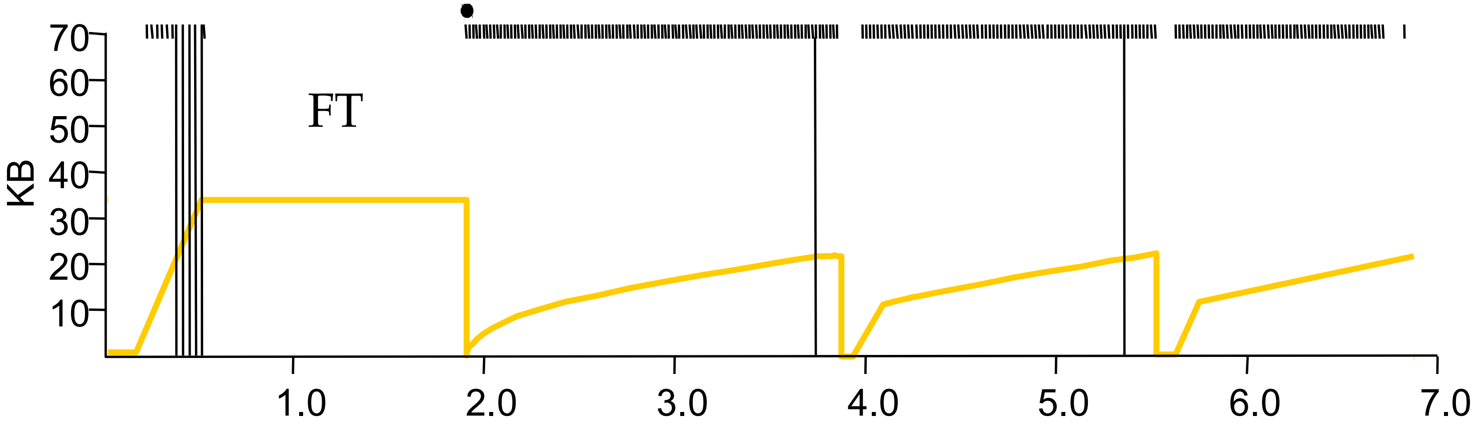
- Would like to detect a lost packet earlier than timeout
  - enable retransmit sooner
- Can we infer that a packet has been lost?
  - Receiver receives an "out of order packet"
  - Good indicator that the one(s) before have been misplaced
- Receiver generates a duplicate ack on receipt of a misordered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

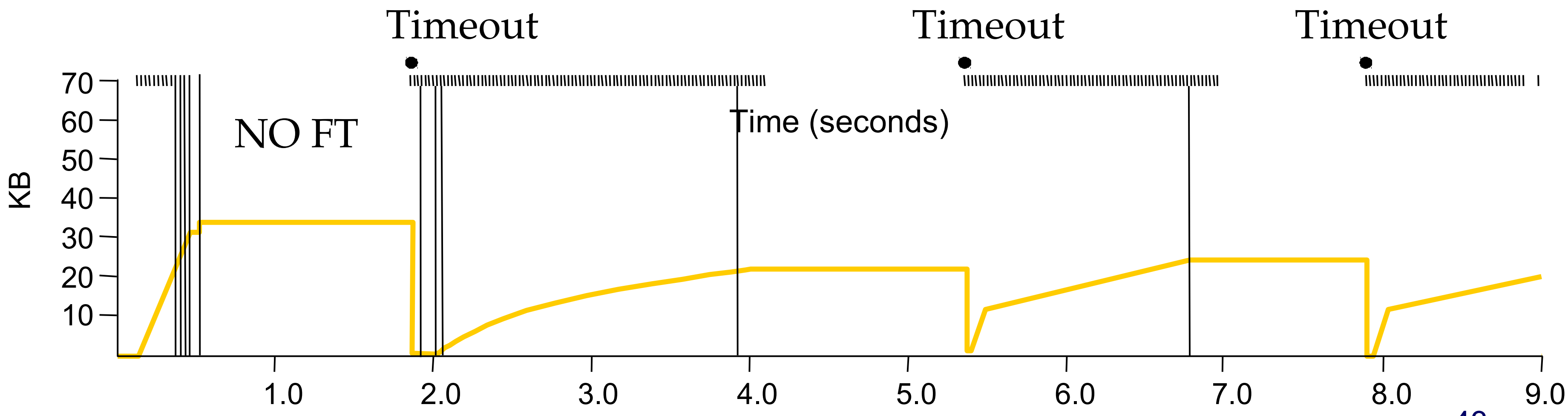
Fast Retransmit

- 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



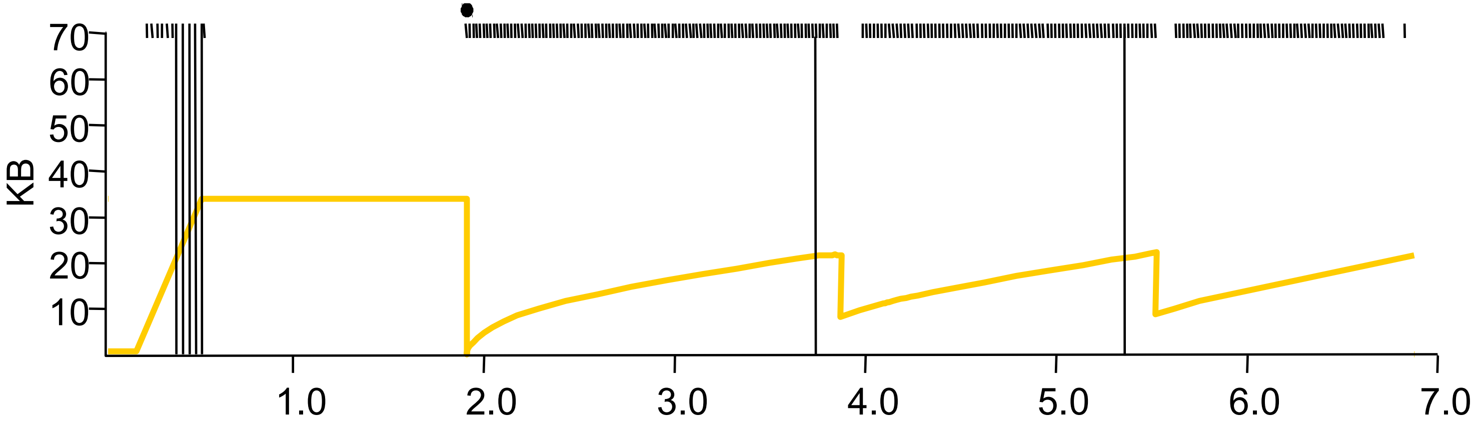
Example (with Fast Retransmit)





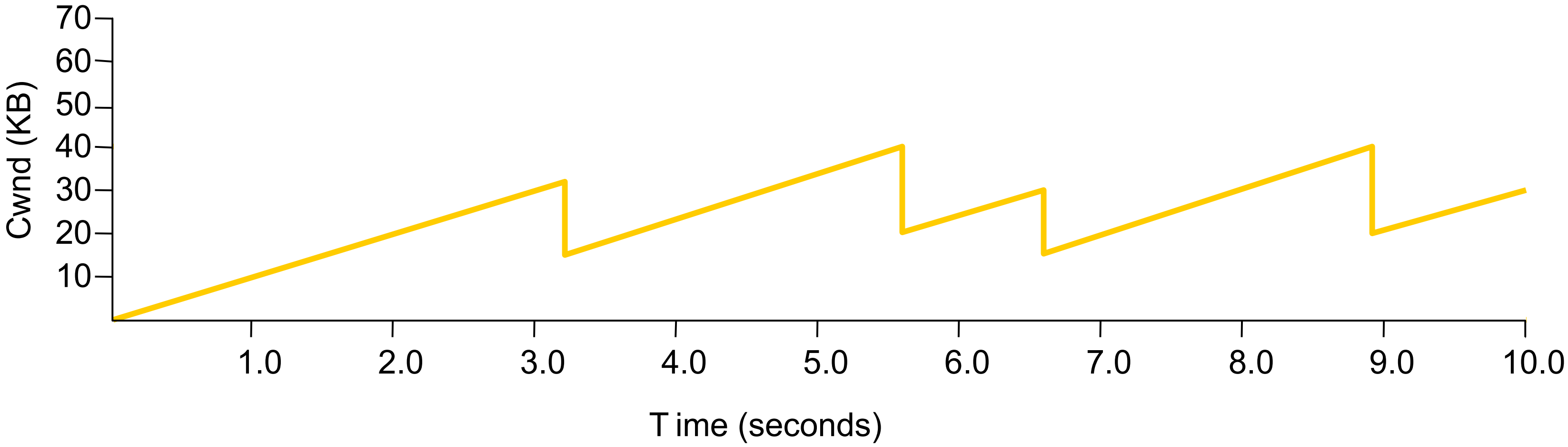
Fast Recovery

- 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.



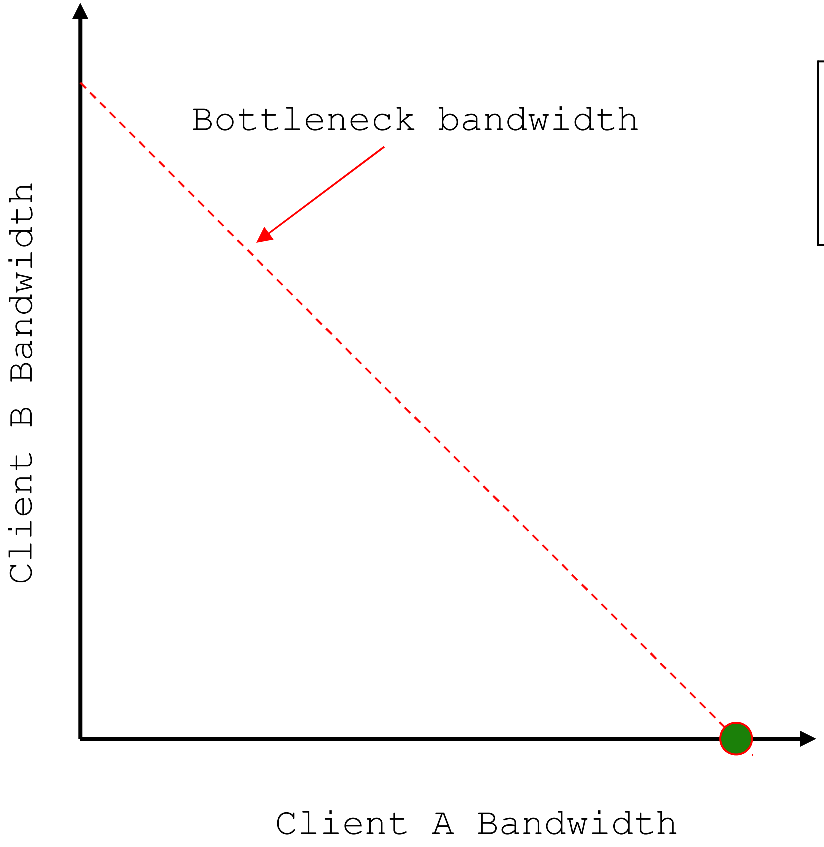
Example (with Fast Recovery)

(Not the same trace as before)



The Familiar Saw Tooth Pattern

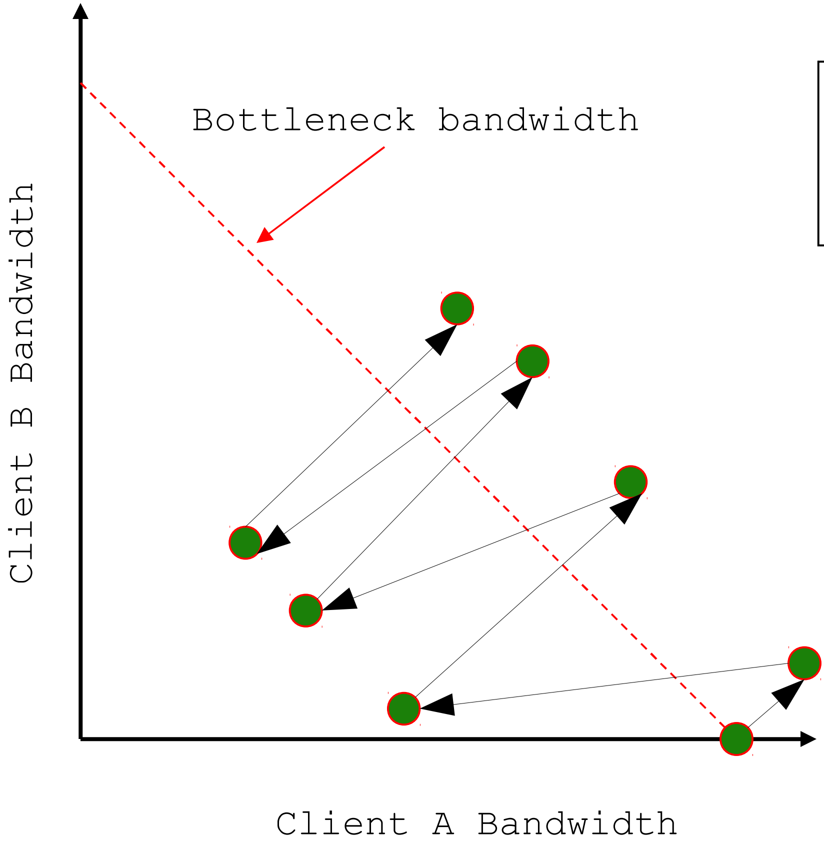
Fairness – an informal argument



Client A has an ongoing flow.
Client B arrives.

What happens?

Fairness – an informal argument



Client A has an ongoing flow.
Client B arrives.

What happens?

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

Key Concepts (next level down)

- TCP probes the network for bandwidth, assuming that loss signals congestion
- The congestion window is managed to be additive increase / multiplicative decrease
  - It took fast retransmit and fast recovery to get there
- Slow start is used to avoid lengthy initial delays
  - Ramp up to near target rate and then switch to AIMD