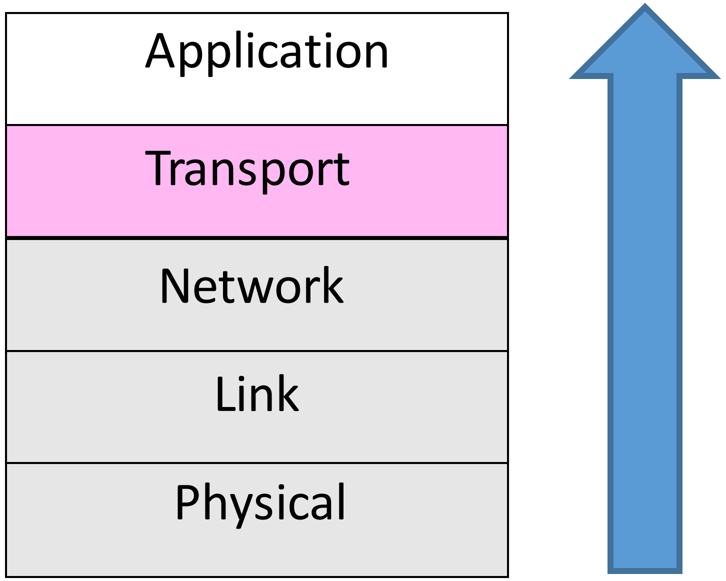
Transport

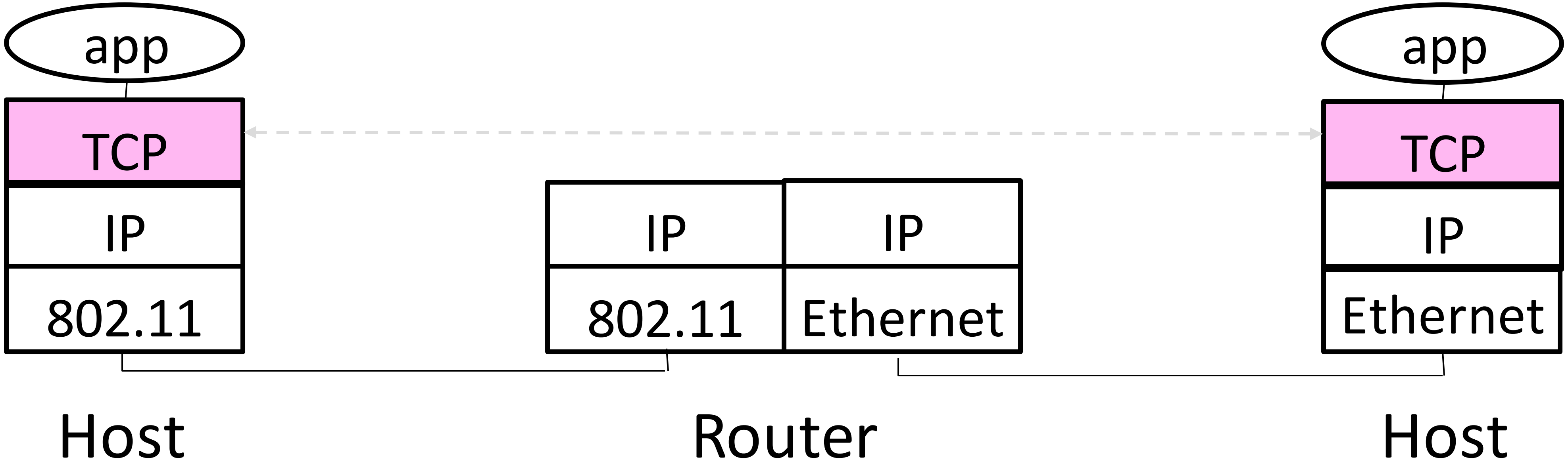
Where we are in the Course

- Moving on up to the Transport Layer!



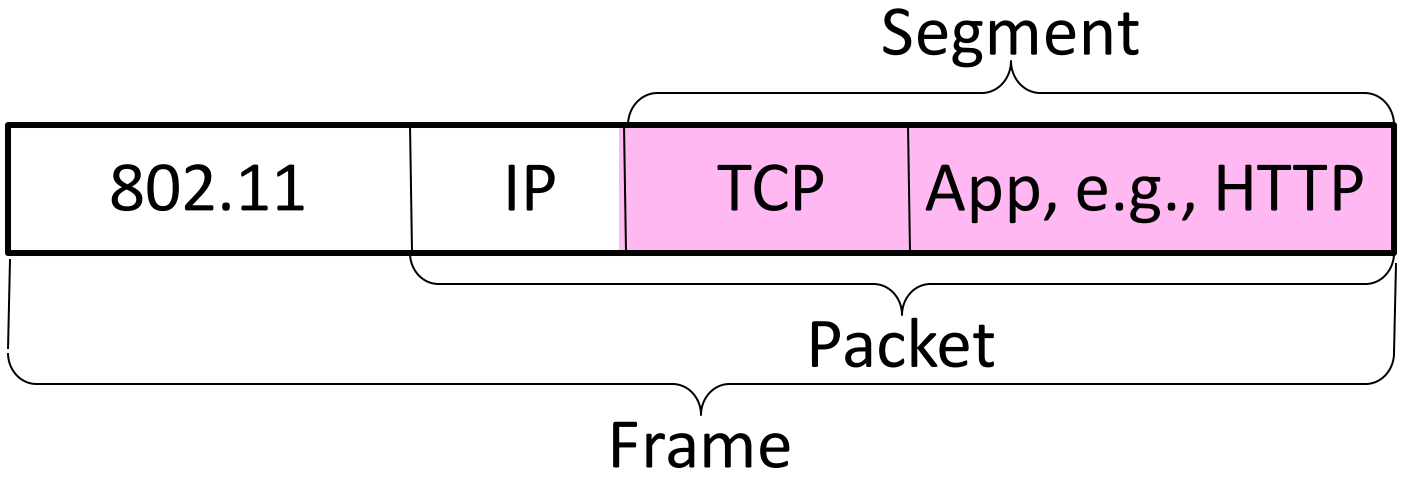
Recall

- Transport layer provides end-to-end connectivity across the network



Recall (2)

- Segments carry application data across the network
- Segments are carried within packets within frames



Transport Layer Services

- Provide different kinds of data delivery across the network to applications

	Unreliable	**Reliable**
Messages	Datagrams (UDP)	
Bytestream		Streams (TCP)

Comparison of Internet Transports

- TCP is full-featured, UDP is a glorified packet

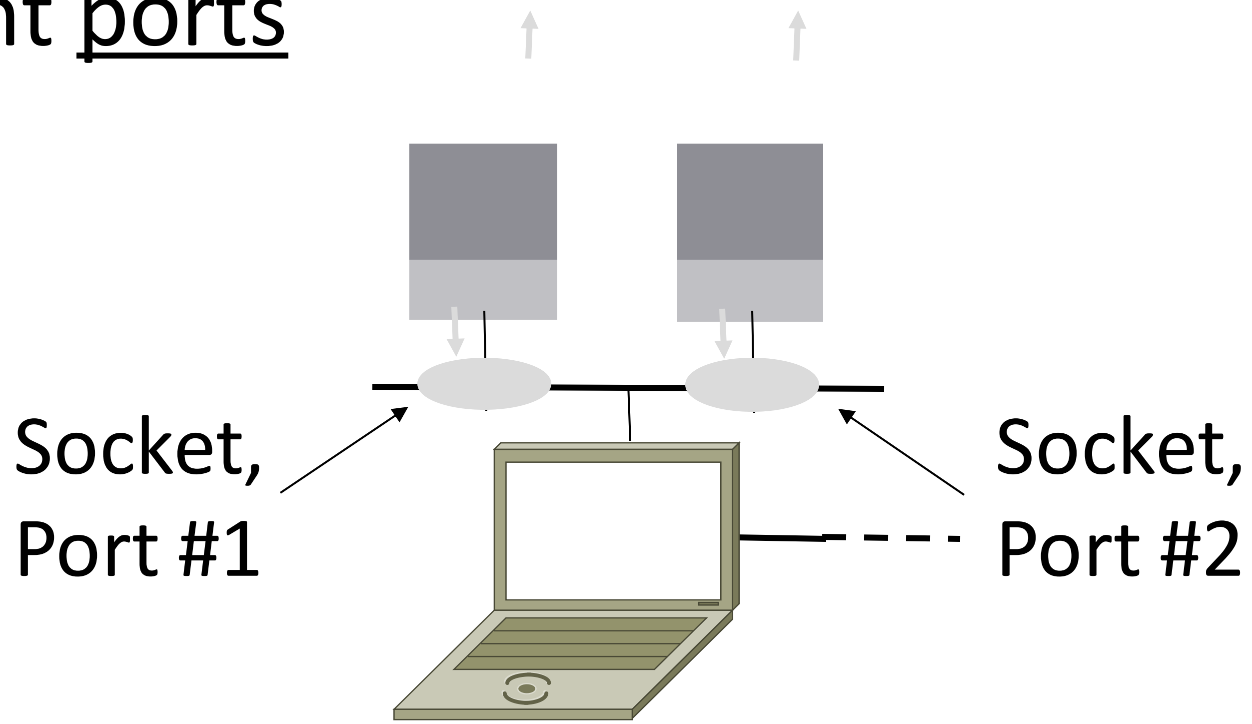
TCP (Streams)	UDP (Datagrams)
Connections	Datagrams
Bytes are delivered once, reliably, and in order	Messages may be lost, reordered, duplicated
Arbitrary length content	Limited message size
Flow control matches sender to receiver	Can send regardless of receiver state
Congestion control matches sender to network	Can send regardless of network state

Socket API

- Simple abstraction to use the network
  - The “network” API (really Transport service) used to write all Internet apps
  - Part of all major OSes and languages; originally Berkeley (Unix) ~1983
- Supports both Internet transport services (Streams and Datagrams)

Socket API (2)

- <u>Sockets</u> let apps attach to the local network at different <u>ports</u>



Socket API (3)

- Same API used for Streams and Datagrams

Only needed for Streams

To/From for Datagrams

Primitive	Meaning
SOCKET	Create a new communication endpoint
BIND	Associate a local address (port) with a socket
LISTEN	Announce willingness to accept connections
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND(TO)	Send some data over the socket
RECEIVE(FROM)	Receive some data over the socket
CLOSE	Release the socket

Ports

- Application process is identified by the tuple IP address, transport protocol, and port
  - Ports are 16-bit integers representing local “mailboxes” that a process leases
- Servers often bind to “well-known ports”
  - <1024, require administrative privileges
- Clients often assigned “ephemeral” ports
  - Chosen by OS, used temporarily

Some Well-Known Ports

Port	Protocol	Use
20, 21	FTP	File transfer
22	SSH	Remote login, replacement for Telnet
25	SMTP	Email
80	HTTP	World Wide Web
110	POP-3	Remote email access
143	IMAP	Remote email access
443	HTTPS	Secure Web (HTTP over SSL/TLS)
543	RTSP	Media player control
631	IPP	Printer sharing

UDP

User Datagram Protocol (UDP)

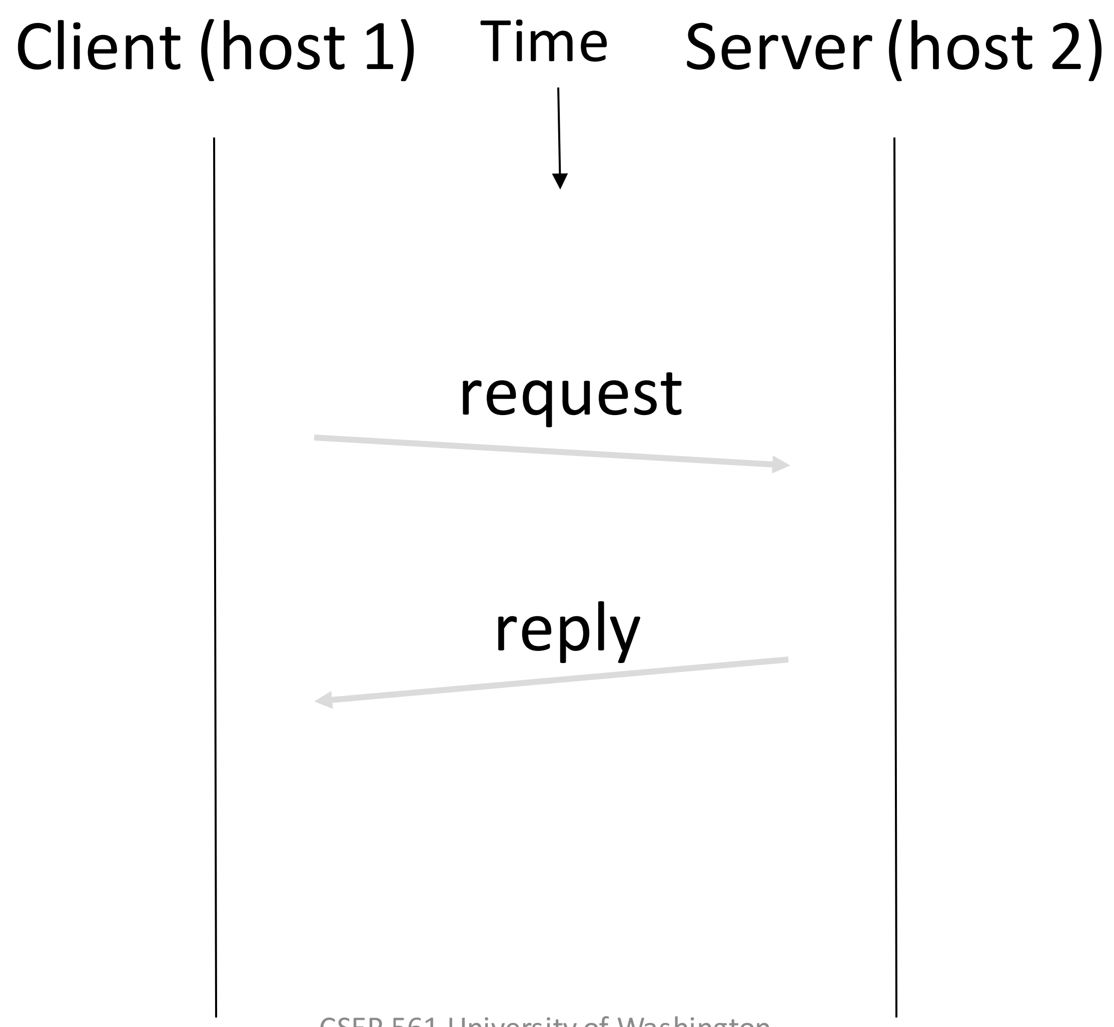
- Used by apps that don’t want reliability or bytestreams
  - Like what?

User Datagram Protocol (UDP)

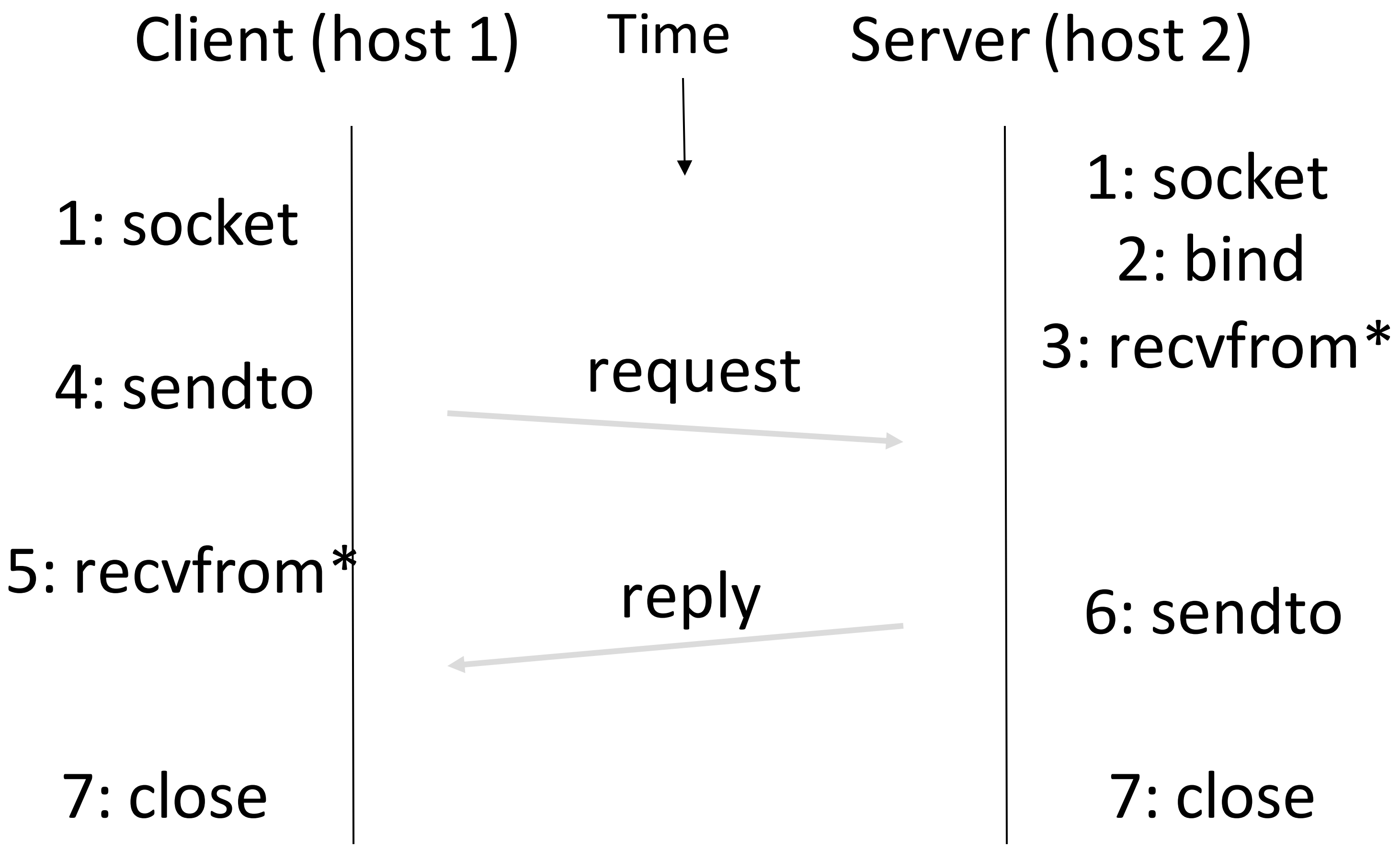
- Used by apps that don’t want reliability or bytestreams
  - Voice-over-IP
  - DNS, RPC
  - DHCP

(If application wants reliability and messages then it has work to do!)

Datagram Sockets

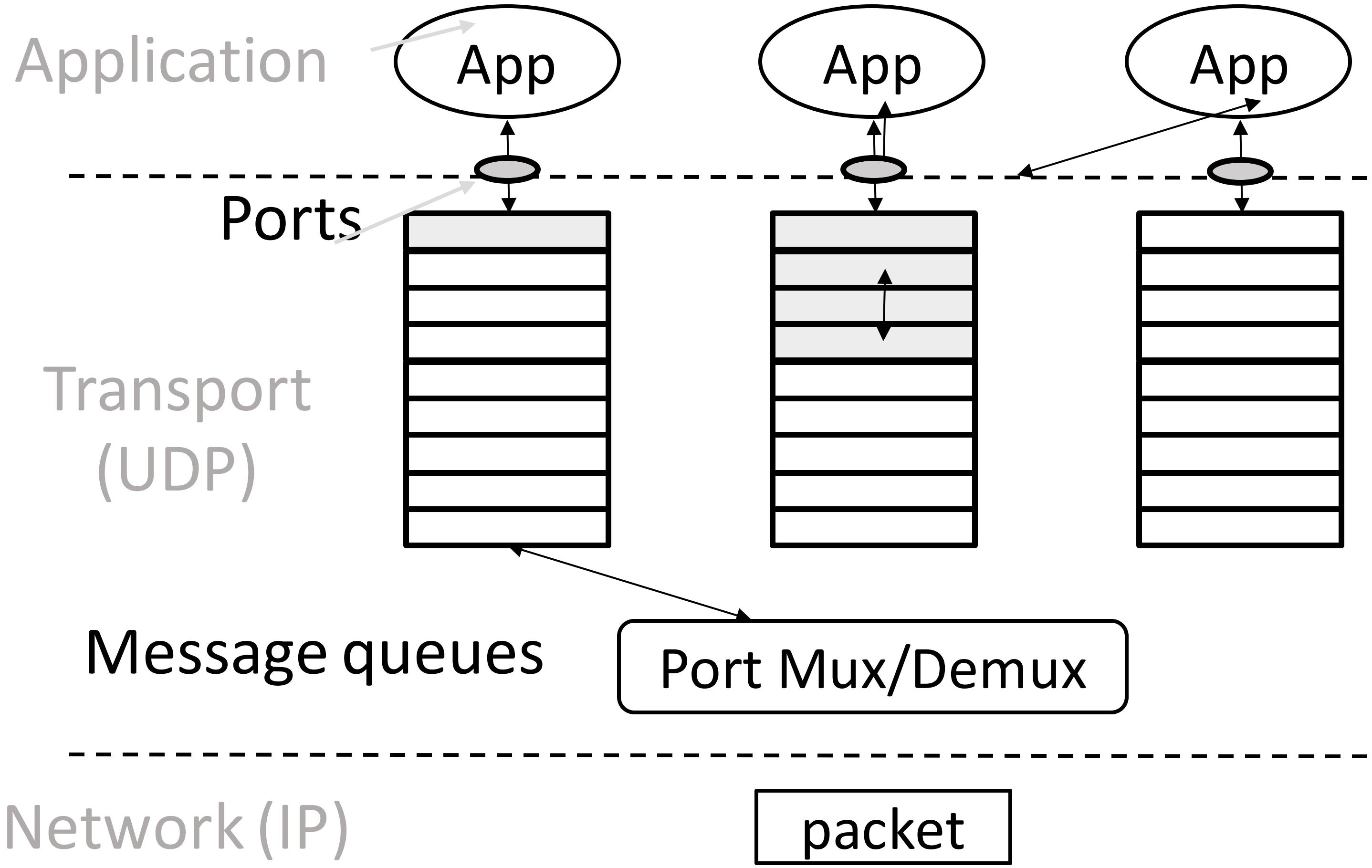


Datagram Sockets (2)



*= call blocks

UDP Buffering



UDP Header

- Uses ports to identify sending and receiving application processes
- Datagram length up to 64K
- Checksum (16 bits) for reliability

32 Bits

Source port	Destination port
UDP length	UDP checksum

UDP Header (2)

- Optional checksum covers UDP segment and IP pseudoheader
  - Checks key IP fields (addresses)
  - Value of zero means “no checksum”

32 Bits

<table>
<tr><td colspan="3">Source address</td></tr>
<tr><td colspan="3">Destination address</td></tr>
<tr><td>0 0 0 0 0 0 0 0</td><td>Protocol = 17</td><td>UDP length</td></tr>
</table>

TCP

TCP

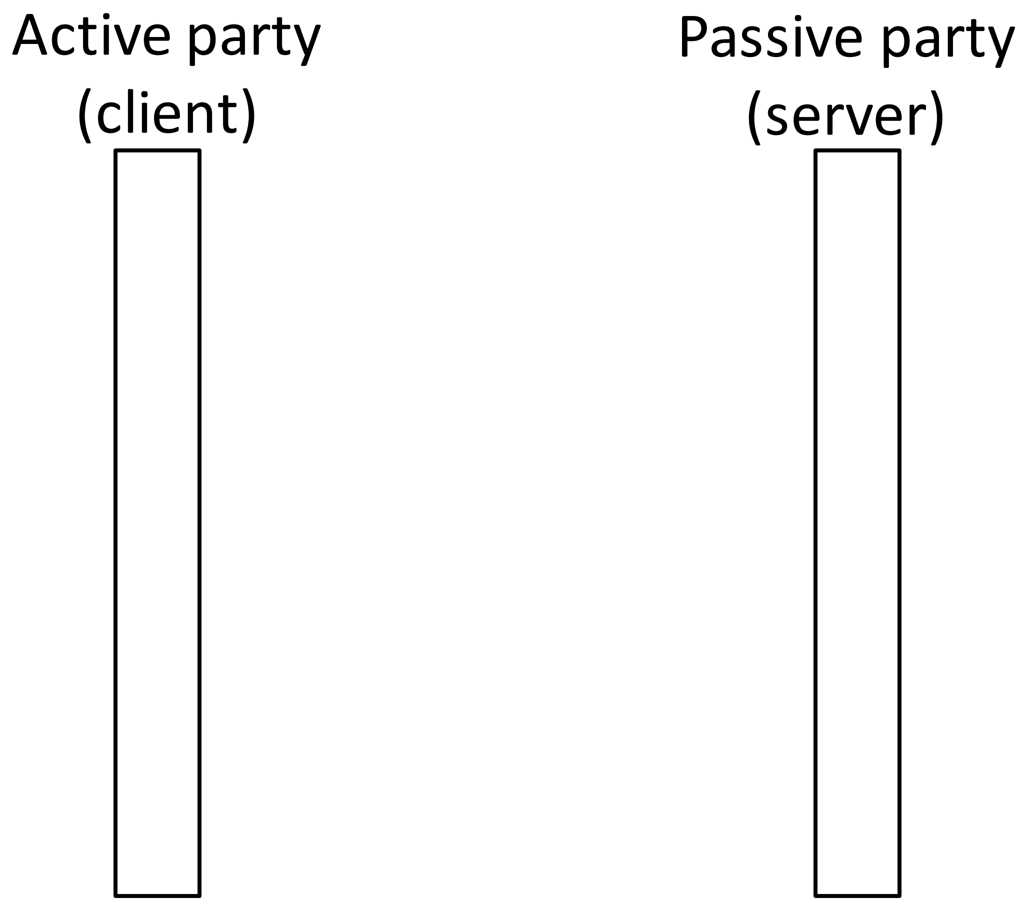
- TCP Consists of 3 primary phases:
  - Connection Establishment (Setup)
  - Sliding Windows/Flow Control
  - Connection Release (Teardown)

Connection Establishment

- Both sender and receiver must be ready before we start the transfer of data
  - Need to agree on a set of parameters
  - e.g., the Maximum Segment Size (MSS)
- This is signaling
  - It sets up state at the endpoints
  - Like “dialing” for a telephone call

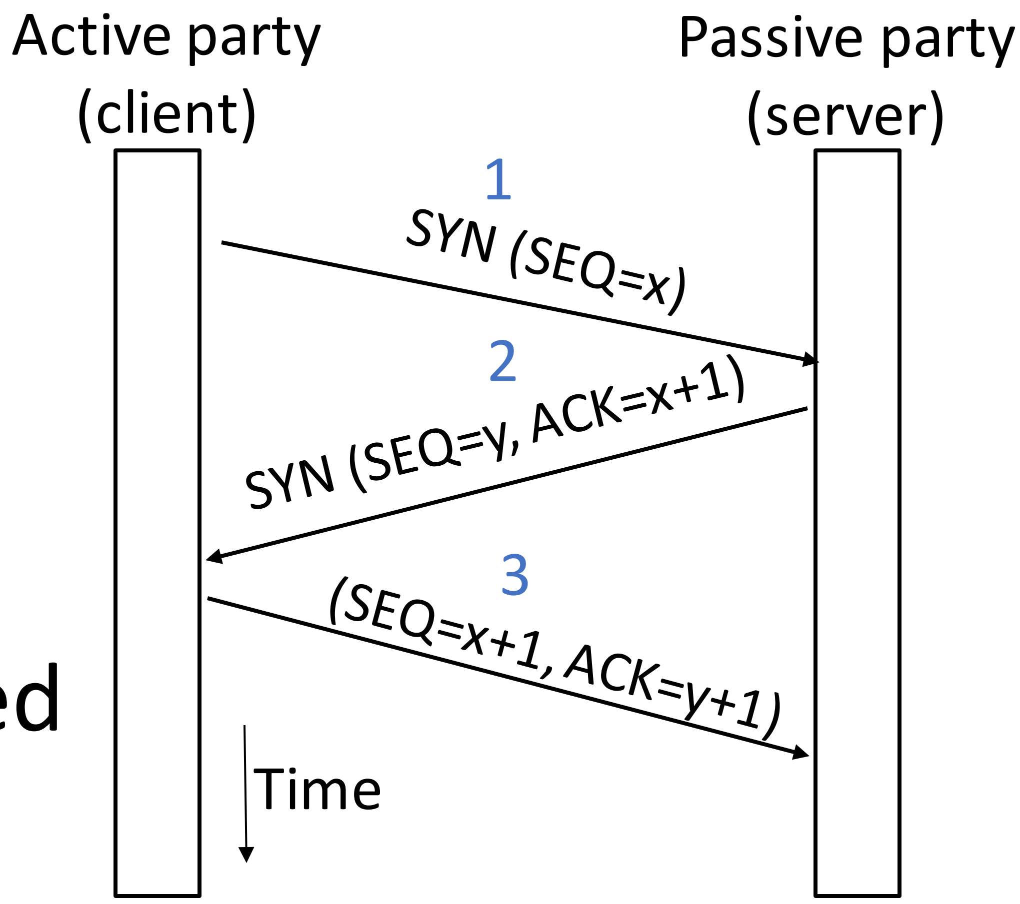
Three-Way Handshake

- Used in TCP; opens connection for data in both directions
- Each side probes the other with a fresh Initial Sequence Number (ISN)
  - Sends on a SYNchronize segment
  - Echo on an ACKnowledge segment
- Chosen to be robust even against delayed duplicates



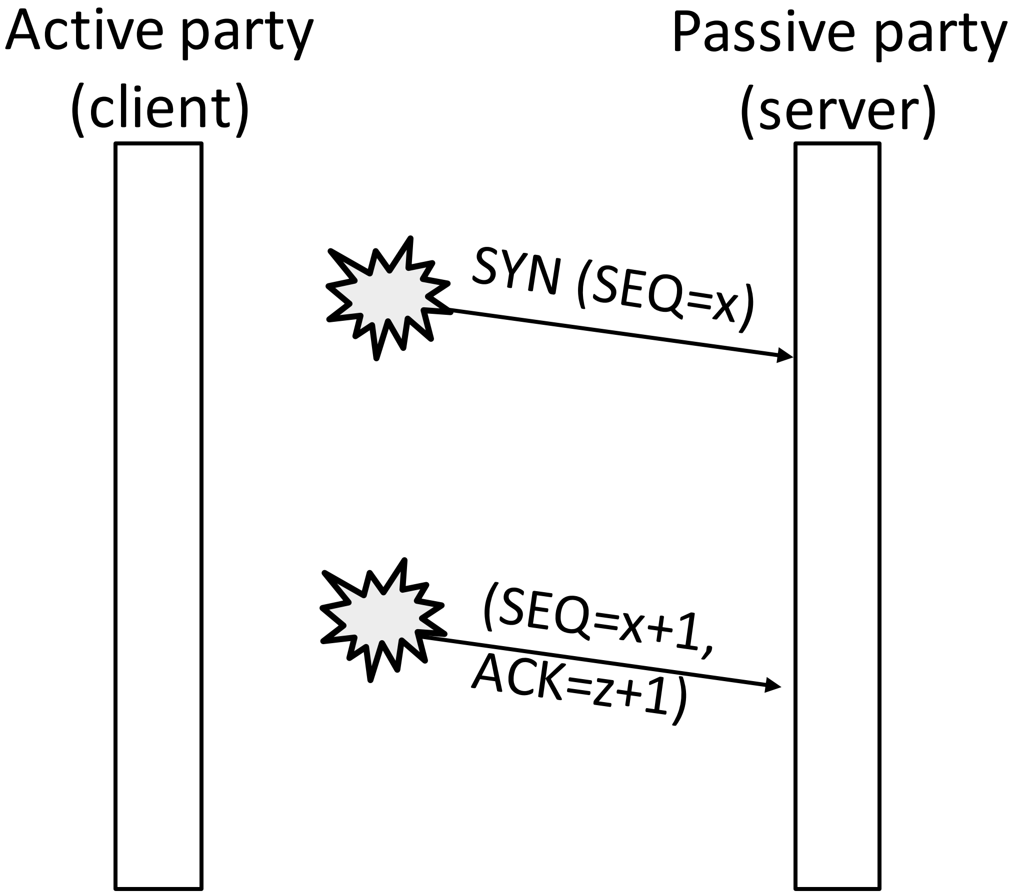
Three-Way Handshake (2)

- Three steps:
  - Client sends SYN(x)
  - Server replies with SYN(y)ACK(x+1)
  - Client replies with ACK(y+1)
  - SYNs are retransmitted if lost
- Sequence and ack numbers carried on further segments



Three-Way Handshake (3)

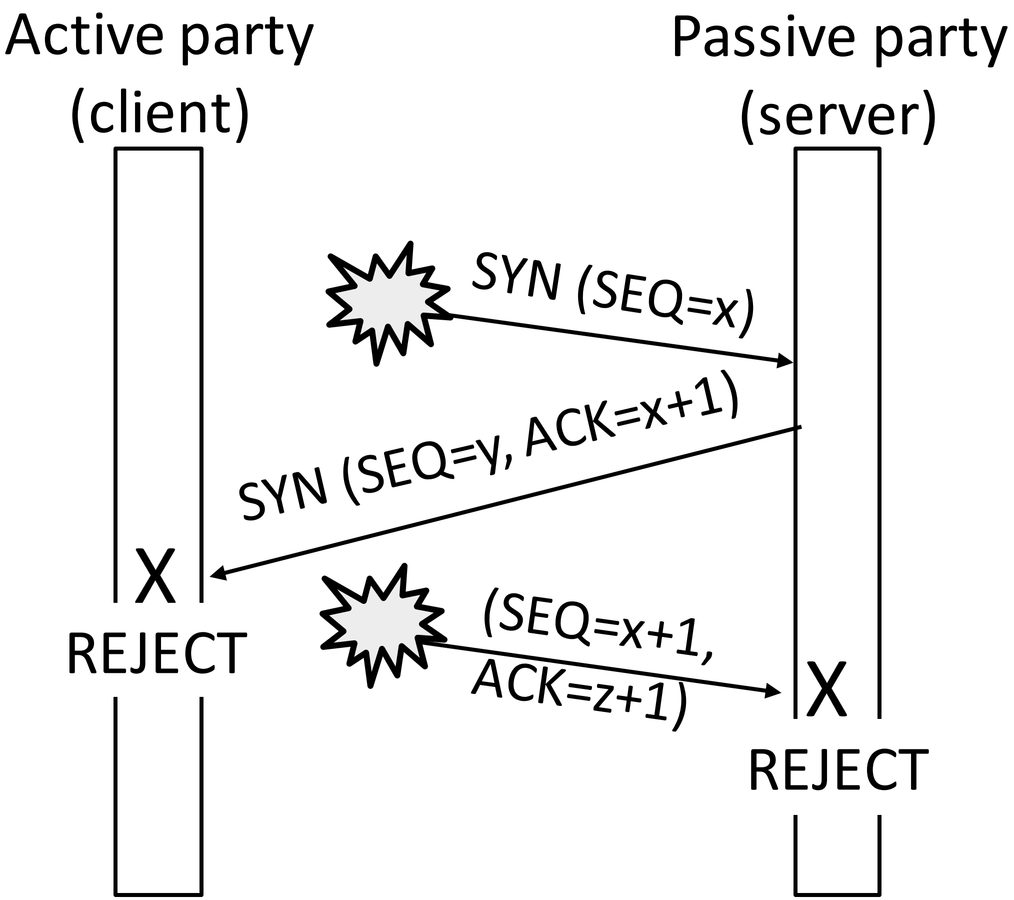
- Suppose delayed, duplicate copies of the SYN and ACK arrive at the server!
  - Improbable, but anyhow …



Three-Way Handshake (4)

- Suppose delayed, duplicate copies of the SYN and ACK arrive at the server!
  - Improbable, but anyhow …

- Connection will be cleanly rejected on both sides



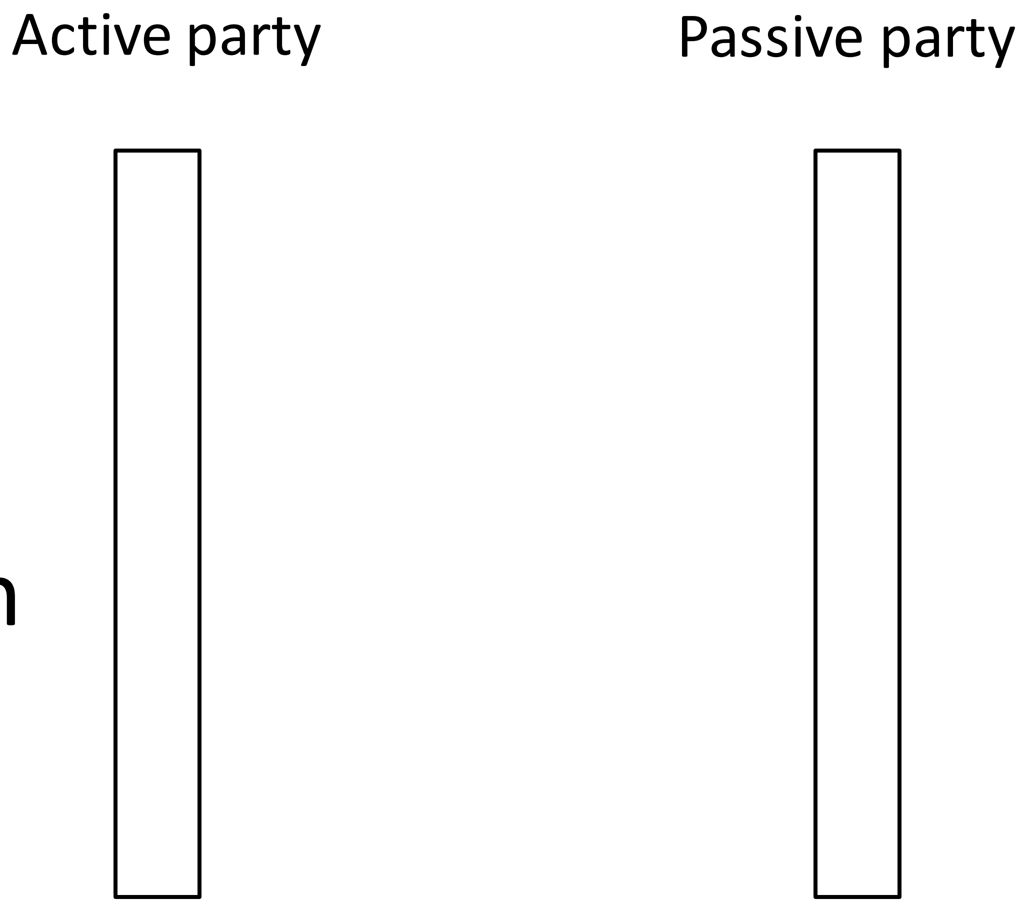
Connection Release

- Orderly release by both parties when done
  - Delivers all pending data and “hangs up”
  - Cleans up state in sender and receiver
- Key problem is to provide reliability while releasing
  - TCP uses a “symmetric” close in which both sides shutdown independently

TCP Connection Release

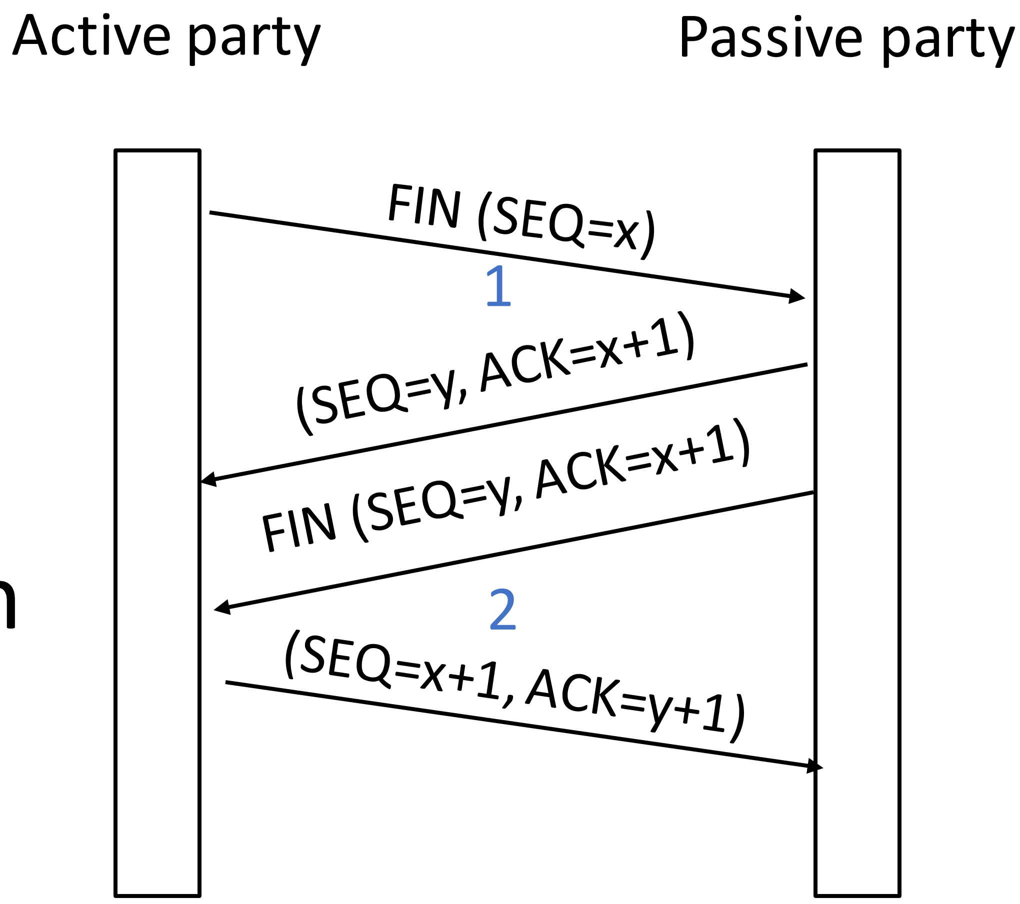
- Two steps:
  - Active sends FIN(x), passive ACKs
  - Passive sends FIN(y), active ACKs
  - FINs are retransmitted if lost

- Each FIN/ACK closes one direction of data transfer



TCP Connection Release (2)

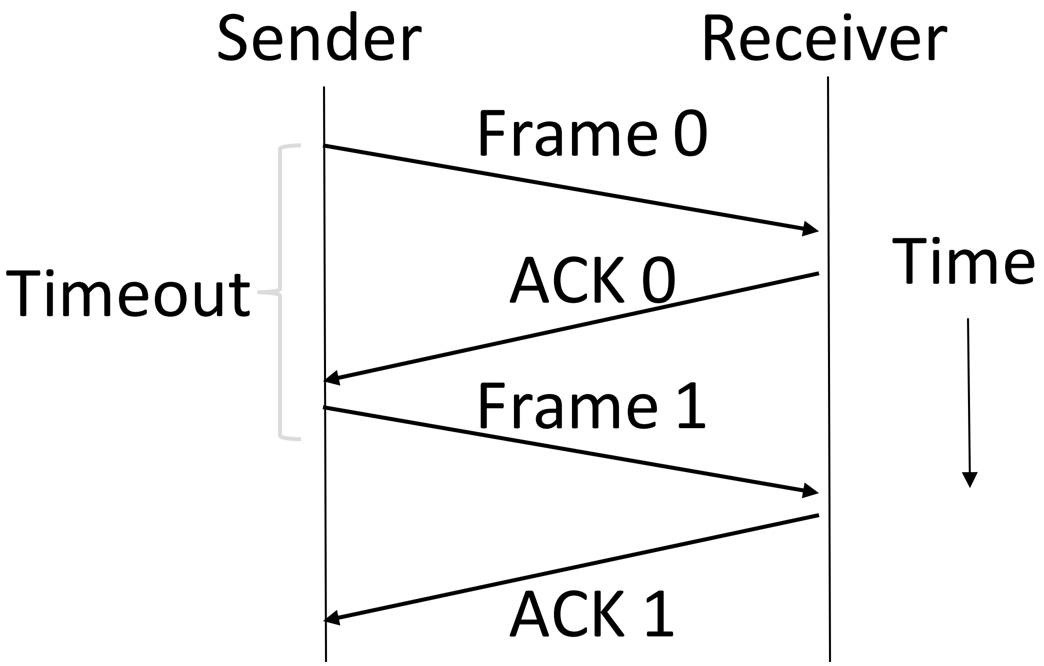
- Two steps:
  - Active sends FIN(x), passive ACKs
  - Passive sends FIN(y), active ACKs
  - FINs are retransmitted if lost
- Each FIN/ACK closes one direction of data transfer



Flow Control

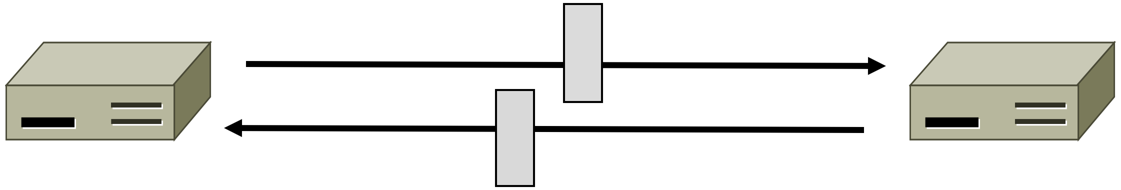
Recall

- ARQ with one message at a time is Stop-and-Wait (normal case below)



Limitation of Stop-and-Wait

- It allows only a single message to be outstanding from the sender:
  - Fine for LAN (only one frame fits in network anyhow)
  - Not efficient for network paths with BD >> 1 packet

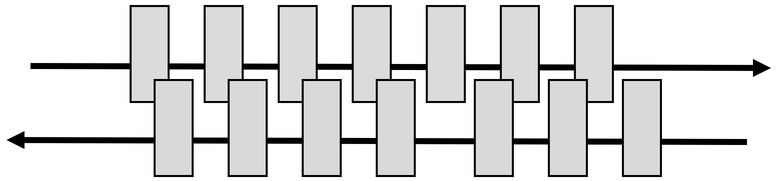


Limitation of Stop-and-Wait (2)

- Example: R=1 Mbps, D = 50 ms, 10kb packets
  - RTT (Round Trip Time) = 2D = 100 ms
  - How many packets/sec?
  - What if R=10 Mbps?

Sliding Window

- Generalization of stop-and-wait
  - Allows W packets to be outstanding
  - Can send W packets per RTT (=2D)



  - Pipelining improves performance
  - Need W=2BD to fill network path

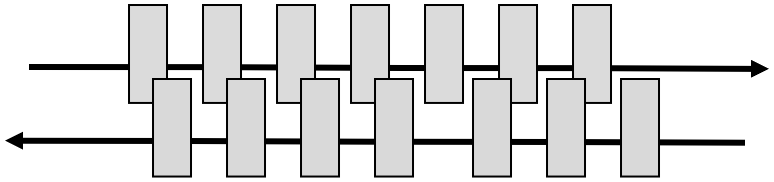
Sliding Window (2)

- What W will use the network capacity?
  - Assume 10kb packets
- Ex: R=1 Mbps, D = 50 ms

- Ex: What if R=10 Mbps?

Sliding Window (3)

- Ex: R=1 Mbps, D = 50 ms
  - 2BD = 10^6 b/sec x 100. 10^{-3} sec = 100 kbit
  - W = 2BD = 10 packets of 1250 bytes



- Ex: What if R=10 Mbps?
  - 2BD = 1000 kbit
  - W = 2BD = 100 packets of 1250 bytes

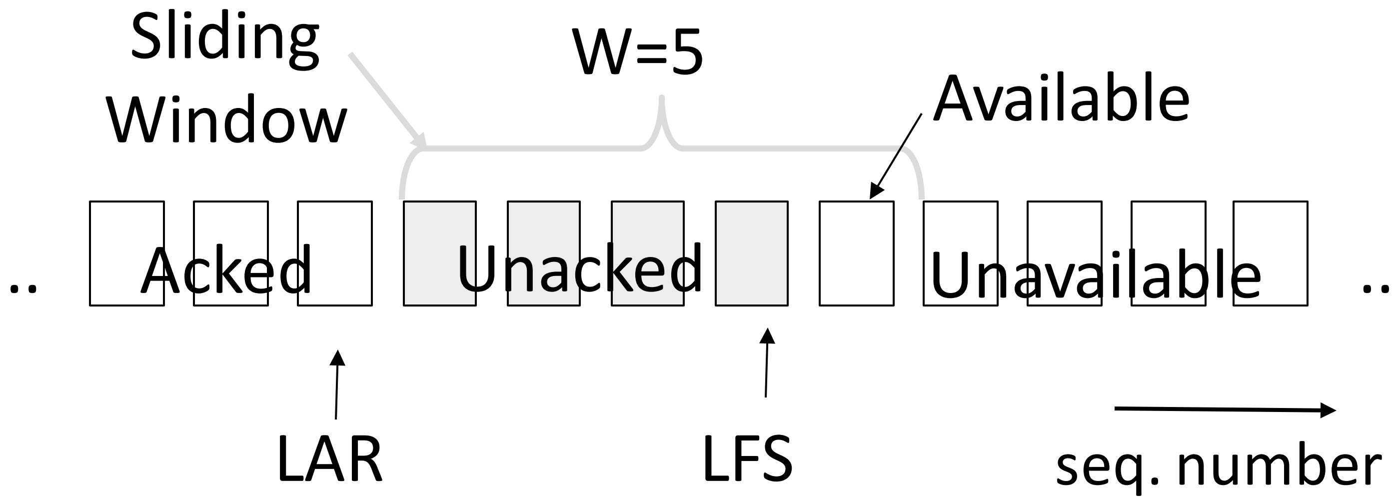
Sliding Window Protocol

- Many variations, depending on how buffers, acknowledgements, and retransmissions are handled

- <u>Go-Back-N</u>
  - Simplest version, can be inefficient
- <u>Selective Repeat</u>
  - More complex, better performance

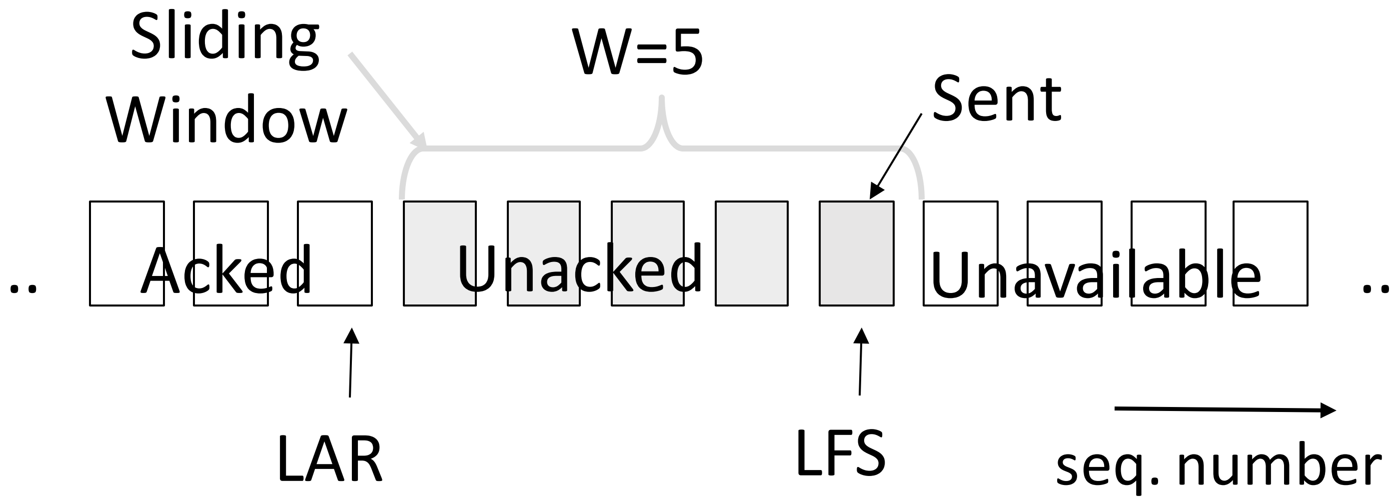
Sliding Window – Sender

- Sender buffers up to W segments until they are acknowledged
  - LFS=LAST FRAME SENT, LAR=LAST ACK REC'D
  - Sends while LFS – LAR ≤ W



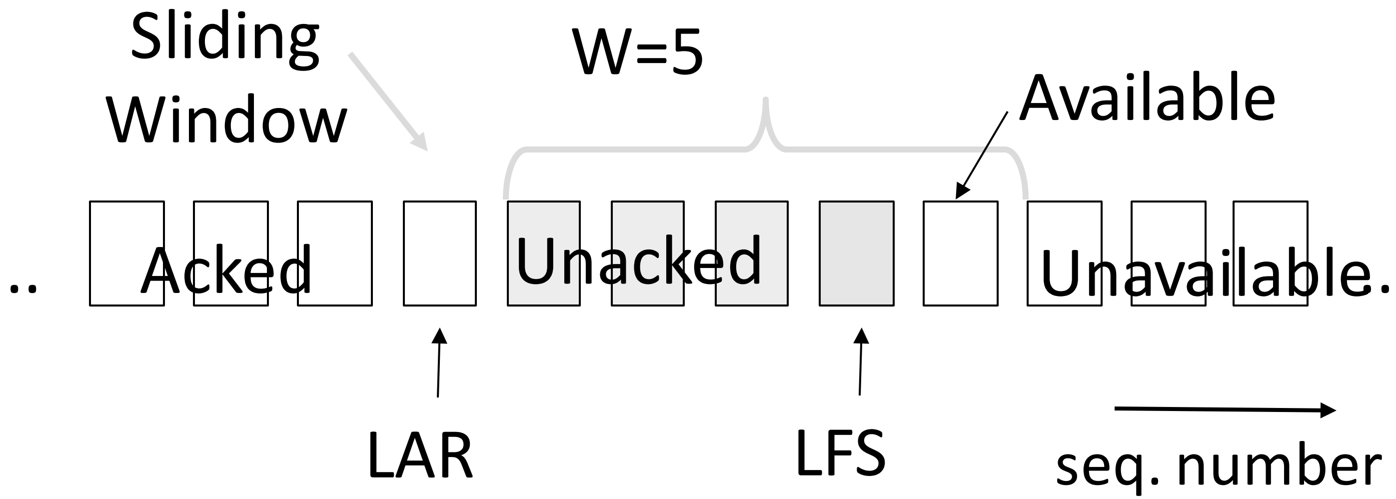
Sliding Window – Sender (2)

- Transport accepts another segment of data from the Application ...
  - Transport sends it (as LFS–LAR 🖬 5)



Sliding Window – Sender (3)

- Next higher ACK arrives from peer…
  - Window advances, buffer is freed
  - LFS–LAR 4 (can send one more)



Sliding Window – Go-Back-N

- Receiver keeps only a single packet buffer for the next segment
  - State variable, LAS = LAST ACK SENT
- On receive:
  - If seq. number is LAS+1, accept and pass it to app, update LAS, send ACK
  - Otherwise discard (as out of order)

Sliding Window – Selective Repeat

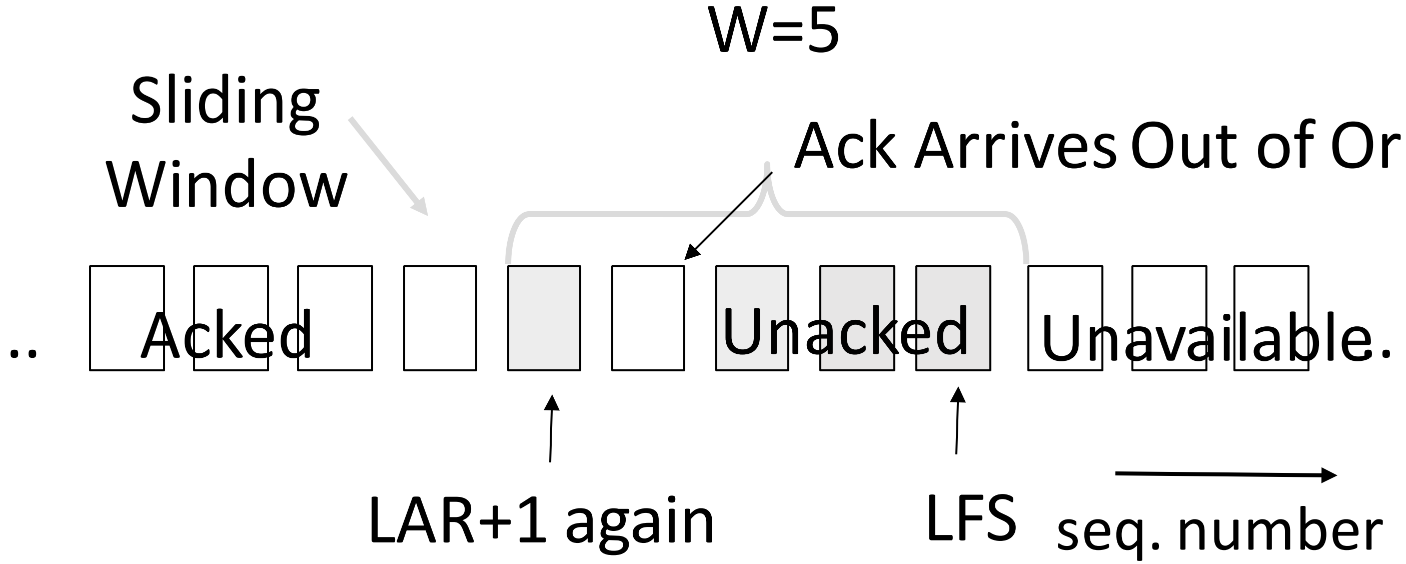
- Receiver passes data to app in order, and buffers out-of-order segments to reduce retransmissions
- ACK conveys highest in-order segment, plus hints about out-of-order segments
- TCP uses a selective repeat design; we'll see the details later

Sliding Window – Selective Repeat (2)

- Buffers W segments, keeps state variable LAS = LAST ACK SENT
- On receive:
  - Buffer segments [LAS+1, LAS+W]
  - Send app in-order segments from LAS+1, and update LAS
  - Send ACK for LAS regardless

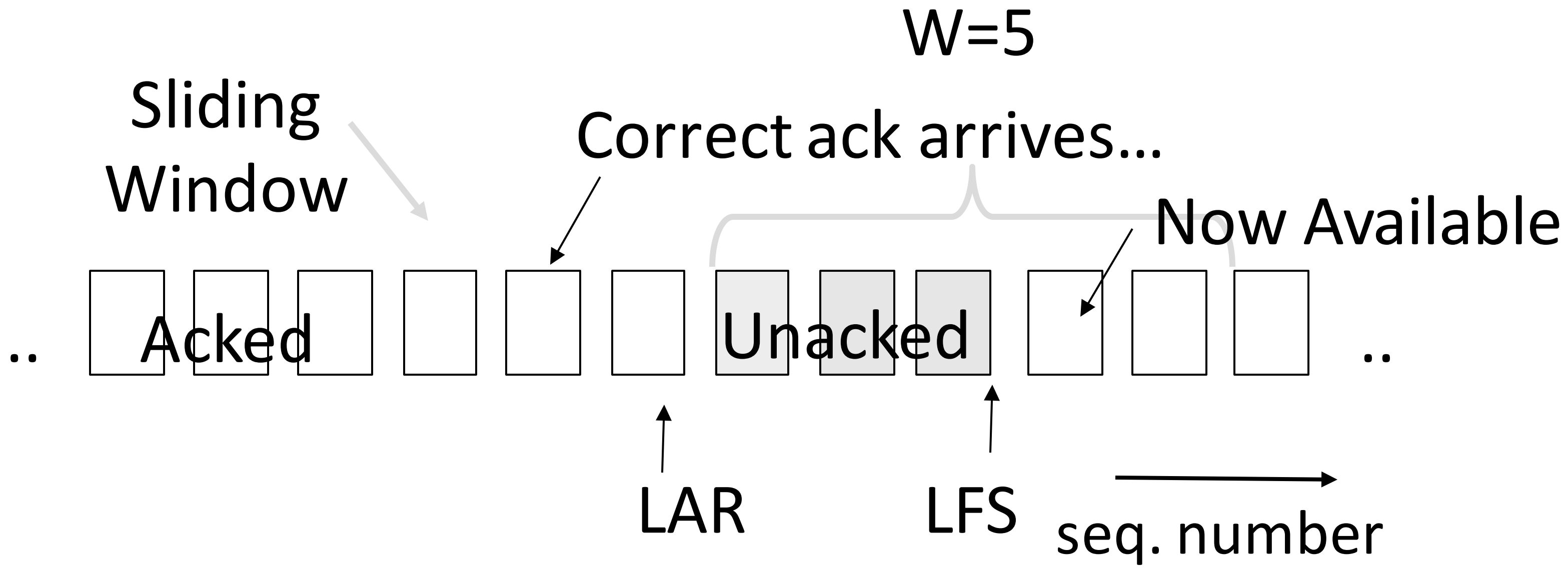
Sliding Window – Selective Retransmission (3)

- Keep normal sliding window
- If receive something out of order
  - Send last unacked packet again!



Sliding Window – Selective Retransmission (4)

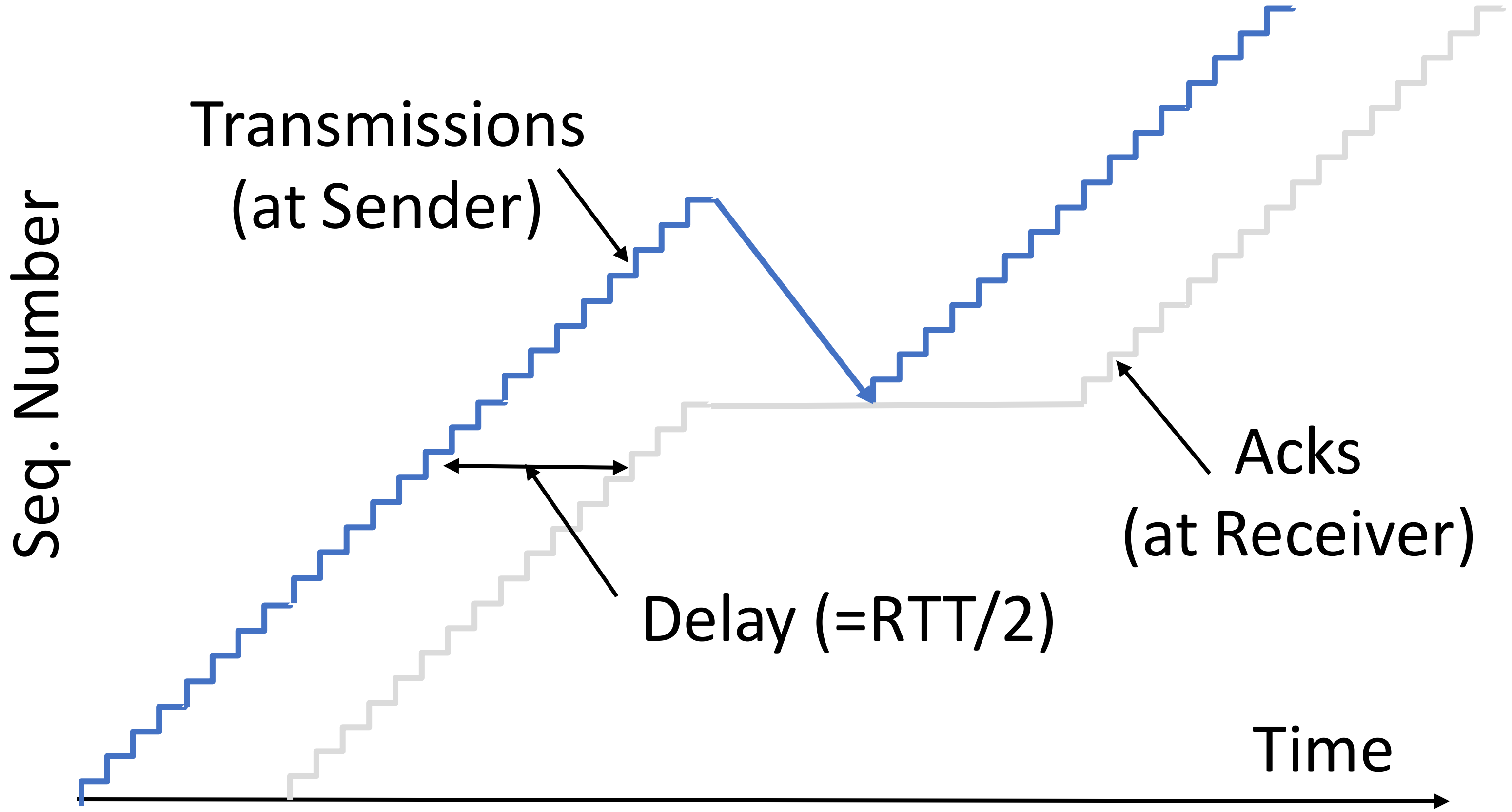
- Keep normal sliding window
- If correct packet arrives, move window and LAR, send more messages



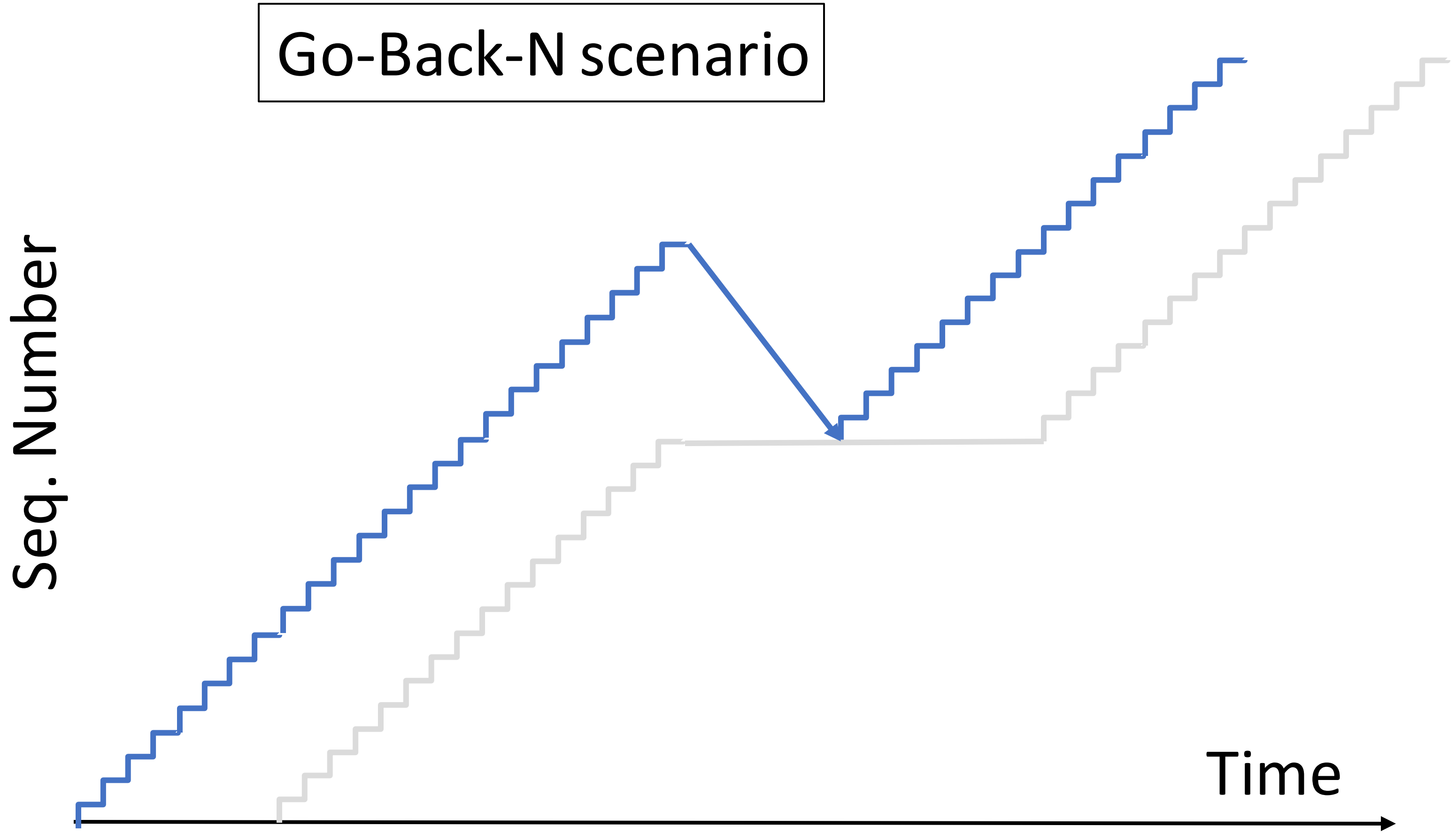
Sliding Window – Retransmissions

- Go-Back-N uses a single timer to detect losses
  - On timeout, resends buffered packets starting at LAR+1
- Selective Repeat uses a timer per unacked segment to detect losses
  - On timeout for segment, resend it
  - Hope to resend fewer segments

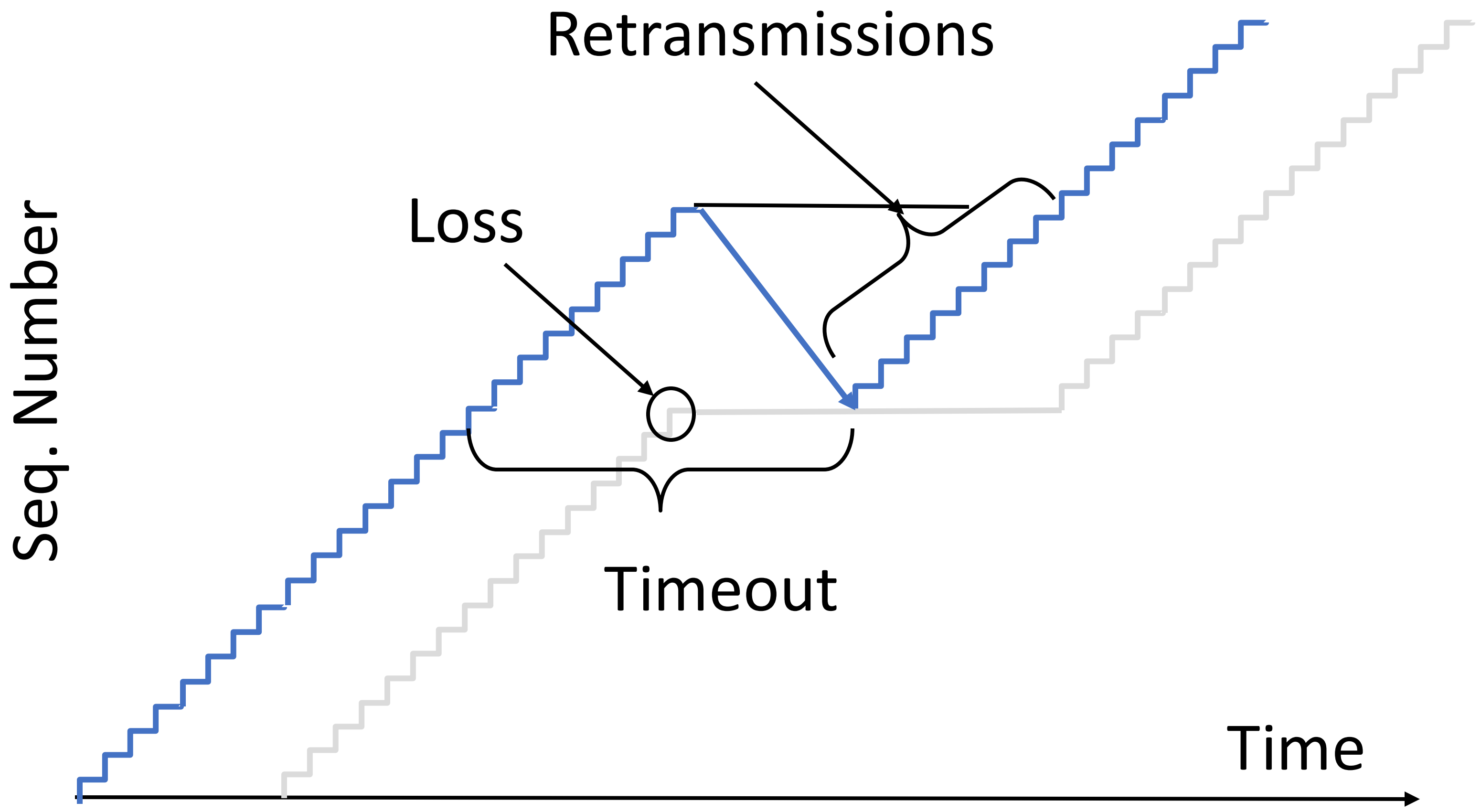
Sequence Time Plot



Sequence Time Plot (2)

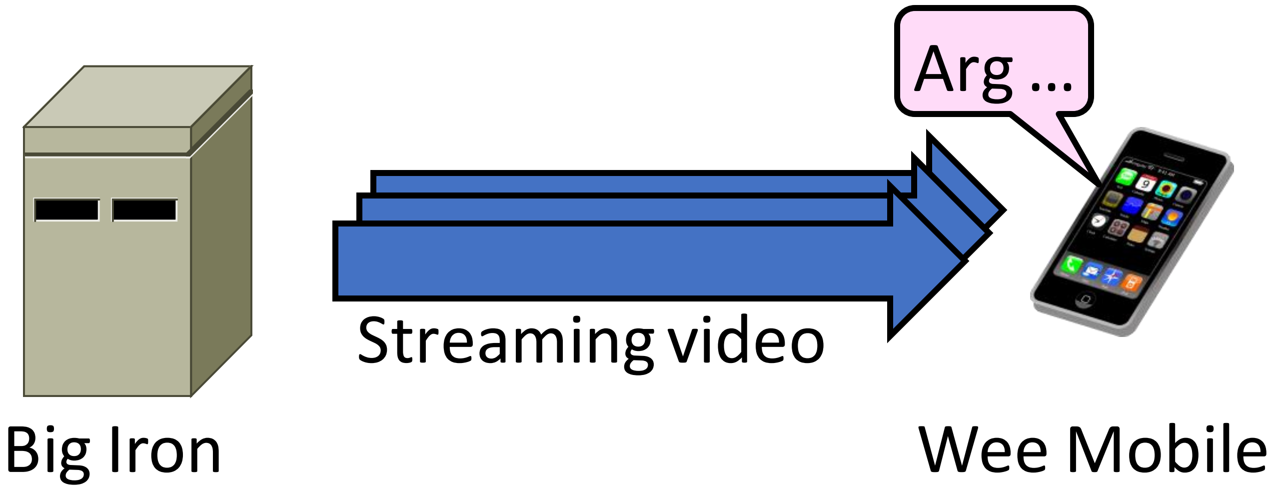


Sequence Time Plot (3)



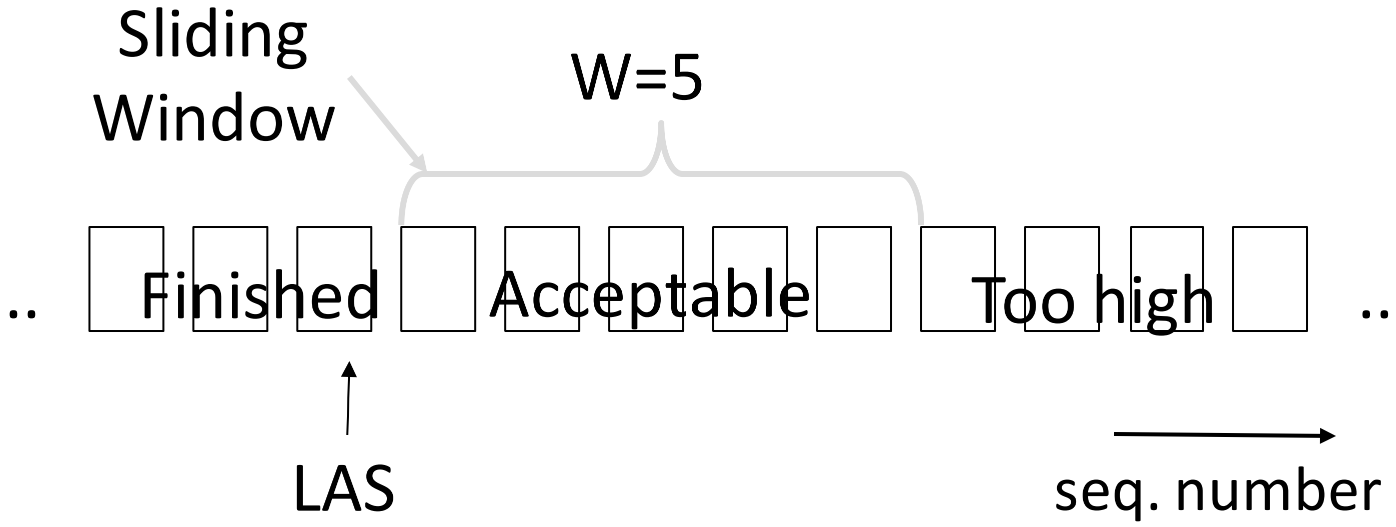
Problem

- Sliding window has pipelining to keep network busy
  - What if the receiver is overloaded?



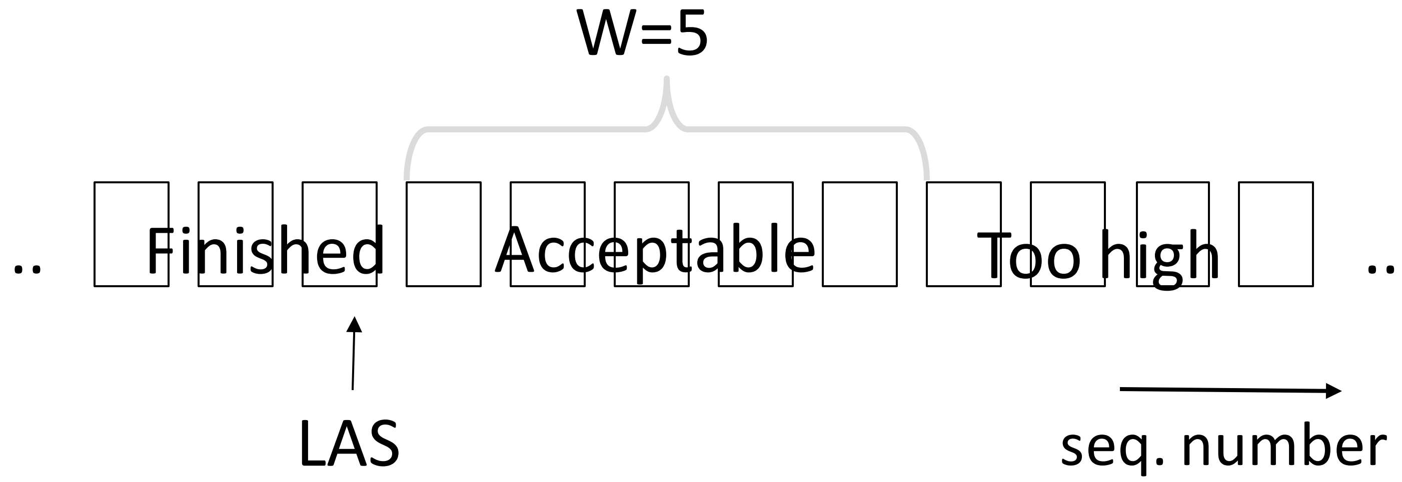
Sliding Window – Receiver

- Consider receiver with W buffers
  - LAS=LAST ACK SENT, app pulls in-order data from buffer with recv() call



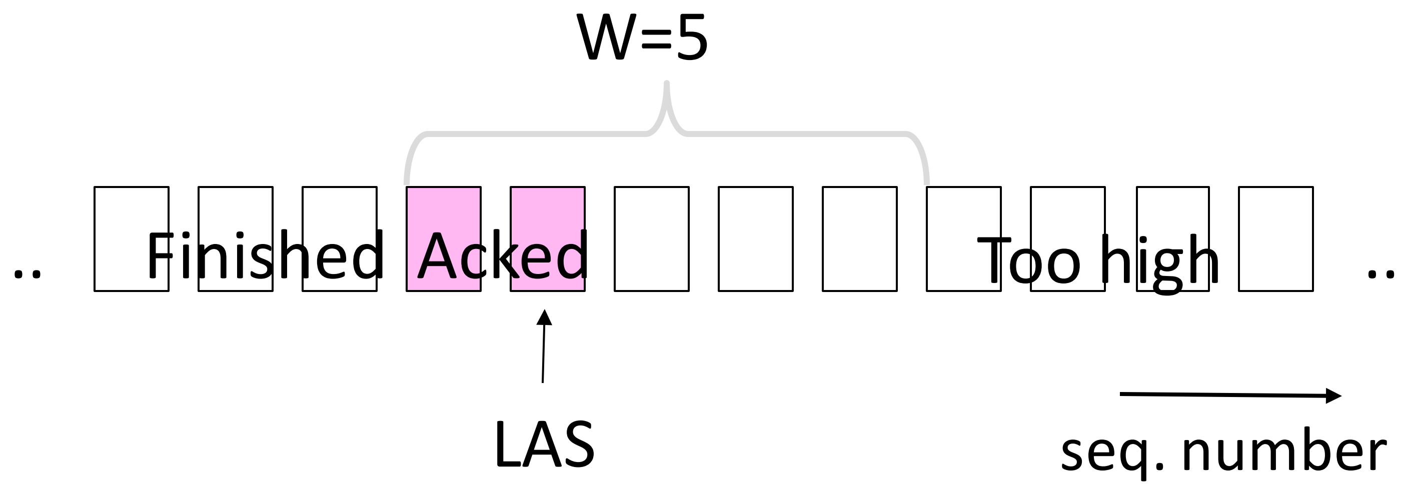
Sliding Window – Receiver (2)

- Suppose the next two segments arrive but app does not call recv()



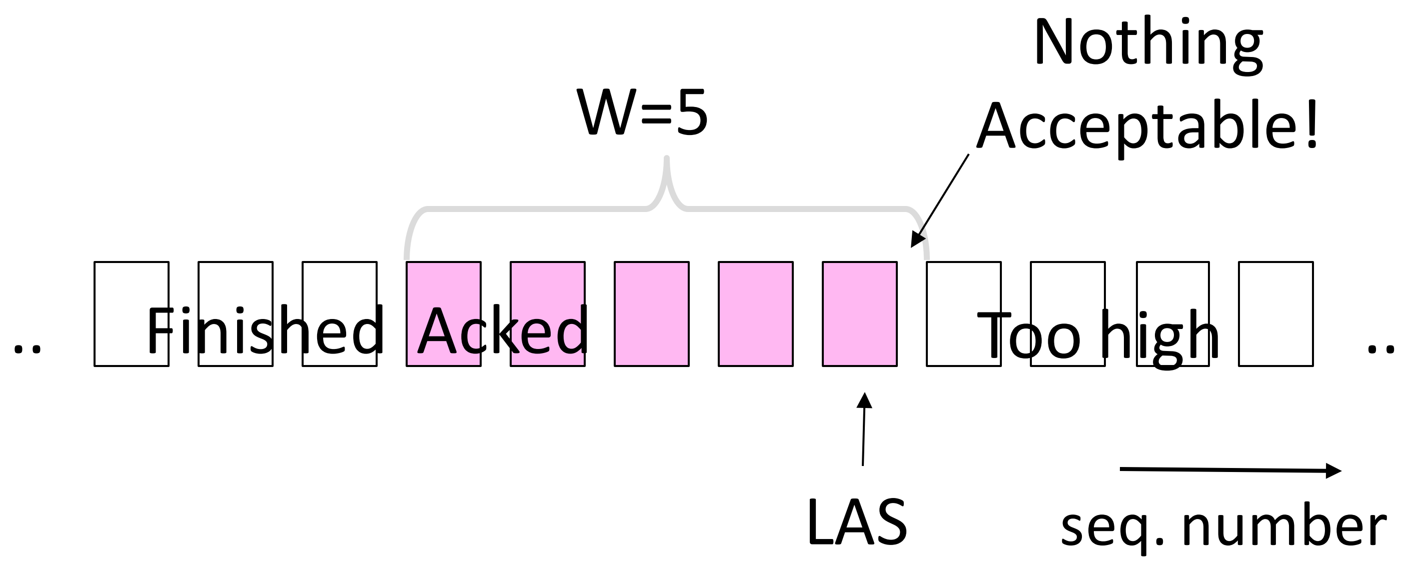
Sliding Window – Receiver (3)

- Suppose the next two segments arrive but app does not call recv()
  - LAS rises, but we can't slide window!



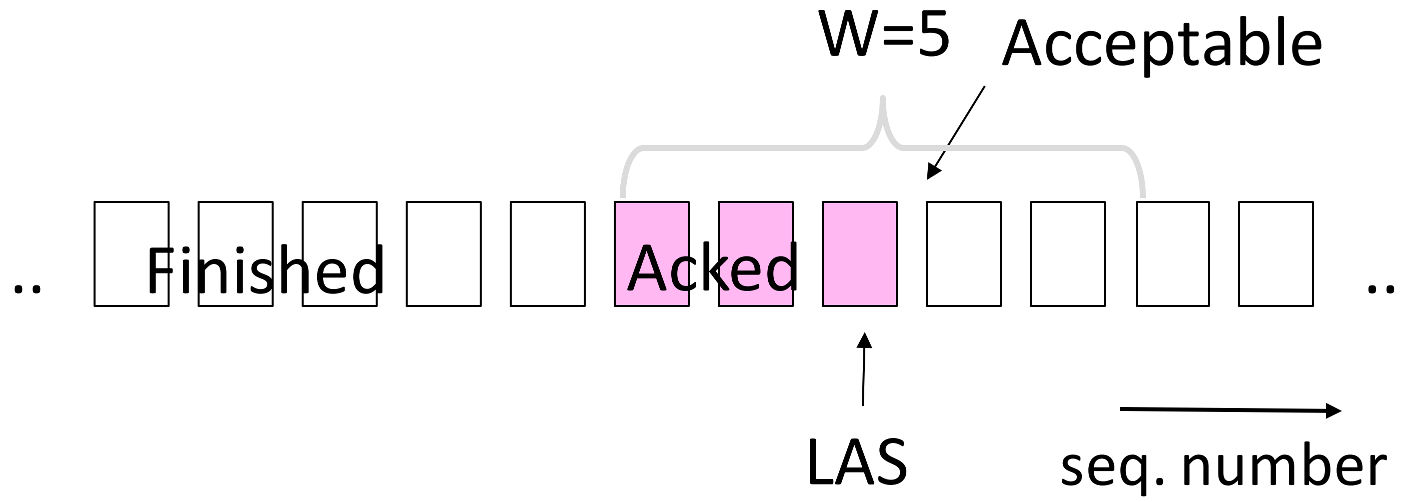
Sliding Window – Receiver (4)

- Further segments arrive (in order) we fill buffer
  - Must drop segments until app recvs!



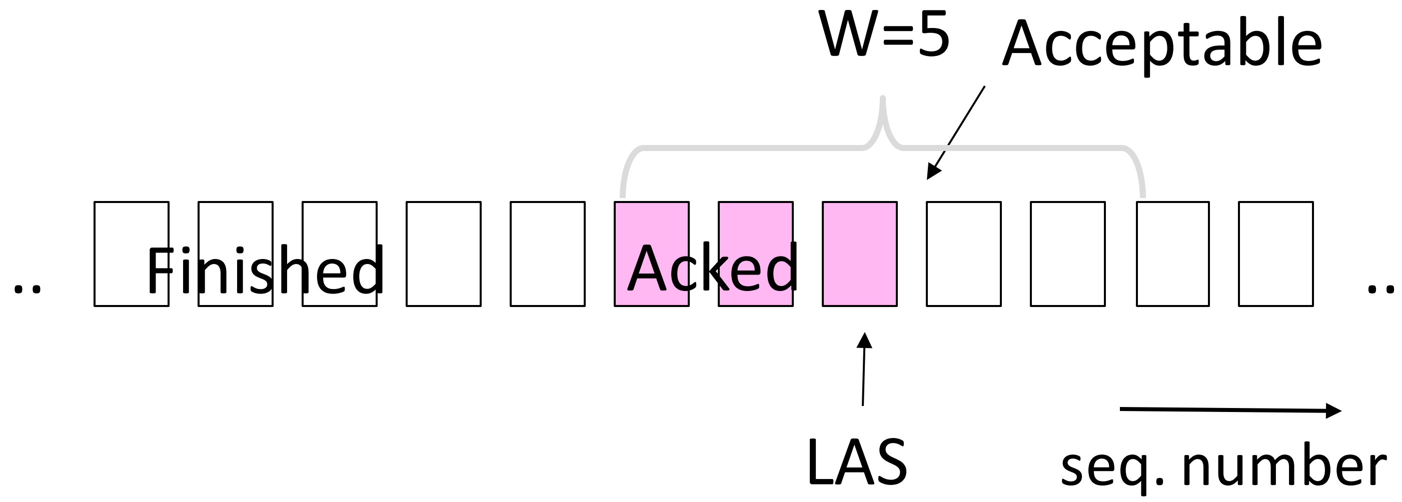
Sliding Window – Receiver (5)

- App recv() takes two segments
  - Window slides (phew)



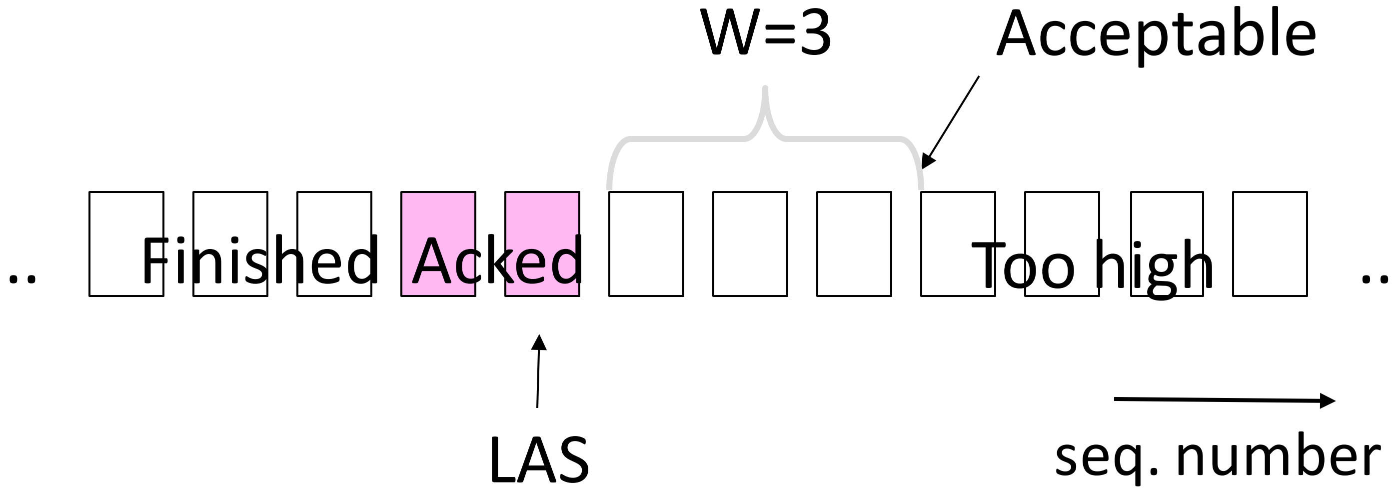
Flow Control

- Avoid loss at receiver by telling sender the available buffer space
  - WIN=#Acceptable, not W (from LAS)



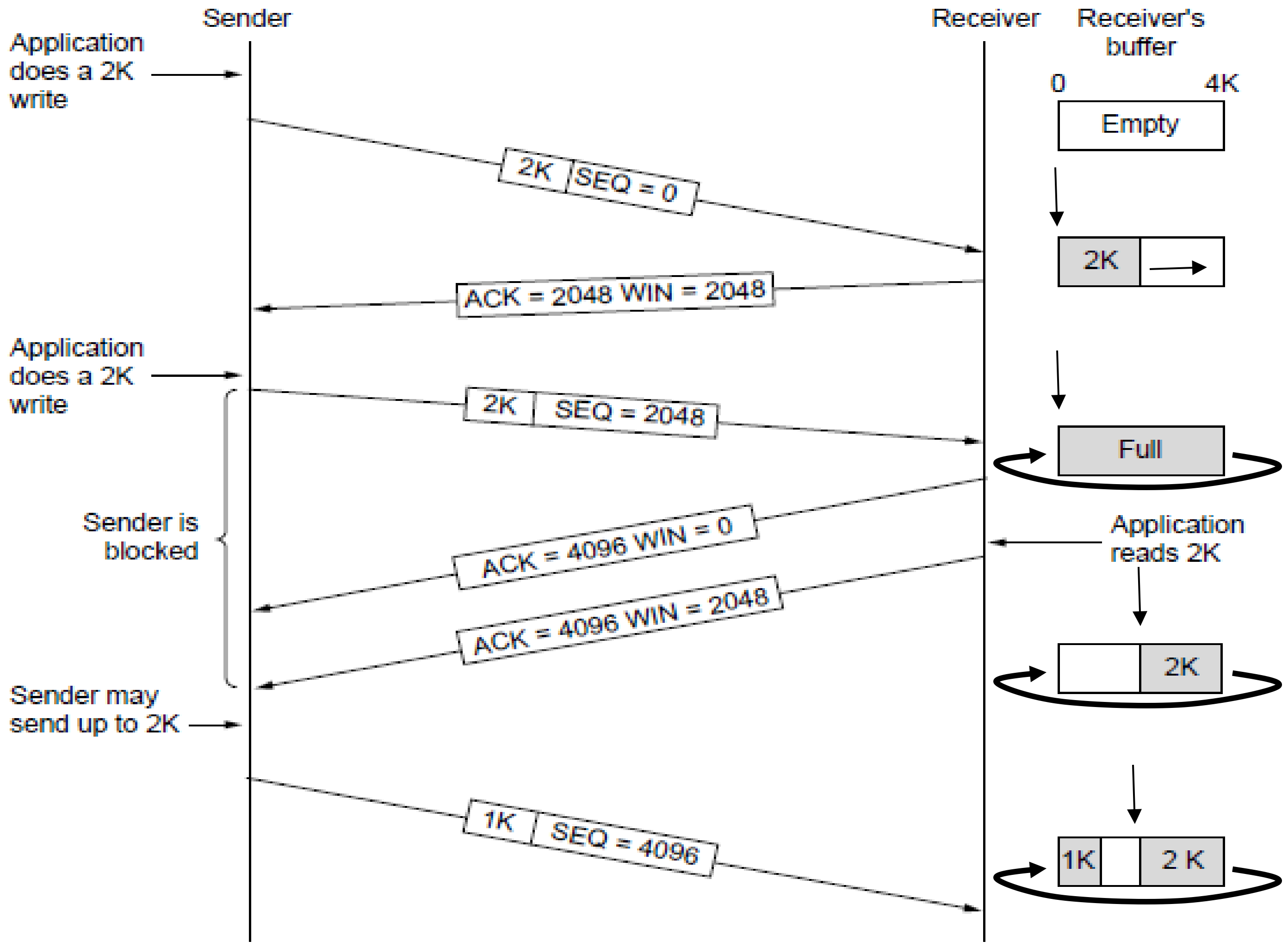
Flow Control (2)

- Sender uses lower of the sliding window and flow control window (WIN) as the effective window size



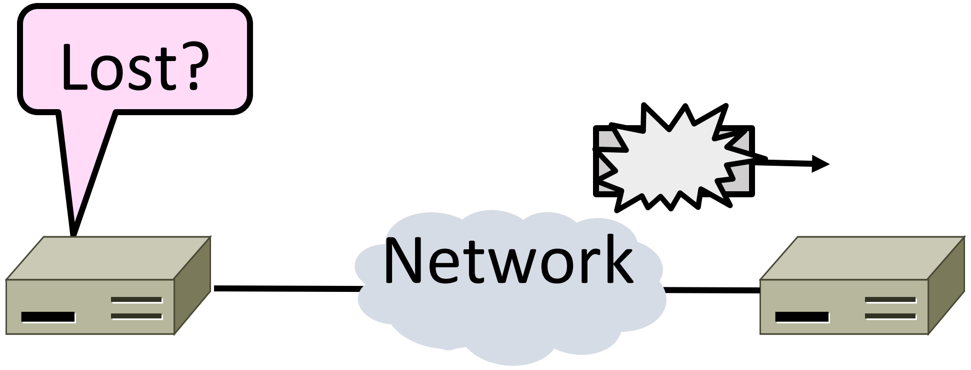
Flow Control (3)

- TCP-style example
  - SEQ/ACK sliding window
  - Flow control with WIN
  - SEQ + length < ACK+WIN
  - 4KB buffer at receiver
  - Circular buffer of bytes



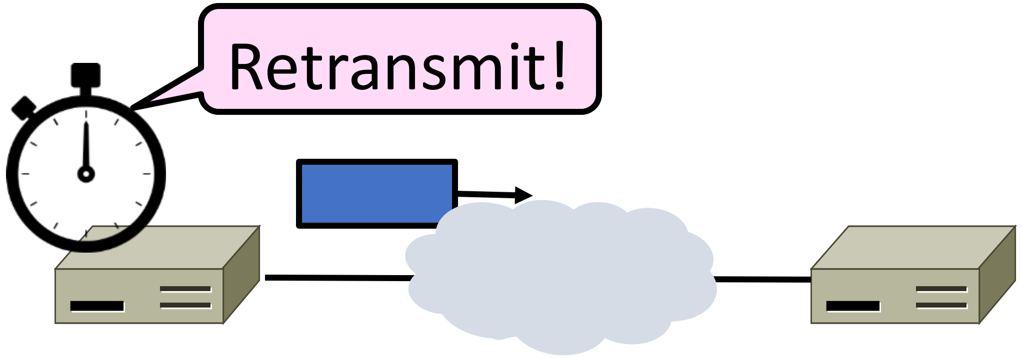
Topic

- How to set the timeout for sending a retransmission
  - Adapting to the network path



Retransmissions

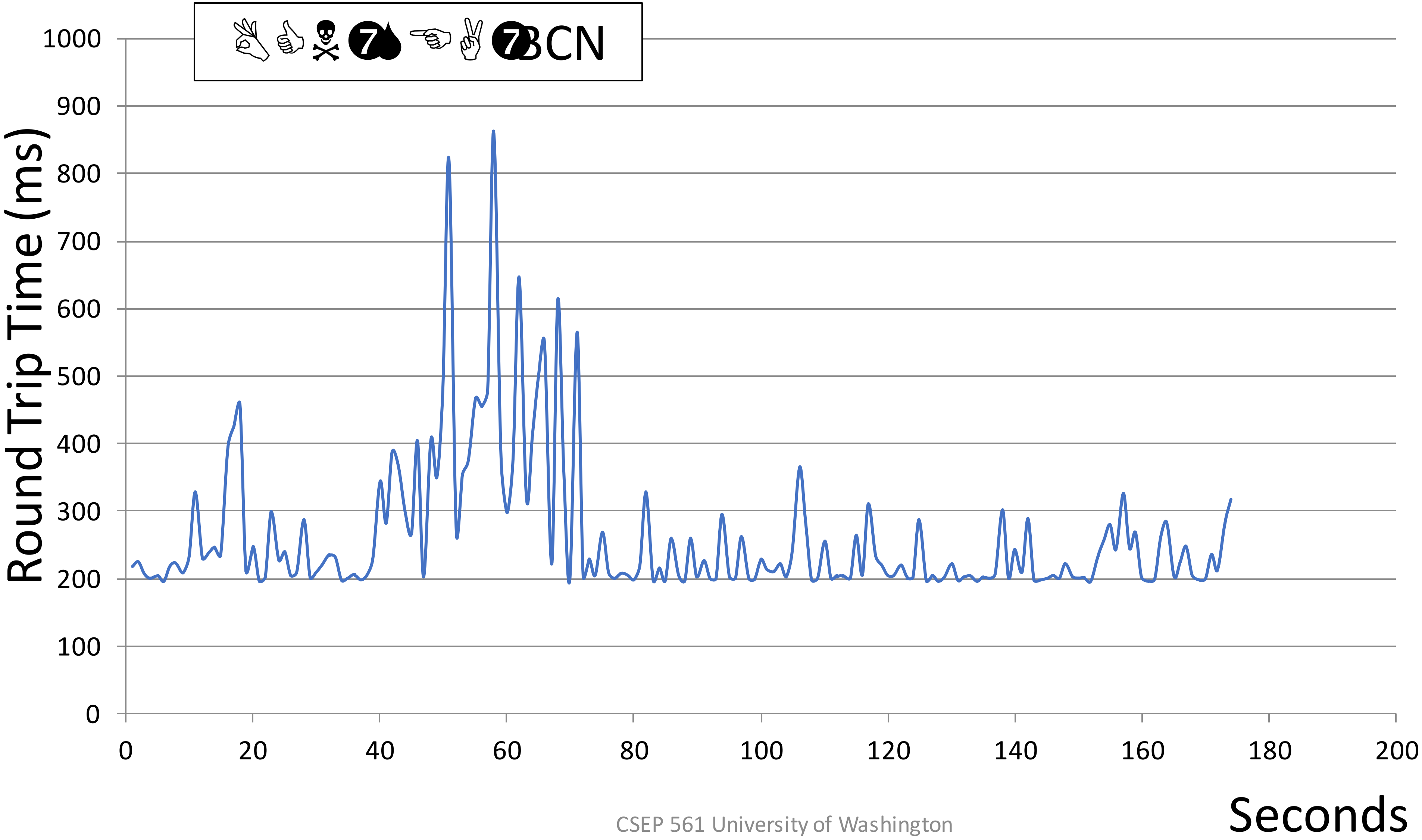
- With sliding window, detecting loss with timeout
  - Set timer when a segment is sent
  - Cancel timer when ack is received
  - If timer fires, retransmit data as lost



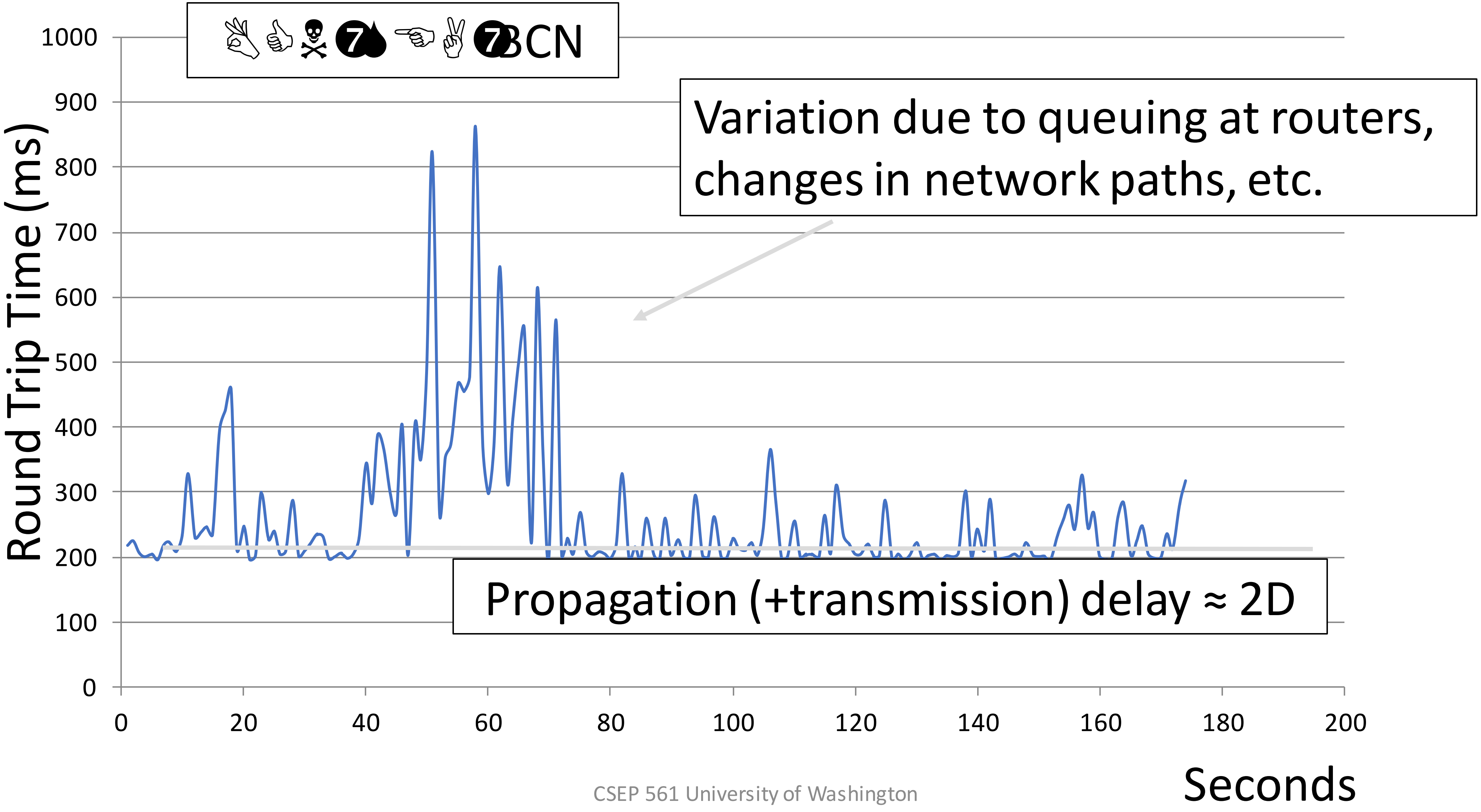
Timeout Problem

- Timeout should be “just right”
  - Too long wastes network capacity
  - Too short leads to spurious resends
  - But what is “just right”?
- Easy to set on a LAN (Link)
  - Short, fixed, predictable RTT
- Hard on the Internet (Transport)
  - Wide range, variable RTT

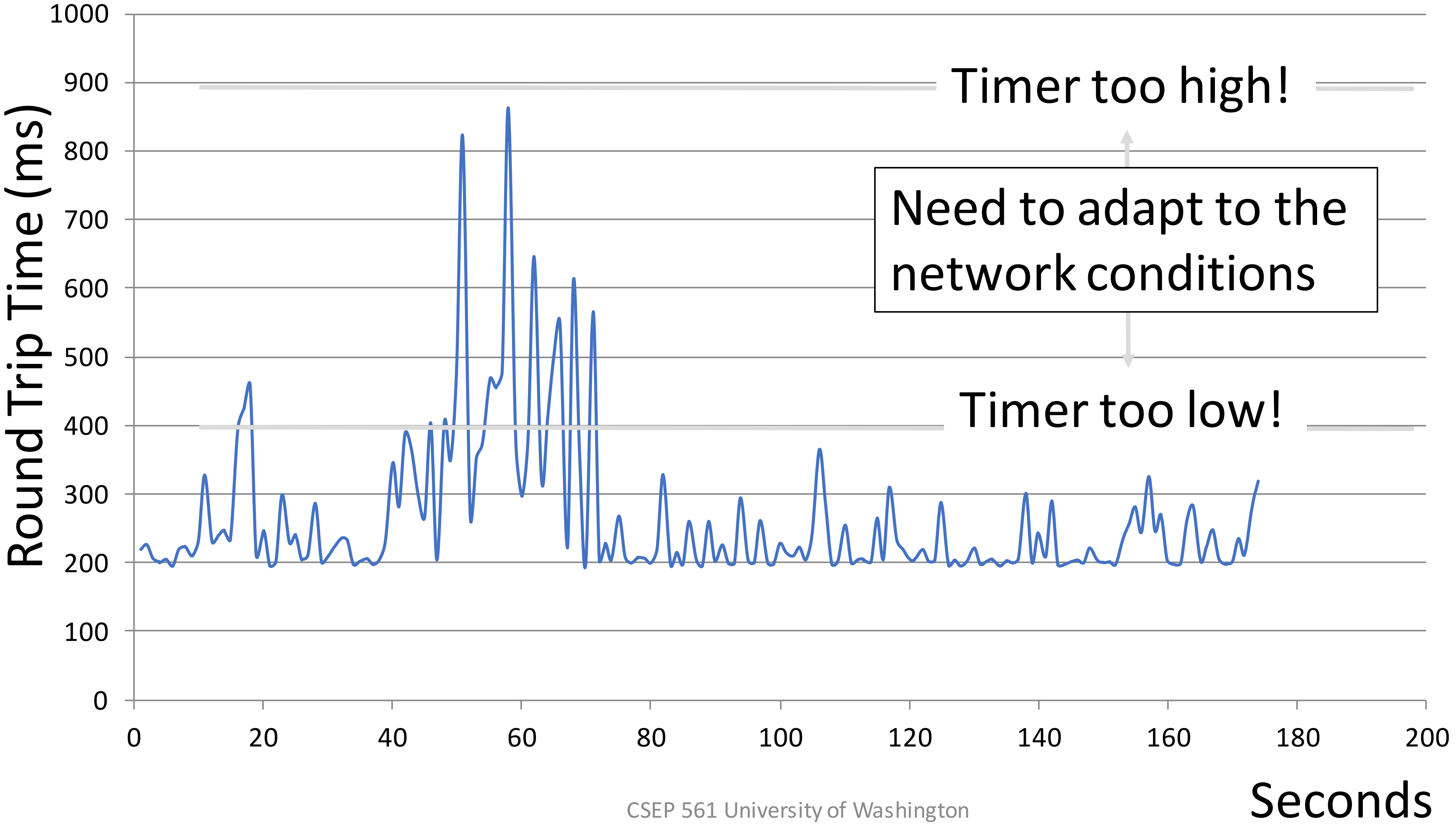
Example of RTTs



Example of RTTs (2)



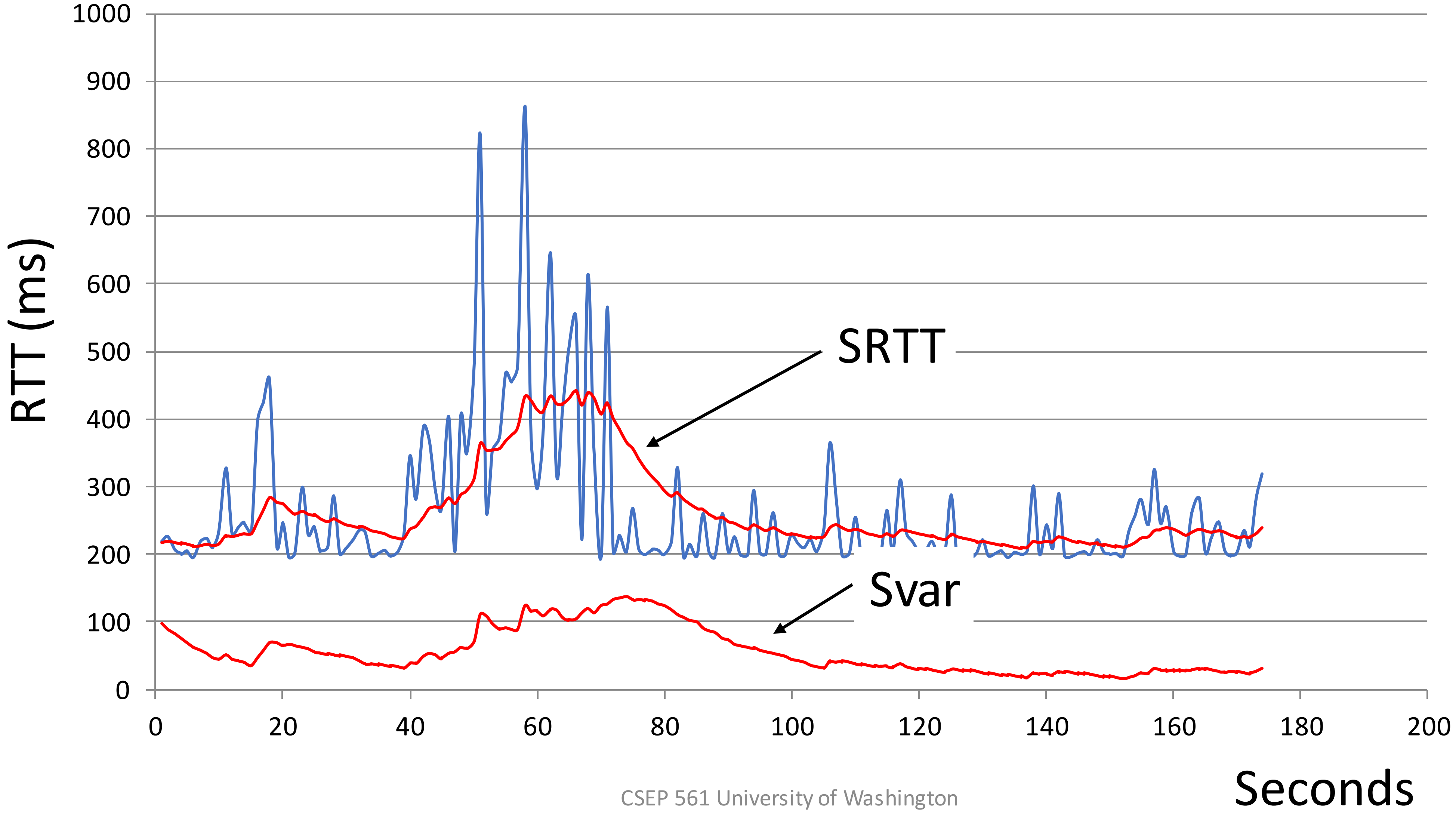
Example of RTTs (3)



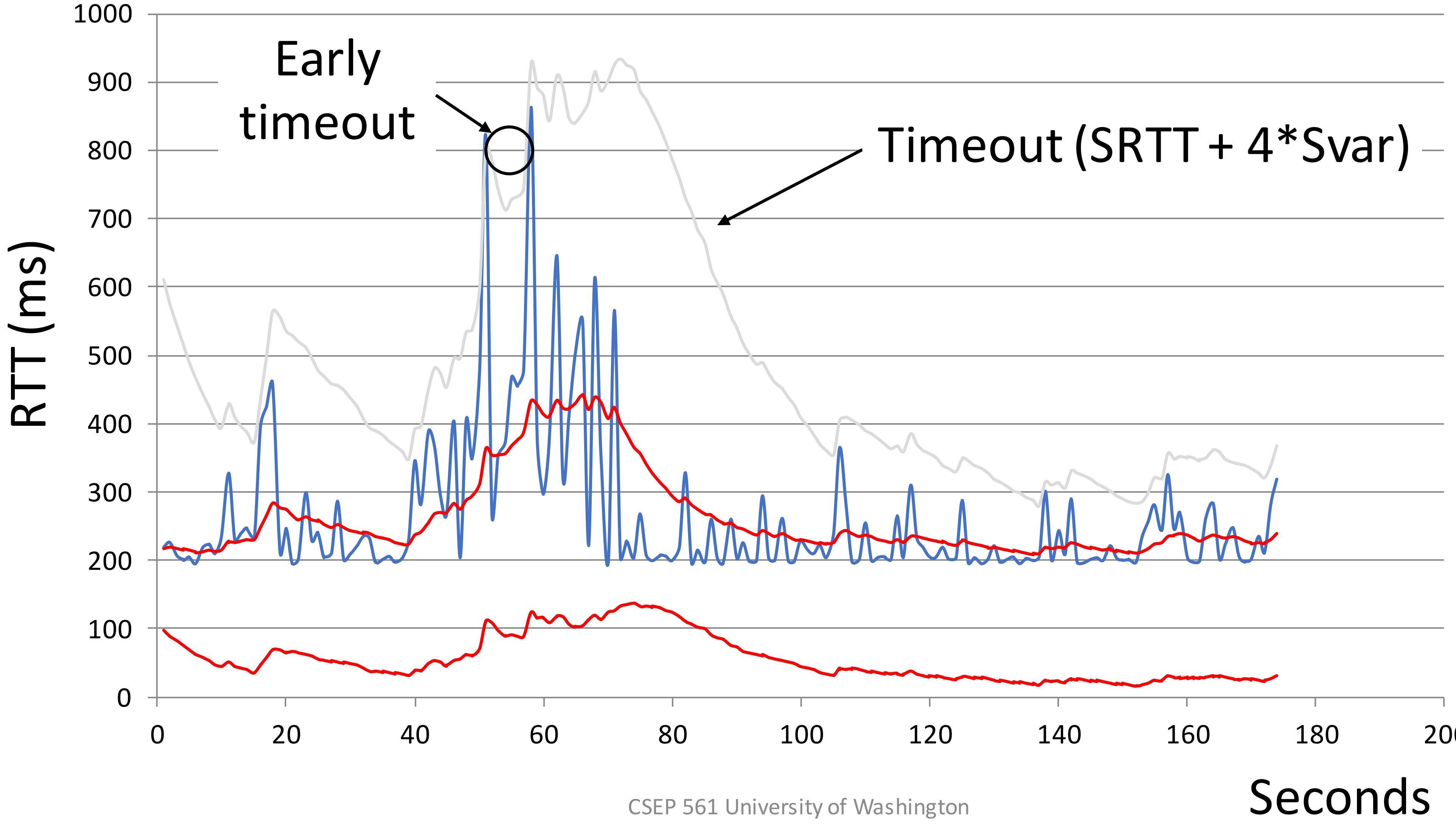
Adaptive Timeout

- Smoothed estimates of the RTT (1) and variance in RTT (2)
  - Update estimates with a moving average
  1. $SRTT_{N+1} = 0.9*SRTT_N + 0.1*RTT_{N+1}$
  2. $Svar_{N+1} = 0.9*Svar_N + 0.1*|RTT_{N+1} - SRTT_{N+1}|$
- Set timeout to a multiple of estimates
  - To estimate the upper RTT in practice
  - TCP $Timeout_N = SRTT_N + 4*Svar_N$

Example of Adaptive Timeout



Example of Adaptive Timeout (2)



Adaptive Timeout (2)

- Simple to compute, does a good job of tracking actual RTT
  - Little “headroom” to lower
  - Yet very few early timeouts

- Turns out to be important for good performance and robustness

Congestion

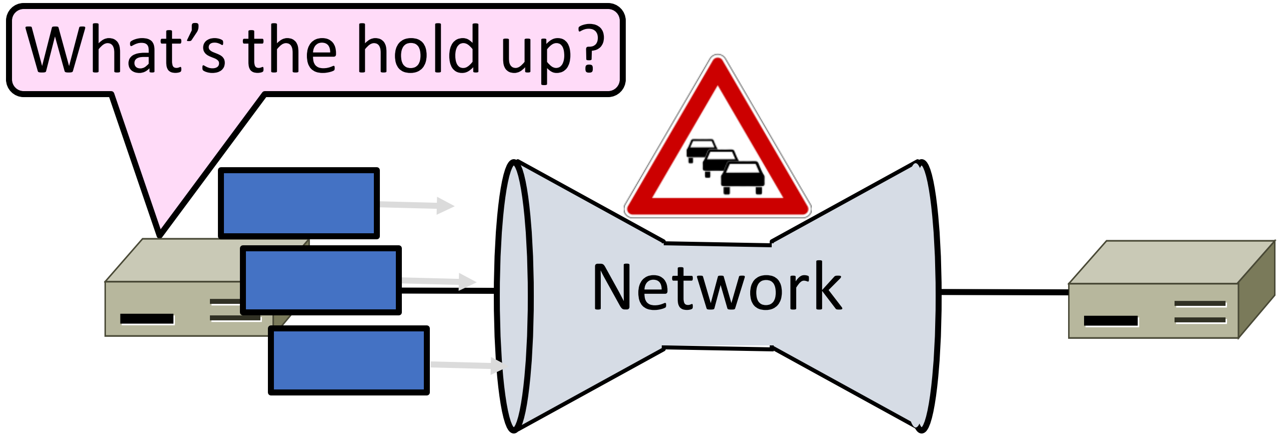
TCP to date:

- We can set up a connection (connection establishment)
- Tear down a connection (connection release)
- Keep the sending and receiving buffers from overflowing (flow control)

What’s missing?

Network Congestion

- A “traffic jam” in the network
  - Later we will learn how to control it

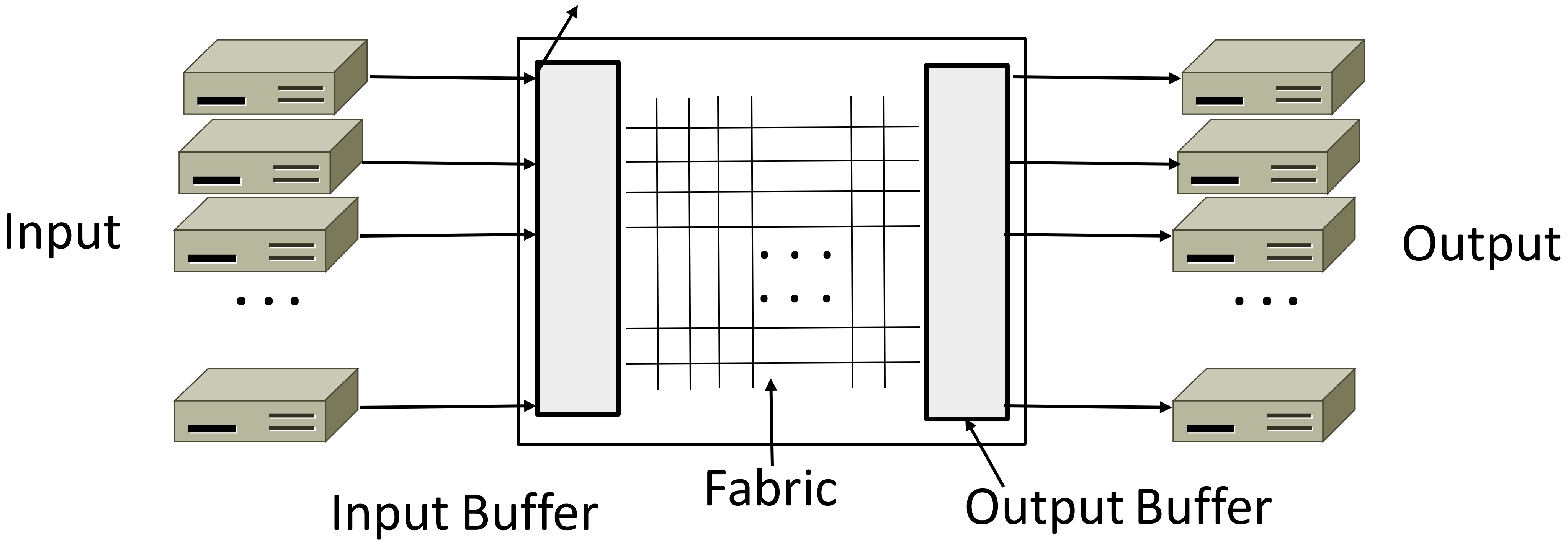


Congestion Collapse in the 1980s

- Early TCP used fixed size window (e.g., 8 packets)
  - Initially fine for reliability
- But something happened as the ARPANET grew
  - Links stayed busy but transfer rates fell by orders of magnitude!

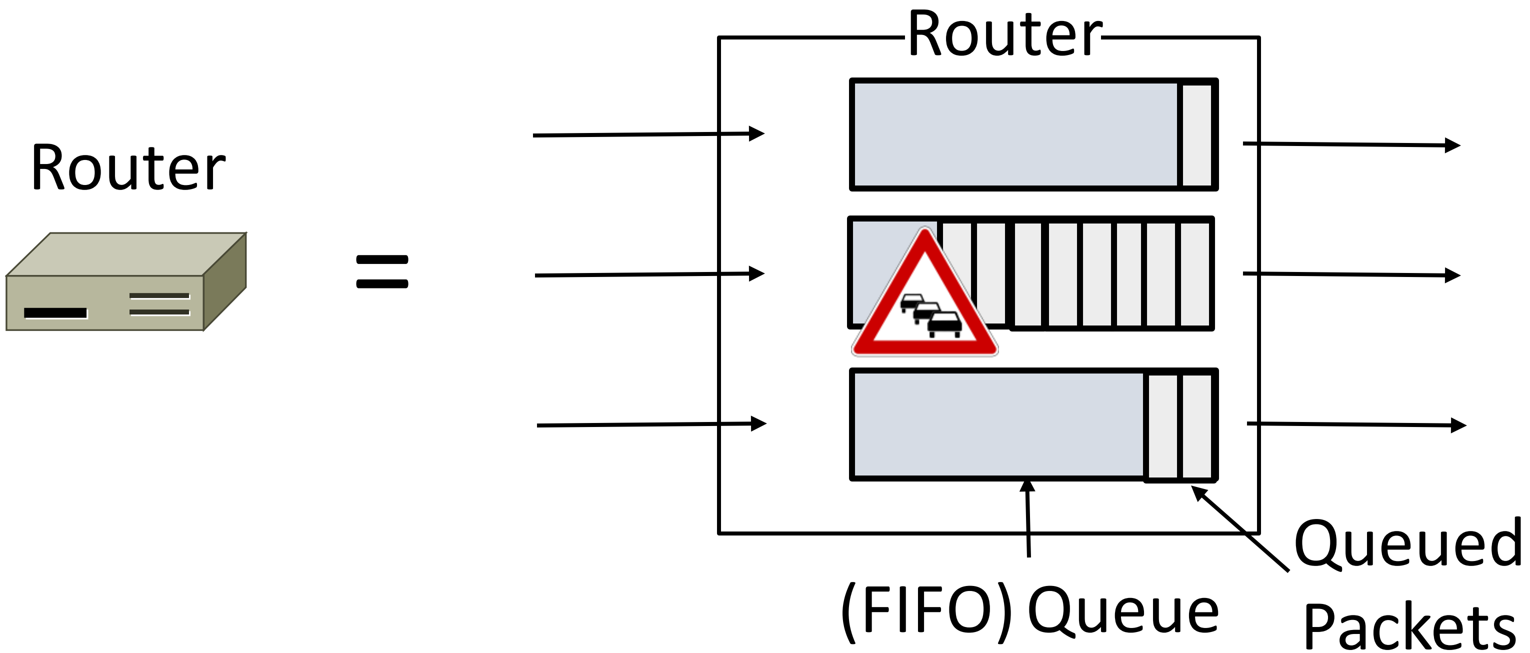
Nature of Congestion

- Routers/switches have internal buffering



Nature of Congestion (2)

- Simplified view of per port output queues
  - Typically FIFO (First In First Out), discard when full

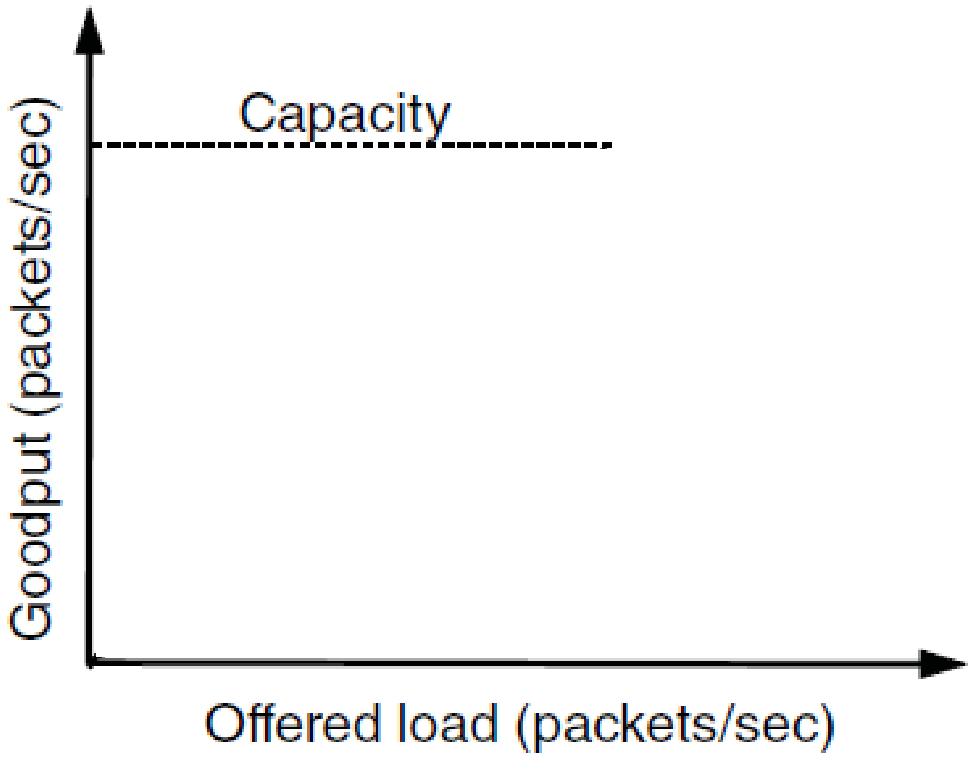


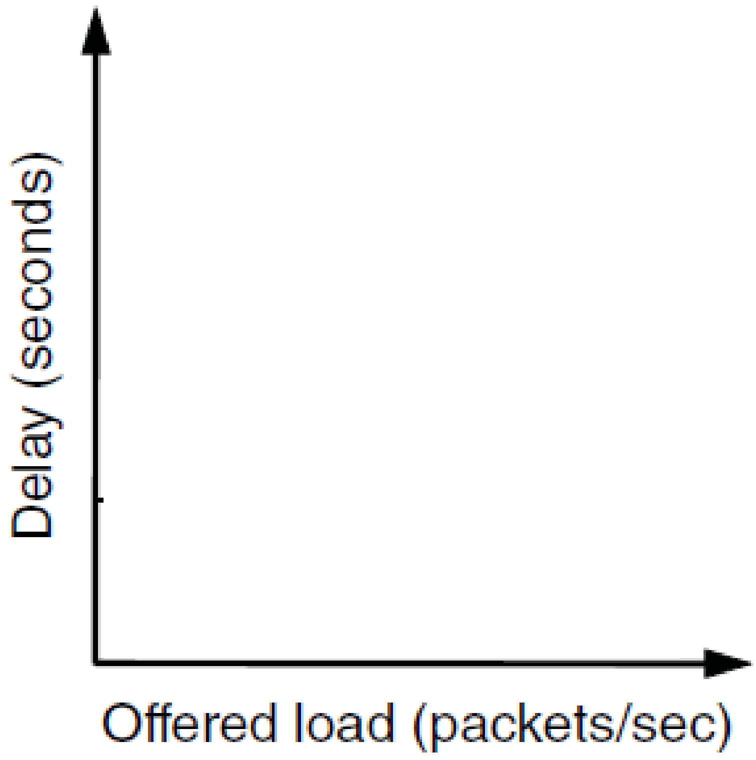
Nature of Congestion (3)

- Queues help by absorbing bursts when input > output rate
- But if input > output rate persistently, queue will overflow
  - This is congestion
- Congestion is a function of the traffic patterns – can occur even if every link has the same capacity

Effects of Congestion

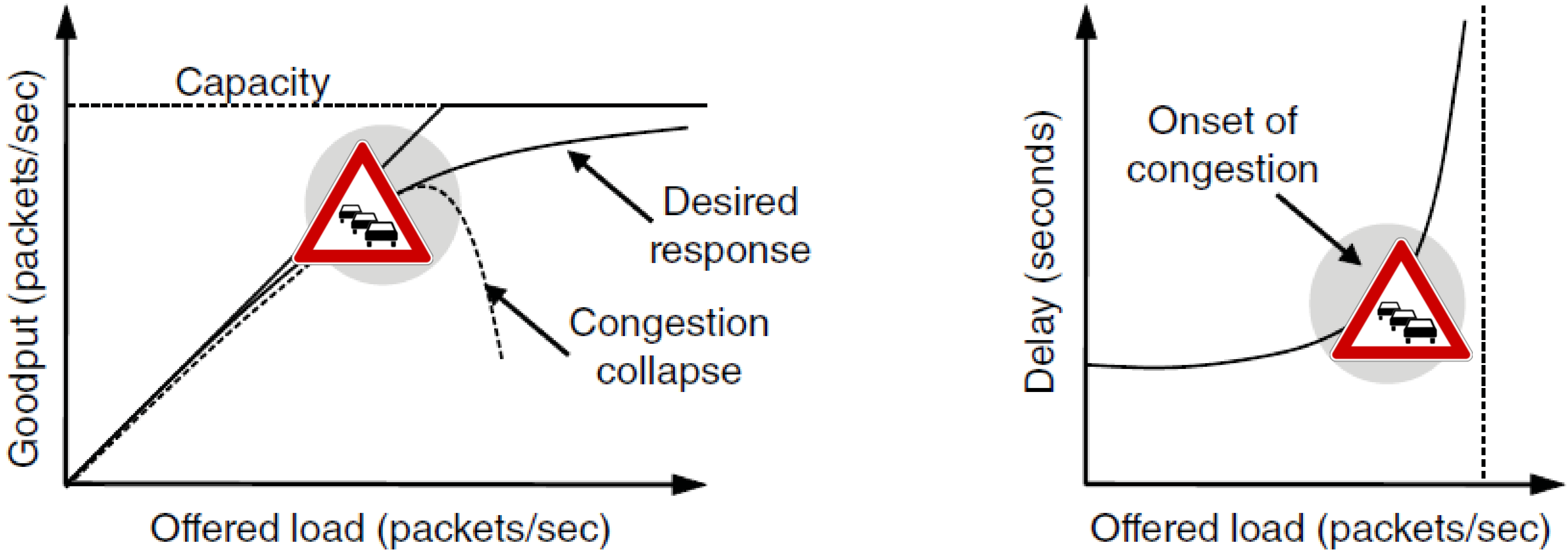
- What happens to performance as we increase load?





Effects of Congestion (2)

- What happens to performance as we increase load?



Effects of Congestion (3)

- As offered load rises, congestion occurs as queues begin to fill:
  - Delay and loss rise sharply with more load
  - Throughput falls below load (due to loss)
  - Goodput may fall below throughput (due to spurious retransmissions)
- None of the above is good!
  - Want network performance just before congestion

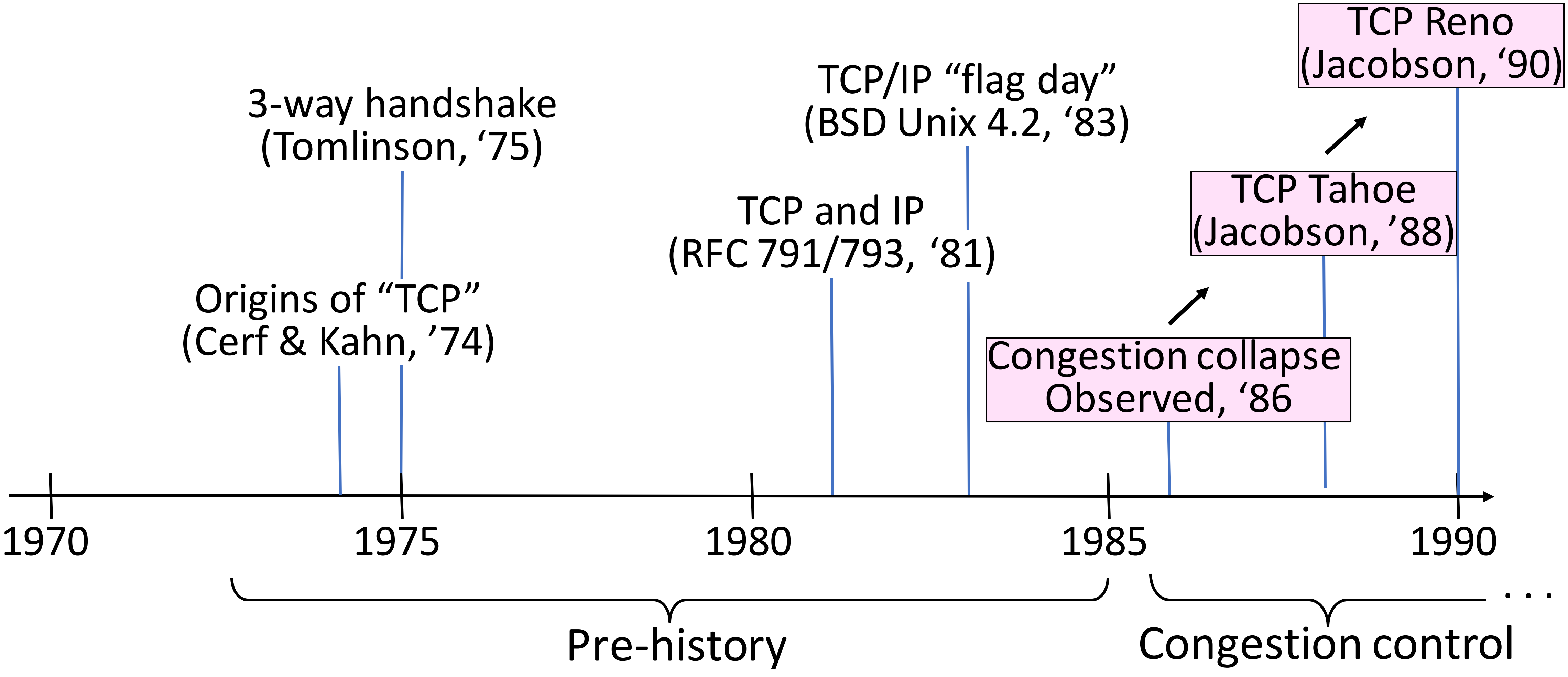
Van Jacobson (1950—)

- Widely credited with saving the Internet from congestion collapse in the late 80s
  - Introduced congestion control principles
  - Practical solutions (TCP Tahoe/Reno)
- Much other pioneering work:
  - Tools like traceroute, tcpdump, pathchar

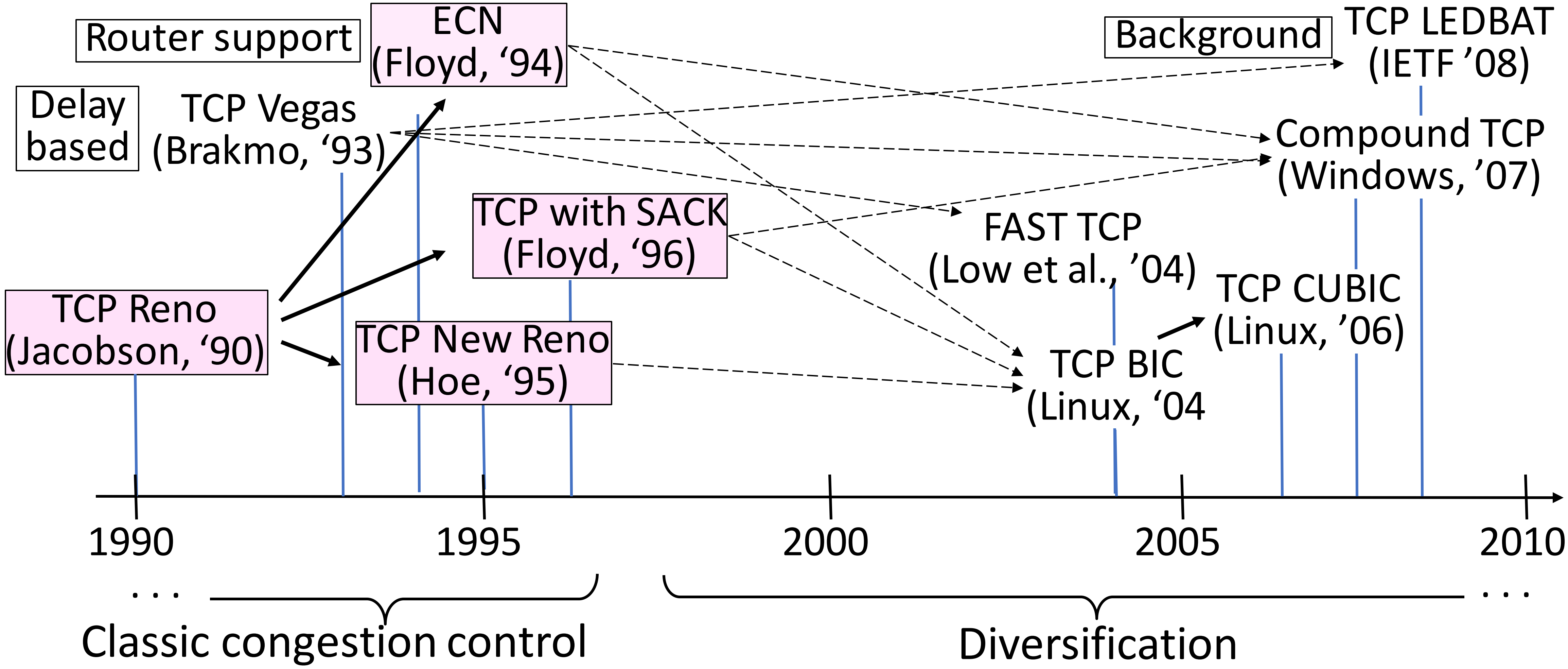
TCP Tahoe/Reno

- TCP extensions and features we will study:
  - AIMD
  - Fair Queuing
  - Slow-start
  - Fast Retransmission
  - Fast Recovery

TCP Timeline



TCP Timeline (2)



Bandwidth Allocation

- Important task for network is to allocate its capacity to senders
  - Good allocation is both efficient and fair
- <u>Efficient</u> means most capacity is used but there is no congestion
- <u>Fair</u> means every sender gets a reasonable share the network

Bandwidth Allocation (2)

- Key observation:
  - In an effective solution, Transport and Network layers must work together
- Network layer witnesses congestion
  - Only it can provide direct feedback
- Transport layer causes congestion
  - Only it can reduce offered load

Bandwidth Allocation (3)

- Why is it hard? (Just split equally!)
  - Number of senders and their offered load changes
  - Senders may lack capacity in different parts of network
  - Network is distributed; no single party has an overall picture of its state

Bandwidth Allocation (4)

- Solution context:
  - Senders adapt concurrently based on their own view of the network
  - Design this adaption so the network usage as a whole is efficient and fair
  - Adaption is continuous since offered loads continue to change over time

Fair Allocations

Fair Allocation

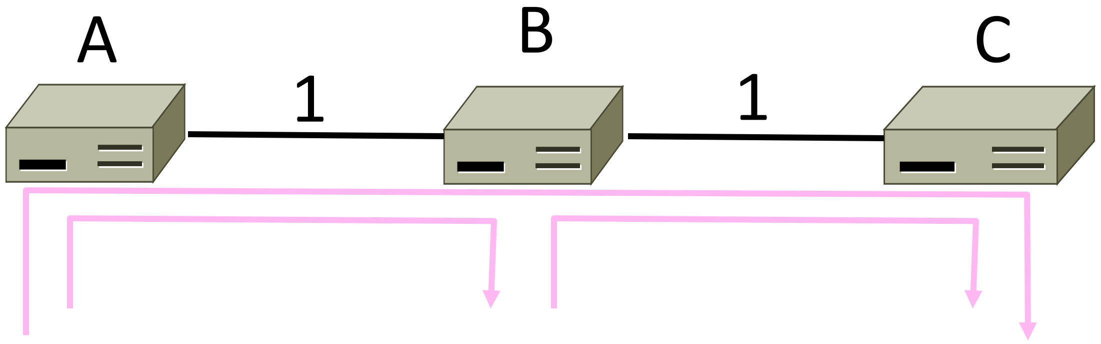
- What's a "fair" bandwidth allocation?
  - The max-min fair allocation

Recall

- We want a good bandwidth allocation to be both fair and efficient
  - Now we learn what fair means
- Caveat: in practice, efficiency is more important than fairness

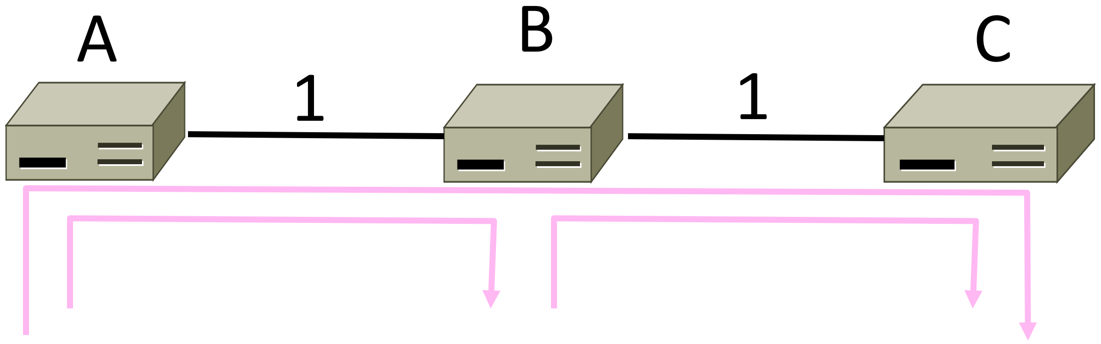
Efficiency vs. Fairness

- Cannot always have both!
  - Example network with traffic:
    - ✌❼B, B→C and A→ C
  - How much traffic can we carry?



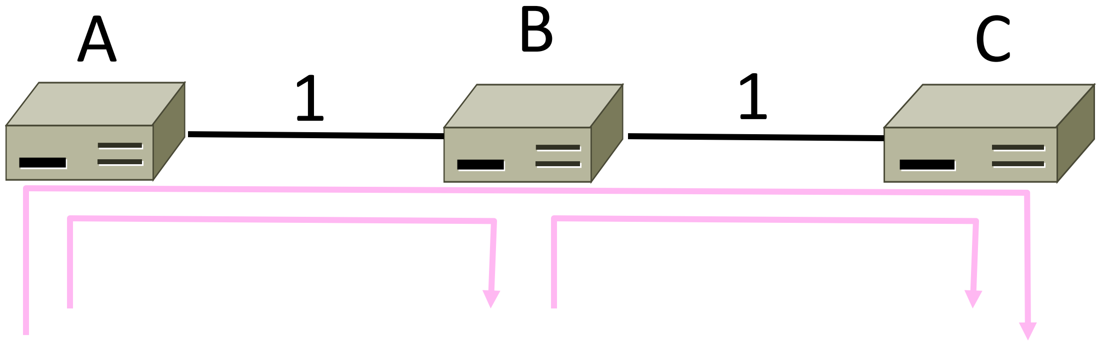
Efficiency vs. Fairness (2)

- If we care about fairness:
  - Give equal bandwidth to each flow
  - ✌❼B: ½ unit, B→C: ½, and A→C, ½
  - Total traffic carried is 1 ½ units



Efficiency vs. Fairness (3)

- If we care about efficiency:
  - Maximize total traffic in network
  - ✌❼B: 1 unit, B→C: 1, and A→C, 0
  - Total traffic rises to 2 units!

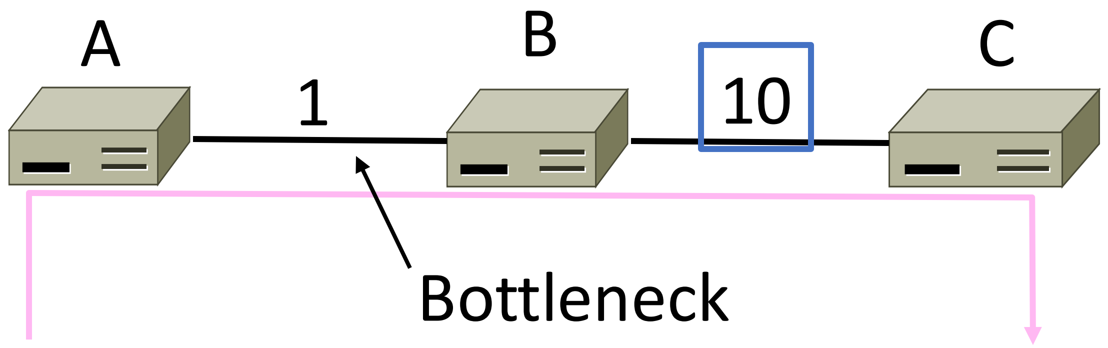


The Slippery Notion of Fairness

- Why is “equal per flow” fair anyway?
  - ✌❼C uses more network resources than A→B or B→C
  - Host A sends two flows, B sends one
- Not productive to seek exact fairness
  - More important to avoid <u>starvation</u>
    - A node that cannot use any bandwidth
  - “Equal per flow” is good enough

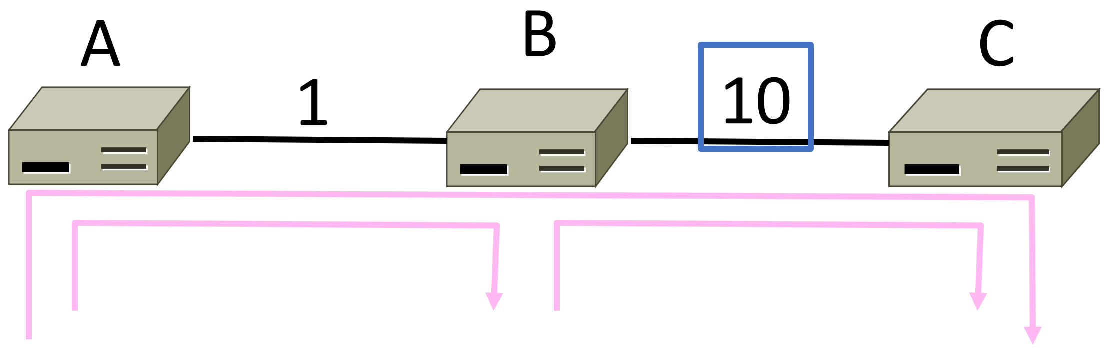
Generalizing "Equal per Flow"

- Bottleneck for a flow of traffic is the link that limits its bandwidth
  - Where congestion occurs for the flow
  - For A→C, link A–B is the bottleneck



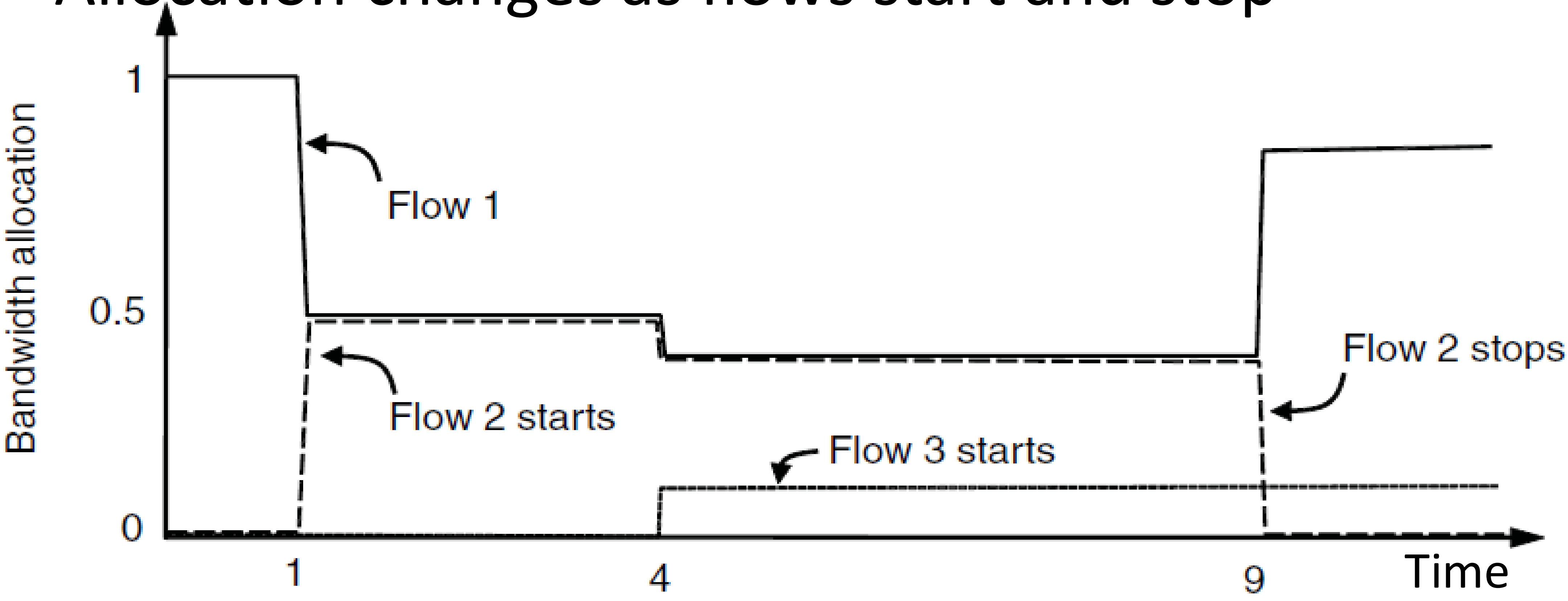
Generalizing "Equal per Flow" (2)

- Flows may have different bottlenecks
  - For A→C, link A–B is the bottleneck
  - For B→C, link B–C is the bottleneck
  - Can no longer divide links equally …

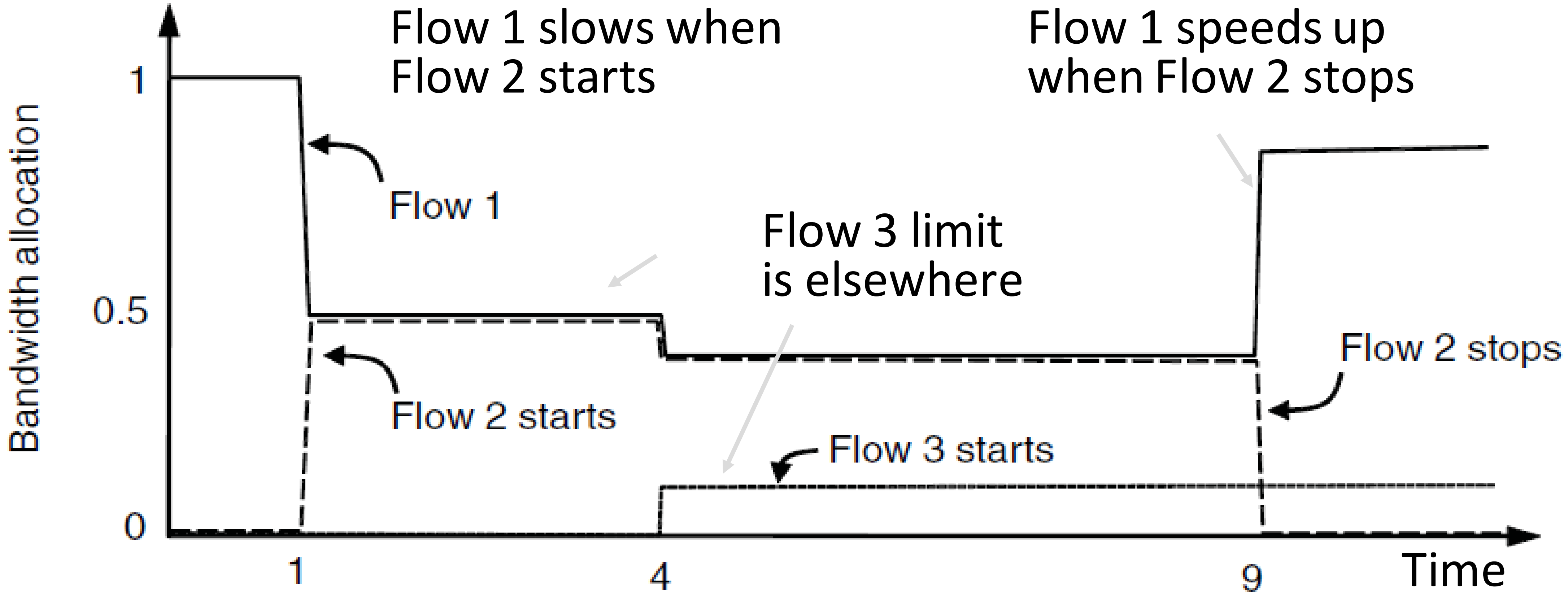


Adapting over Time

- Allocation changes as flows start and stop



Adapting over Time (2)



Bandwidth Allocation Models

- Open loop versus closed loop
  - Open: reserve bandwidth before use
  - Closed: use feedback to adjust rates
- Host versus Network support
  - Who is sets/enforces allocations?
- Window versus Rate based
  - How is allocation expressed?

TCP is a closed loop, host-driven, and window-based

Bandwidth Allocation Models (2)

- We'll look at closed-loop, host-driven, and window-based too
- Network layer returns feedback on current allocation to senders
  - For TCP signal is "a packet dropped"
- Transport layer adjusts sender's behavior via window in response
  - How senders adapt is a <u>control law</u>

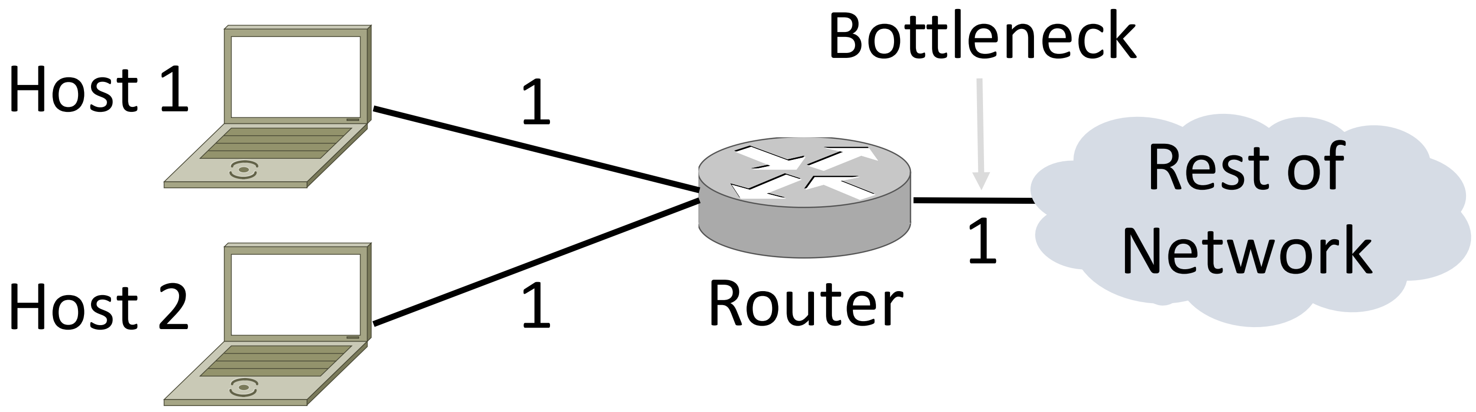
Additive Increase Multiplicative Decrease

- AIMD is a control law hosts can use to reach a good allocation
  - Hosts additively increase rate while network not congested
  - Hosts multiplicatively decrease rate when congested
  - Used by TCP

- Let's explore the AIMD game …

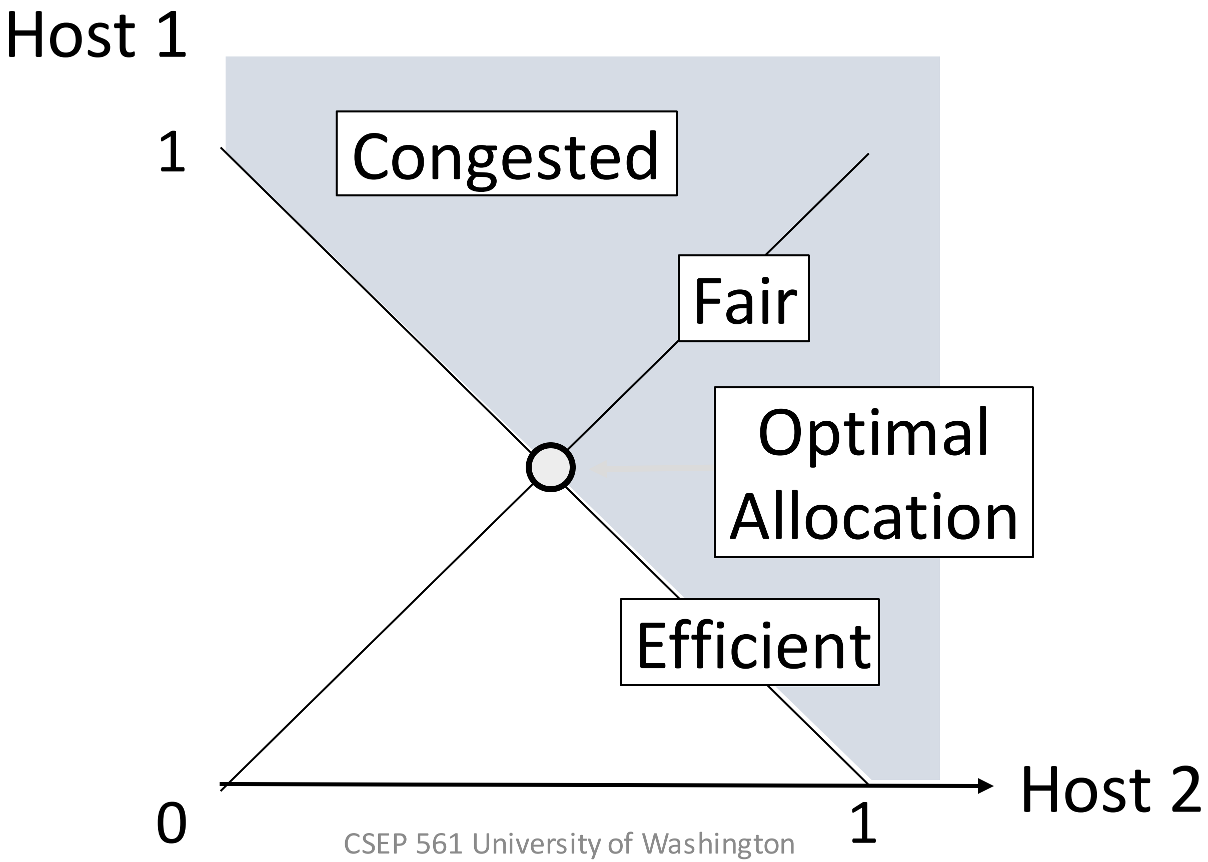
AIMD Game

- Hosts 1 and 2 share a bottleneck
  - But do not talk to each other directly
- Router provides binary feedback
  - Tells hosts if network is congested



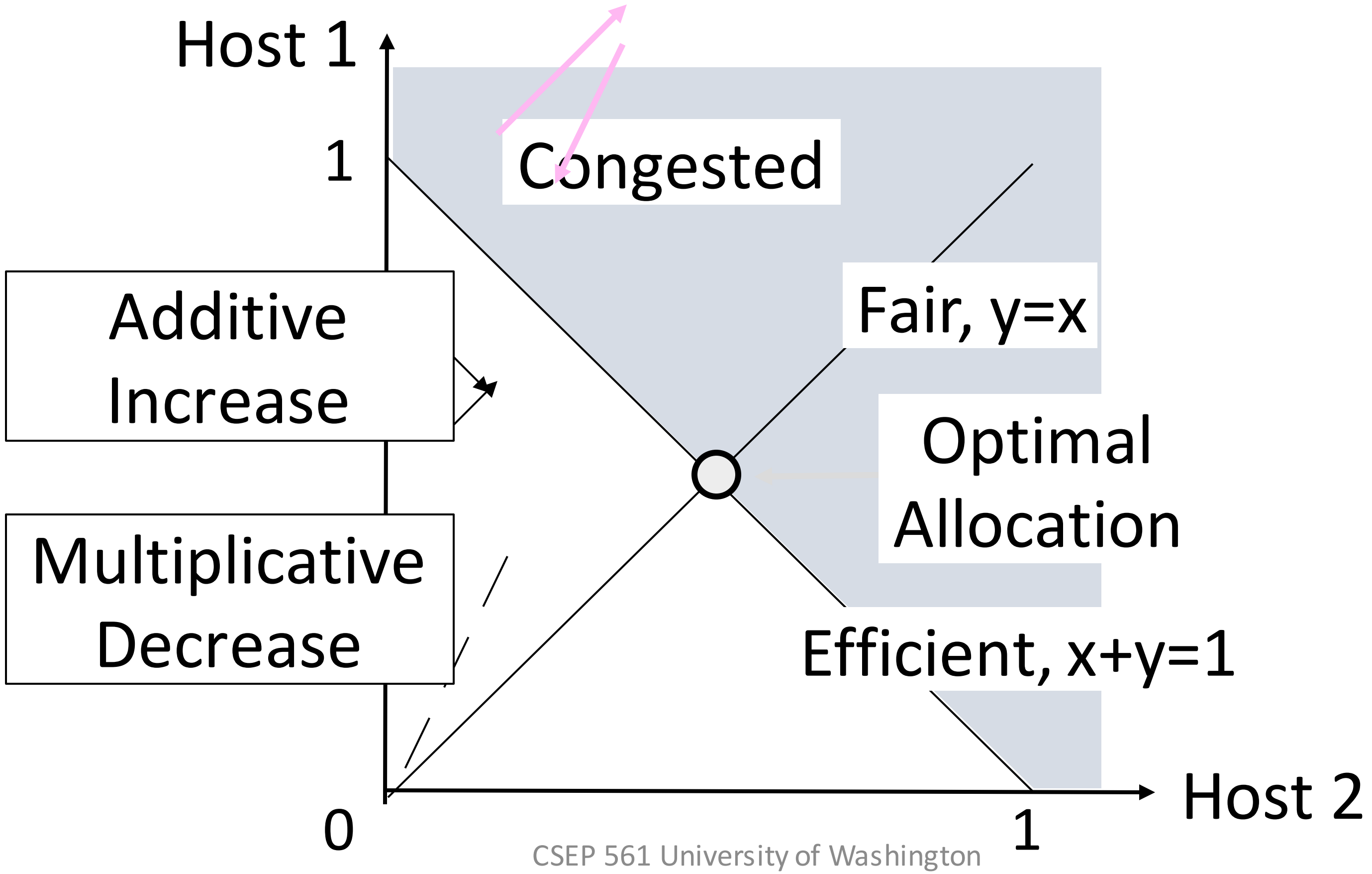
AIMD Game (2)

- Each point is a possible allocation



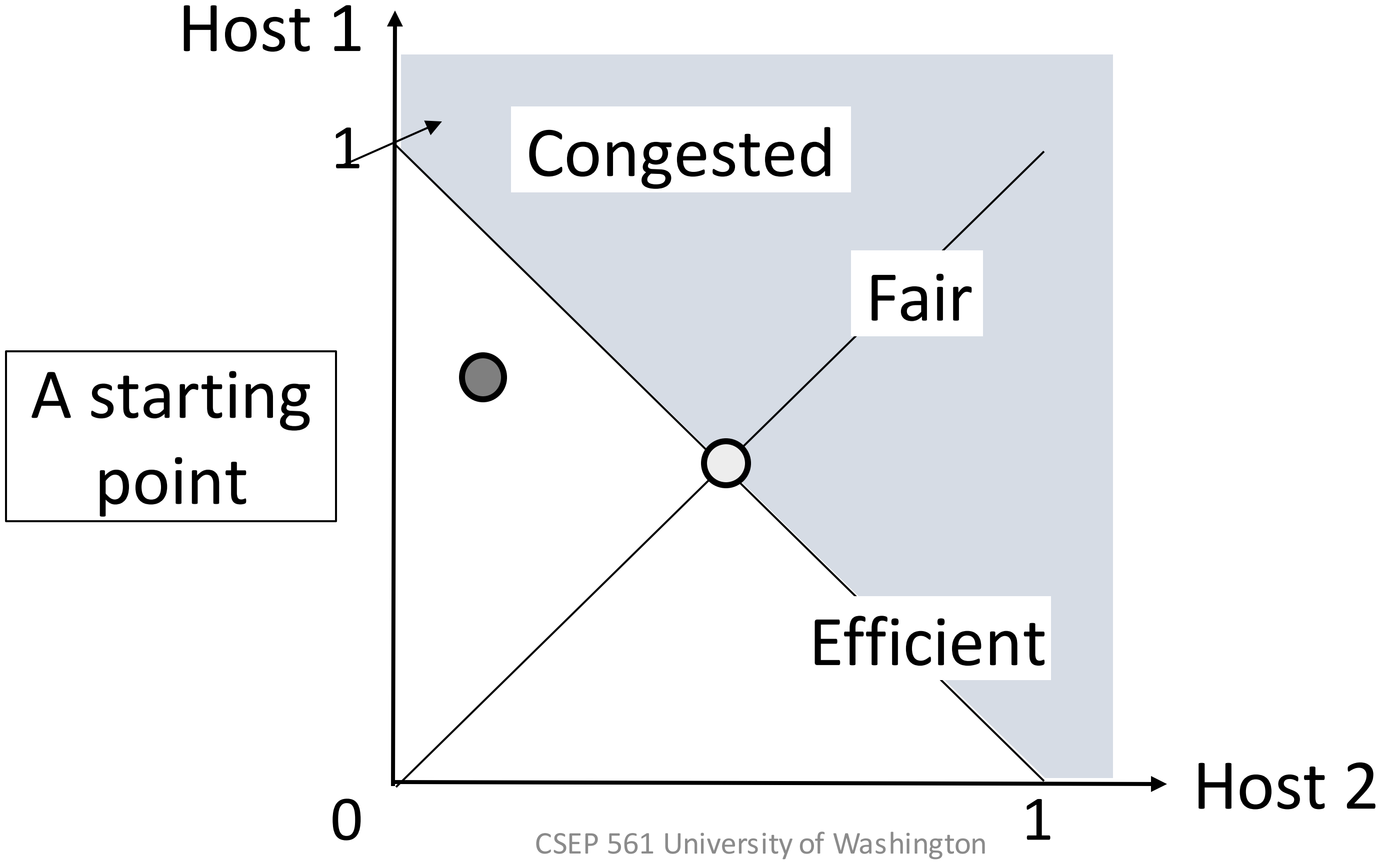
AIMD Game (3)

- AI and MD move the allocation



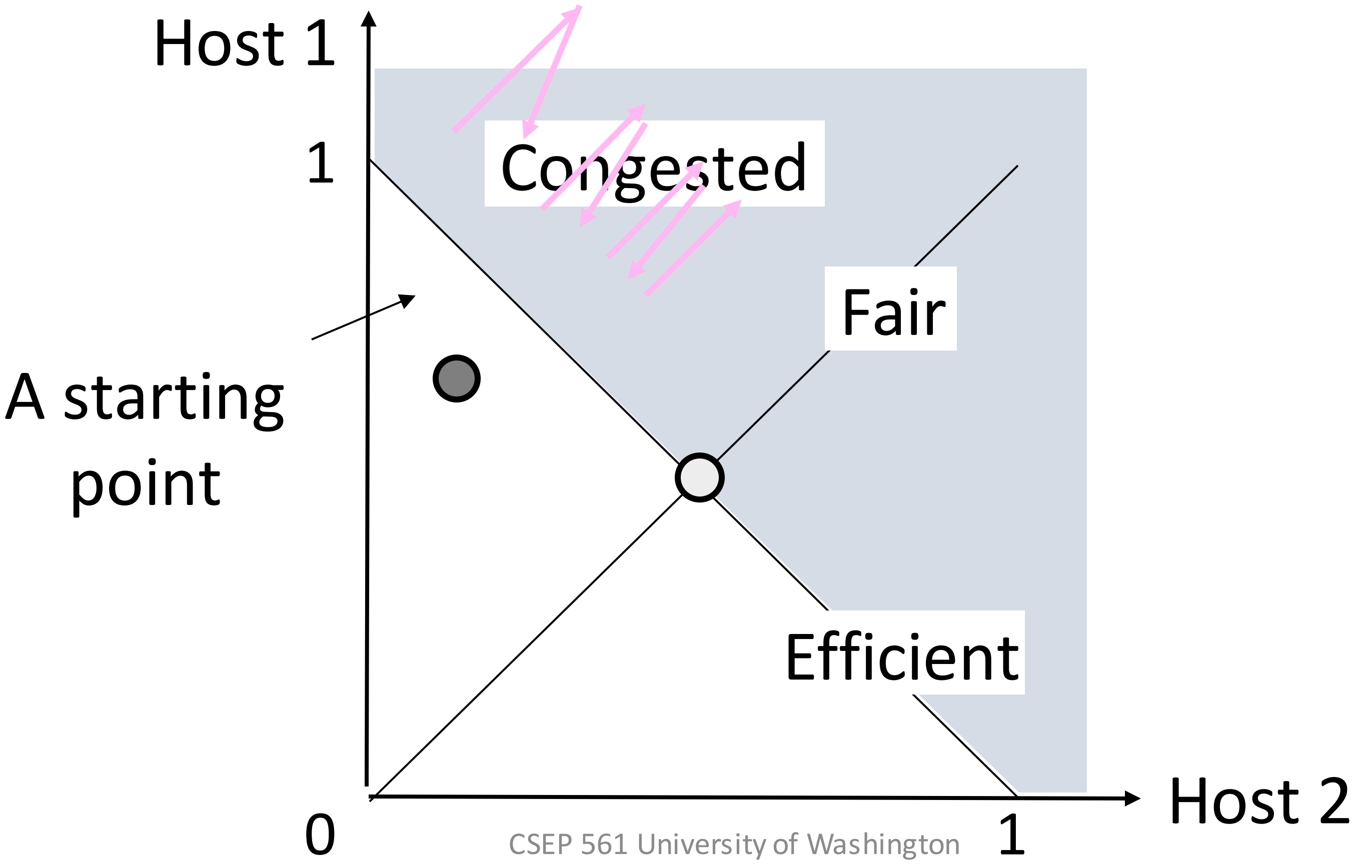
AIMD Game (4)

- Play the game!



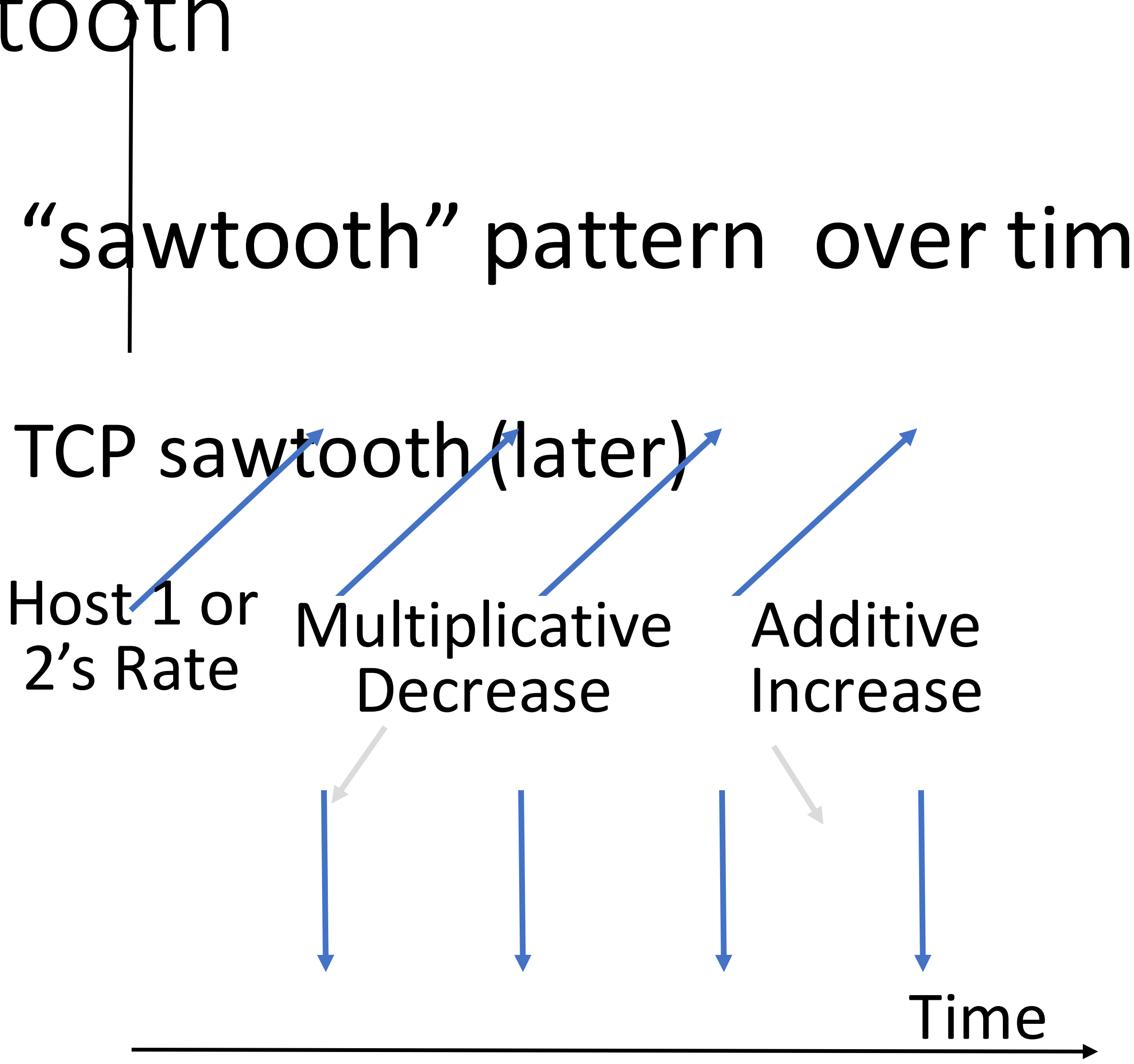
AIMD Game (5)

- Always converge to good allocation!



AIMD Sawtooth

- Produces a “sawtooth” pattern over time for rate of each host
  - This is the TCP sawtooth (later)



AIMD Properties

- Converges to an allocation that is efficient and fair when hosts run it
  - Holds for more general topologies
- Other increase/decrease control laws do not! (Try MIAD, MIMD, MIAD)
- Requires only binary feedback from the network

Feedback Signals

- Several possible signals, with different pros/cons
  - We'll look at classic TCP that uses packet loss as a signal

Signal	Example Protocol	Pros / Cons
Packet loss	TCP NewReno Cubic TCP (Linux)	Hard to get wrong Hear about congestion late
Packet delay	TCP BBR (Youtube)	Hear about congestion early Need to infer congestion
Router indication	TCPs with Explicit Congestion Notification	Hear about congestion early Require router support

Slow Start (TCP Additive Increase)

Practical AIMD

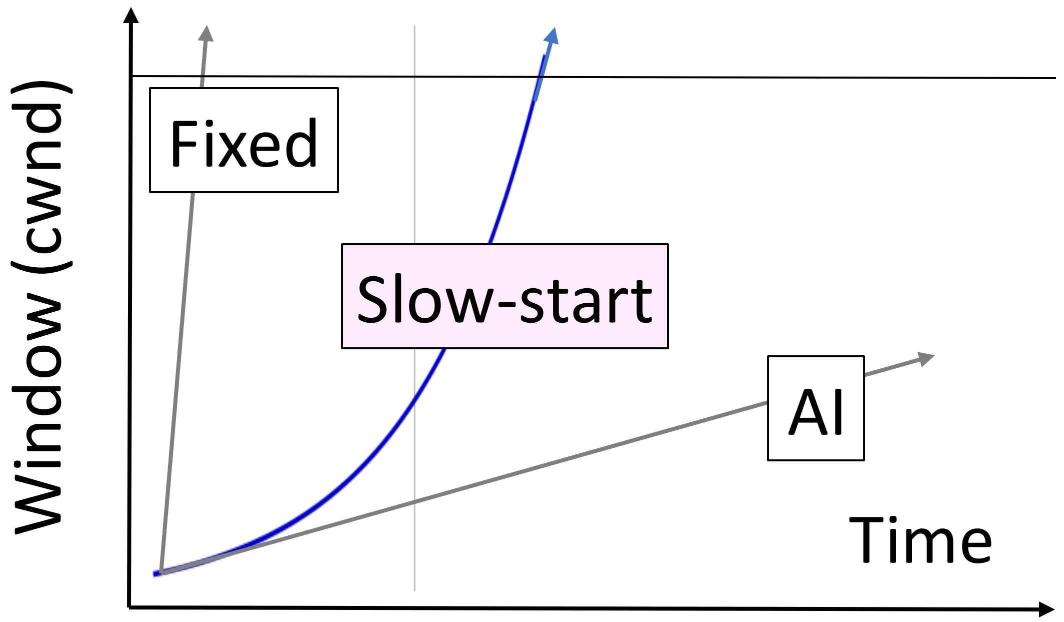
- We want TCP to follow an AIMD control law for a good allocation
- Sender uses a congestion window or cwnd to set its rate (≈cwnd/RTT)
- Sender uses loss as network congestion signal
- Need TCP to work across a very large range of rates and RTTs

TCP Startup Problem

- We want to quickly near the right rate, $\text{cwnd}_{\text{IDEAL}}$, but it varies greatly
  - Fixed sliding window doesn’t adapt and is rough on the network (loss!)
  - Additive Increase with small bursts adapts cwnd gently to the network, but might take a long time to become efficient

Slow-Start Solution

- Start by doubling cwnd every RTT
  - Exponential growth (1, 2, 4, 8, 16, …)
  - Start slow, quickly reach large values

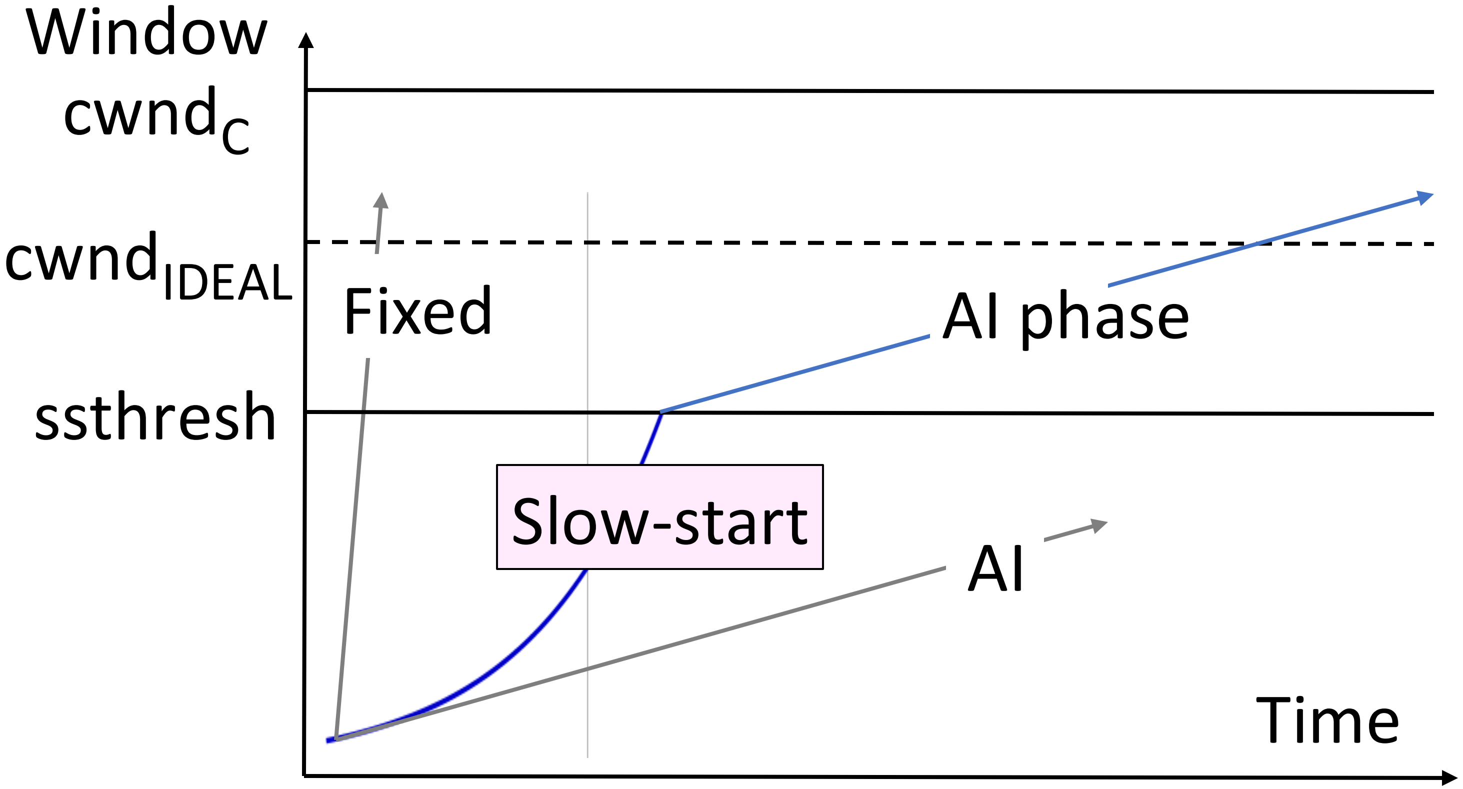


Slow-Start Solution (2)

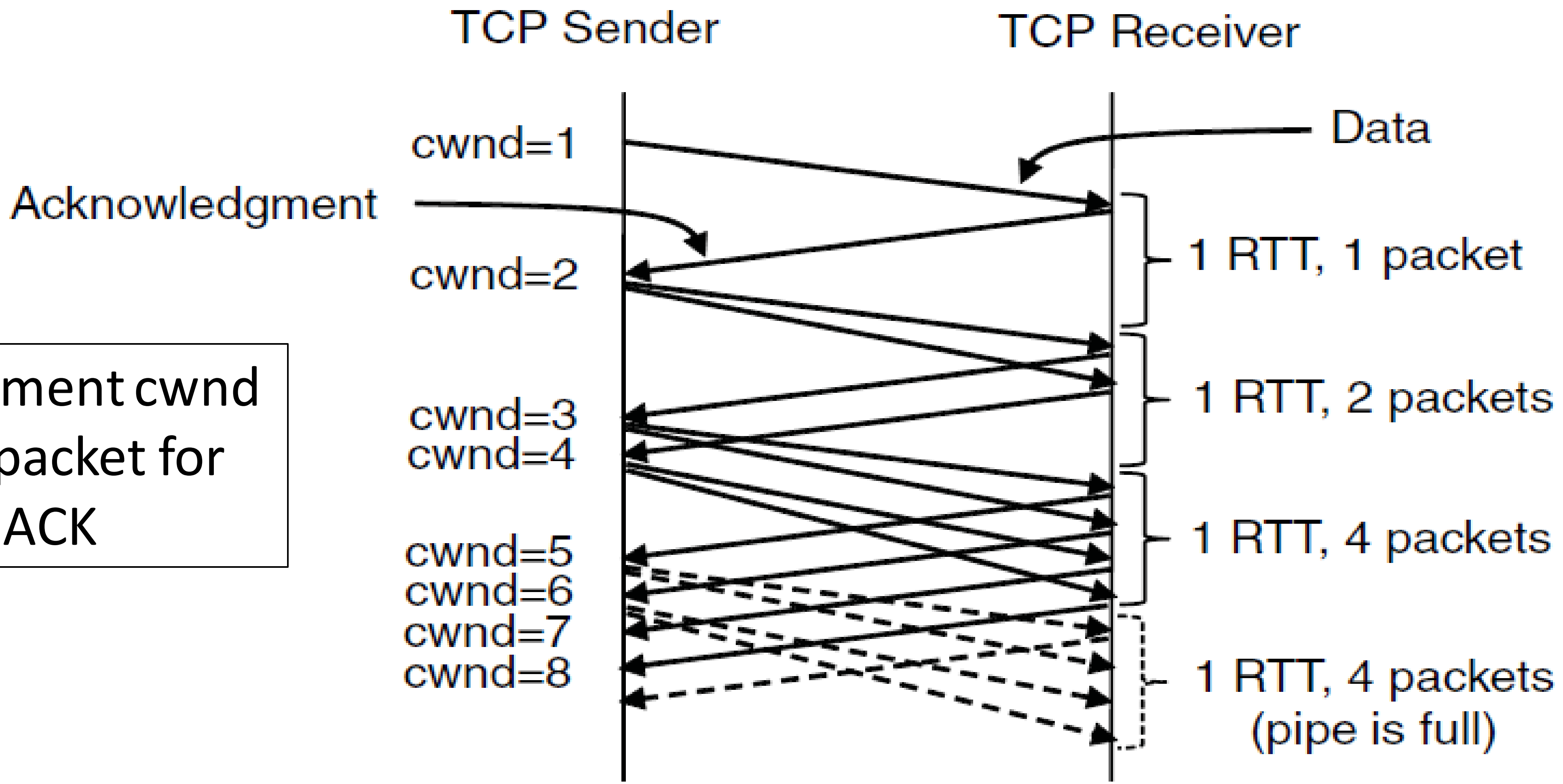
- Eventually packet loss will occur when the network is congested
  - Loss timeout tells us cwnd is too large
  - Next time, switch to AI beforehand
  - Slowly adapt cwnd near right value
- In terms of cwnd:
  - Expect loss for $\text{cwnd}_C \approx$ 2BD+queue
  - Use ssthresh = $\text{cwnd}_C/2$ to switch to AI

Slow-Start Solution (3)

- Combined behavior, after first time
  - Most time spend near right value

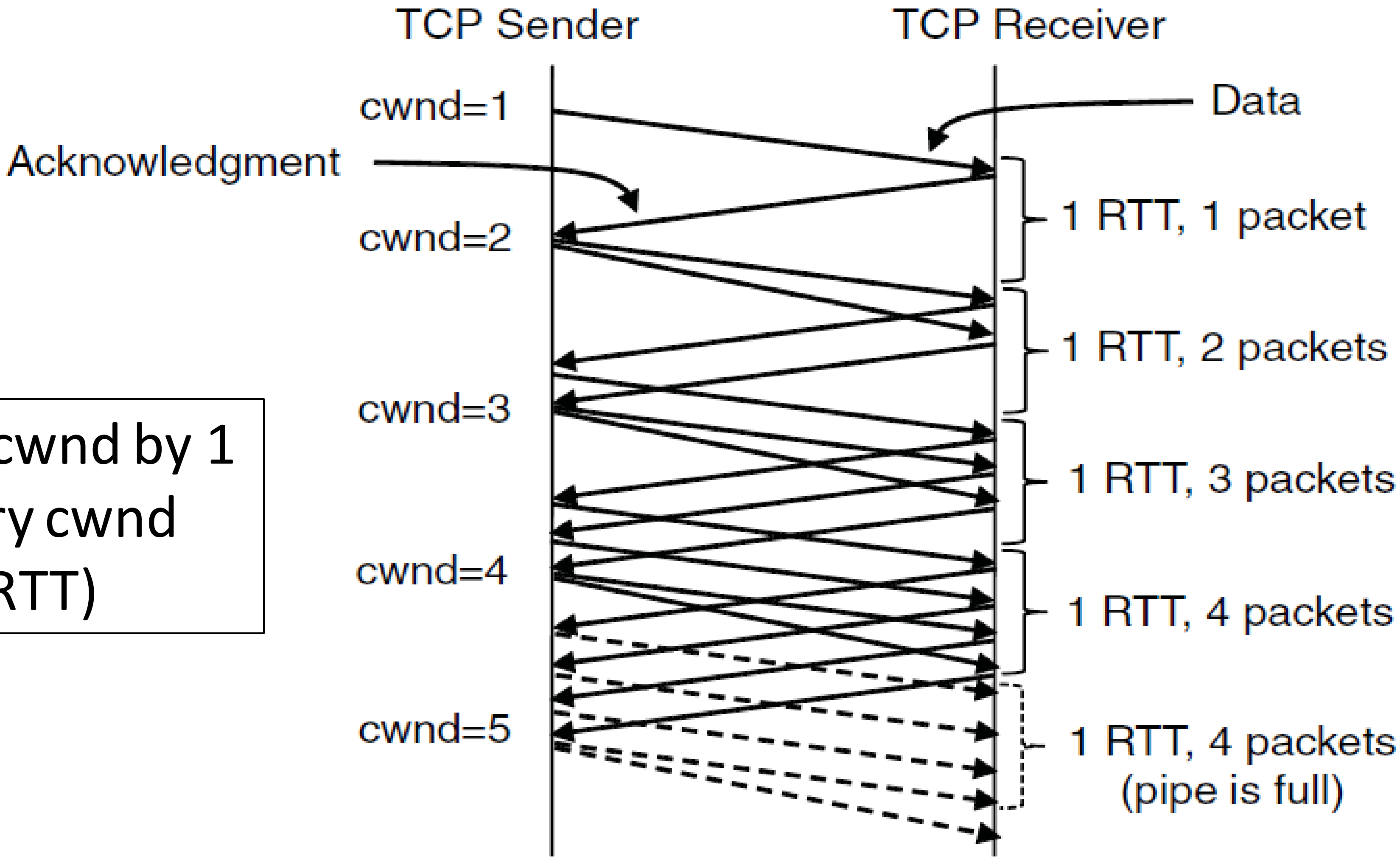


Slow-Start (Doubling) Timeline



Increment cwnd by 1 packet for each ACK

Additive Increase Timeline



Increment cwnd by 1 packet every cwnd ACKs (or 1 RTT)

TCP Tahoe (Implementation)

- Initial slow-start (doubling) phase
  - Start with cwnd = 1 (or small value)
  - cwnd += 1 packet per ACK
- Later Additive Increase phase
  - cwnd += 1/cwnd packets per ACK
  - Roughly adds 1 packet per RTT
- Switching threshold (initially infinity)
  - Switch to AI when cwnd > ssthresh
  - Set ssthresh = cwnd/2 after loss

Timeout Misfortunes

- Why do a slow-start after timeout?
  - Instead of MD cwnd (for AIMD)
- Timeouts are sufficiently long that the ACK clock will have run down
  - Slow-start ramps up the ACK clock
- We need to detect loss before a timeout to get to full AIMD

Fast Recovery
(TCP Multiplicative Decrease)

Practical AIMD (2)

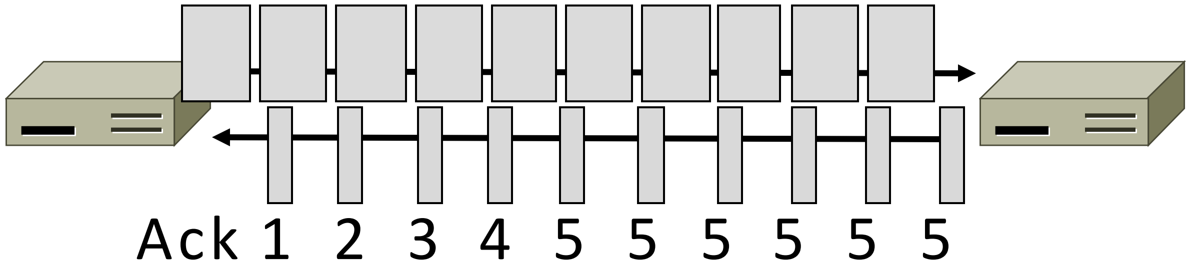
- We want TCP to follow an AIMD control law for a good allocation
- Sender uses a <u>congestion window</u> or <u>cwnd</u> to set its rate (≈cwnd/RTT)
- Sender uses slow-start to ramp up the ACK clock, followed by Additive Increase
- But after a timeout, sender slow-starts again with cwnd=1 (as it no ACK clock)

Inferring Loss from ACKs

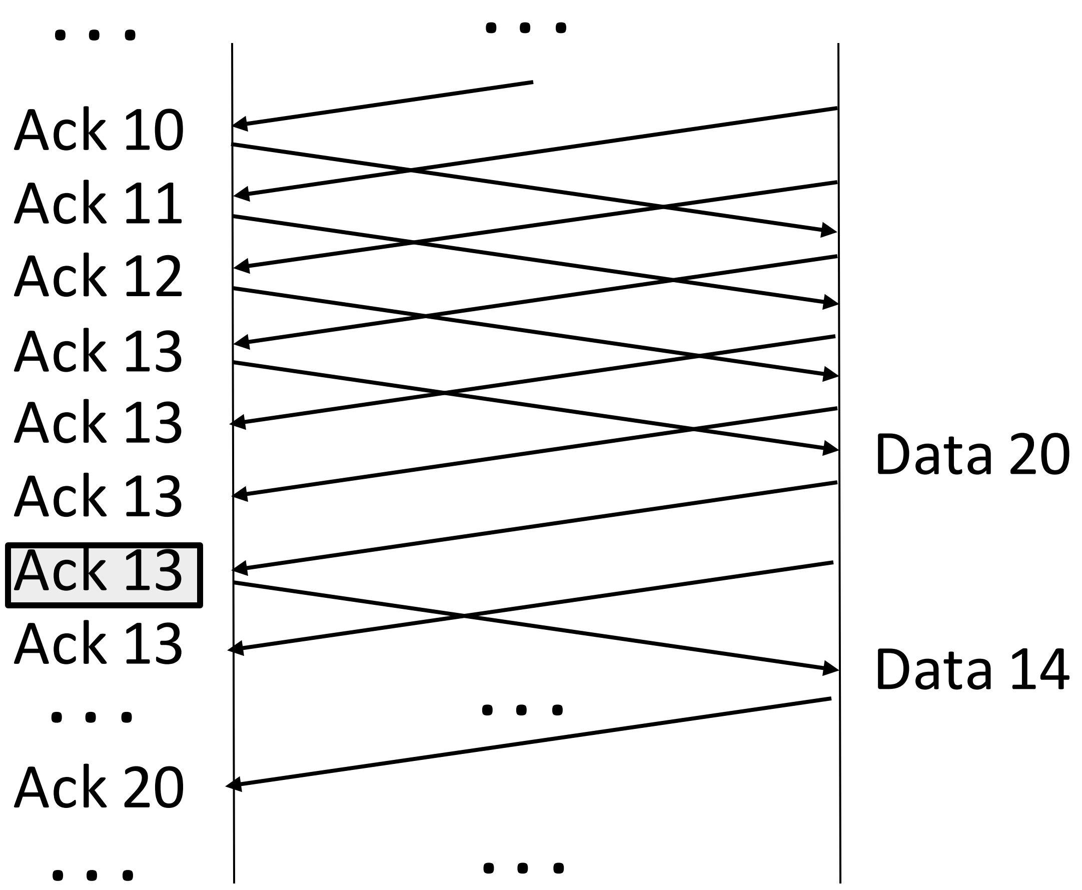
- TCP uses a cumulative ACK
  - Carries highest in-order seq. number
  - Normally a steady advance
- Duplicate ACKs give us hints about what data hasn't arrived
  - Tell us some new data did arrive, but it was not next segment
  - Thus the next segment may be lost

Fast Retransmit

- Treat three duplicate ACKs as a loss
  - Retransmit next expected segment
  - Some repetition allows for reordering, but still detects loss quickly



Fast Retransmit (2)



Data 14 was lost earlier, but got 15 to 20

Third duplicate ACK, so send 14

Retransmission fills in the hole at 14

ACK jumps after loss is repaired

Fast Retransmit (3)

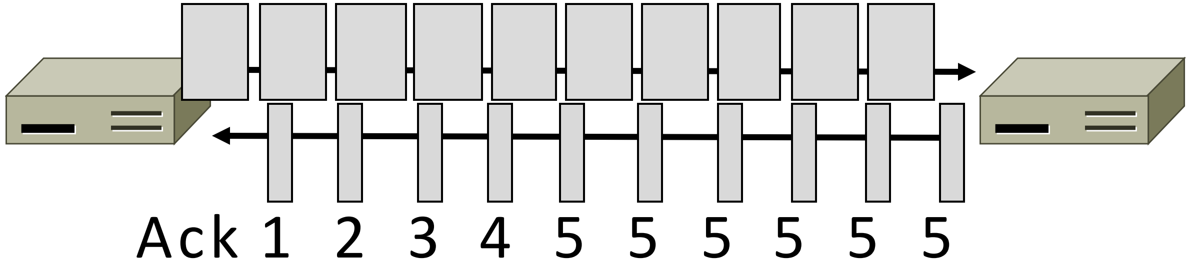
- It can repair single segment loss quickly, typically before a timeout
- However, we have quiet time at the sender/receiver while waiting for the ACK to jump
- And we still need to MD cwnd …

Inferring Non-Loss from ACKs

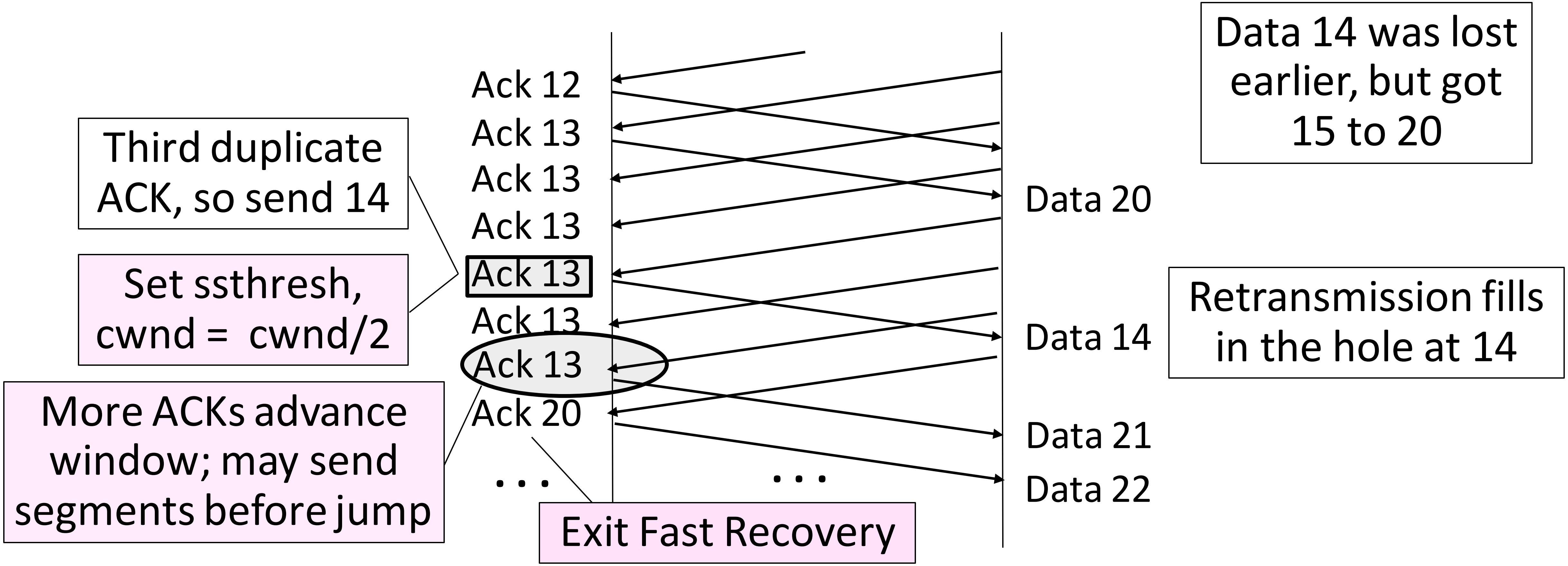
- Duplicate ACKs also give us hints about what data has arrived
  - Each new duplicate ACK means that some new segment has arrived
  - It will be the segments after the loss
  - Thus advancing the sliding window will not increase the number of segments stored in the network

Fast Recovery

- First fast retransmit, and MD cwnd
- Then pretend further duplicate ACKs are the expected ACKs
    - Lets new segments be sent for ACKs
    - Reconcile views when the ACK jumps



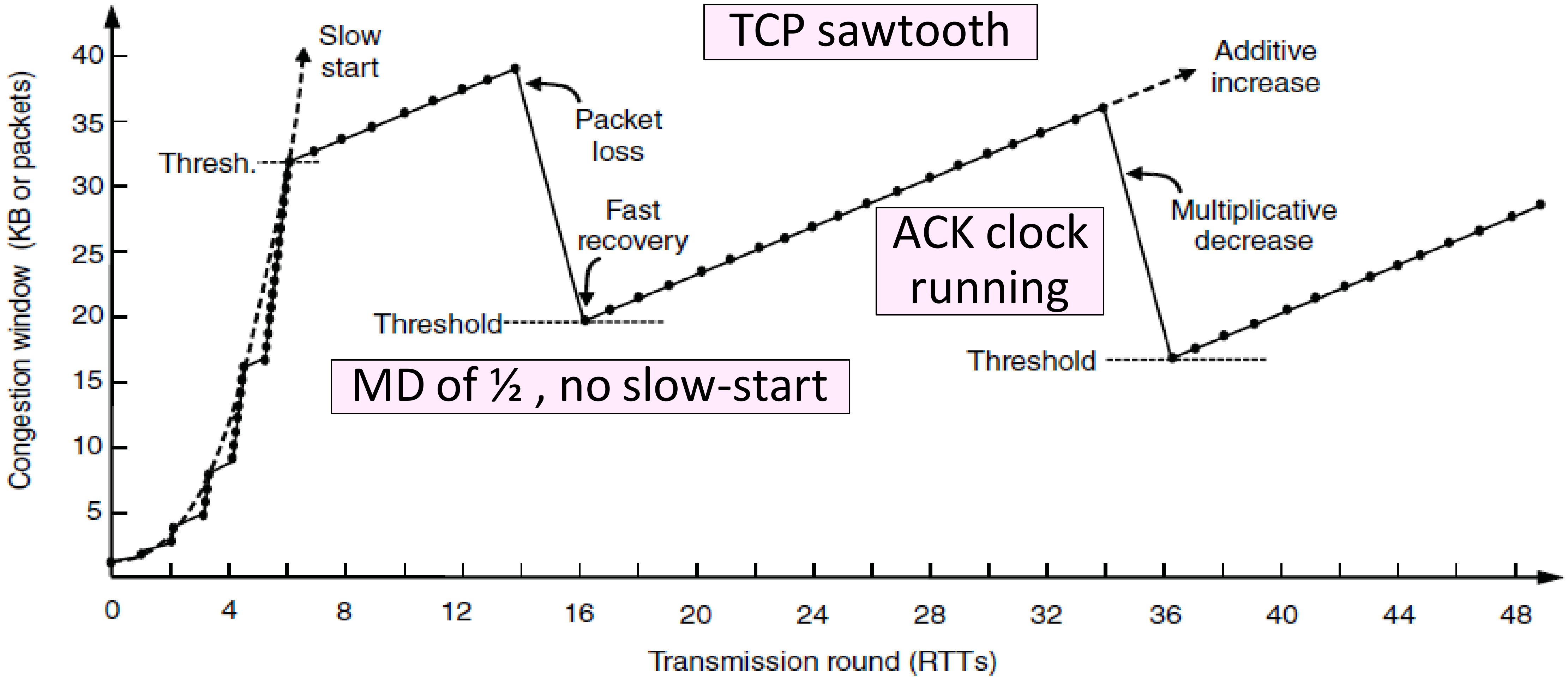
Fast Recovery (2)



Fast Recovery (3)

- With fast retransmit, it repairs a single segment loss quickly and keeps the ACK clock running
- This allows us to realize AIMD
  - No timeouts or slow-start after loss, just continue with a smaller cwnd
- TCP Reno combines slow-start, fast retransmit and fast recovery
  - Multiplicative Decrease is ½

TCP Reno



TCP sawtooth
Slow start
Additive increase
Thresh.
Packet loss
Fast recovery
Threshold
ACK clock running
Multiplicative decrease
Threshold
MD of ½ , no slow-start
Threshold
Congestion window (KB or packets)
40
35
30
25
20
15
10
5
0
4
8
12
16
20
24
28
32
36
40
44
48
Transmission round (RTTs)

TCP Reno, NewReno, and SACK

- Reno can repair one loss per RTT
  - Multiple losses cause a timeout
- NewReno further refines ACK heuristics
  - Repairs multiple losses without timeout
- Selective ACK (SACK) is a better idea
  - Receiver sends ACK ranges so sender can retransmit without guesswork

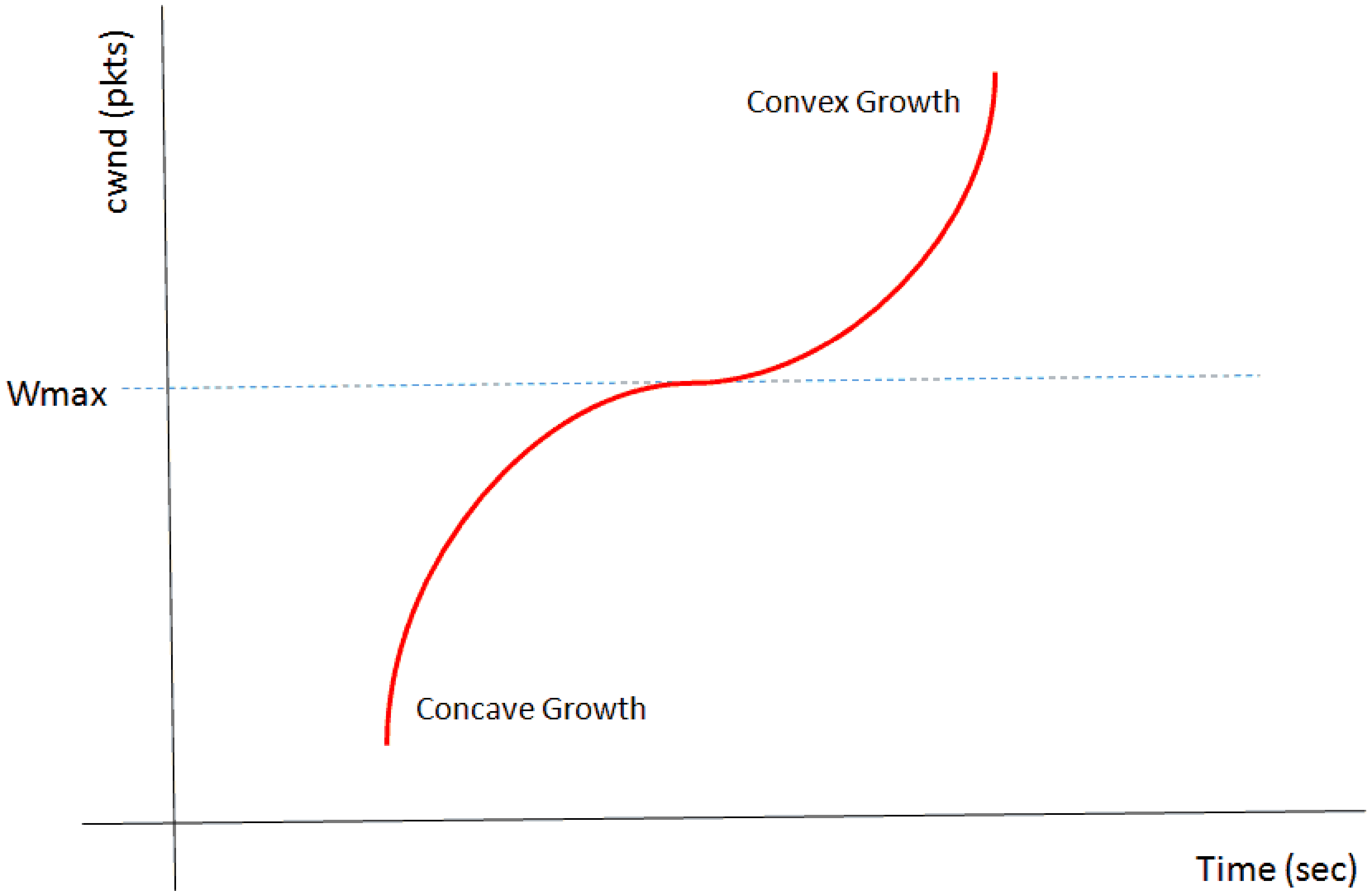
TCP CUBIC

- Standard TCP Stack in Linux (> 2.6.19) and Windows (> 10.1709)
- Internet grows to have more long-distance, high bandwidth connections
- Seeks to resolve two key problems with "standard" TCP:
  - Flows with lower RTT's "grow" faster than those with higher RTTs
  - Flows grow too "slowly" (linearly) after congestion

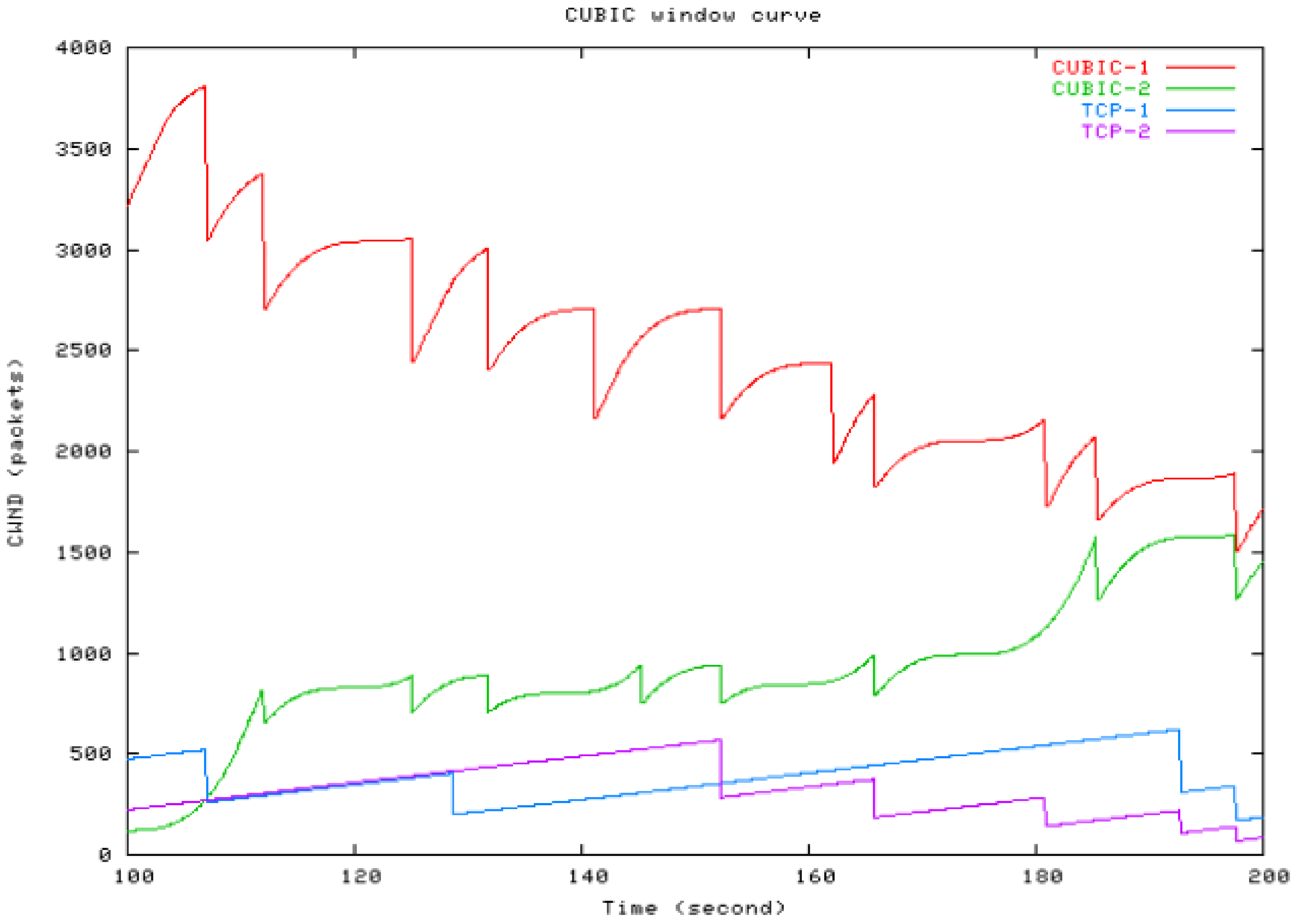
TCP CUBIC

1) At the time of experiencing congestion event the window size for that instant will be recorded as *Wmax* or the maximum window size.
2) The *Wmax* value will be set as the inflection point of the cubic function that will govern the growth of the congestion window.
3) The transmission will then be restarted with a smaller window value (20%) and, if no congestion is experienced, this value will increase according to the concave portion of the cubic function **(not depending on received ACKs for**

TCP CUBIC



TCP CUBIC vs Everyone



TCP BBR

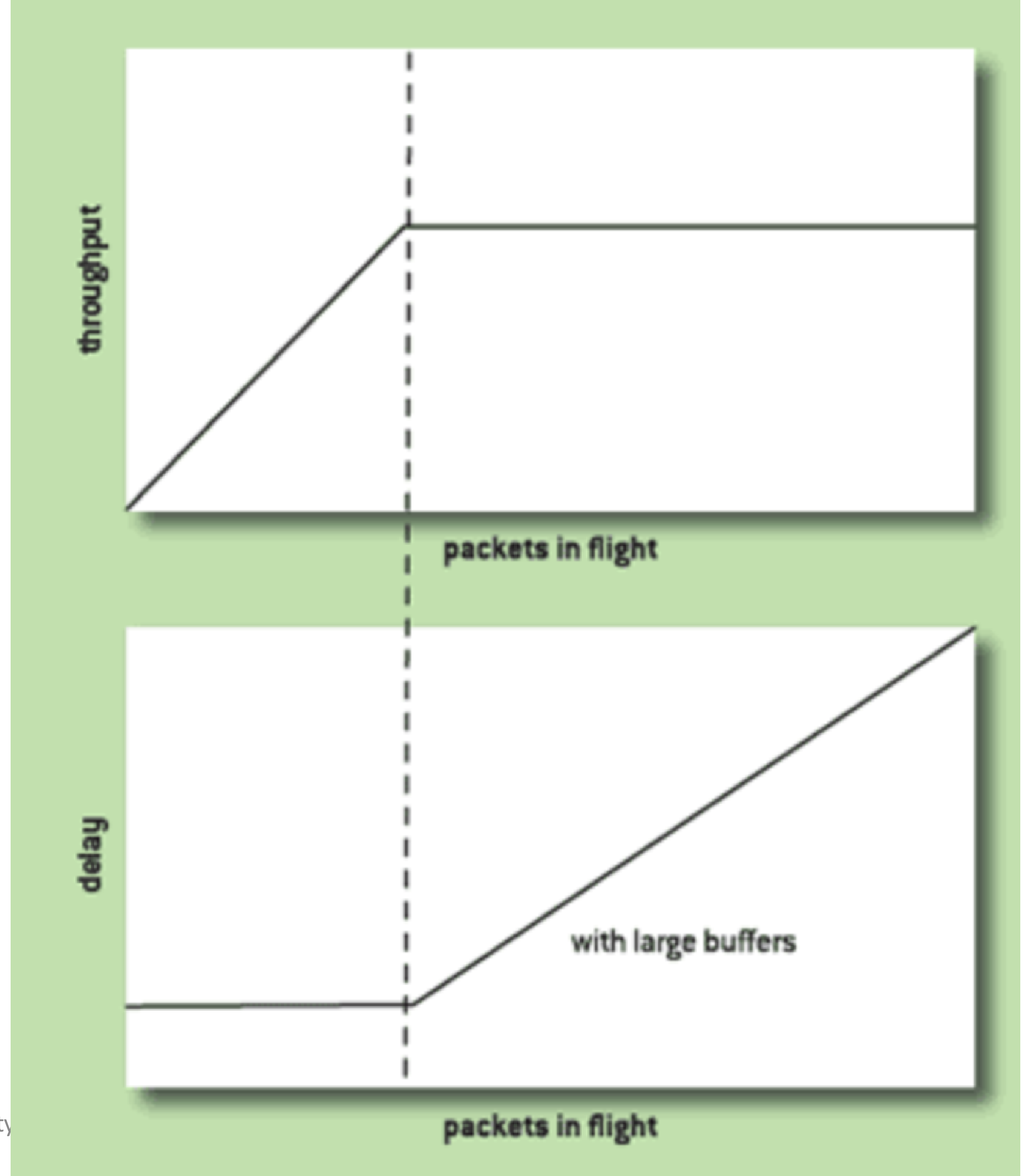
- Bottleneck Bandwidth and Round-trip propagation time
- Developed at Google in 2016 primarily for YouTube traffic
- Attempting to solve “bufflerbloat” problem
- “Model-based” (Vegas) rather than “Loss-based” (CUBIC)
  - Measure RTT, latency, bottleneck bandwidth
  - Use this to predict window size

Bufferbloat

- Larger queues are better than smaller queues right?

Bufferbloat

- Given TCP loss semantics…
- Performance can *decrease* buffer size is *increased*
- Consider a full buffer:
  - New packets arrive and are dropped ('tail drop')
  - SACK doesn't arrive until entire buffer sent

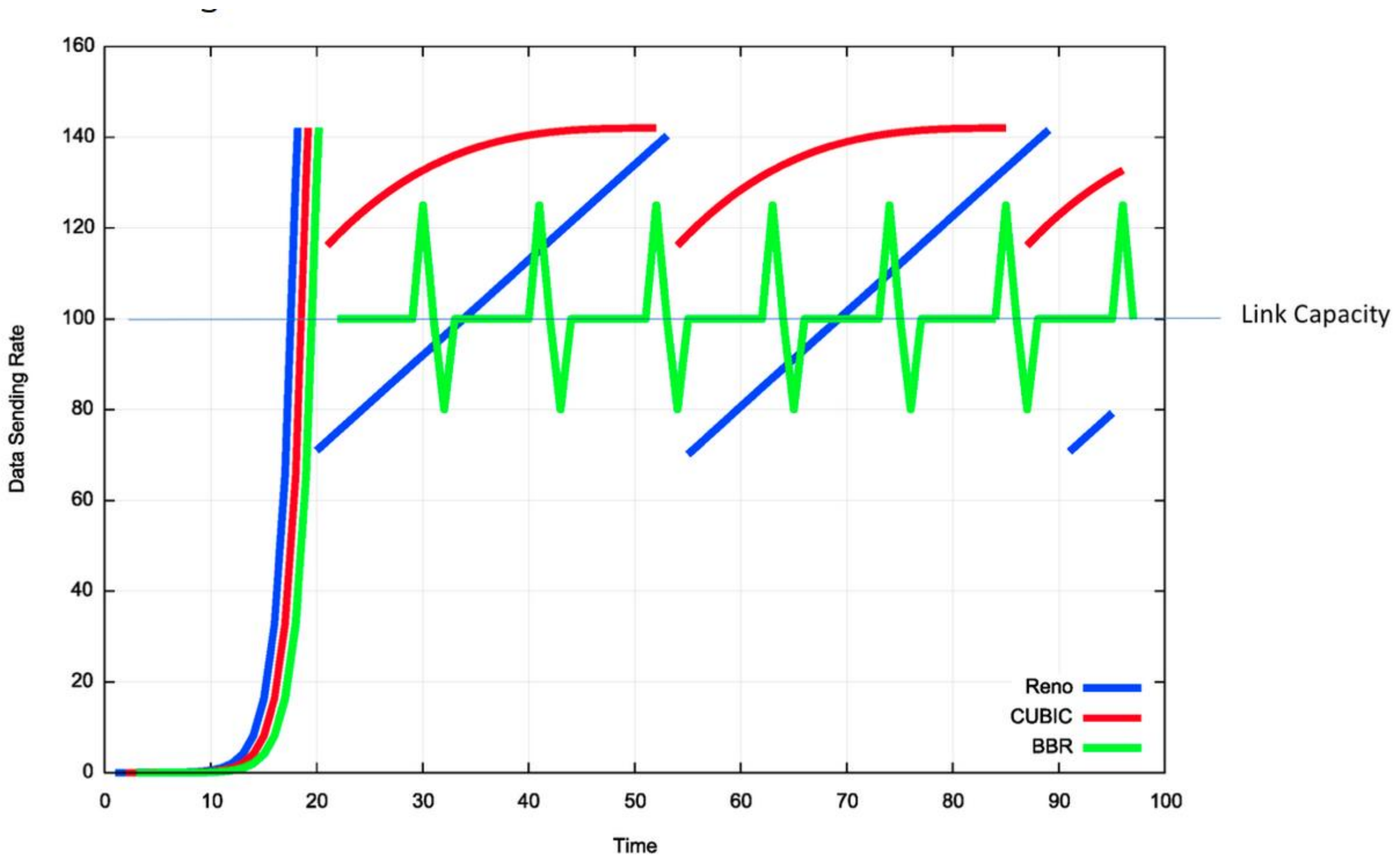


TCP BBR

- BBR Has 4 Distinct Phases

1) Startup: Basically identical to Cubic. Expontentially grow until RTTs start to increase (instead of dropped packet). Set *cwnd*.
2) Drain: Startup filled a queue. Temporarily reduce sending rate (known as "pacing gain")
3) Probe Bandwidth: Increase sending rate to see if there's more capacity. If not, *drain* again.
4) Probe RTT: Reduce rate dramatically (4 packets) to measure RTT. Use this as our baseline for above.

TCP BBR vs Everyone



Network-Side Congestion Control

Congestion Avoidance vs. Control

- Classic TCP drives the network into congestion and then recovers
  - Needs to see loss to slow down
- Would be better to use the network but avoid congestion altogether!
  - Reduces loss and delay
- But how can we do this?

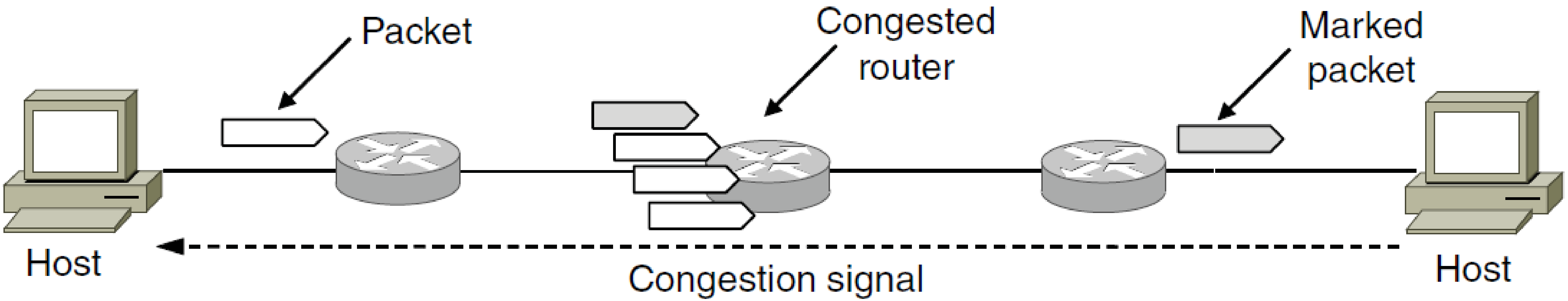
Feedback Signals

- Delay and router signals can let us avoid congestion

Signal	Example Protocol	Pros / Cons
Packet loss	Classic TCP Cubic TCP (Linux)	Hard to get wrong Hear about congestion late
Packet delay	TCP BBR (Youtube)	Hear about congestion early Need to infer congestion
Router indication	TCPs with Explicit Congestion Notification	Hear about congestion early Require router support

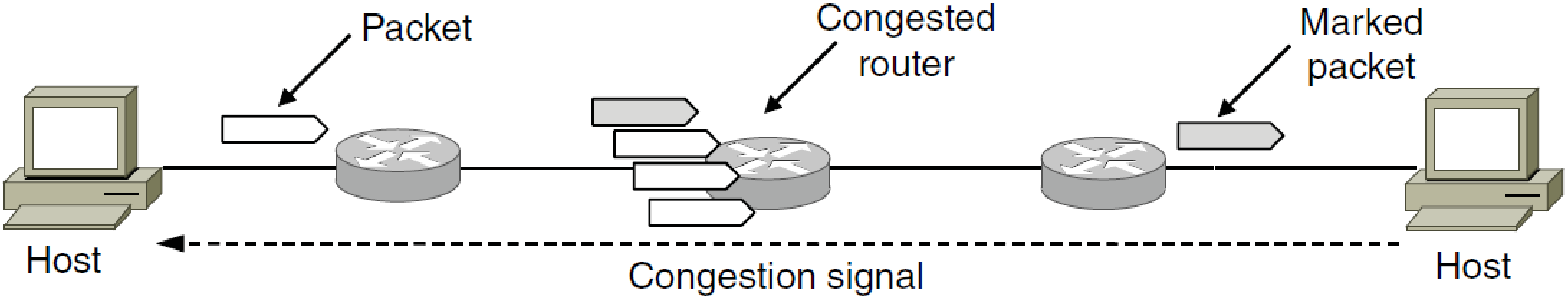
ECN (Explicit Congestion Notification)

- Router detects the onset of congestion via its queue
  - When congested, it marks affected packets (IP header)



ECN (2)

- Marked packets arrive at receiver; treated as loss
  - TCP receiver reliably informs TCP sender of the congestion



ECN (3)

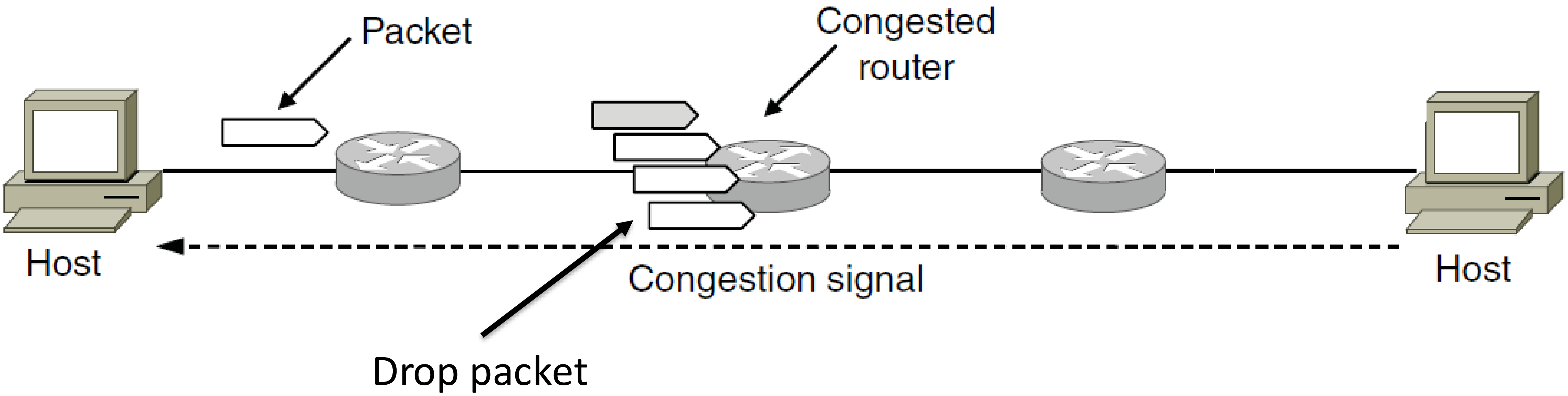
- Advantages:
  - Routers deliver clear signal to hosts
  - Congestion is detected early, no loss
  - No extra packets need to be sent
- Disadvantages:
  - Routers and hosts must be upgraded (currently 1%)
  - More work at router

Random Early Detection (RED)

- Jacobson (again!) and Floyd
- Alternative idea: instead of marking packets, drop
  - We know they're using TCP, make use of that fact
- Signals congestion to sender
  - But without adding headers or doing packet inspection
- Drop at random, depending on queue size
  - If queue empty, accept packet always
  - If queue full, always drop
  - As queue approaches full, increase likelihood of packet drop

RED (Random Early Detection)

- Router detects the onset of congestion via its queue
  - Prior to congestion, drop a packet to signal



RED (Random Early Detection)

- Sender enters MD, slows packet flow
  - We shed load, everyone is happy