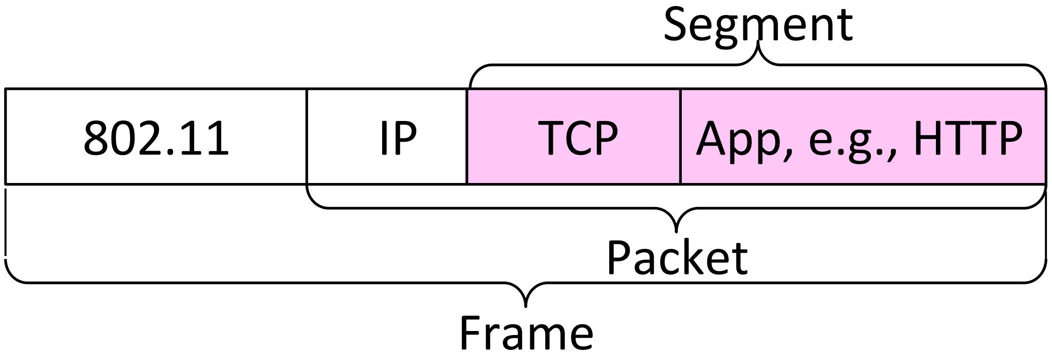
Where we are in the Course

- Starting the Transport Layer!
  - Builds on the network layer to deliver data across networks for applications with the desired reliability or quality

Application
Transport
Network
Link
Physical

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)

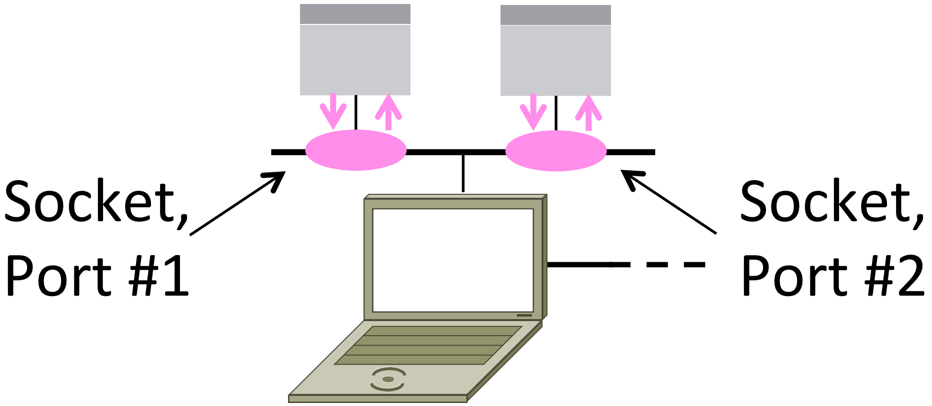
Comparison of Internet Transports

- 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

- Sockets let apps attach to the local network at different ports



Socket API (3)

- Same API used for Streams and Datagrams

	Primitive	Meaning
	SOCKET	Create a new communication endpoint
	BIND	Associate a local address (port) with a socket
Only needed for Streams	LISTEN	Announce willingness to accept connections
	ACCEPT	Passively establish an incoming connection
	CONNECT	Actively attempt to establish a connection
To/From forms for Datagrams	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, 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

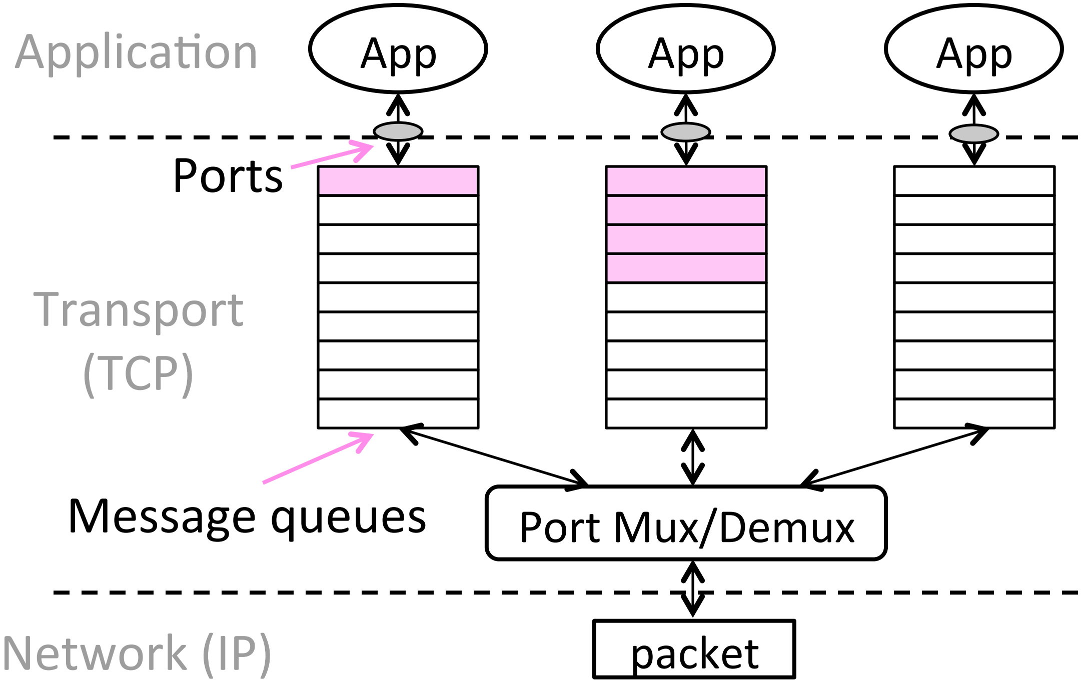
Port	Protocol	Use
20, 21	FTP	File transfer
22	SSH	Remote login, replacement for Telnet
25	SMTP	Email
80	HTTP	World Wide Web
110	POP-3	Remote email access
143	IMAP	Remote email access
443	HTTPS	Secure Web (HTTP over SSL/TLS)
543	RTSP	Media player control
631	IPP	Printer sharing

User Datagram Protocol (UDP)

- Used by apps that don't want reliability or bytestreams
  - Voice-over-IP (unreliable)
  - DNS, RPC (message-oriented)
  - DHCP (bootstrapping)

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

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

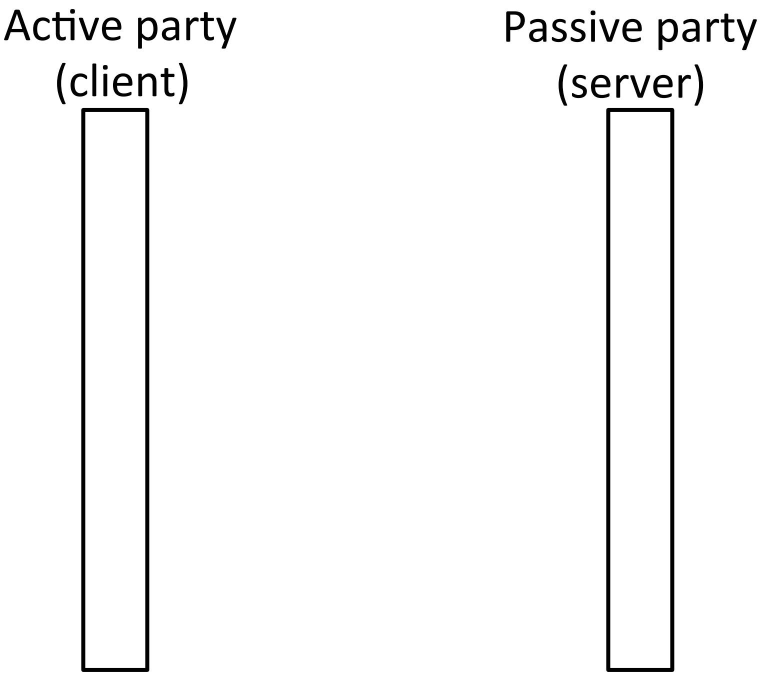
Connection Establishment

- 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

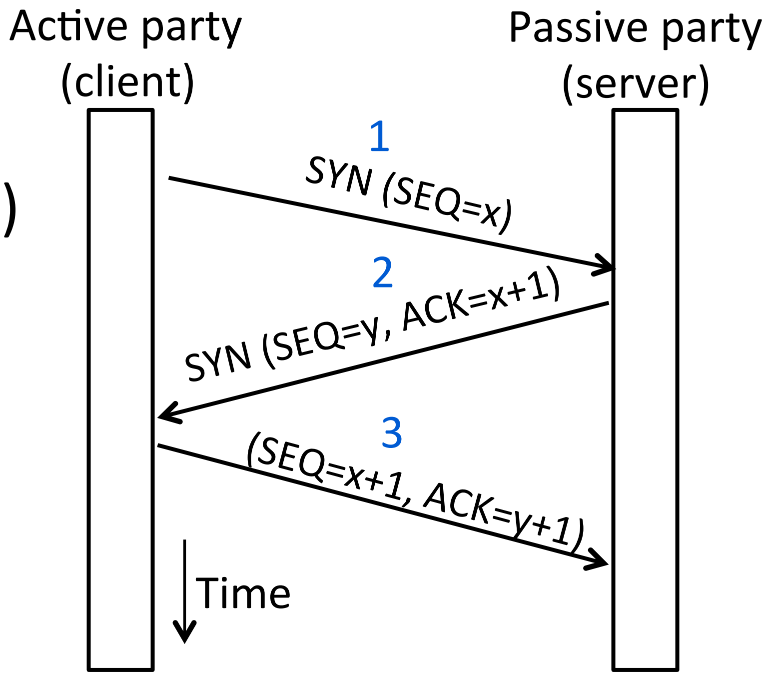
Three-Way Handshake

- 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



Connection Release

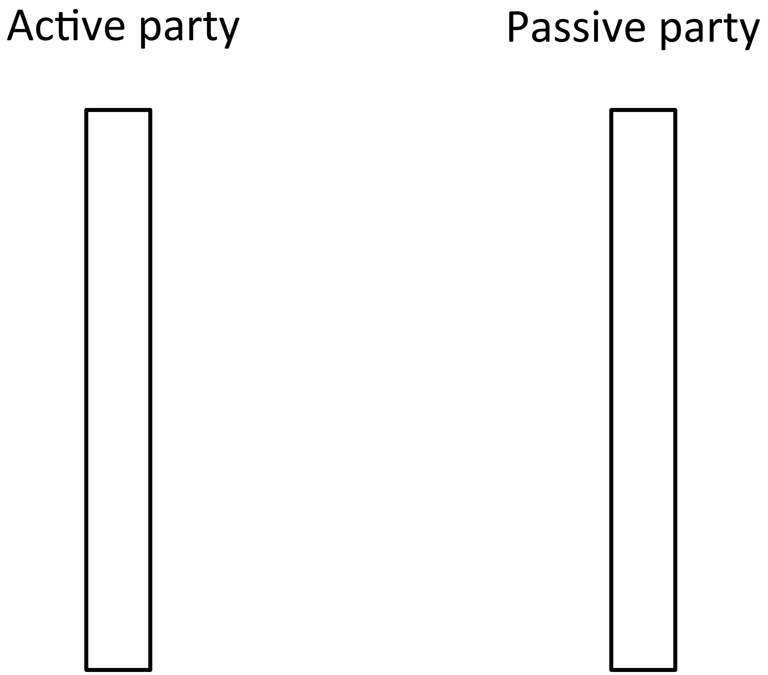
- Orderly release by both parties when done
  - Delivers all pending data and “hangs up”
  - Cleans up state in sender and receiver

- Key problem is to provide reliability while releasing
  - TCP uses a “symmetric” close in which both sides shutdown independently

TCP Connection Release

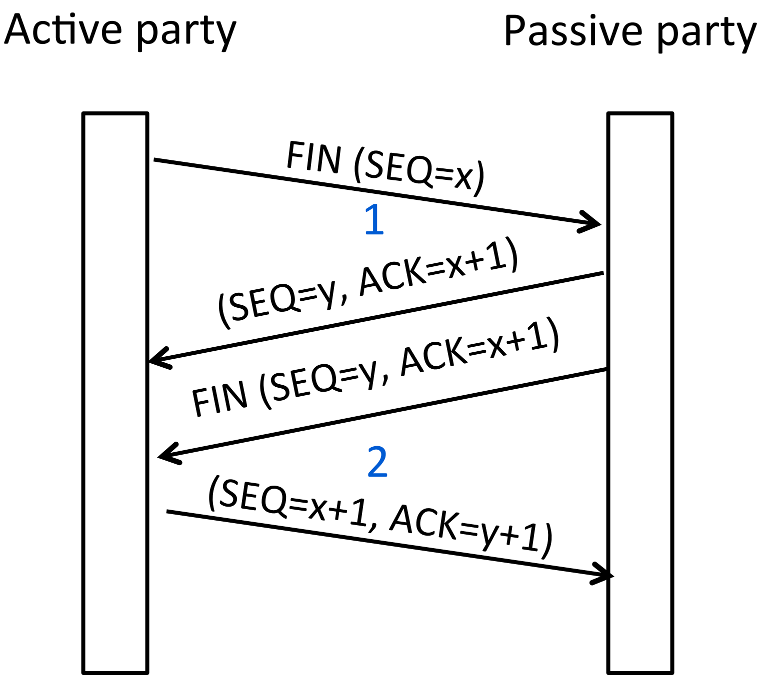
- Two steps:
  – Active sends FIN(x), passive ACKs
  – Passive sends FIN(y), active ACKs
  – FINs are retransmitted if lost

- 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

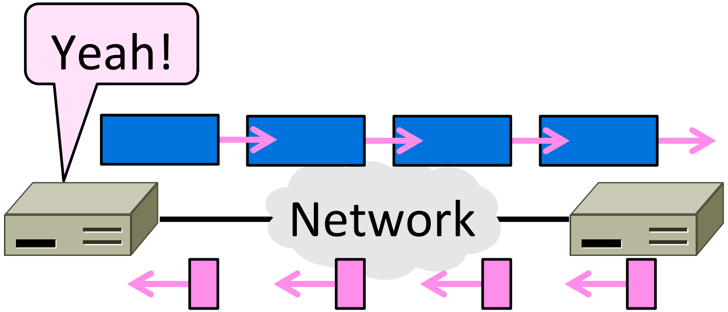


TIME_WAIT State

- We wait a long time (two times the maximum segment lifetime of 60 seconds) after sending all segments and before completing the close
- Why?
  - ACK might have been lost, in which case FIN will be resent for an orderly close
  - Could otherwise interfere with a subsequent connection

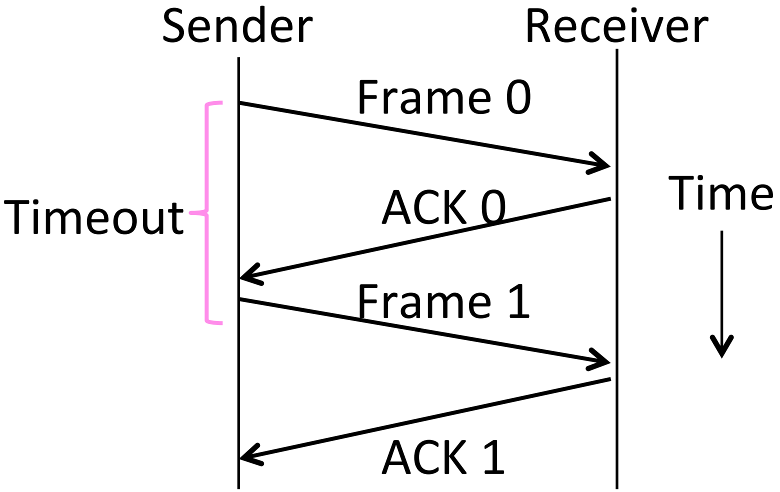
Sliding Window

- The sliding window algorithm
  - Pipelining and reliability
  - Building on Stop-and-Wait



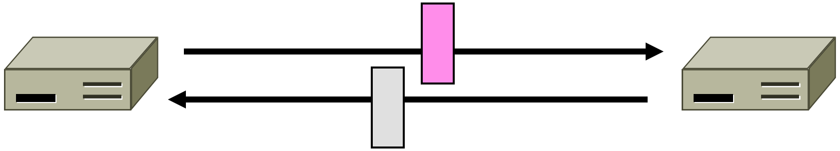
Recall

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



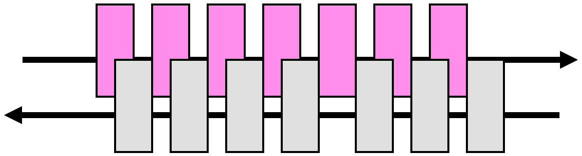
Limitation of Stop-and-Wait

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



Sliding Window

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



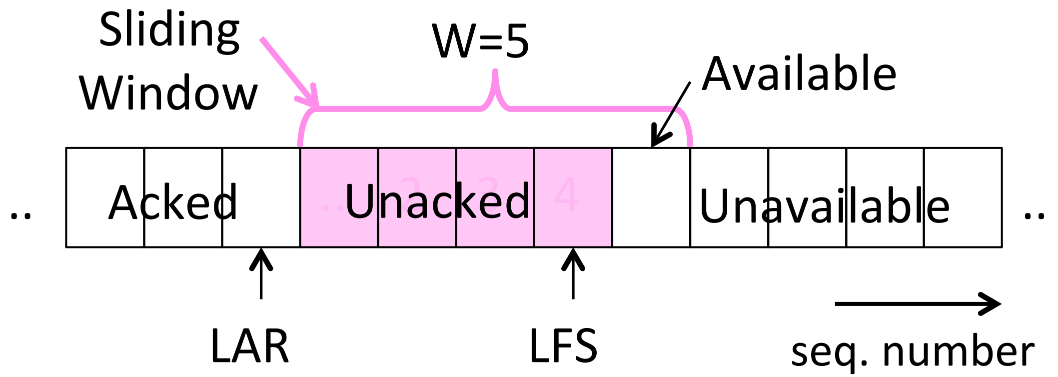
  - Pipelining improves performance
  - Need W=2BD to fill network path

Sliding Window Protocol

- Many variations, depending on how buffers, acknowledgements, and retransmissions are handled
- Go-Back-N »
  - Simplest version, can be inefficient
- Selective Repeat »
  - More complex, better performance

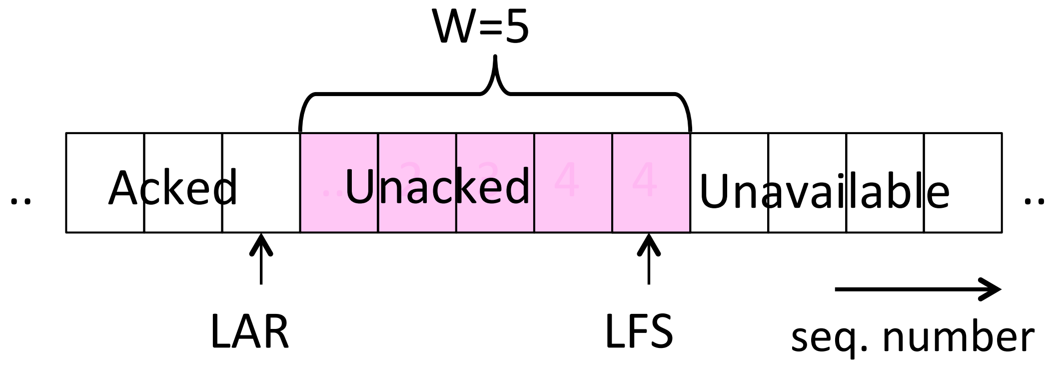
Sliding Window – Sender

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



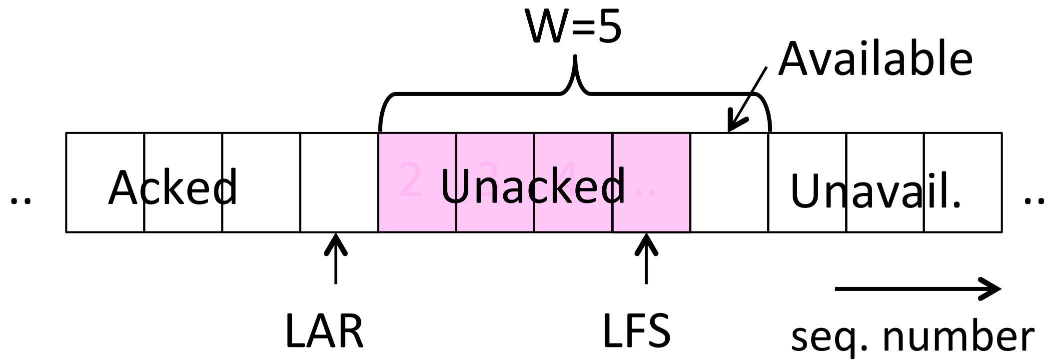
Sliding Window – Sender (2)

- 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]
  - Pass up to app in-order segments from LAS+1, and update LAS
  - Send ACK for LAS regardless

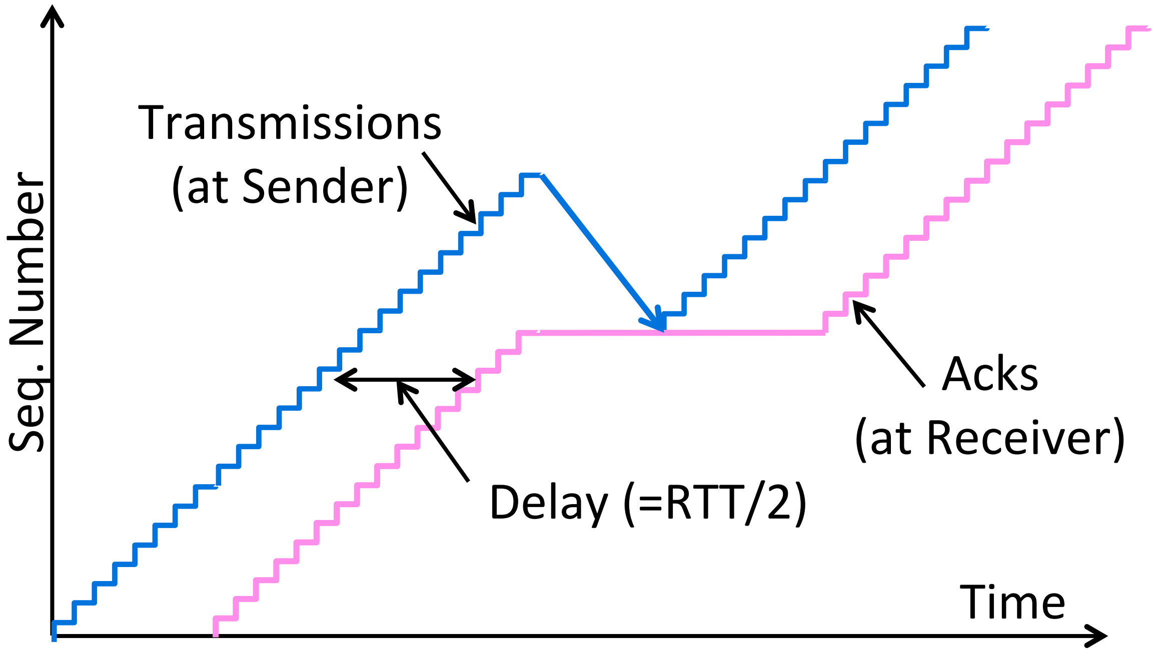
Sliding Window – Retransmissions

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

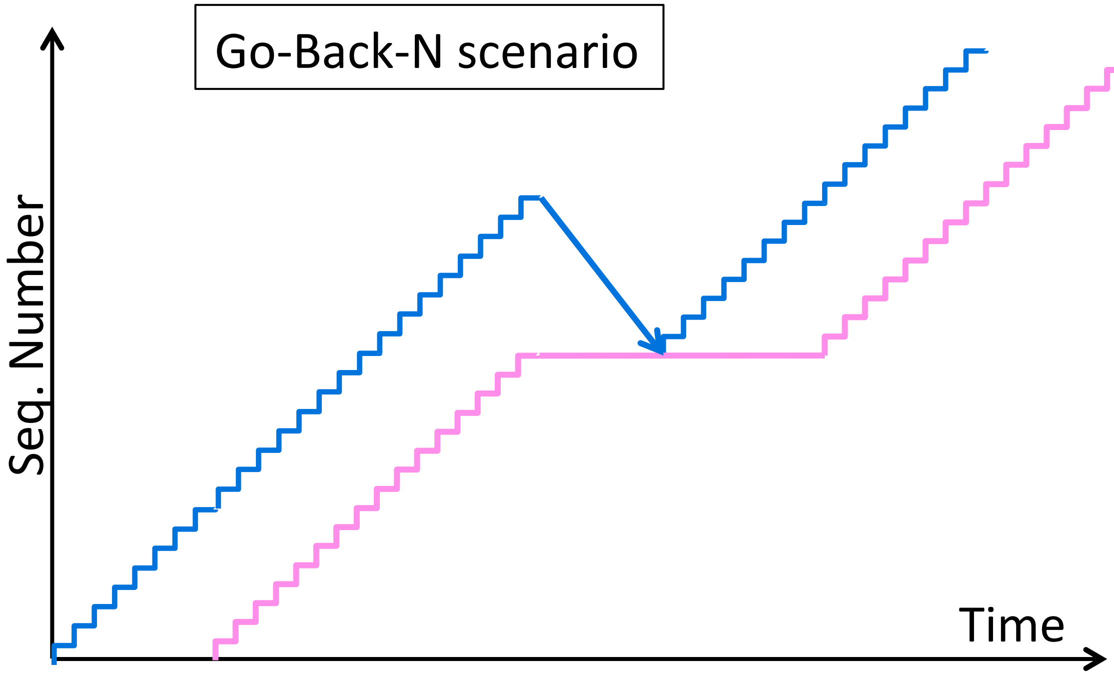
Sequence Numbers

- Need more than 0/1 for Stop-and-Wait …
  - But how many?
- For Selective Repeat, need W numbers for packets, plus W for acks of earlier packets
  - 2W seq. numbers
  - Fewer for Go-Back-N (W+1)
- Typically implement seq. number with an N-bit counter that wraps around at 2^N—1
  - E.g., N=8: …, 253, 254, 255, 0, 1, 2, 3, …

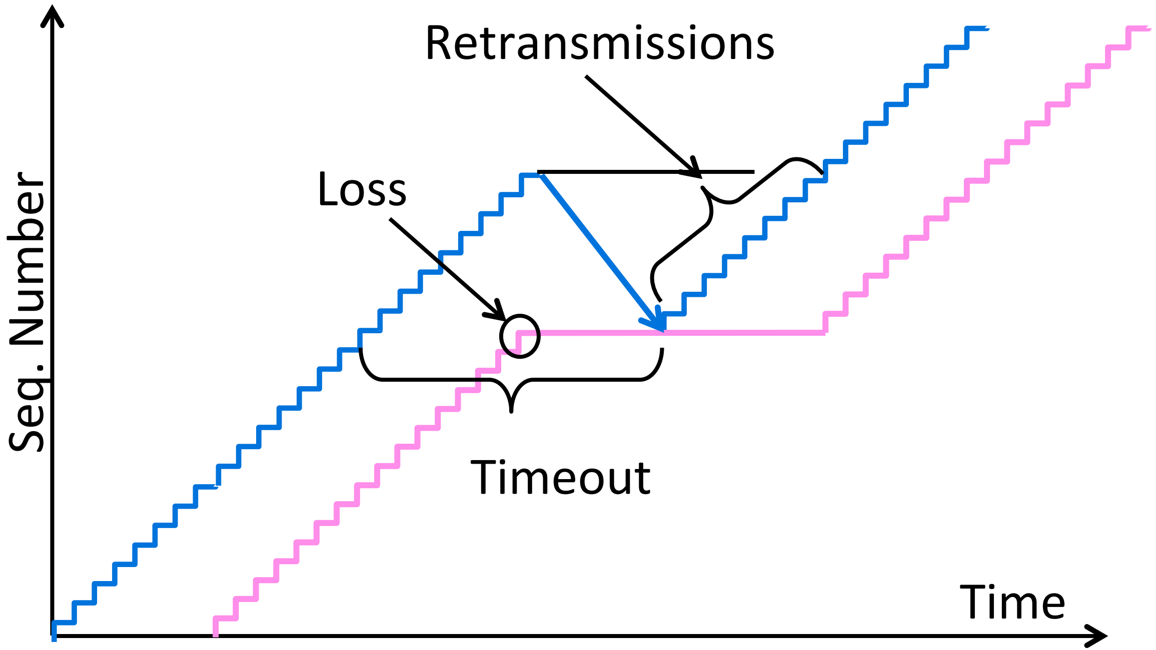
Sequence Time Plot



Sequence Time Plot (2)

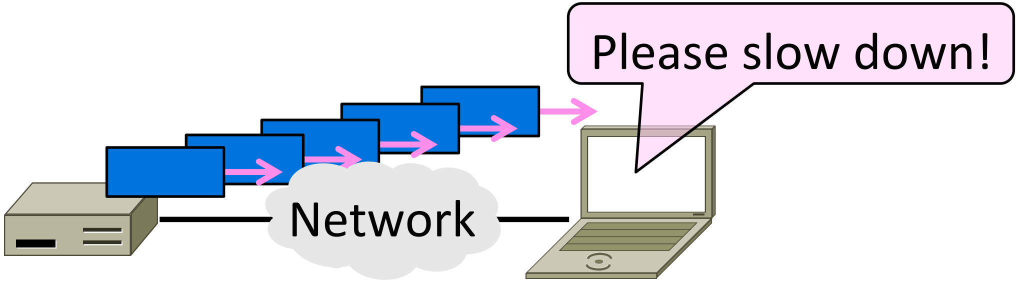


Sequence Time Plot (3)



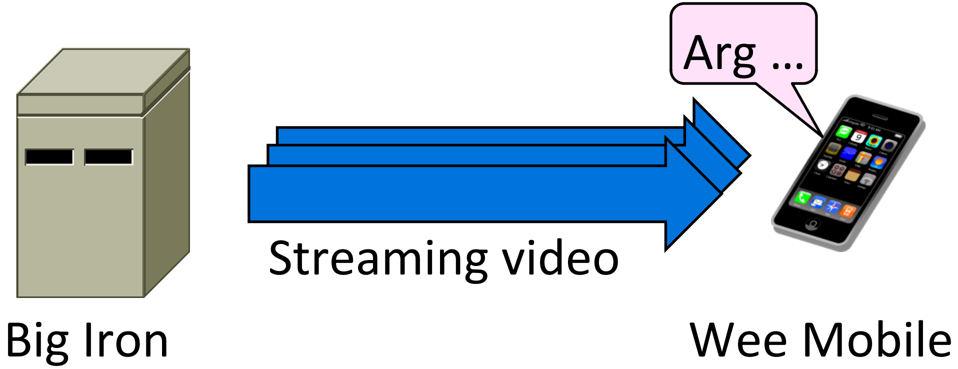
Flow Control

- Adding flow control to the sliding window algorithm
  - To slow the over-enthusiastic sender



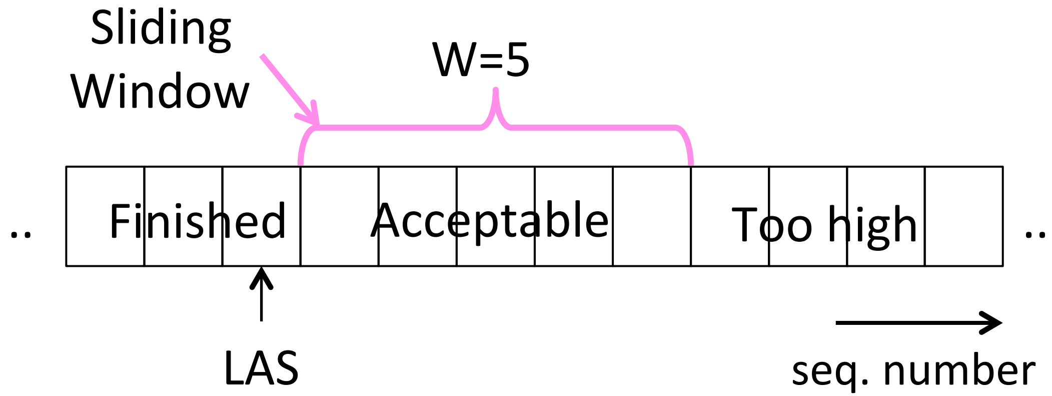
Problem

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



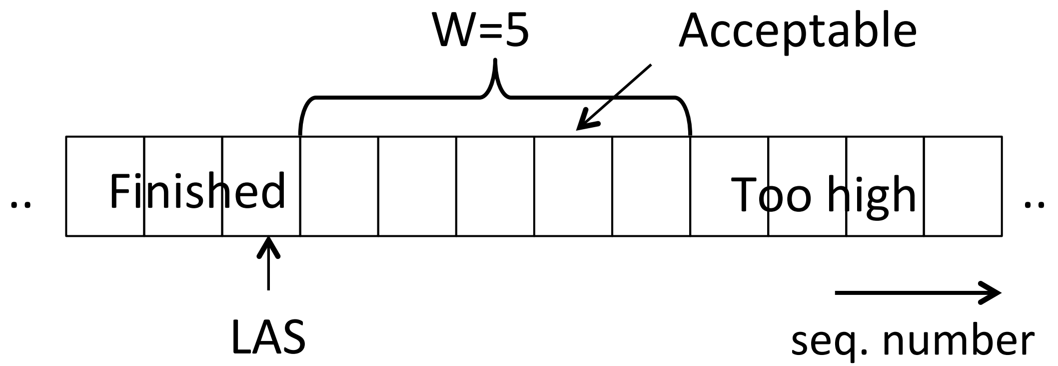
Sliding Window – Receiver

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



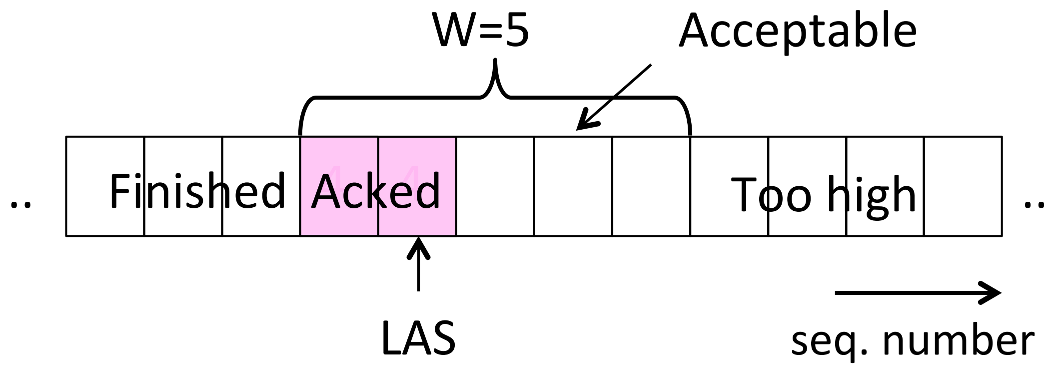
Sliding Window – Receiver (2)

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



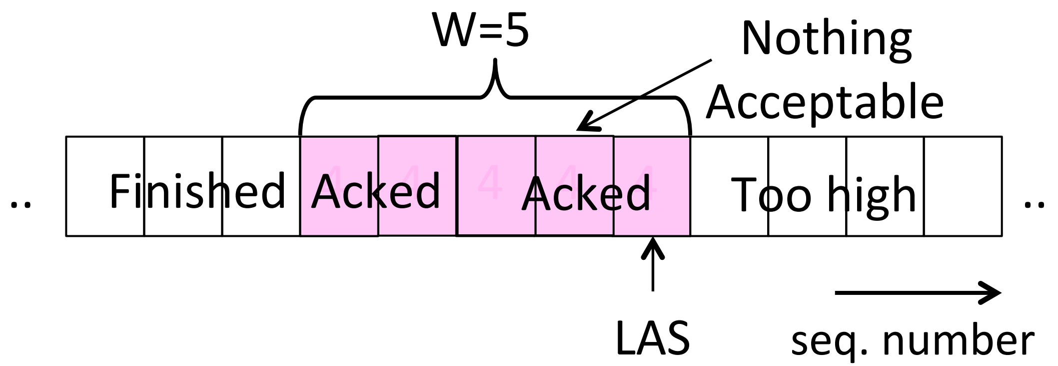
Sliding Window – Receiver (3)

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



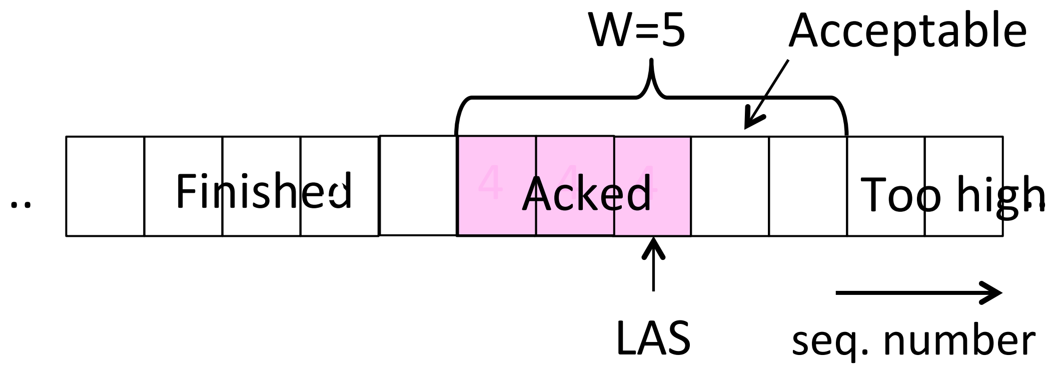
Sliding Window – Receiver (4)

- If further segments arrive (even in order) we can fill the buffer
  - Must drop segments until app recvs!



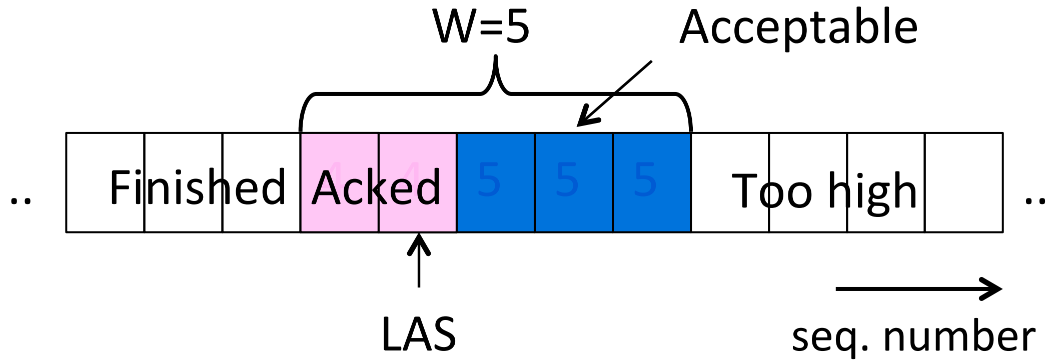
Sliding Window – Receiver (5)

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



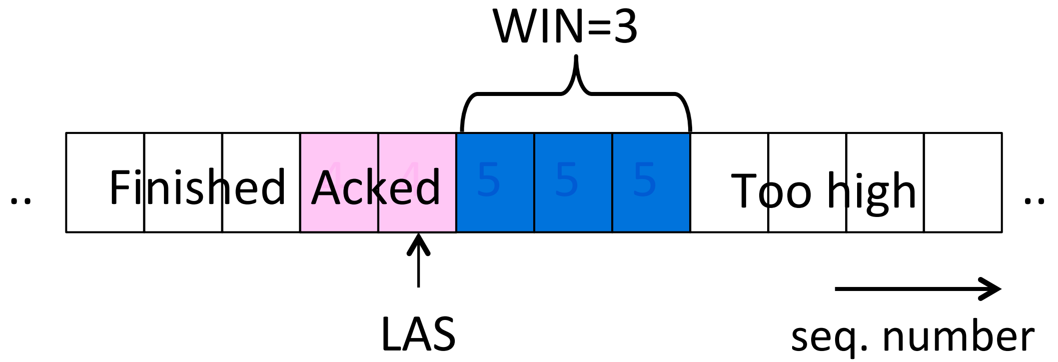
Flow Control

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



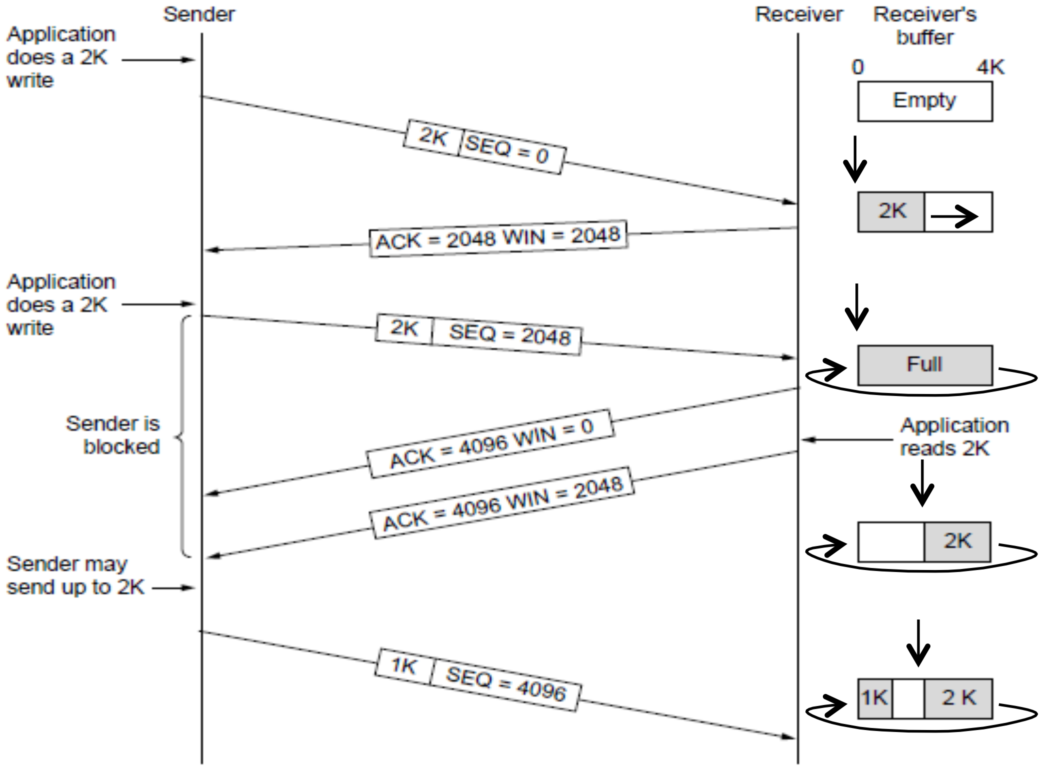
Flow Control (2)

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



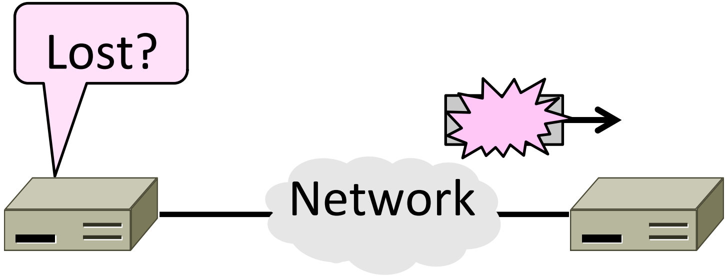
Flow Control (3)

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



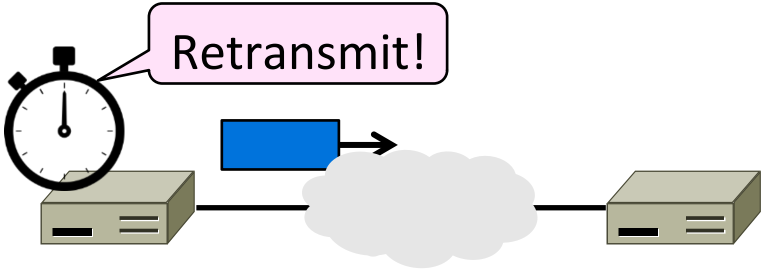
Topic

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



Retransmissions

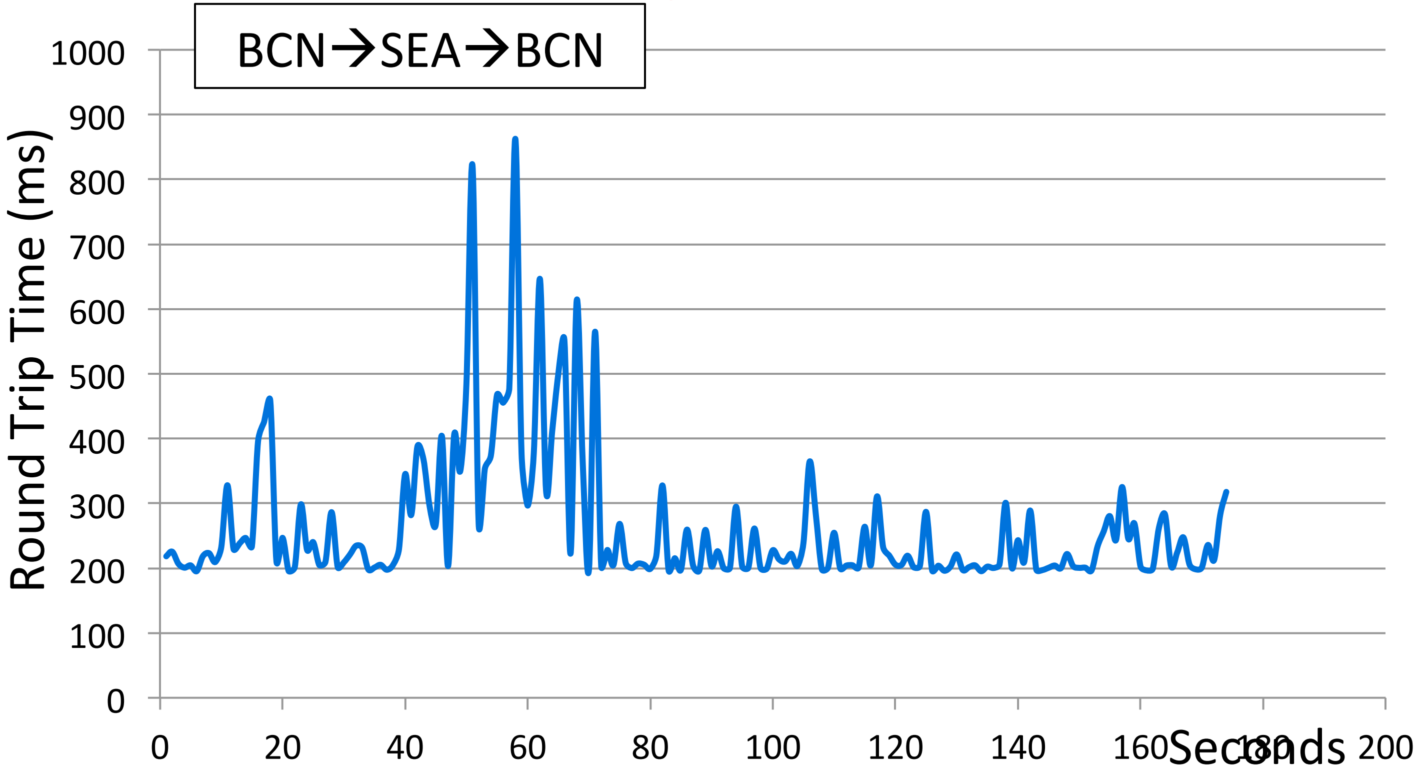
- With sliding window, the strategy for detecting loss is the timeout
  - Set timer when a segment is sent
  - Cancel timer when ack is received
  - If timer fires, retransmit data as lost



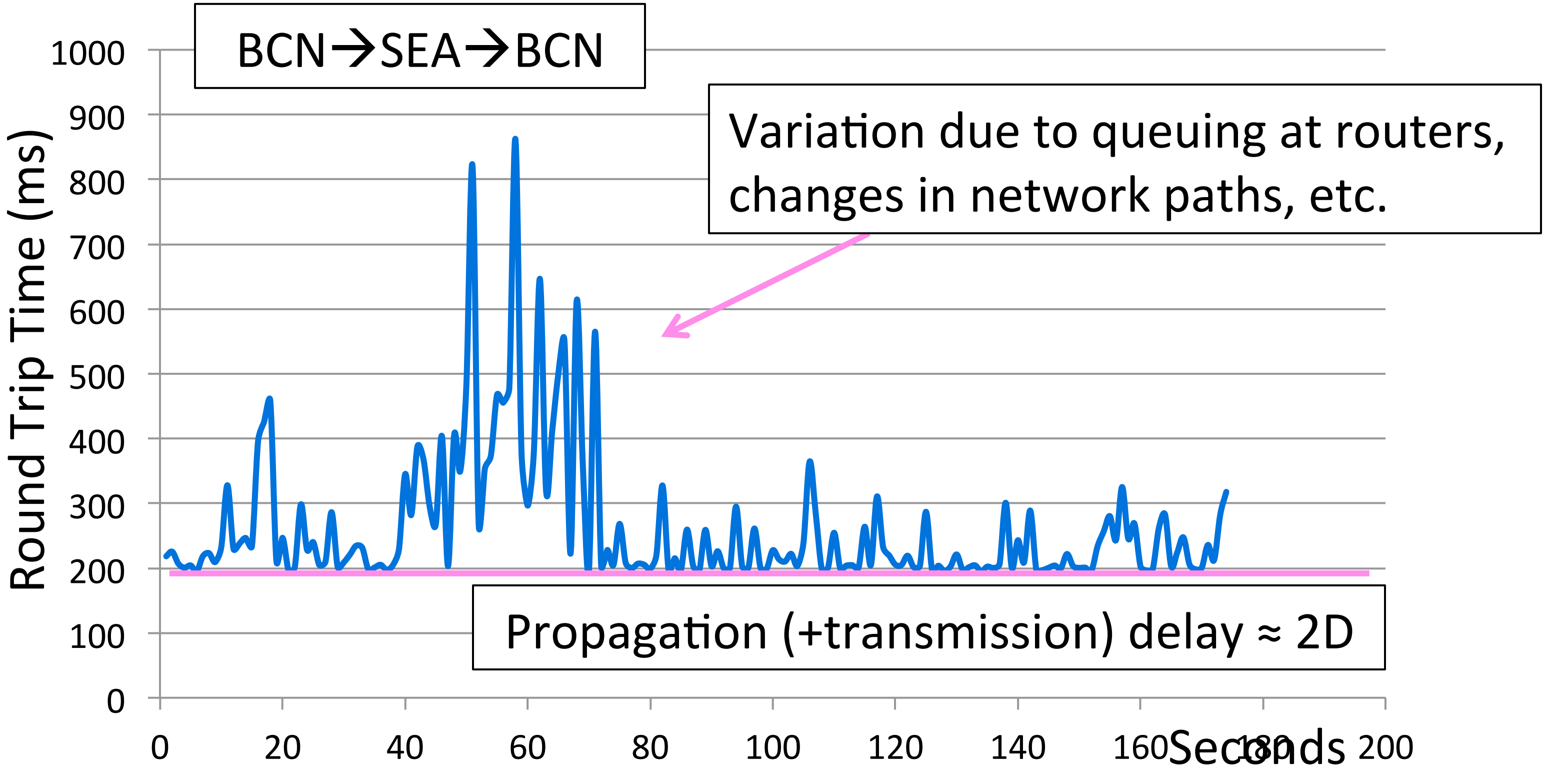
Timeout Problem

- 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

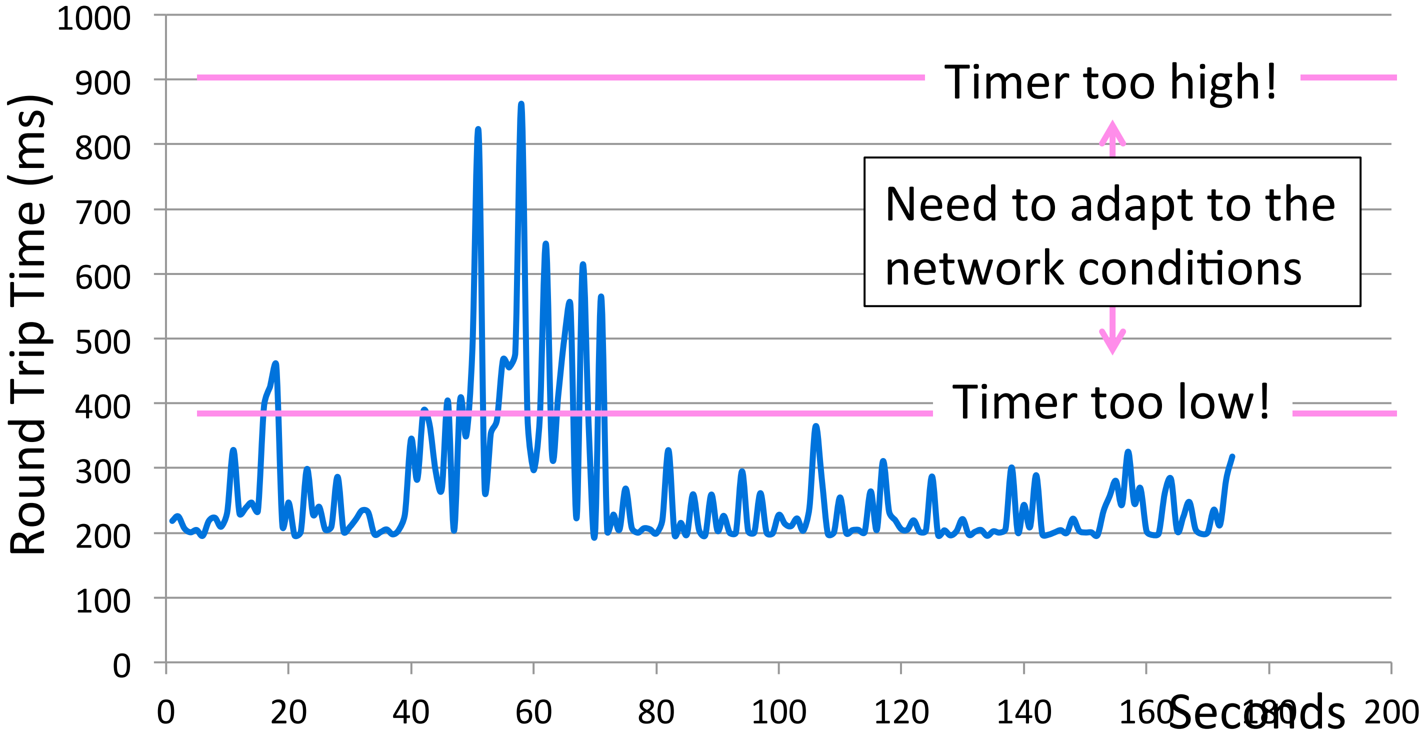
Example of RTTs



Example of RTTs (2)



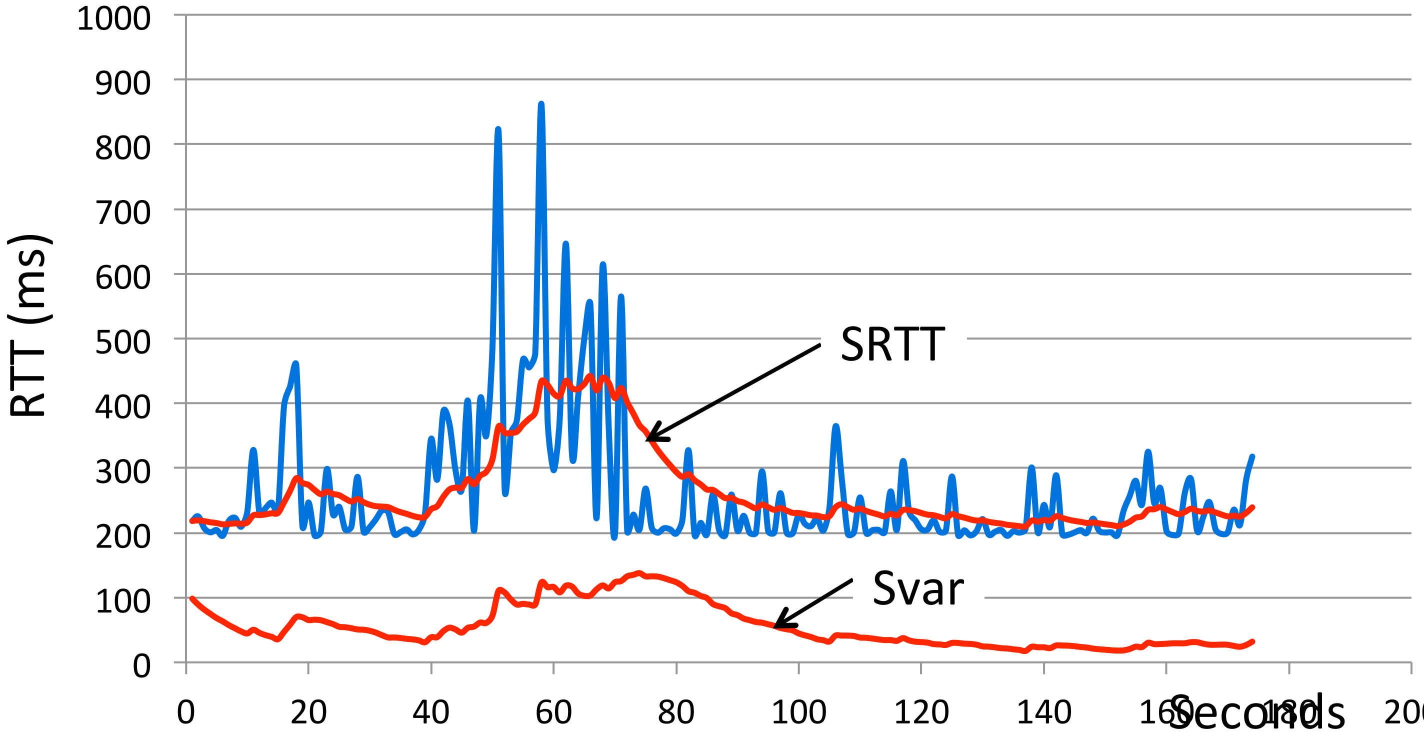
Example of RTTs (3)



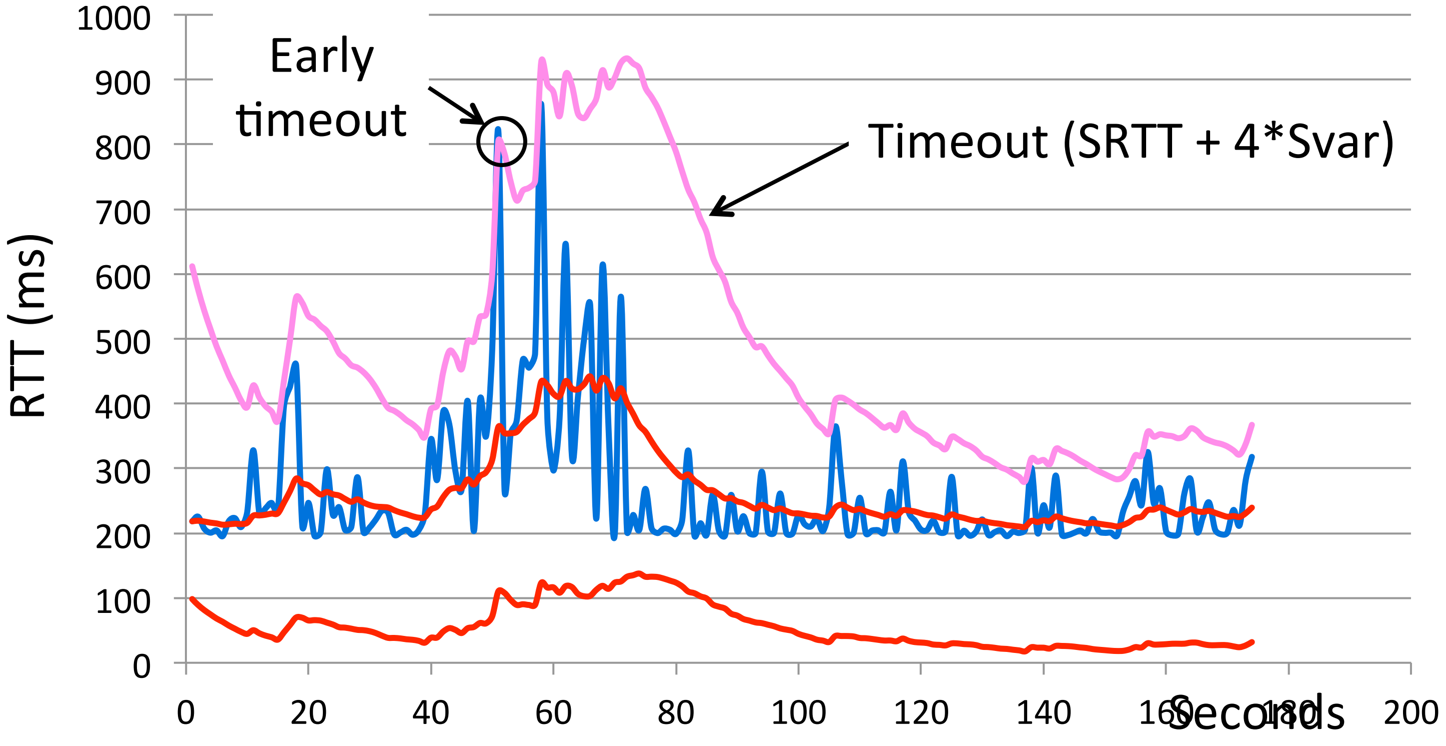
Adaptive Timeout

- Keep 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)



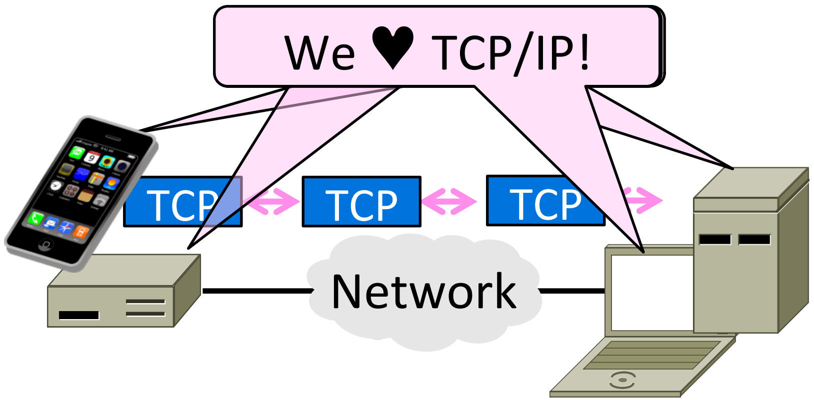
Adaptive Timeout (2)

- 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

Topic

- How TCP works!
  - The transport protocol used for most content on the Internet



TCP Features

- A reliable bytestream service »
- Based on connections
- Sliding window for reliability »
  - With adaptive timeout
- Flow control for slow receivers

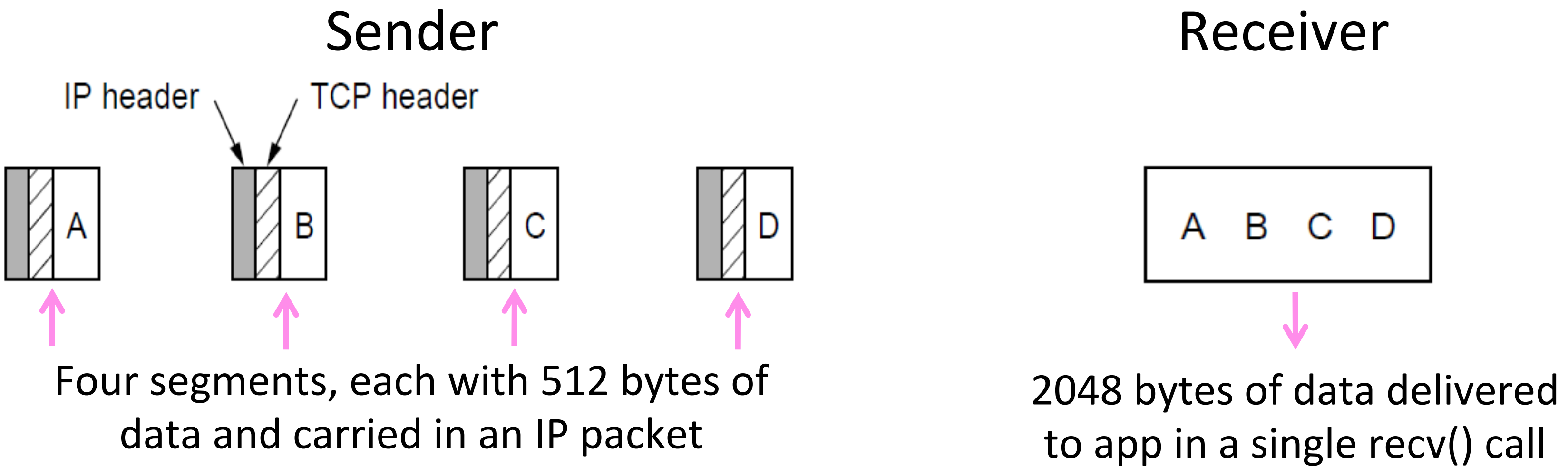
This time

- Congestion control to allocate network bandwidth

Next time

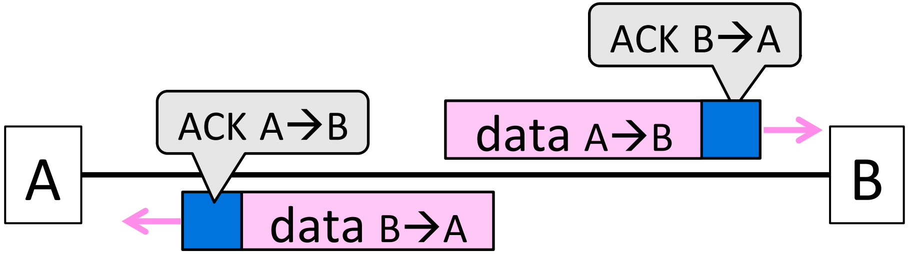
Reliable Bytestream

- Message boundaries not preserved from send() to recv()
  - But reliable and ordered (receive bytes in same order as sent)



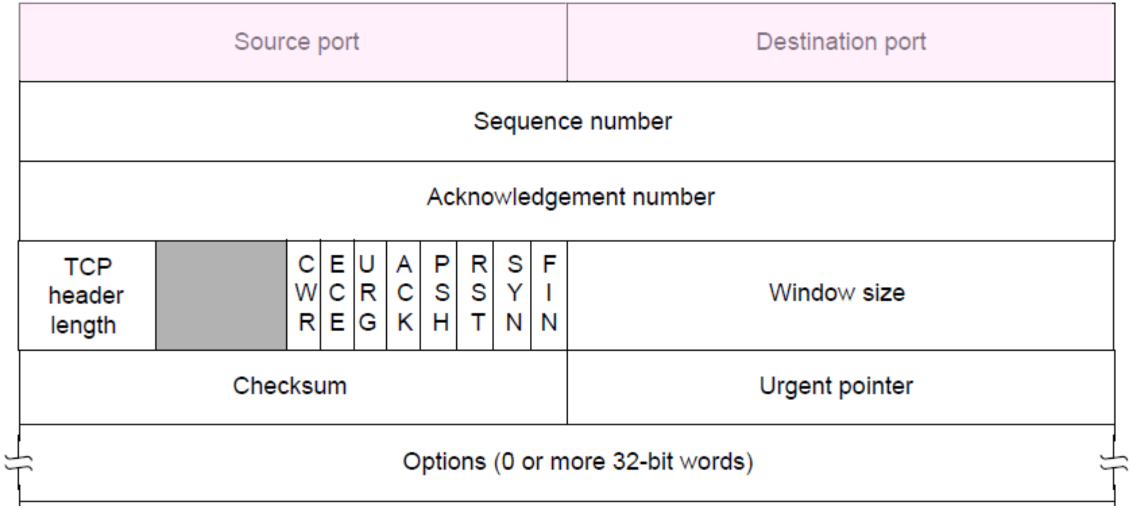
Reliable Bytestream (2)

- Bidirectional data transfer
  - Control information (e.g., ACK) piggybacks on data segments in reverse direction



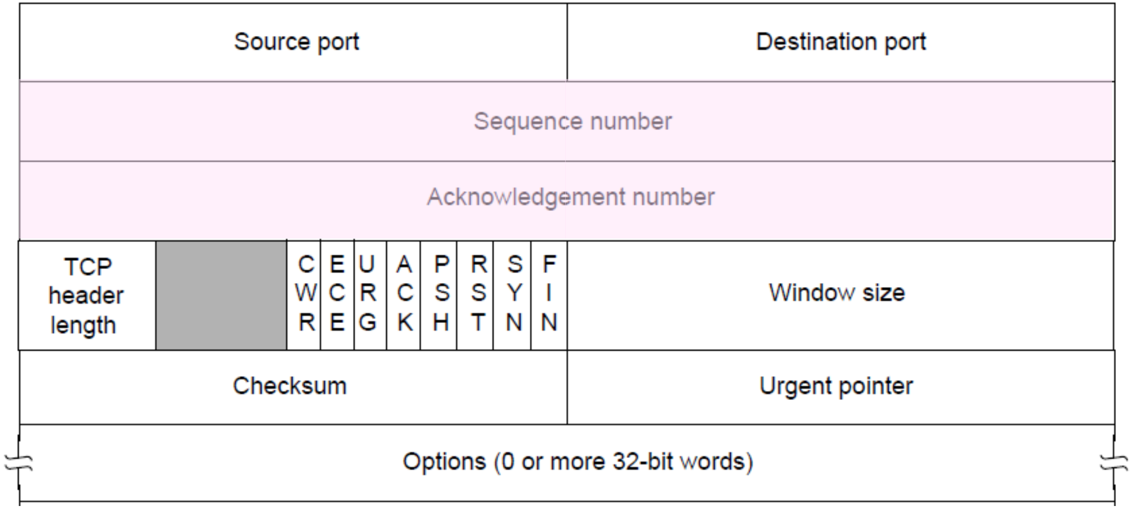
TCP Header (1)

- Ports identify apps (socket API)
  - 16-bit identifiers



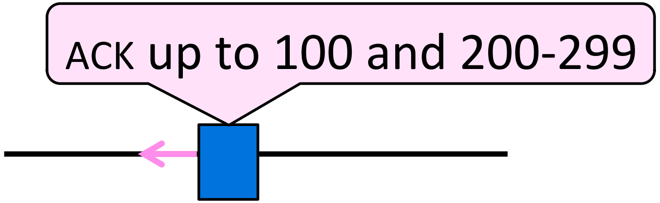
TCP Header (2)

- SEQ/ACK used for sliding window
  - Selective Repeat, with byte positions



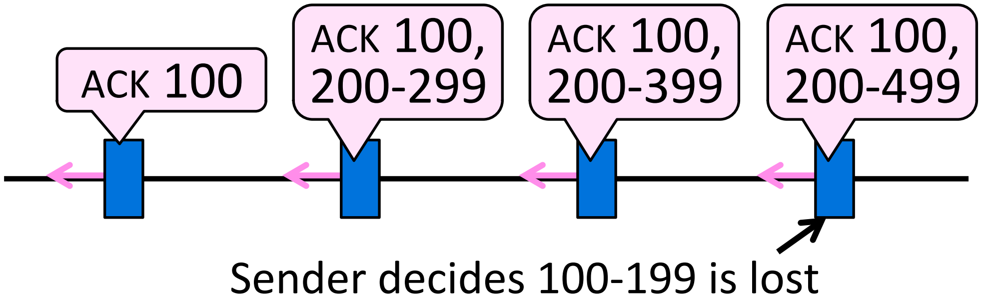
TCP Sliding Window – Receiver

- Cumulative ACK tells next expected byte sequence number ("LAS+1")
- Optionally, selective ACKs (SACK) give hints for receiver buffer state
  - List up to 3 ranges of received bytes



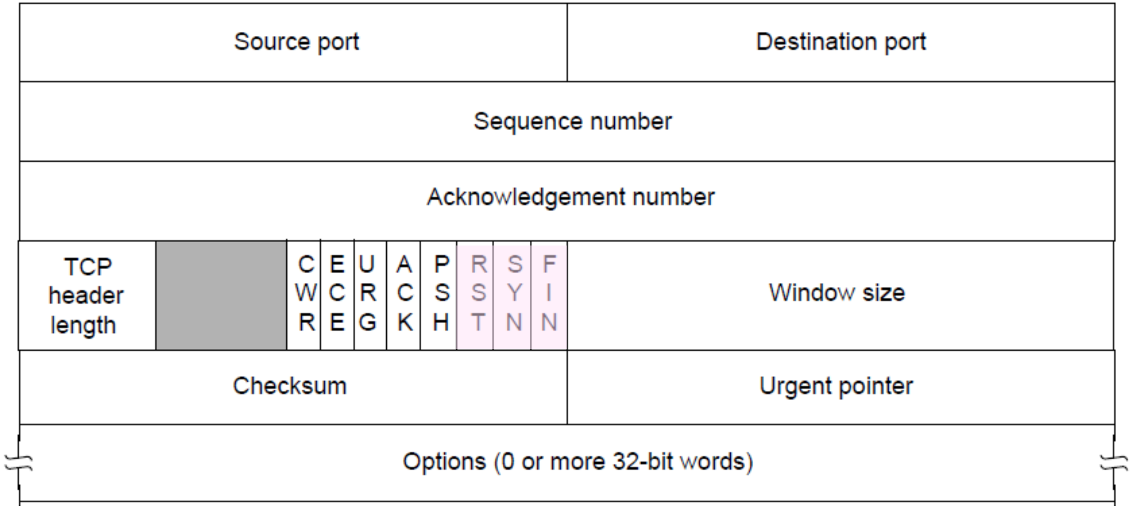
TCP Sliding Window – Sender

- Uses an adaptive retransmission timeout to resend data from LAS+1
- Uses heuristics to infer loss quickly and resend to avoid timeouts
  - "Three duplicate ACKS" treated as loss



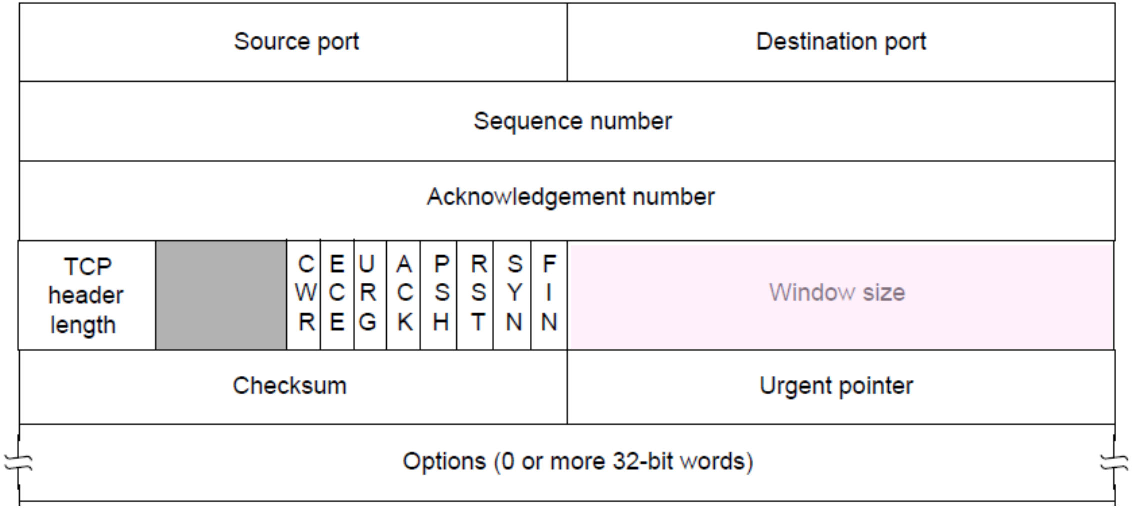
TCP Header (3)

- SYN/FIN/RST flags for connections
  - Flag indicates segment is a SYN etc.



TCP Header (4)

- Window size for flow control
  - Relative to ACK, and in bytes

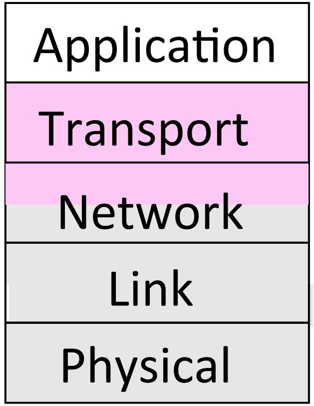


Other TCP Details

- Many, many quirks you can learn about its operation
  - But they are the details
- Biggest remaining mystery is the workings of congestion control
  - We'll tackle this next time!

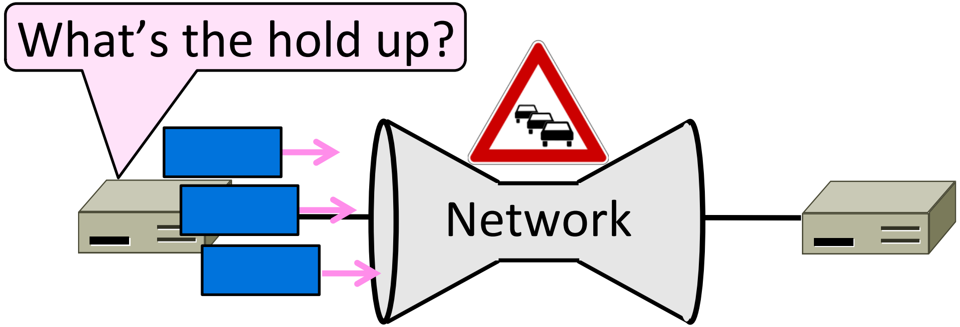
Where we are in the Course

- More fun in the Transport Layer!
  - The mystery of congestion control
  - Depends on the Network layer too



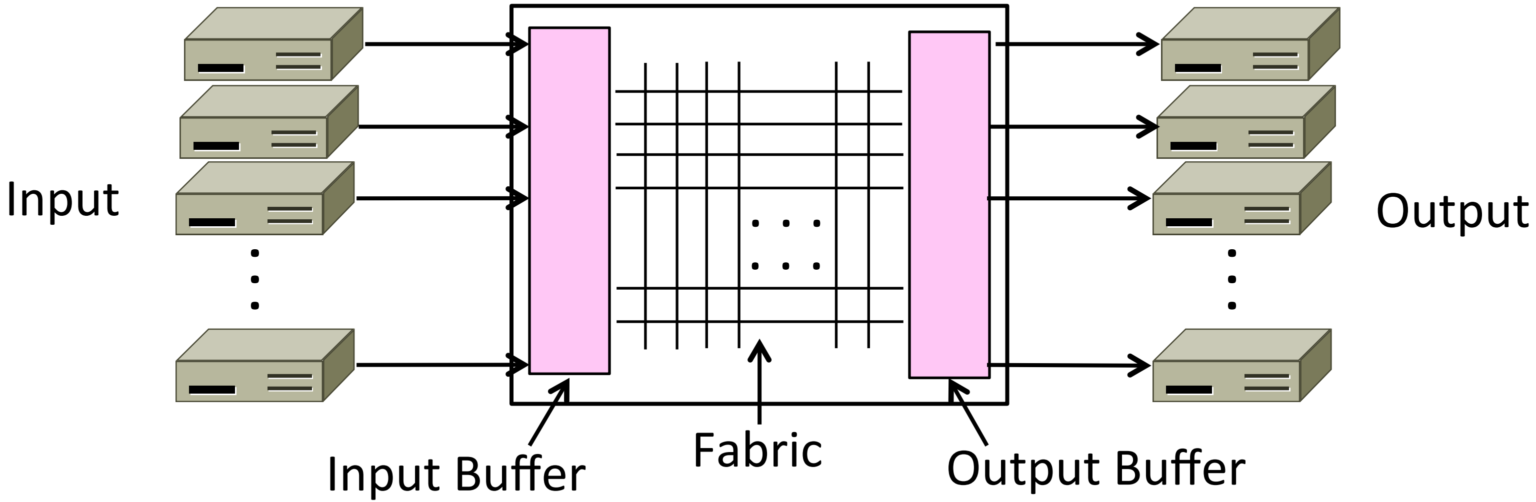
Topic

- Understanding congestion, a "traffic jam" in the network
  - Later we will learn how to control it



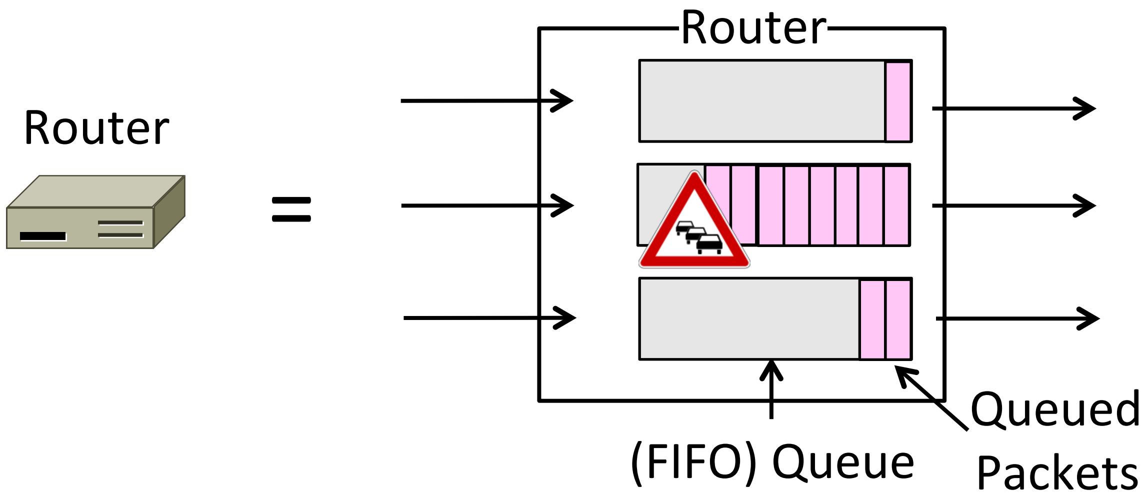
Nature of Congestion

- Routers/switches have internal buffering for contention



Nature of Congestion (2)

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

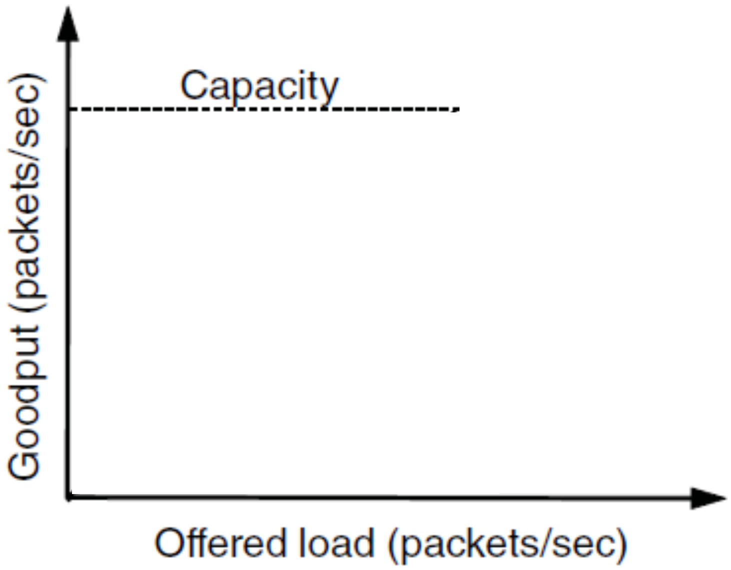


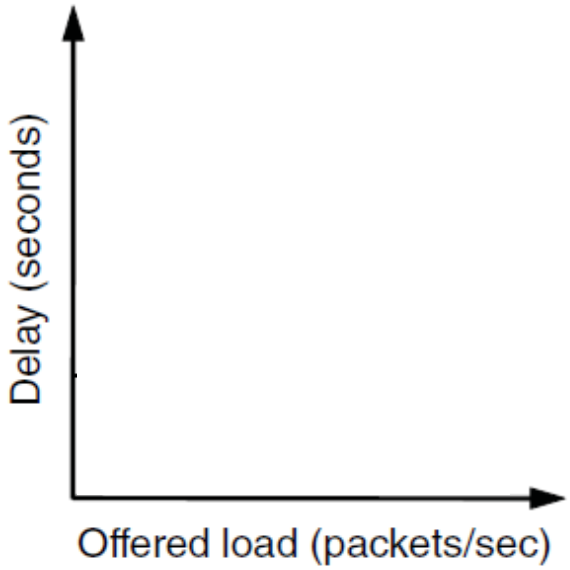
Nature of Congestion (3)

- 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 have the same capacity

Effects of Congestion

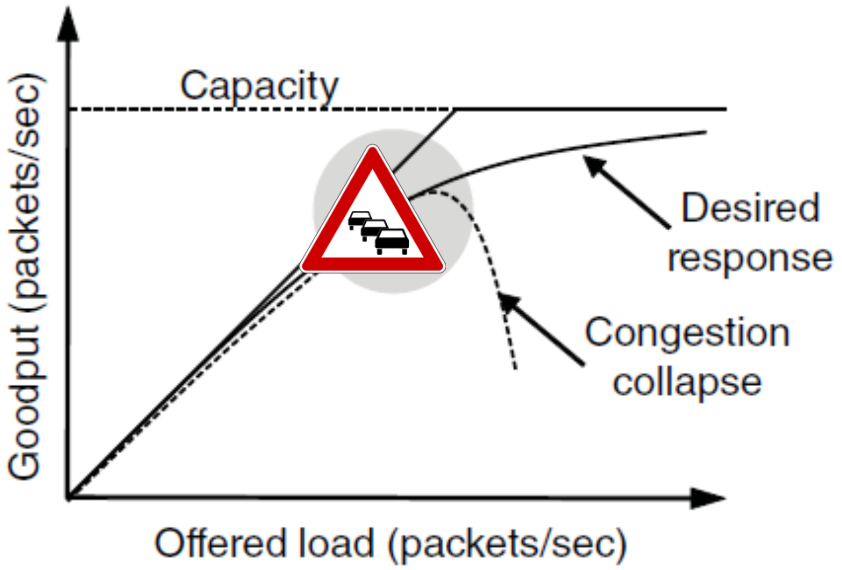
- What happens to performance as we increase the load?

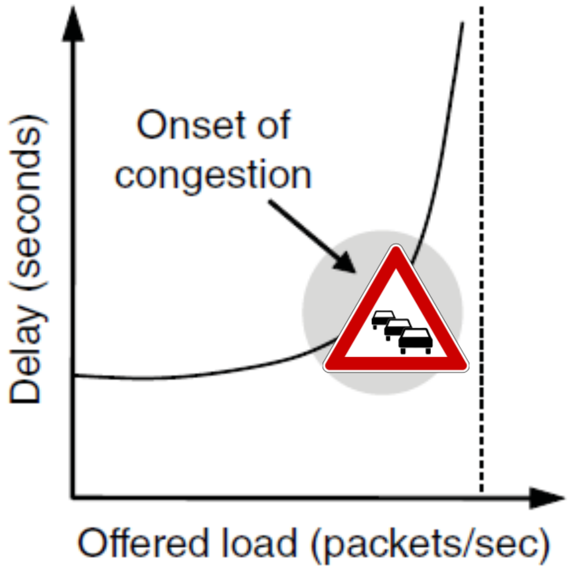




Effects of Congestion (2)

- What happens to performance as we increase the 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 to operate network just before the onset of congestion

Bandwidth Allocation

- Important task for network is to allocate its capacity to senders
  - Good allocation is efficient and fair

- Efficient means most capacity is used but there is no congestion
- Fair 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 is constantly changing
  - Senders may lack capacity in different parts of the 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

Topic

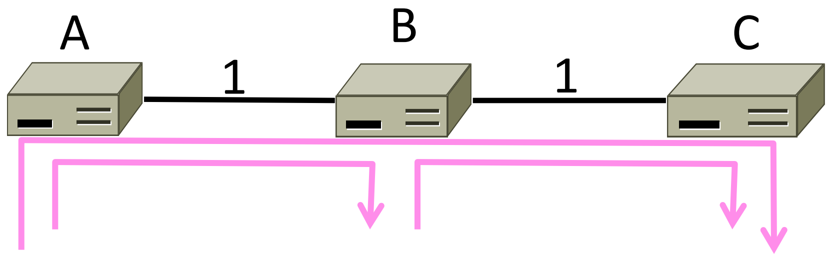
- What's a "fair" bandwidth allocation?
  - The max-min fair allocation

Recall

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

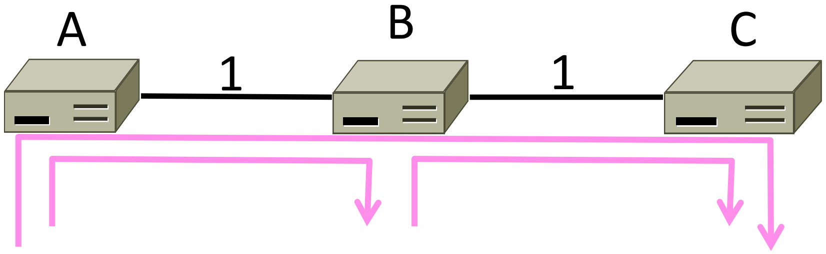
Efficiency vs. Fairness

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



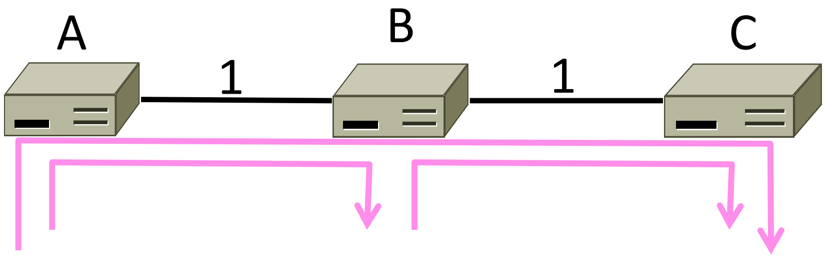
Efficiency vs. Fairness (2)

- If we care about fairness:
  - Give equal bandwidth to each flow
  - A→B: 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
  - A→B: 1 unit, B→C: 1, and A→C, 0
  - Total traffic rises to 2 units!

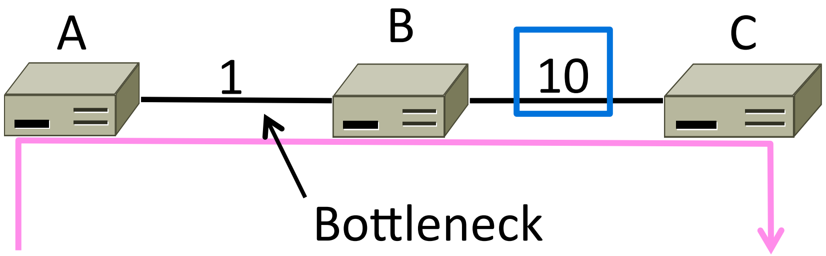


The Slippery Notion of Fairness

- Why is "equal per flow" fair anyway?
  - A→C uses more network resources (two links) 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>
  - "Equal per flow" is good enough

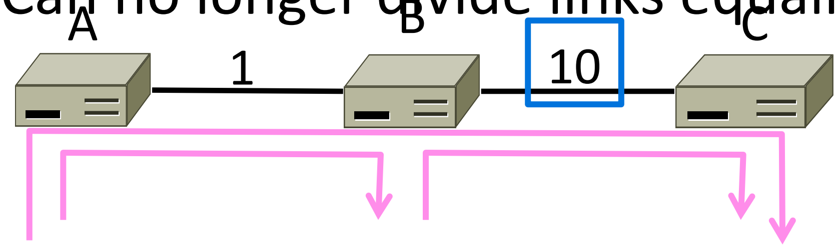
Generalizing "Equal per Flow"

- 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 …



Max-Min Fairness

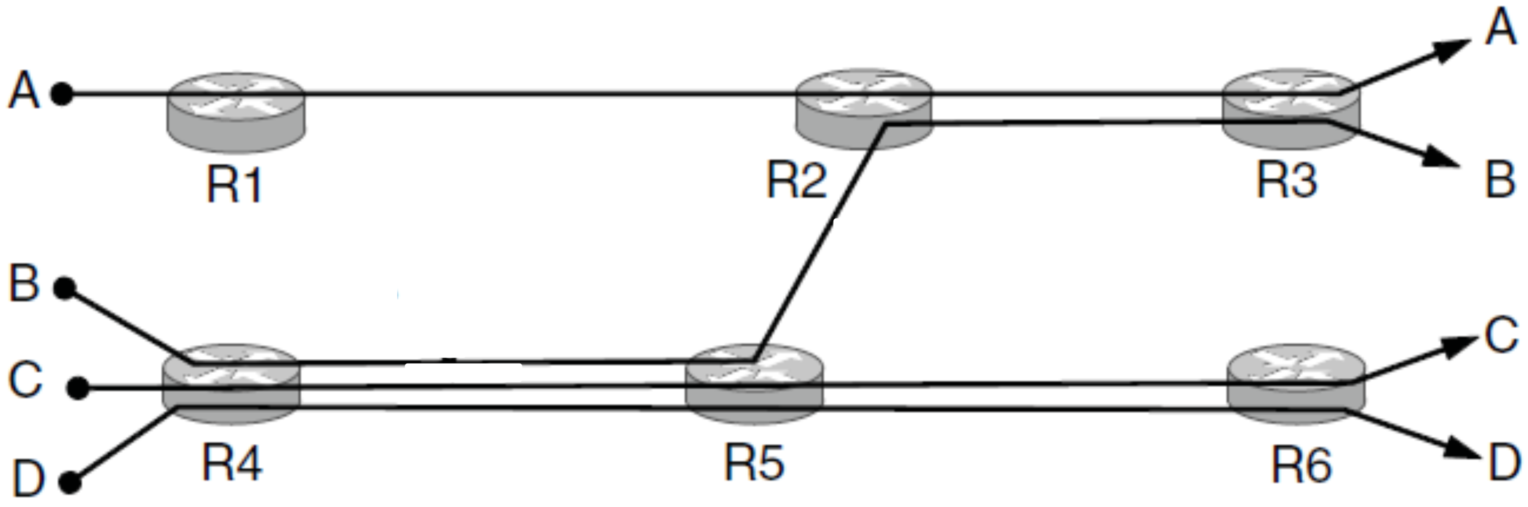
- Intuitively, flows bottlenecked on a link get an equal share of that link
- Max-min fair allocation is one that:
  - Increasing the rate of one flow will decrease the rate of a smaller flow
  - This "maximizes the minimum" flow

Max-Min Fairness (2)

- To find it given a network, imagine "pouring water into the network"
  1. Start with all flows at rate 0
  2. Increase the flows until there is a new bottleneck in the network
  3. Hold fixed the rate of the flows that are bottlenecked
  4. Go to step 2 for any remaining flows

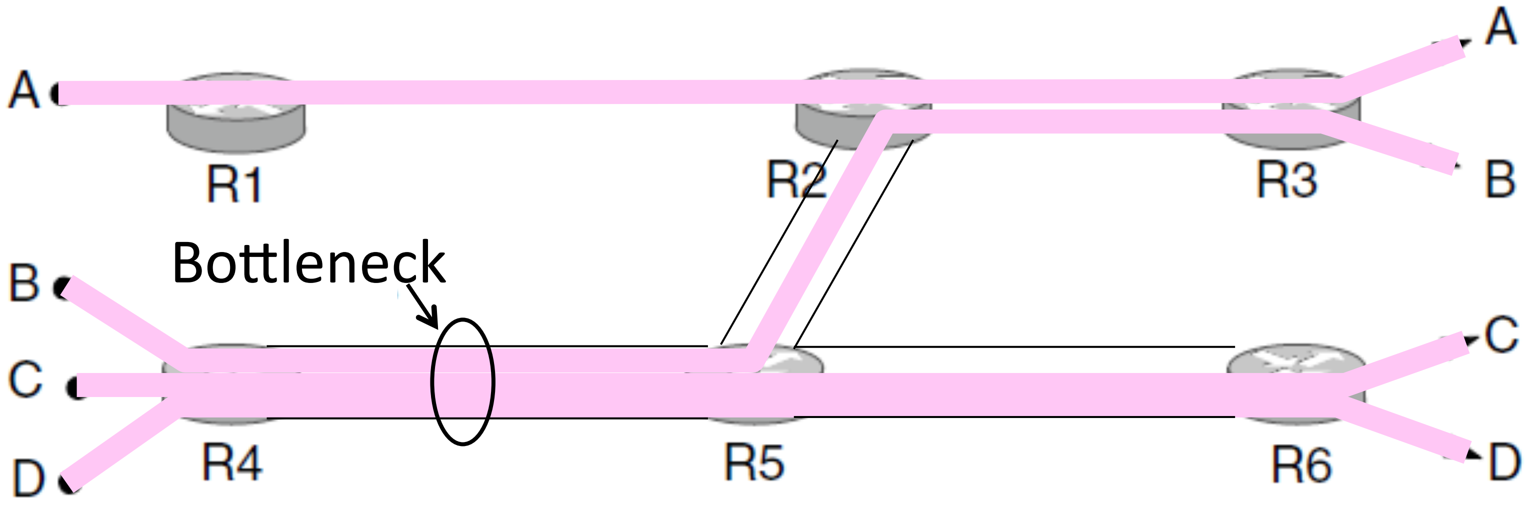
Max-Min Example

- Example: network with 4 flows, links equal bandwidth
  - What is the max-min fair allocation?



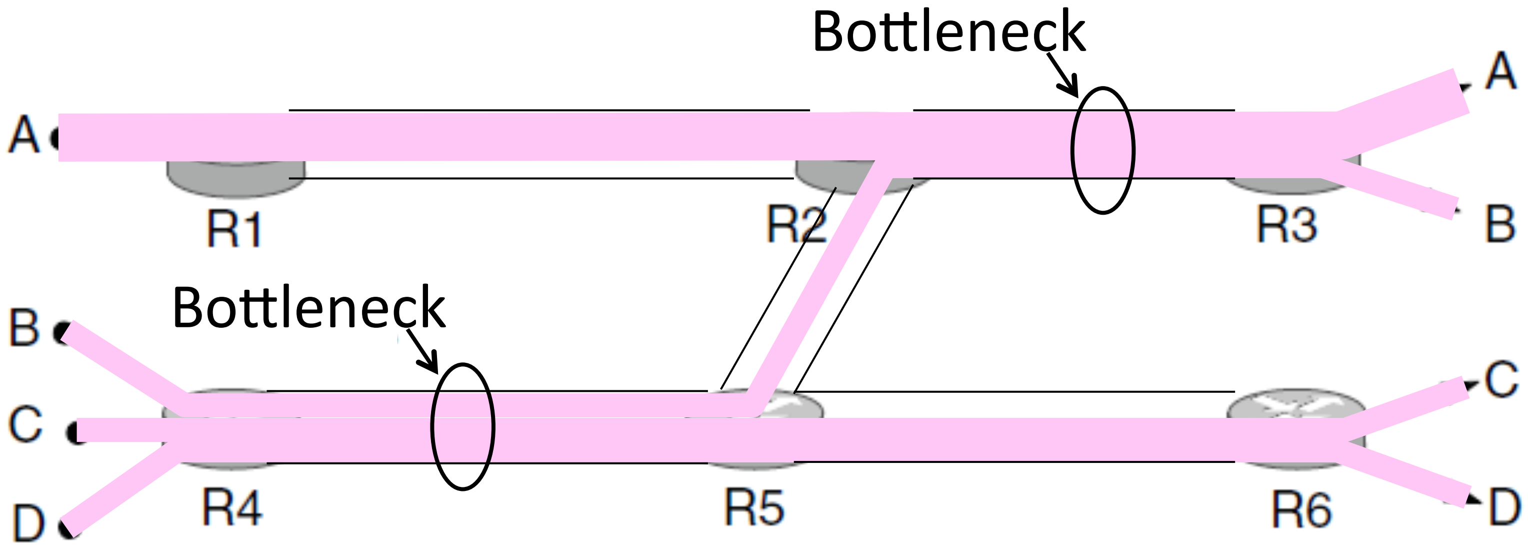
Max-Min Example (2)

- When rate=1/3, flows B, C, and D bottleneck R4—R5
  - Fix B, C, and D, continue to increase A

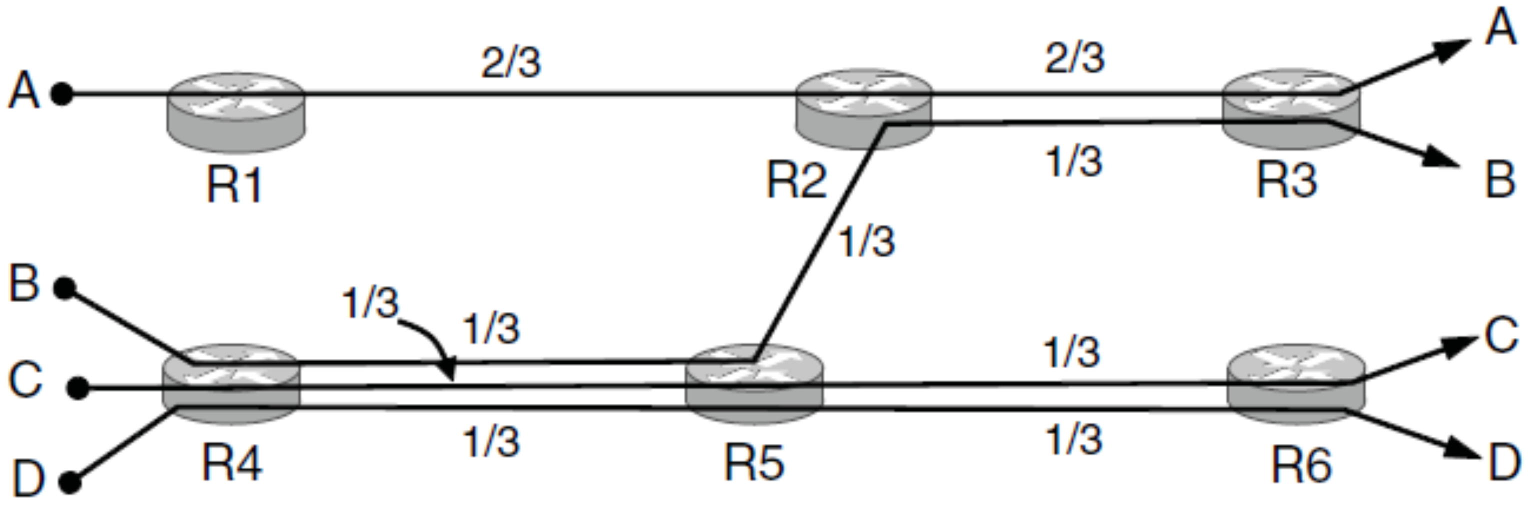


Max-Min Example (3)

- When rate=2/3, flow A bottlenecks R2—R3. Done.

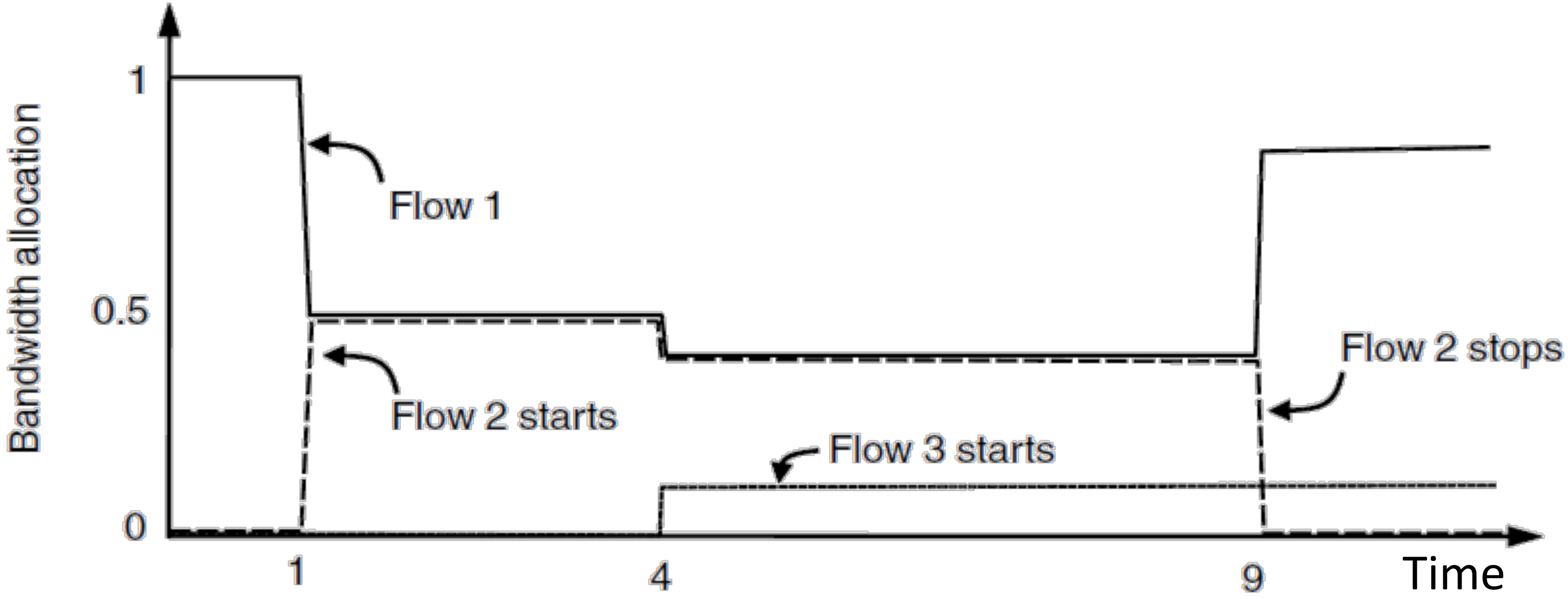


Max-Min Example (4)

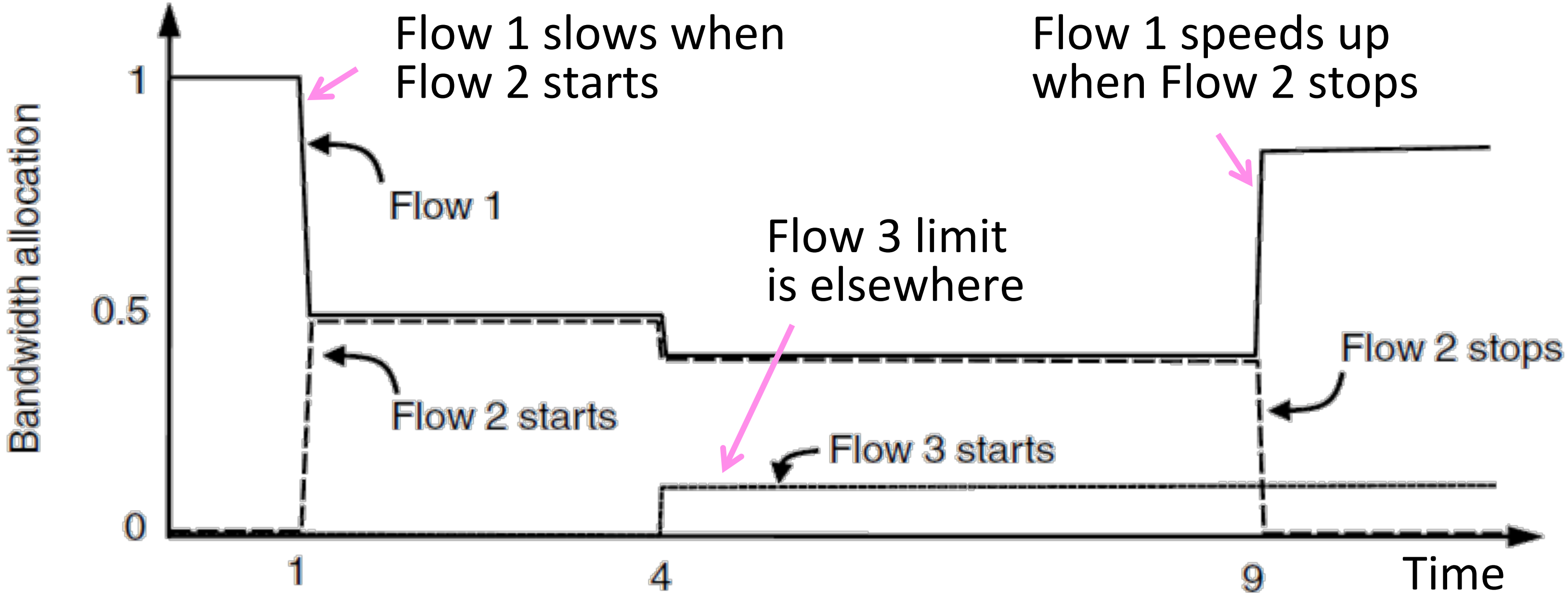
- End with A=2/3, B, C, D=1/3, and R2—R3, R4—R5 full
  - Other links have extra capacity that can't be used
- 

Adapting over Time

- Allocation changes as flows start and stop

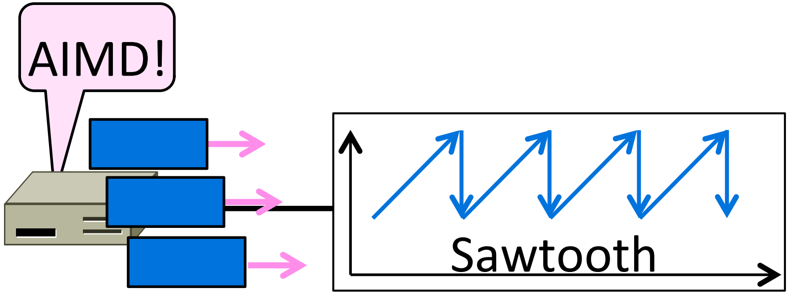


Adapting over Time (2)



Topic

- Bandwidth allocation models
  - Additive Increase Multiplicative Decrease (AIMD) control law



Recall

- Want to allocate capacity to senders
  - Network layer provides feedback
  - Transport layer adjusts offered load
  - A good allocation is efficient and fair
- How should we perform the allocation?
  - Several different possibilities …

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)

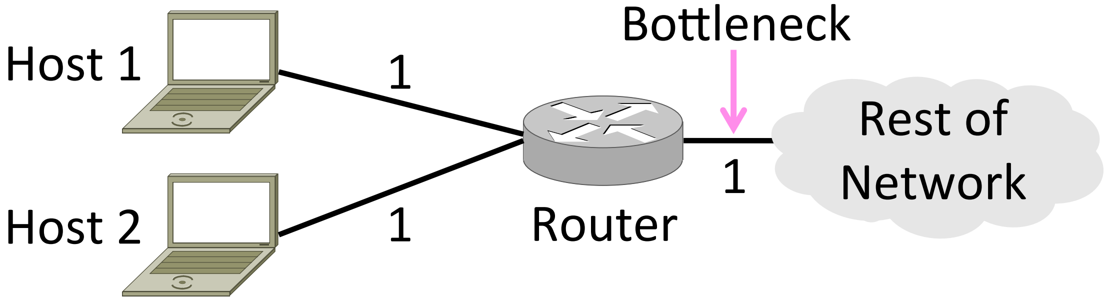
- We'll look at closed-loop, host-driven, and window-based too
- Network layer returns feedback on current allocation to senders
  - At least tells if there is congestion
- Transport layer adjusts sender's behavior via window in response
  - How senders adapt is a control law

Additive Increase Multiplicative Decrease

- AIMD is a control law hosts can use to reach a good allocation
  - Hosts additively increase rate while network is not congested
  - Hosts multiplicatively decrease rate when congestion occurs
  - Used by TCP ☺
- Let’s explore the AIMD game …

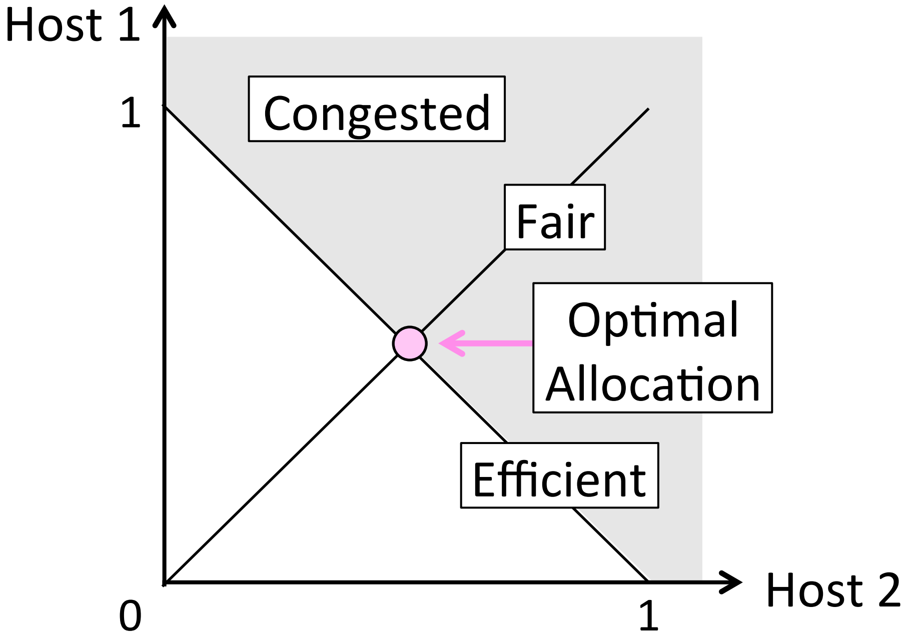
AIMD Game

- 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



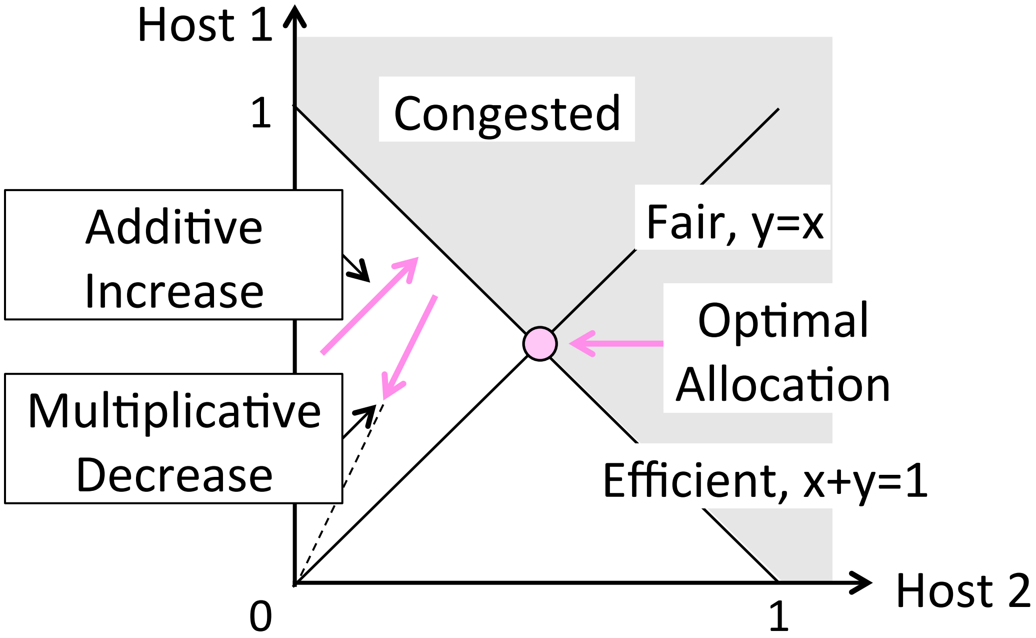
AIMD Game (2)

- Each point is a possible allocation



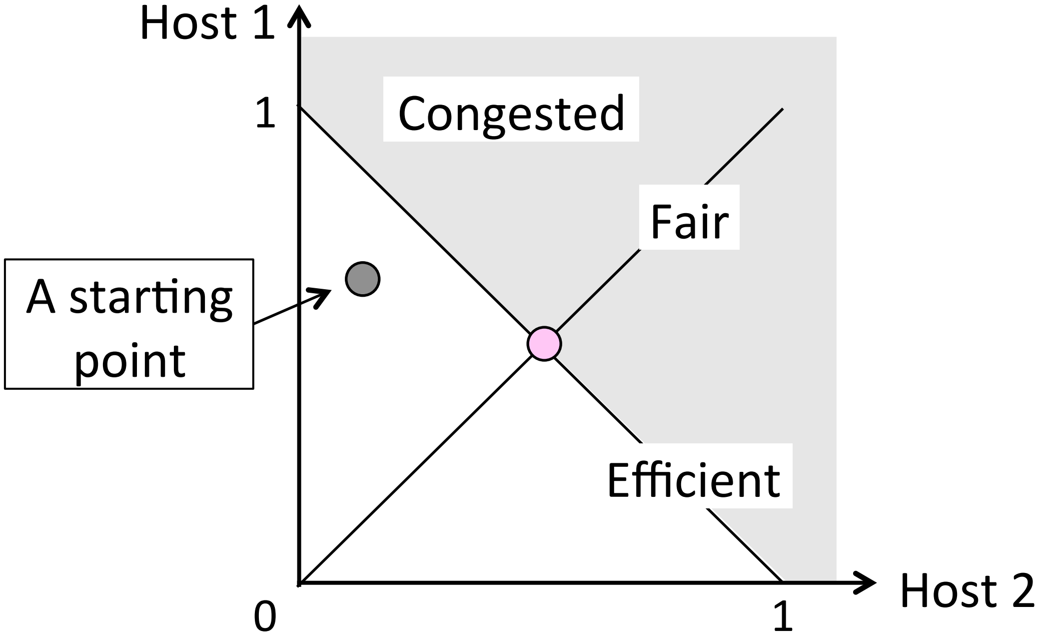
AIMD Game (3)

- AI and MD move the allocation



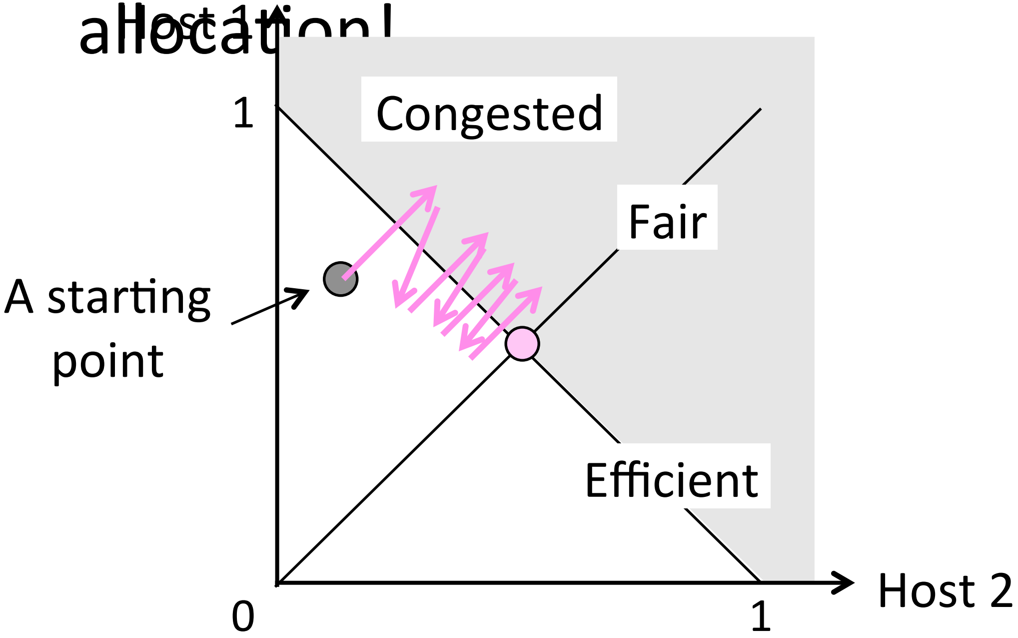
AIMD Game (4)

- Play the game!



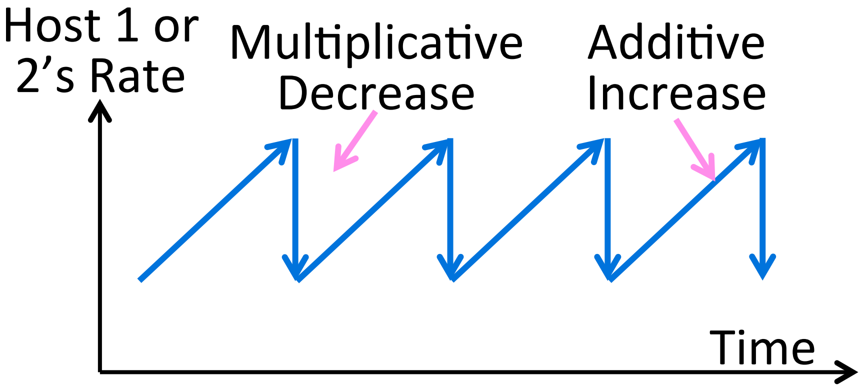
AIMD Game (5)

- 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	Compound TCP (Windows)	Hear about congestion early Need to infer congestion
Router indication	TCPs with Explicit Congestion Notification	Hear about congestion early Require router support