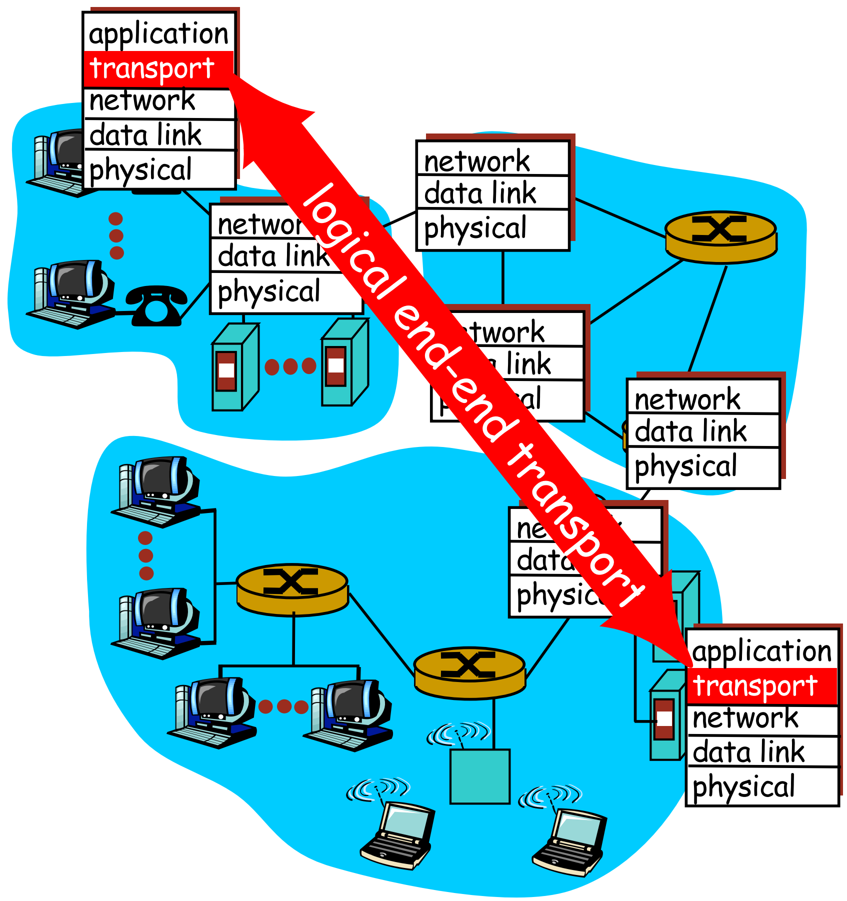
Transport Layer

Objectives

- Understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- TCP and UDP protocols
  - Message format
  - Operations

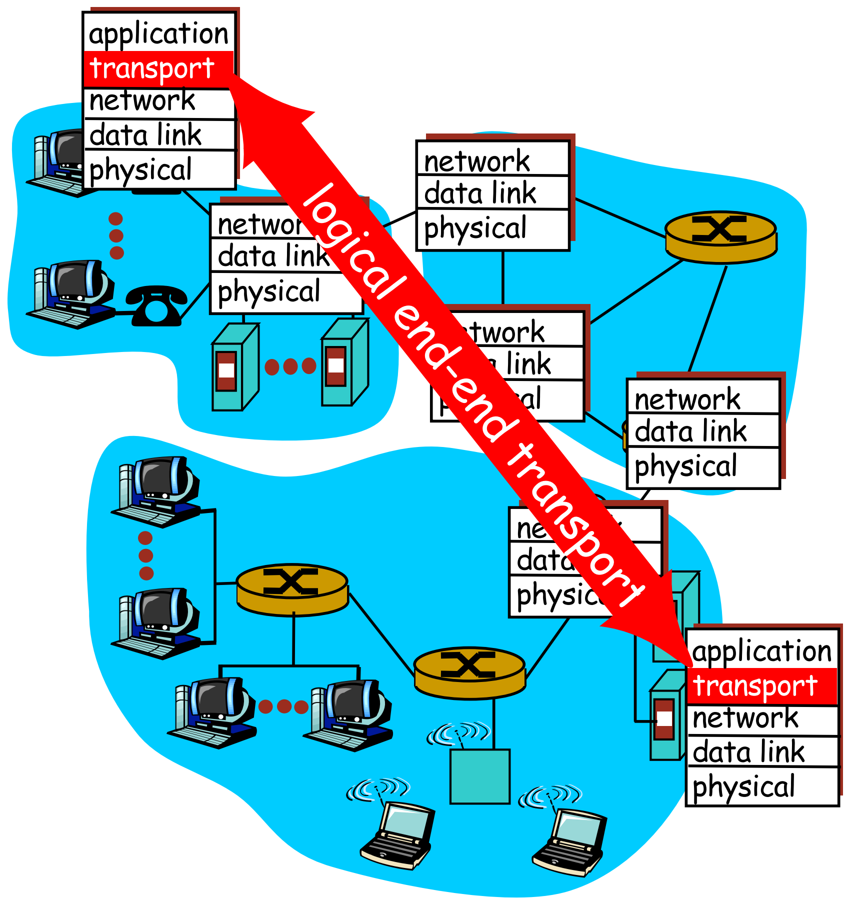
Transport services and protocols

- provide *logical communication* between app *processes* running on different hosts
- transport protocols run in end systems
  - sender side: breaks app messages into segments, passes to network layer
  - receiver side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



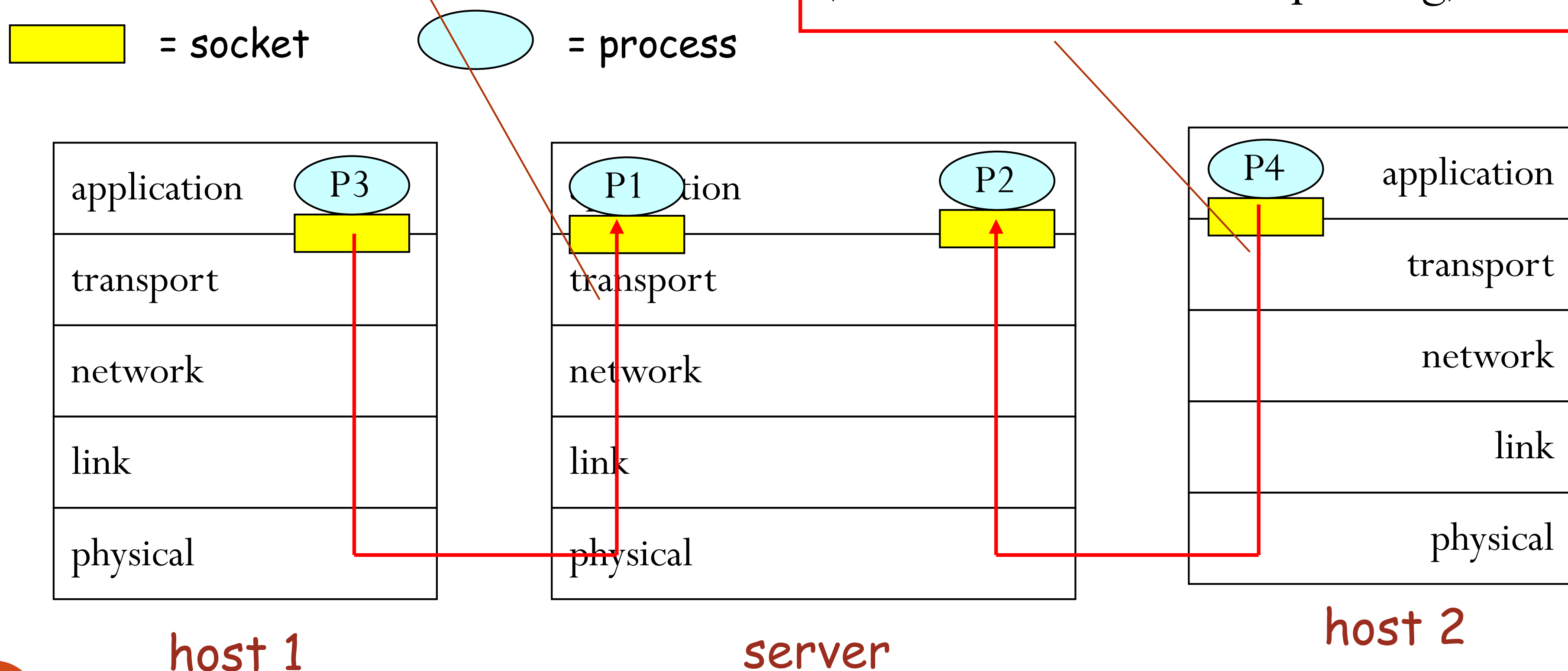
Multiplexing/demultiplexing

Demultiplexing at receiver host:

delivering received segments to correct socket

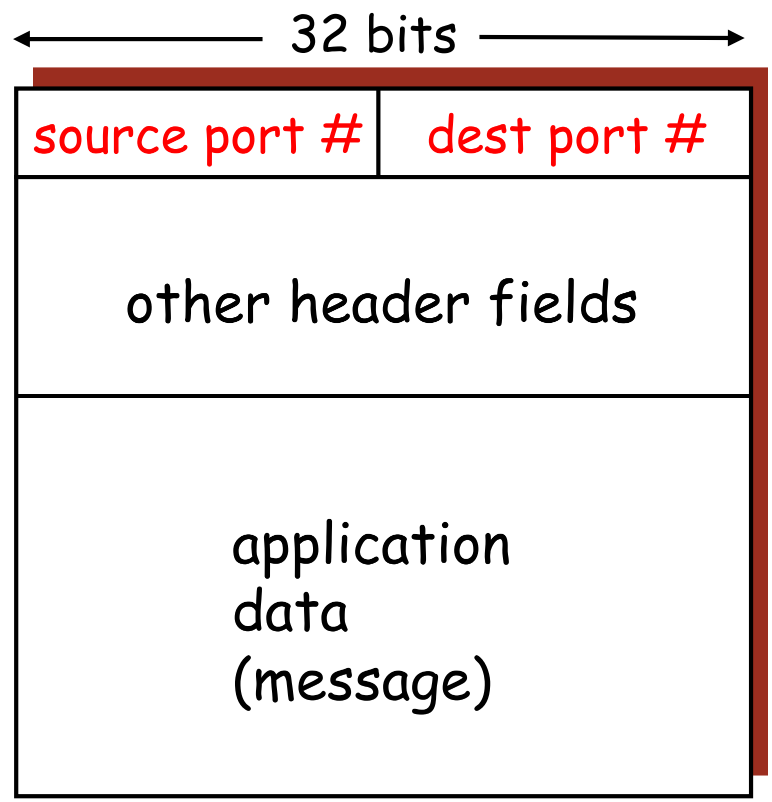
Multiplexing at sender host:

gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)



How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number
- host uses IP addresses & port numbers to direct segment to appropriate socket
  - TCP sockets are identified by (S-IP, D-IP, SP, DP) 4-tuple
  - UDP sockets are identified by (D-IP, DP)



TCP/UDP segment format

Principles of Reliable Data Transfer Protocols

- Important in applications, transport layer and link layers
- Mechanisms:
  - Error detection and correction: a packet is received but may be erroneous
  - Loss detection and recovery: a packet is missing
- Design issues:
  - Where to put the functionality?
  - Efficiency: utilization of bandwidth resource

Utilization = maximum app. data rate/available bandwidth

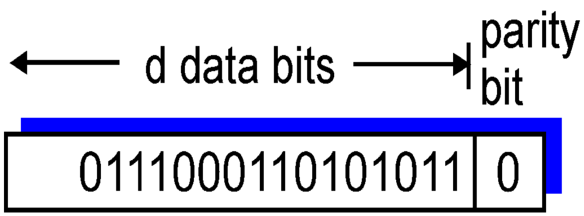
Error Detection

- Problem: detect bit errors in packets (frames)
- Solution: add extra bits to each packet
- Techniques:
  - Parity check
  - Checksum
  - Other sophisticated coding schemes such as Reed-Solomon code

Parity Checking

Single Bit Parity:

Detect single bit errors



Odd parity check:

- Sum up information bits and mod 2
- If zero, add 1 as the parity bit
- Otherwise, add 0

Even parity check

- Sum up information bits and mod 2
- If zero, add 0 as the parity bit
- Otherwise, add 1

- e.g., d = 7, even parity

Received message: 00101010

Redundancy

$$R = \frac{\text{Number of bits used in full}}{\text{Number of bits in message}}$$

Q: Can it detect multiple bit errors?

Can it correct any bit error?

Internet Checksum

- 16-bit one's complement of the one's complement sum of all 16-bit words in the content to be protected
- Two's complement sum: summing the numbers (with carries)
- One's complement sum: summing the numbers and adding the carry (or carries)

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Internet checksum

Sender:

- treat contents to be protected as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into the checksum field

Receiver:

- compute checksum of received data
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. But maybe errors nonetheless? More later ….

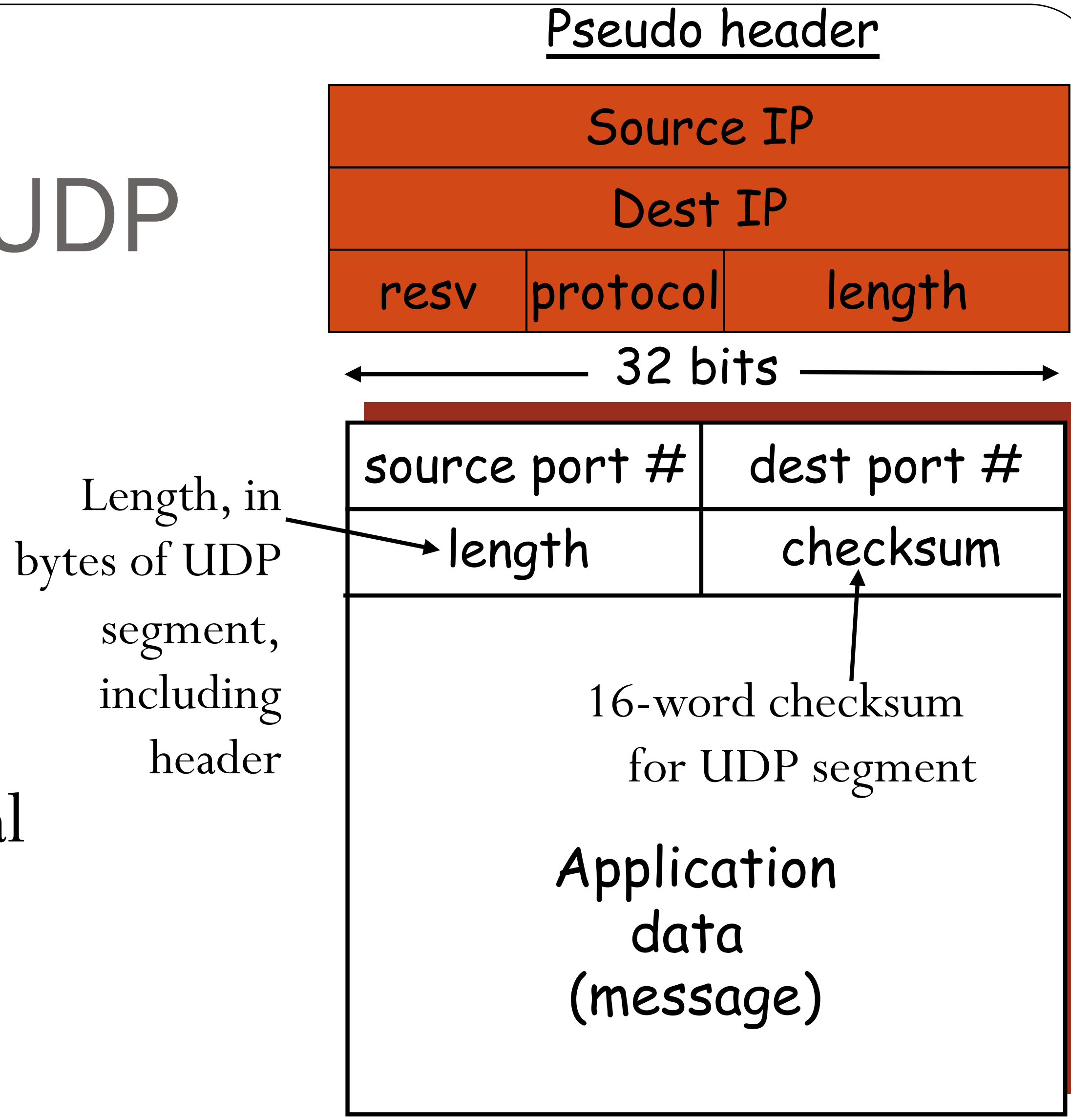
Q: Can it detect multiple bit errors?
Can it correct any bit error?

Example

- Original: 01 00 F2 03 F4 F5 F6 F7 00 00
- 2's complement sum: 0100 + F203 + F4F5 + F6F7 + 0000 = 2DEEF
- 1's complement sum:
- Checksum = ?
- Result:
- Recv: 01 00 F2 03 F4 F5 F6 F7 01 00 210E
- Redundancy? Limitations?

Checksum in UDP

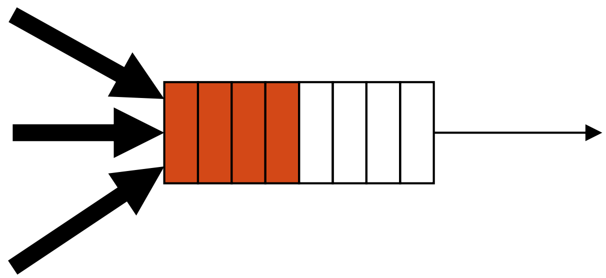
- the checksum is computed using the payload and a "pseudo header" that contains some of the same information from the real IP header.



UDP segment format

Loss Detection

- Causes of packet losses
  - Buffer overflow
  - Drop after error detection
- Detection methods
  - At the crime scene
  - At the receiver
    - How do I know that I am supposed to get certain data?
  - At the sender

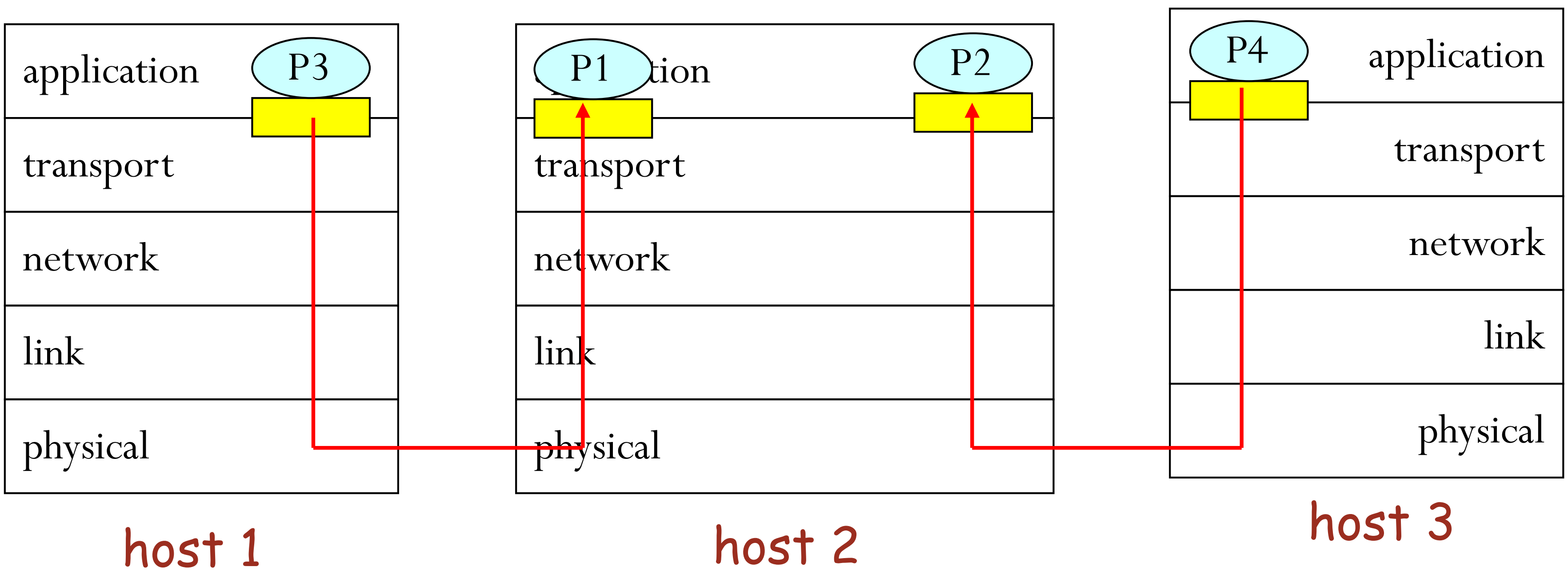


Loss Recovery

- Once packet losses are detected, the source needs to be informed
  - Negative ACK (NACK): "packet xx is missing" vs.
  - Positive ACK: "packet xx is received" -- timeout
- Source action
  - Should I proceed to transmit the next message w/o the knowledge of the reliable delivery of the current one?
- Retransmission
  - Retransmit every unacked packet?
  - Selective retransmit the lost packet only?

Flow Control

- In Internet terms, flow control aims to control the rate of the sender not to overwhelm the receiver
- How?



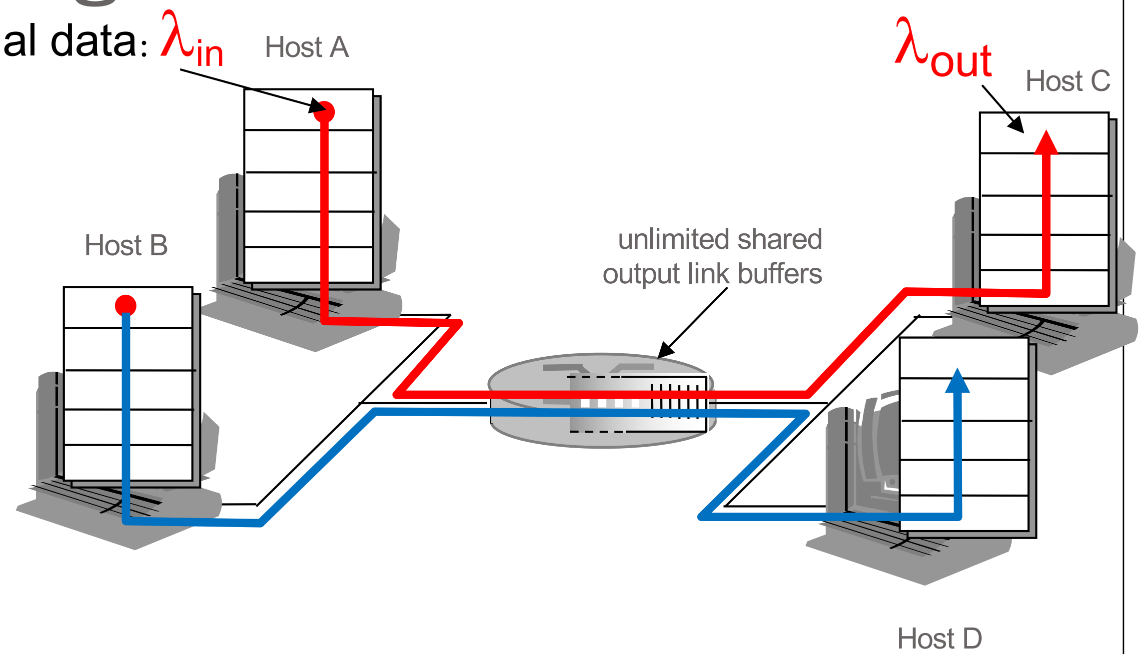
Congestion Control

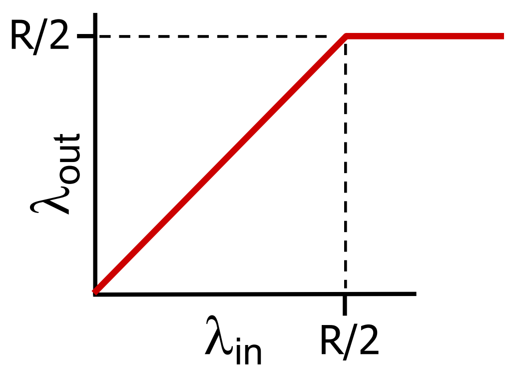
Congestion:

- informally: "too many sources sending too much data too fast for network to handle"
- different from flow control!
  - Flow control concerns not to overload the receiver
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)

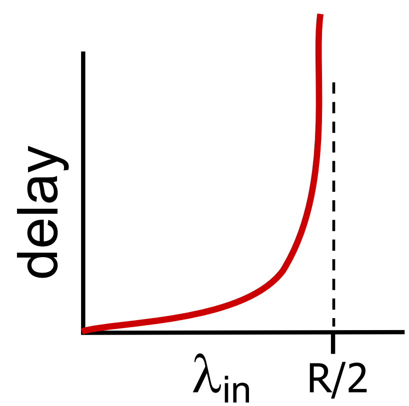
Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- output link capacity: R
- no retransmission





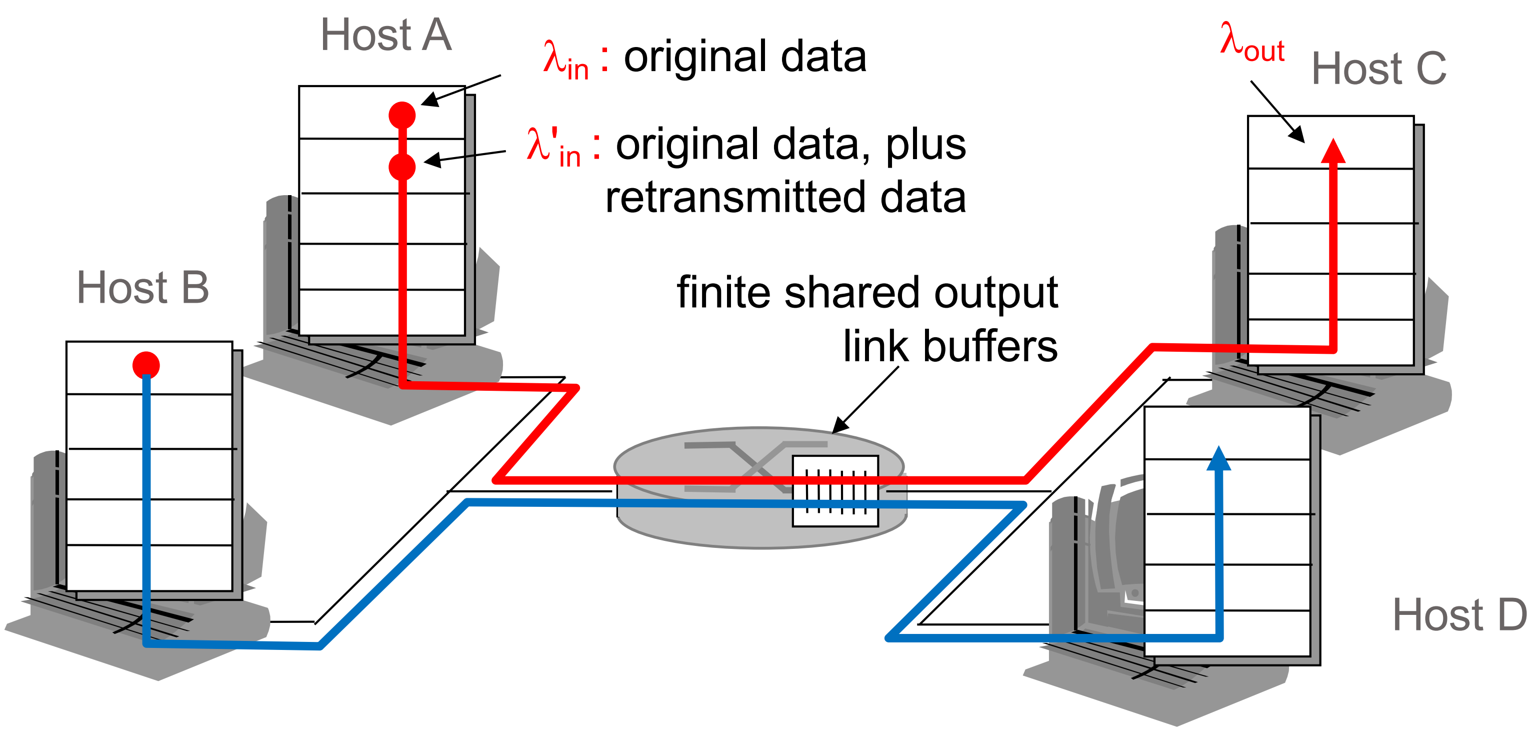
maximum per-connection throughput: R/2



large delays as arrival rate, λ_{in}, approaches capacity

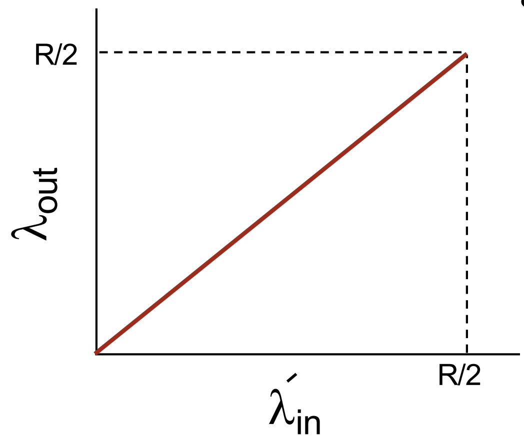
Causes/costs of congestion: scenario 2

- one router, finite buffers
- If a packet is lost at router due to a full buffer, the sender retransmits lost packet
  - application-layer: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes retransmissions: $\lambda'_{in} \geq \lambda_{in}$

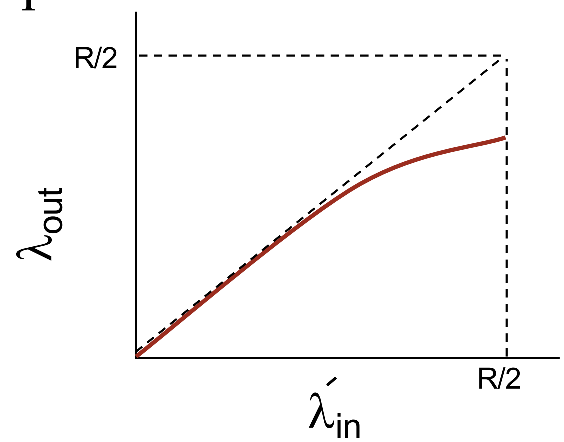


Causes/costs of congestion: scenario 2

- Realistic: when λ_{in} approaching R/2, some packets are retransmissions including duplicated that are delivered!



a. perfect rate control



b. loss & retransmission

"costs" of congestion:

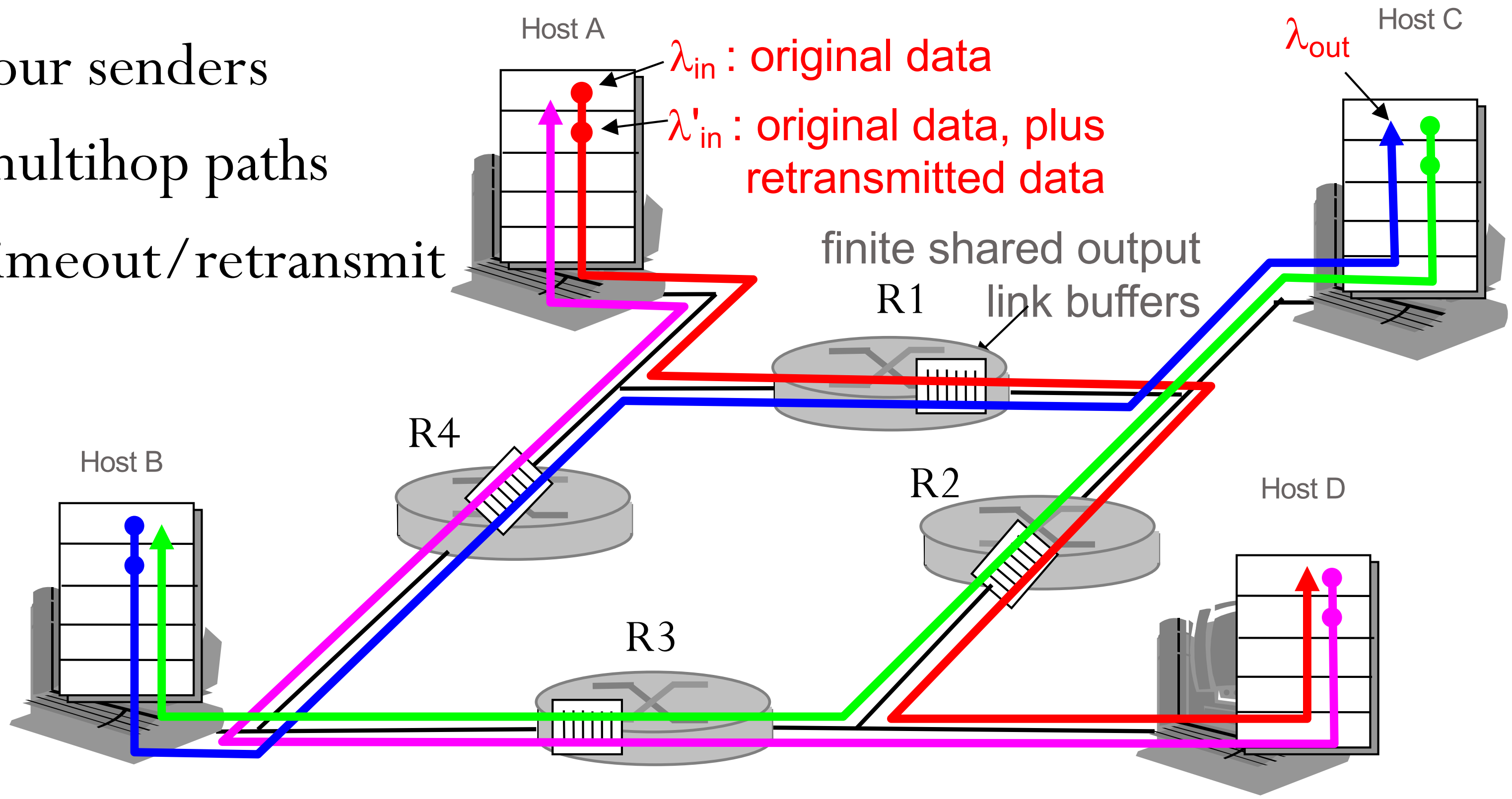
- more work (retransmits) for given "goodput"
- unneeded retransmissions: link carries multiple copies of packet

Causes/costs of congestion: scenario 3

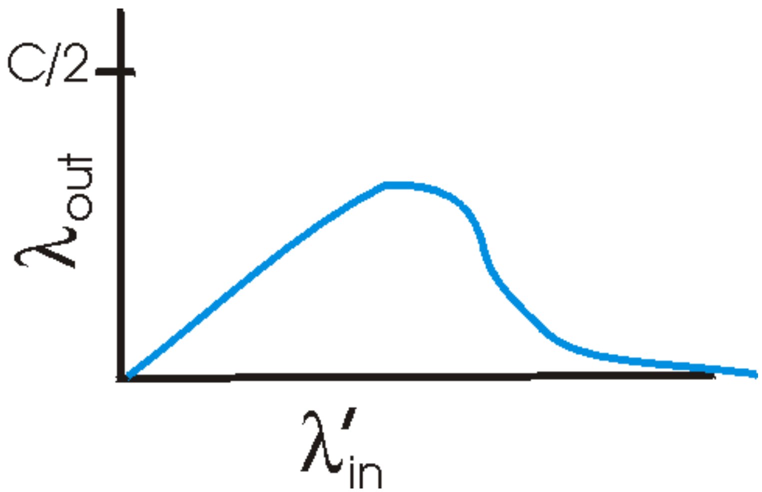
Q: what happens as λ'_{in} and λ_{in} increase?

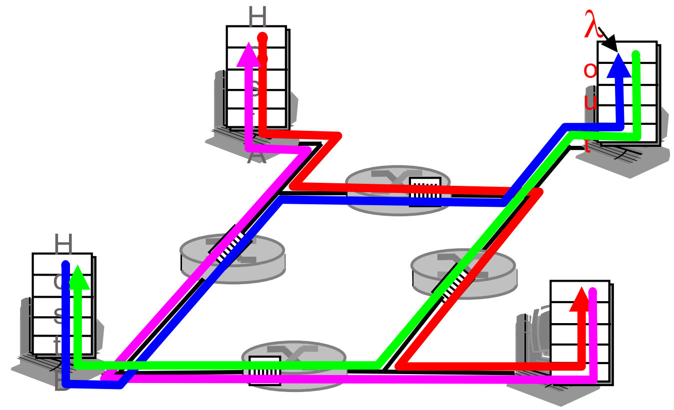
A: As red λ'_{in} increses, nearly all blue packets at higher finite queue are dropped. Blue throughput approaches 0.

- four senders
- multihop paths
- timeout/retransmit



Causes/costs of congestion: scenario 3





Another "cost" of congestion:

- when packet dropped, any "upstream transmission capacity used for that packet was wasted!

Approaches towards congestion control

Two broad approaches towards congestion control:

End-to-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss and delay
- approach taken by TCP

Network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at

Bag of Tricks

Functions	TCP	UDP
Multiplexing/demultiplexing	x	x
Reliable data transfer	• Checksum • Sequence number • ACK from receivers • Retransmission • Buffering outstanding packets	• Checksum
Flow control	Throttled by the receiver, sender reacts	
Congestion control	• End-system estimates and adjusts transmission rate based on congestion "signal" from the network	
Connection establishment/tear down	x	

UDP and TCP

- Understanding the protocol details of UDP and TCP
  - Header format
  - TCP state machine
    - Connection setup and tear-down
  - Sliding window in TCP
  - TCP flow control
  - TCP congestion control

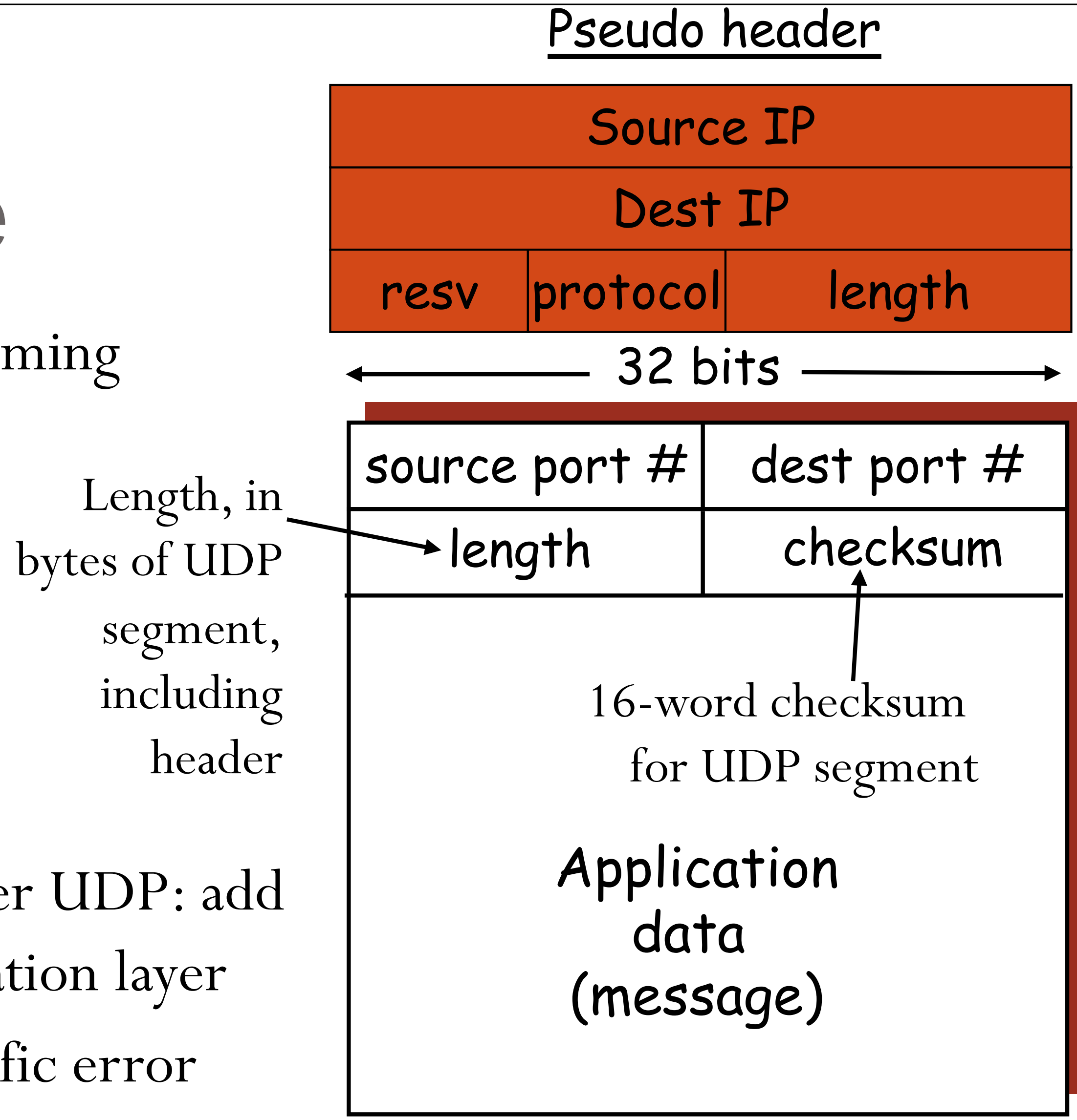
UDP: User Datagram Protocol [RFC 768]

- "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- connectionless:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

- Why is there a UDP?
- no connection establishment (less delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as "desired"

UDP: more

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS, SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!



UDP segment format

Size of UDP header?

TCP Outline

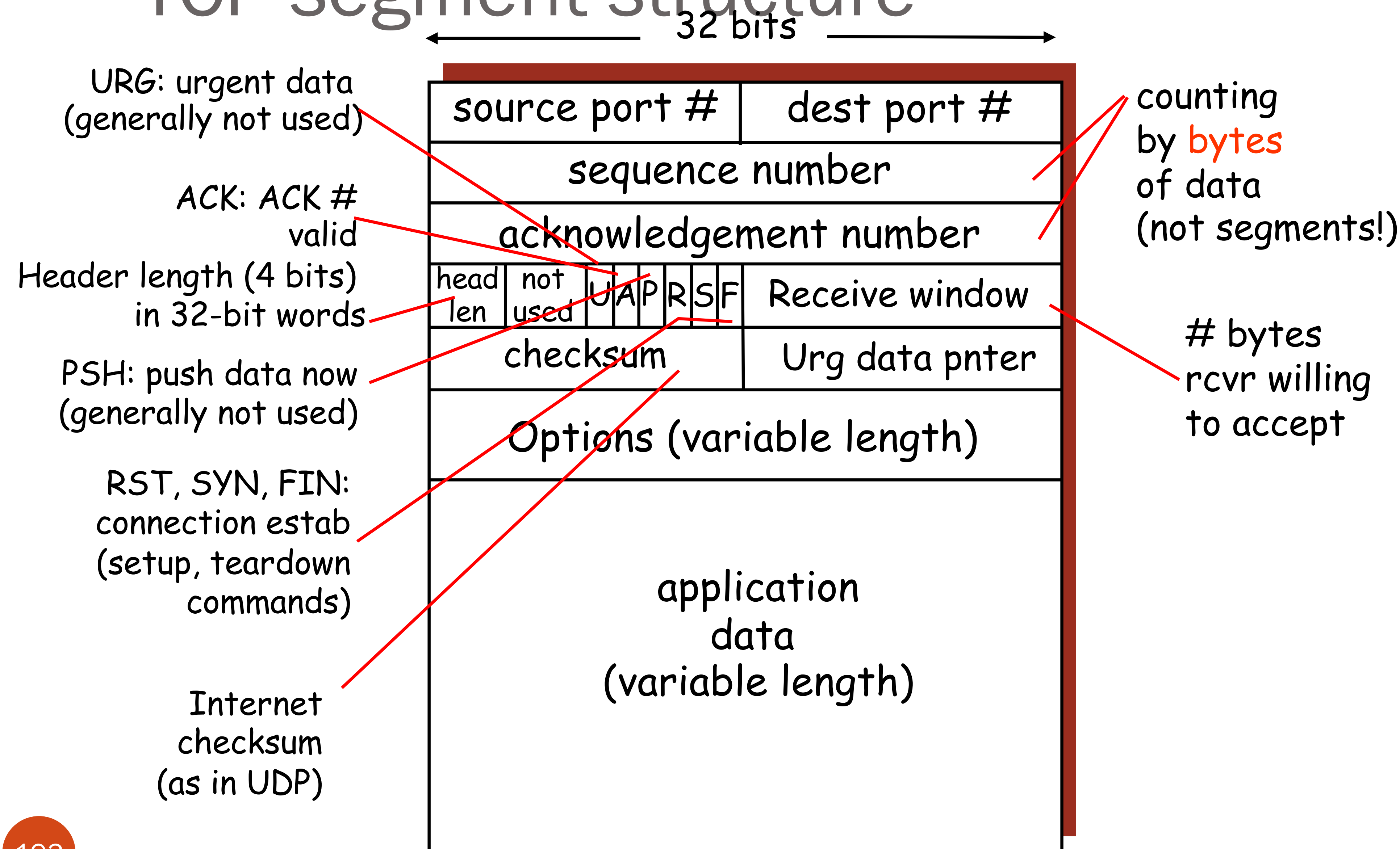
- TCP: Overview
- TCP header format
- TCP connection establishment & tear down
  - What are the error scenarios?
- Reliable data transfer
- Congestion control

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order byte steam:
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- send & receive buffers
- full duplex data:
  - bi-directional data flow in same connection
  - MSS (maximum segment size): largest data payload in TCP
- connection-oriented:
  - handshaking (exchange of control msgs) initiates sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure

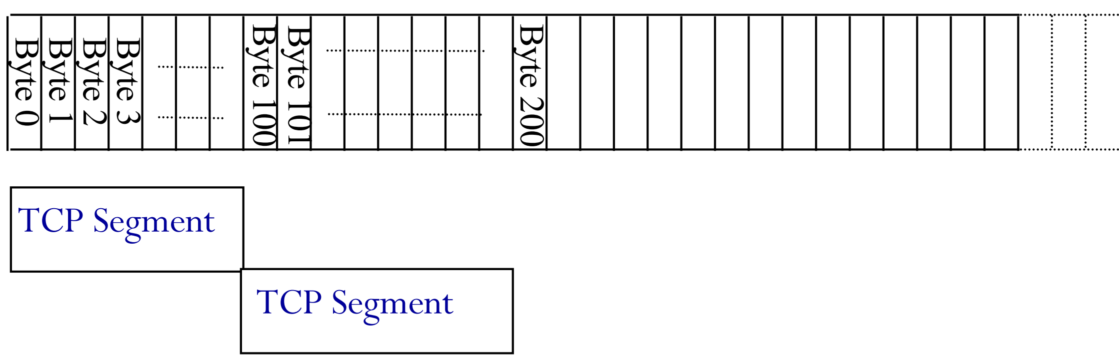


TCP: Segments

- TCP "Stream of Bytes" Service
- TCP segment
  - No more than Maximum Segment Size (MSS) bytes
  - Segment sent when Segment full (MSS) or "Pushed" by application

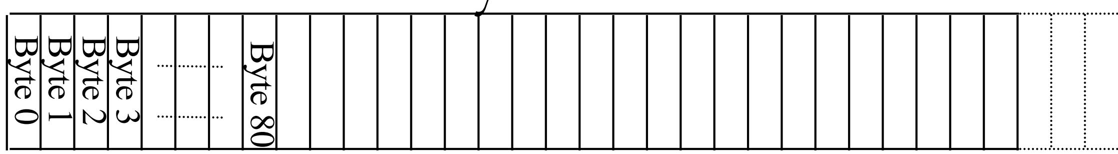
Example:

  - MSS = 100 bytes, Data receive from application



TCP: Sequence Numbers, ACKs

- TCP "Stream of Bytes" Service



- Sequence numbers:
  - Starting byte offset of data carried in this segment
- ACKs: ("What Byte is Next")
  - gives seq# just beyond highest seq# received in order

TCP: Sequence Numbers, ACKs Example

- Sequence Number = 1001. Sender sends 500 bytes. Receiver acknowledges with ACK number:

  A) 501 B) 1002 C) 1500 D) 1501 E) 1502

  Answer: D) 1501

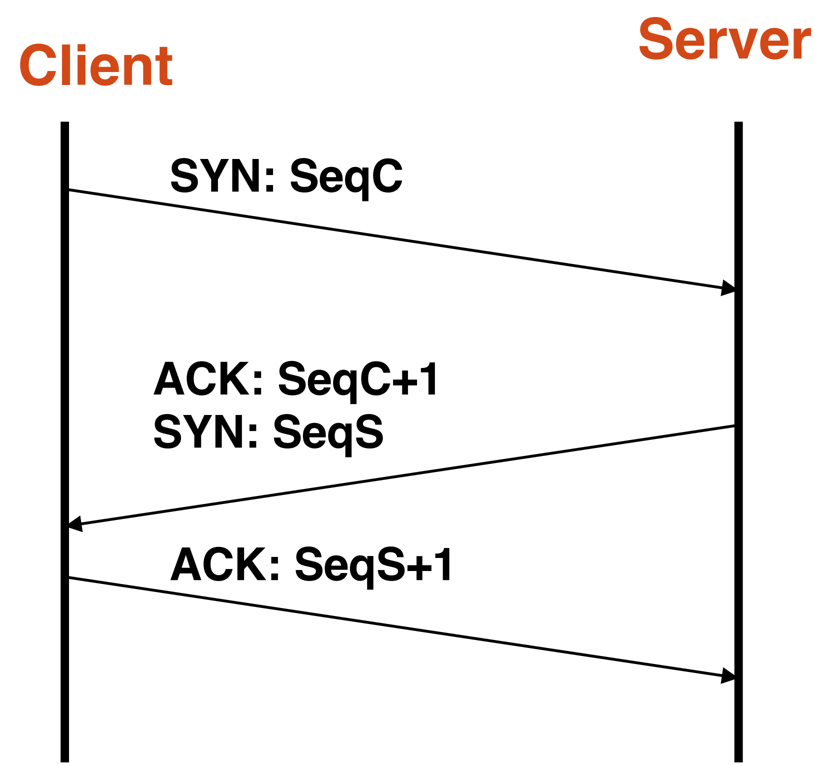
- Next sequence number to send by sender is:

  A) 1500 B) 1501 C) 1502

  Answer: B) 1501

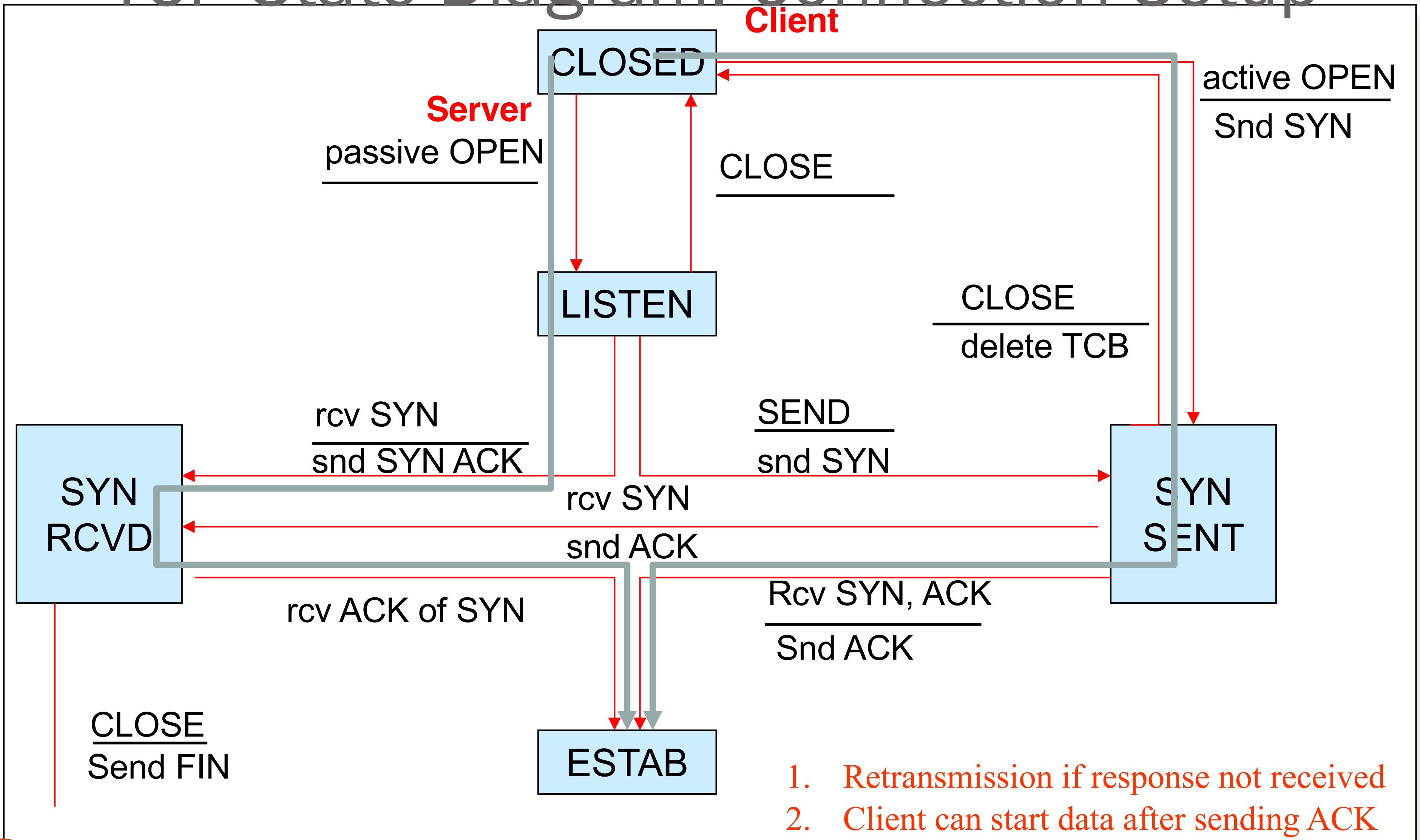
Establishing Connection

- Three-Way Handshake
  - Each side notifies the other of starting sequence number it will use for sending
  - Each side acknowledges other's sequence number
    - SYN-ACK: Acknowledge sequence number + 1
  - The third segment may piggyback some data



- Why 3-way handshake?

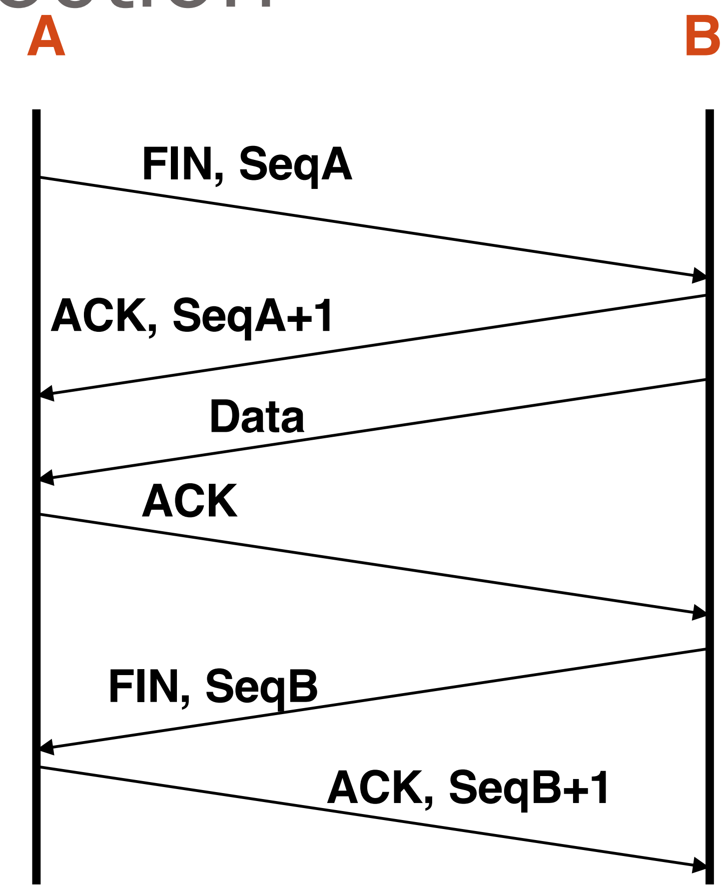
TCP State Diagram: Connection Setup



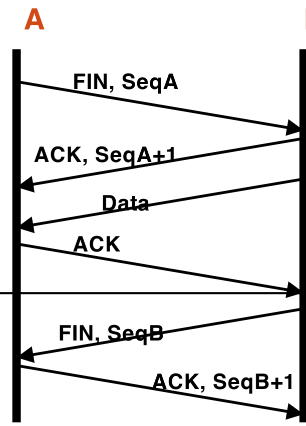
1. Retransmission if response not received
2. Client can start data after sending ACK

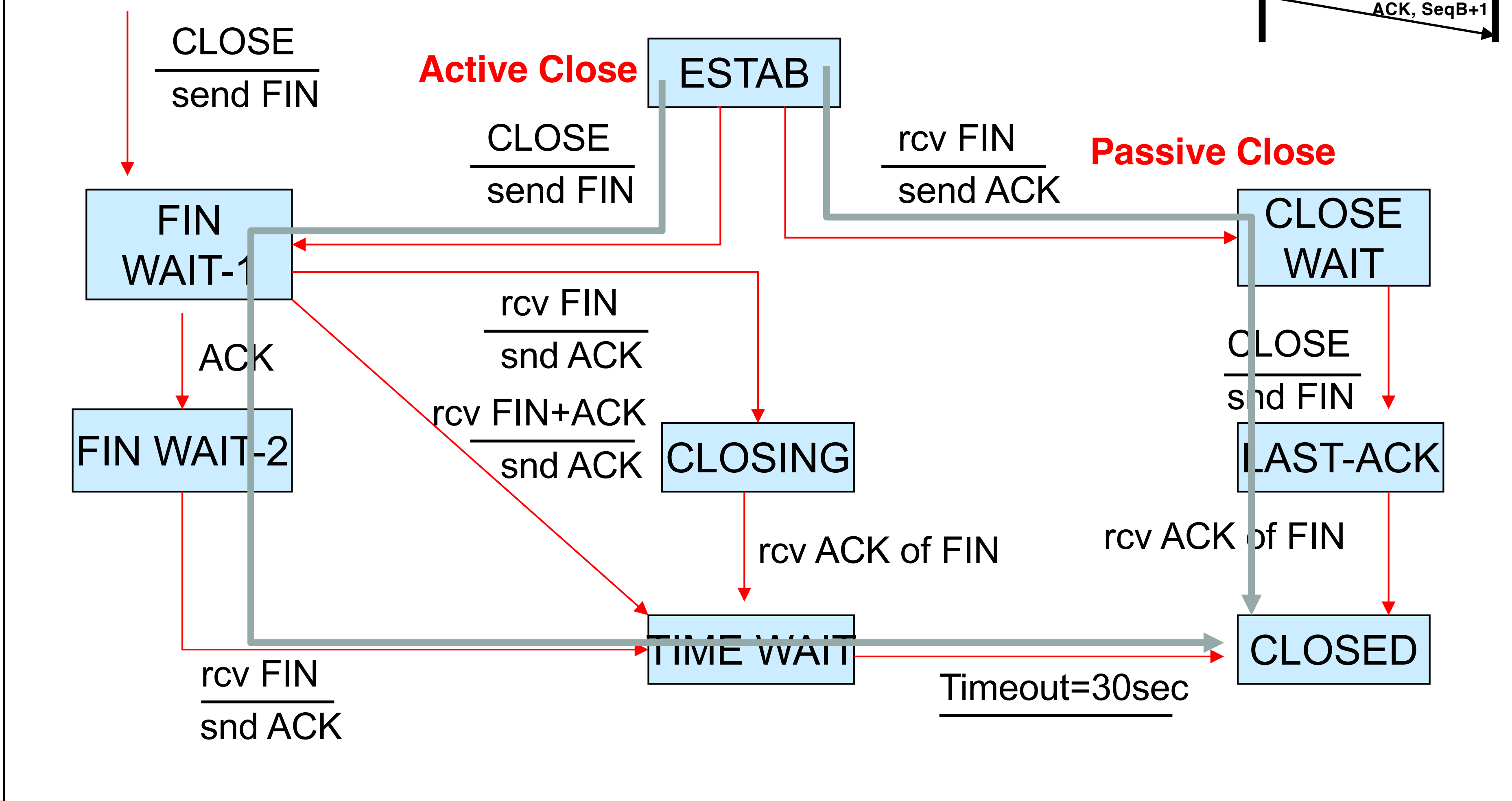
Tearing Down Connection

- Either Side Can Initiate Tear Down
  - Send FIN signal
  - I'm not going to send any more data
- Other Side Can Continue Sending Data
  - Half open connection
  - Must continue to acknowledge
- Acknowledging FIN
  - Acknowledge last sequence number + 1

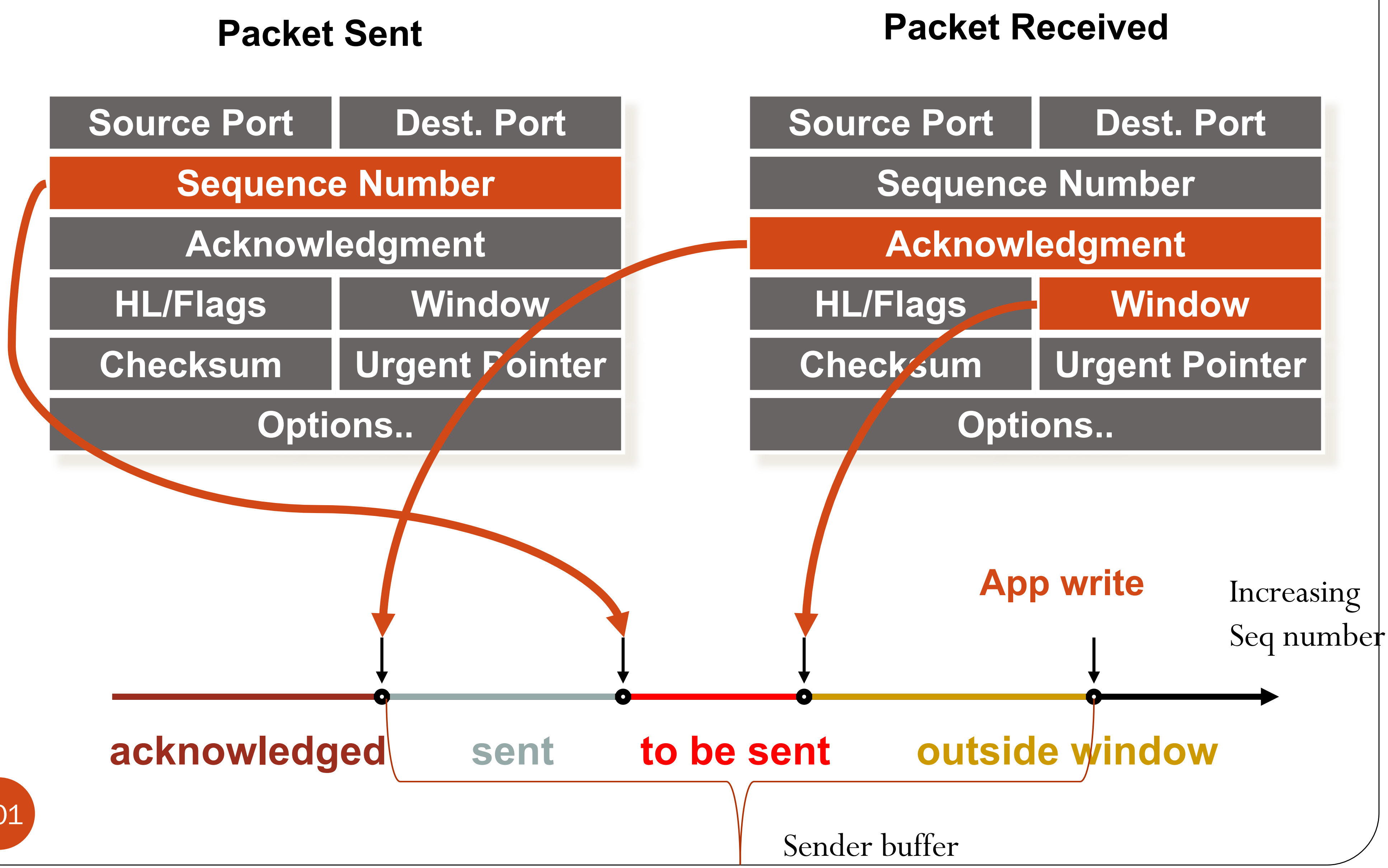


State Diagram: Connection Tear-down



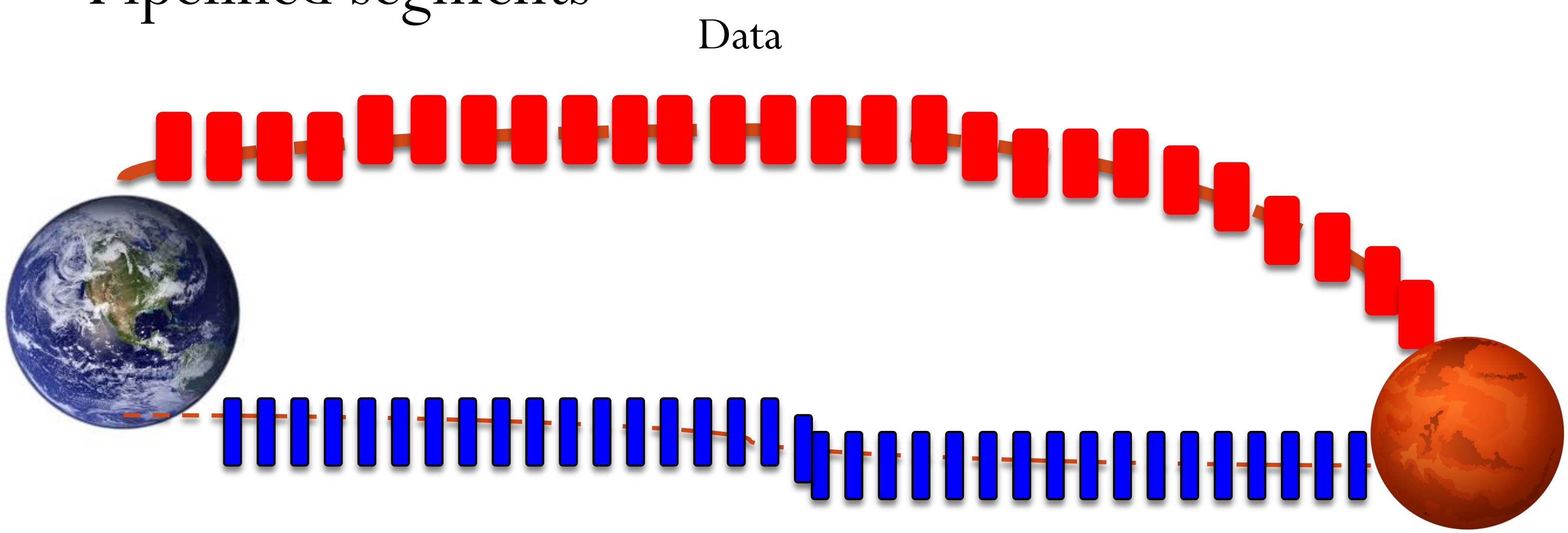


TCP Window Control



TCP reliable data transfer

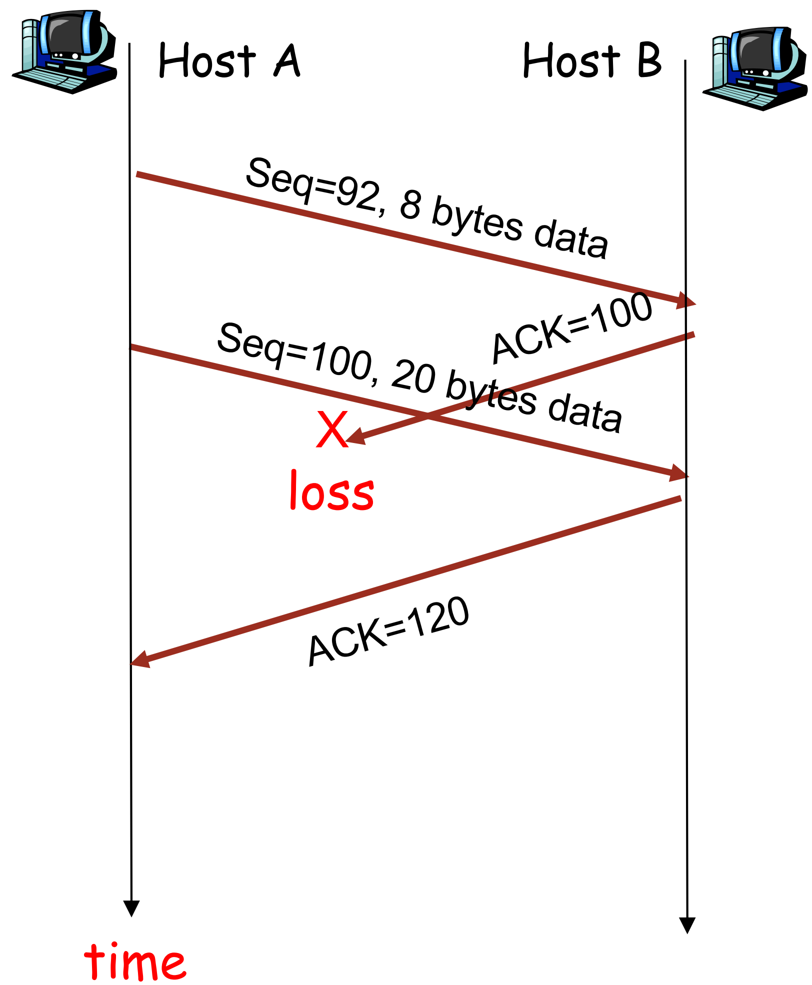
- TCP creates reliable data transport service on top of IP's unreliable service
  - Pipelined segments



In one round trip time (RTT) time, a maximum BWxRTT bits can be sent to fill up the pipe

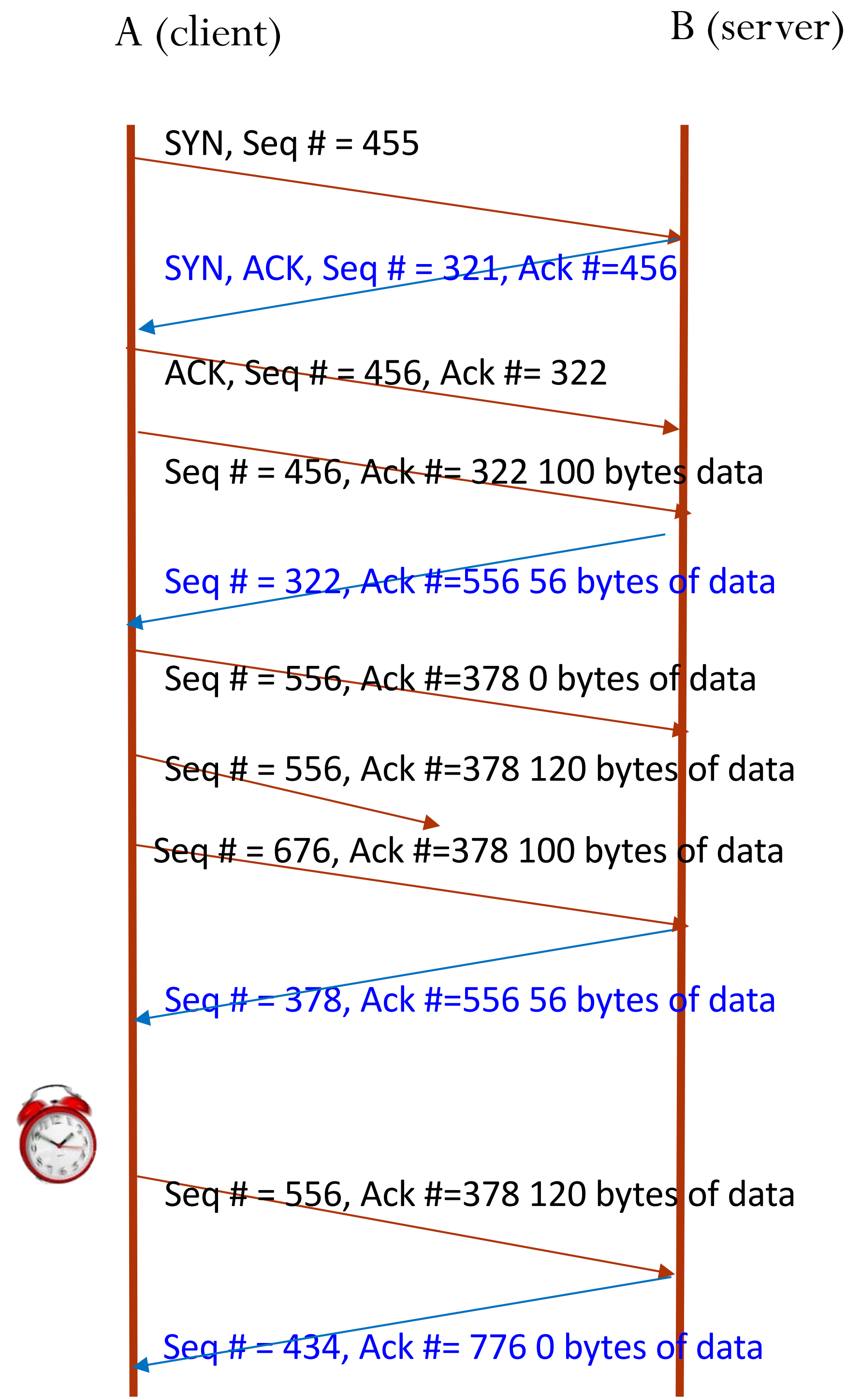
TCP reliable data transfer

- Cumulative acks



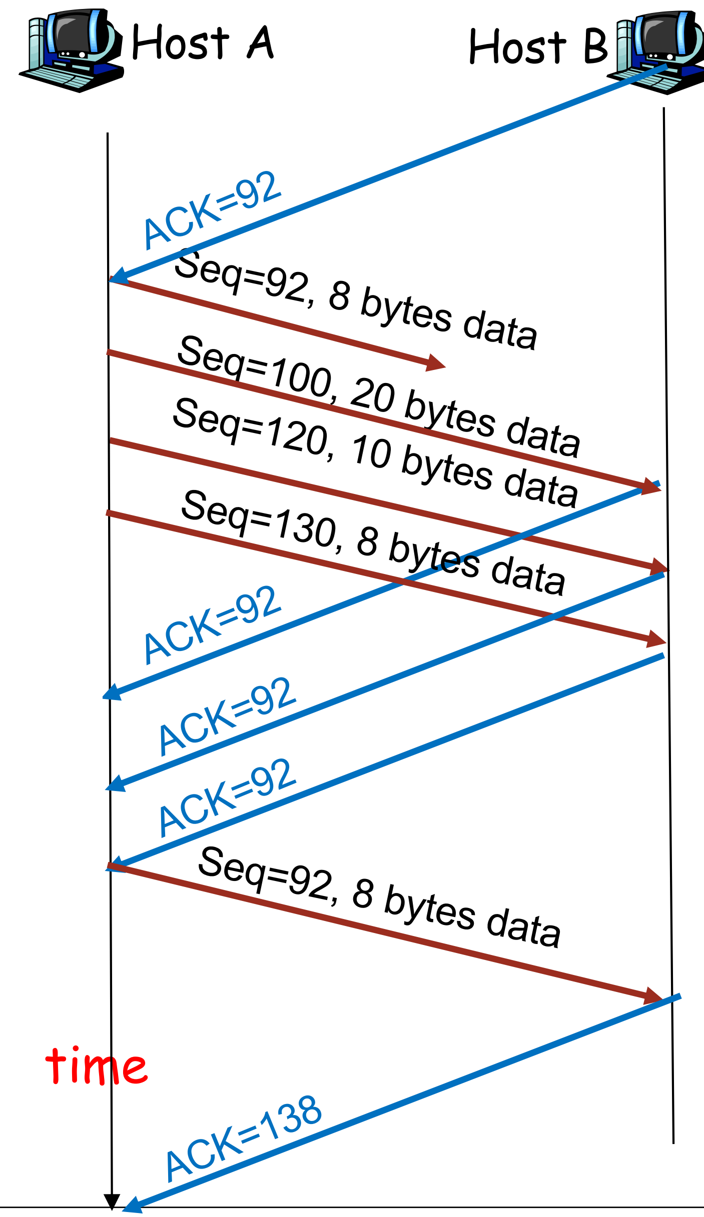
TCP reliable data transfer

- Retransmission triggered by timeout



TCP reliable data transfer

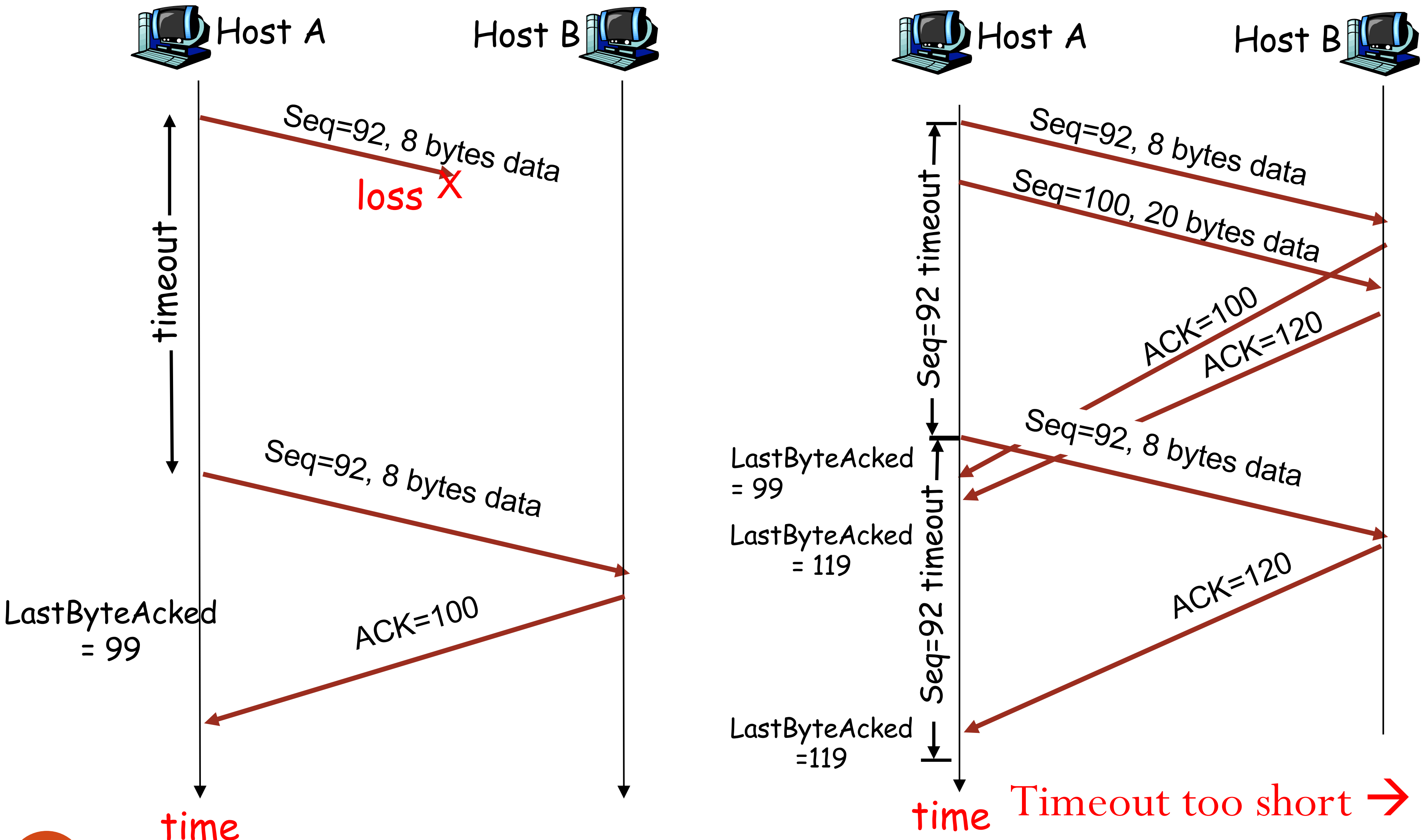
- Retransmissions triggered by:
  - duplicate acks



TCP: retransmission timeout

- If the sender hasn't received an ACK by timeout, retransmit the first packet in the window, restart timer
- How do we pick a timeout value?

TCP: retransmission timeout



Timeout too long → inefficient

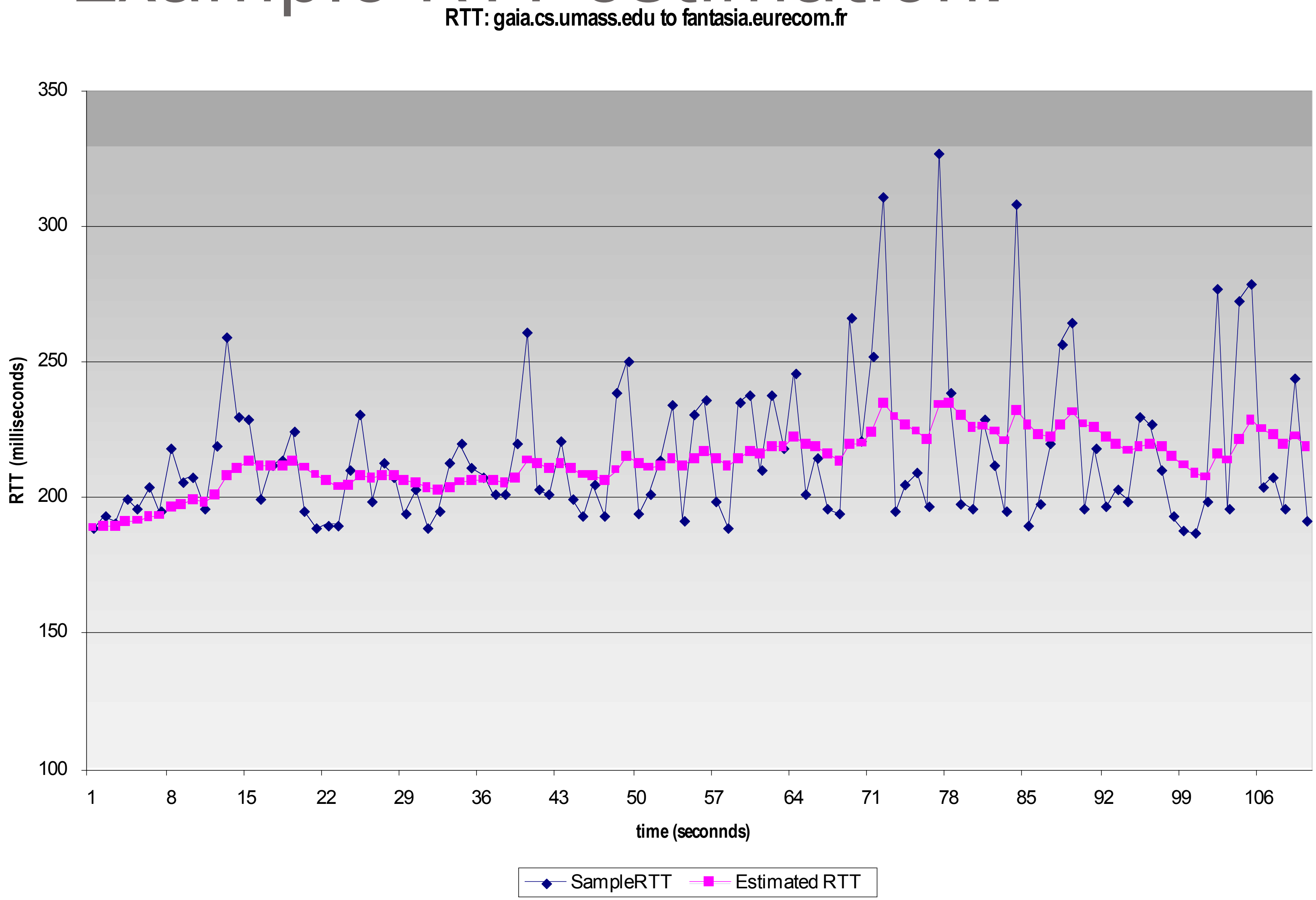
Timeout too short → duplicate packets

TCP Round Trip Time

Q: how to estimate RTT?

- SampleRTT: measured time from segment transmission until ACK receipt
  - ignore retransmissions (?)
- SampleRTT will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current SampleRTT

Example RTT estimation:



TCP Round Trip Time and Timeout

EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT

- Exponential weighted moving average
- influence of past sample decreases exponentially fast (?)
- typical value: α = 0.125

Effect of α?

TCP Round Trip Time and Timeout

Setting the timeout

- EstimatedRTT plus "safety margin"
  - large variation in EstimatedRTT -> larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

DevRTT = (1-β)*DevRTT +
β*|SampleRTT-EstimatedRTT|

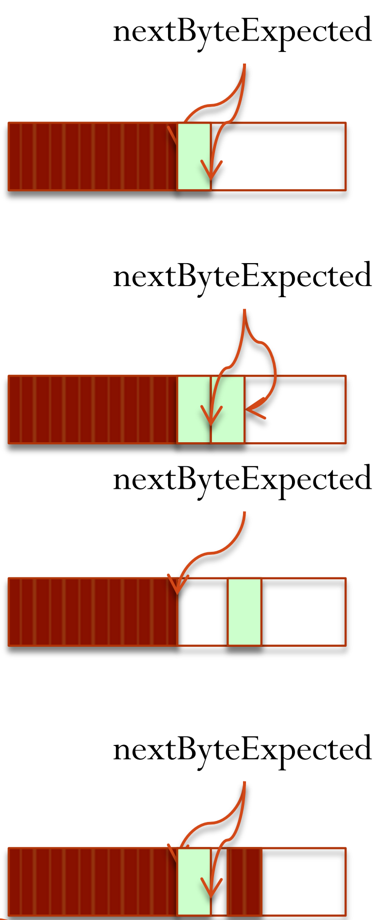
(typically, β = 0.25)

- Then set timeout interval:

TimeoutInterval = EstimatedRTT + 4*DevRTT

TCP Receiver Event/ACK generation [RFC 1122, RFC 2581]

TCP receiver buffer



Accept packets within receiver window

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send duplicate ACK, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment starts lower end of gap

TCP Congestion Control

Congestion detection:

- loss event = timeout or 3 duplicate acks
- TCP sender reduces rate (congestion window) after loss event

Rate adjustment: (probing)

- slow start
- Congestion avoidance: additive Increase and Multiplicative Decrease (AIMD)
- conservative after timeout events

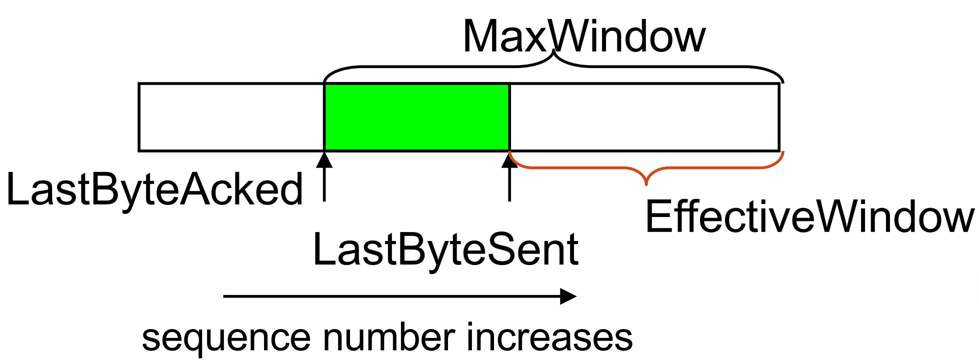
Packet loss == congestion?

Congestion Window (cwnd)

- Limits how much data can be in transit
- Implemented in number of bytes

MaxWindow = min(cwnd, rwnd)

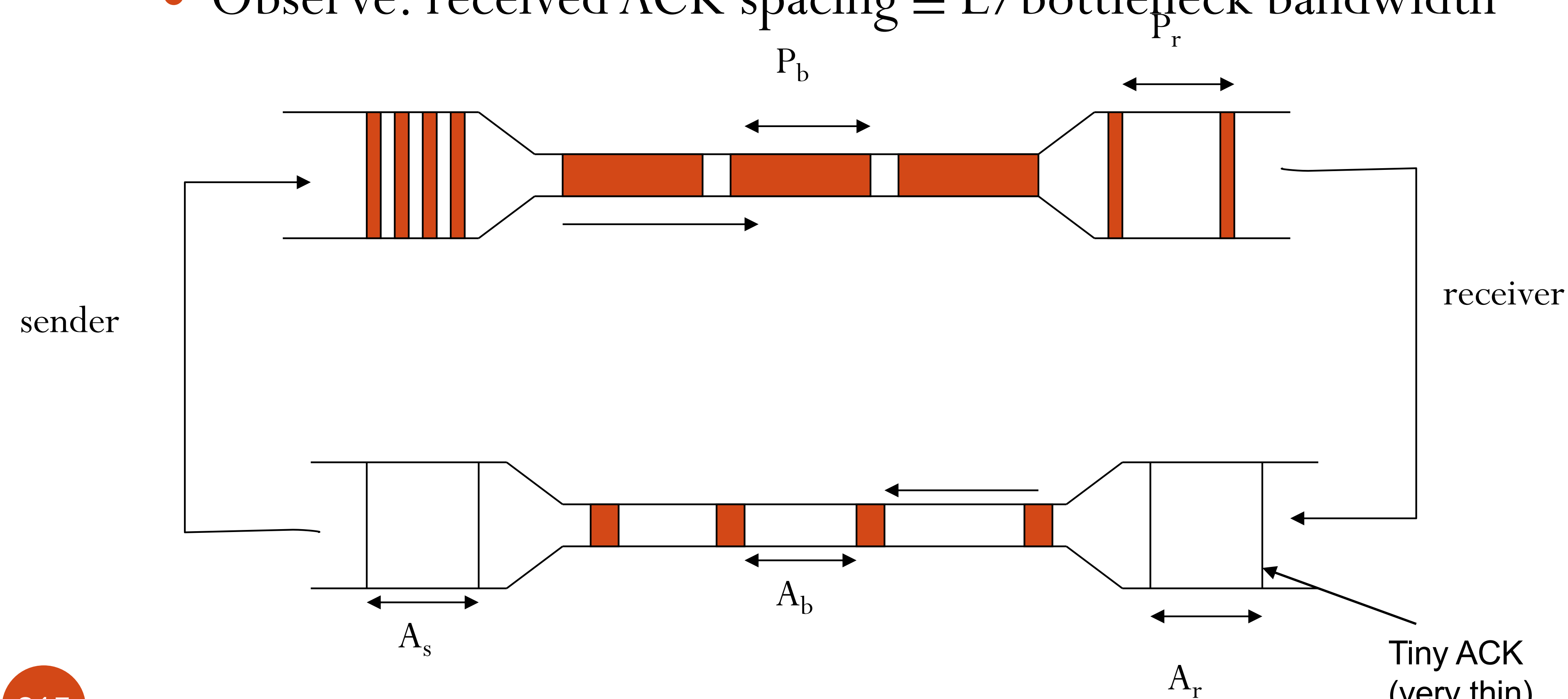
EffectiveWindow = MaxWindow – (LastByteSent – LastByteAcked)



$$\text{rate} \cong \frac{\min(\text{rwnd},\text{cwnd})}{\text{RTT}} \text{ Bytes/sec}$$

Self-clocking

- If we have a large window, ACKs "self-clock" the data to the rate of the bottleneck link
- Observe: received ACK spacing ≅ L/bottleneck bandwidth

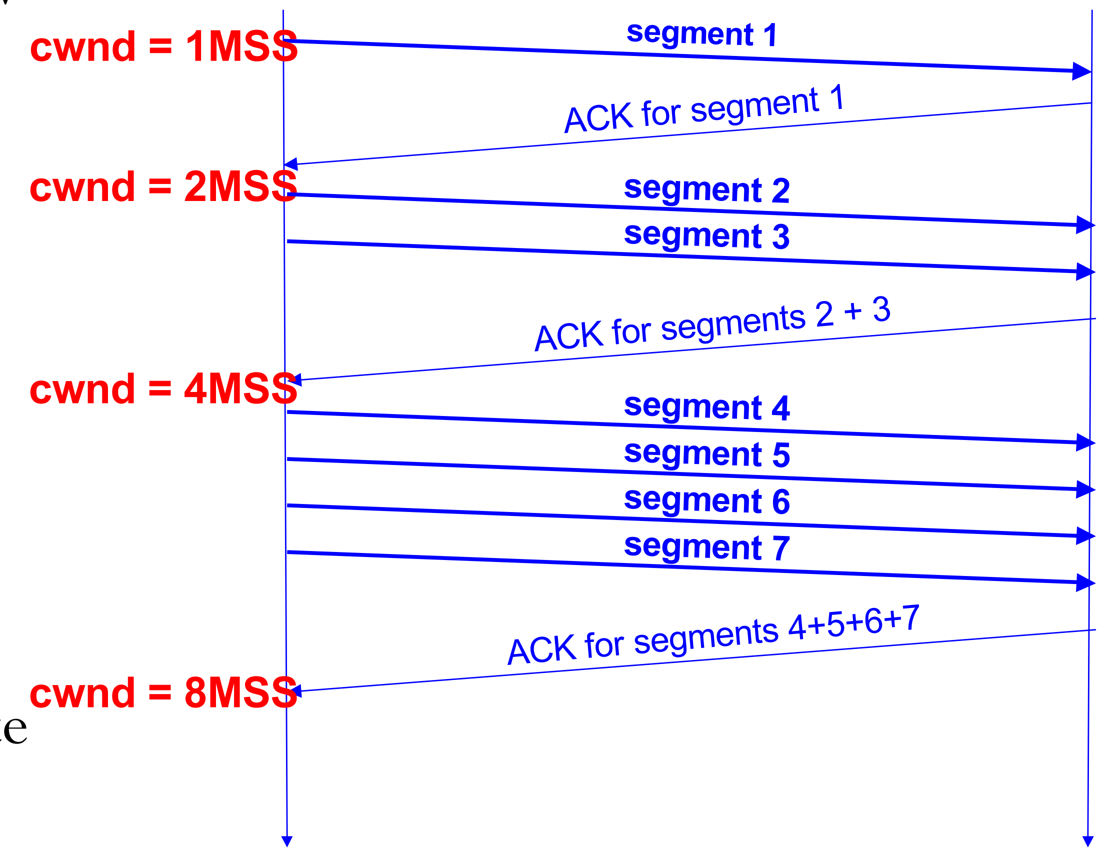


TCP: Slow Start

- Goal: discover roughly the proper sending rate quickly
- Whenever starting traffic on a new connection, or whenever increasing traffic after congestion (timeout) was experienced:
  - Initial cwnd =1MSS
  - Each time a segment is acknowledged, increment cwnd by one MSS
- Continue until
  - Reach ss_thresh
  - Packet loss

Slow Start Illustration

- The congestion window size grows very rapidly
- TCP slows down the increase of cwnd when cwnd >= ss_thresh
- Observe:
  - Each ACK generates two packets
  - slow start increases rate exponentially fast (doubled every RTT)!



assume 1 MSS = 1 byte

Congestion Avoidance (After Slow Start)

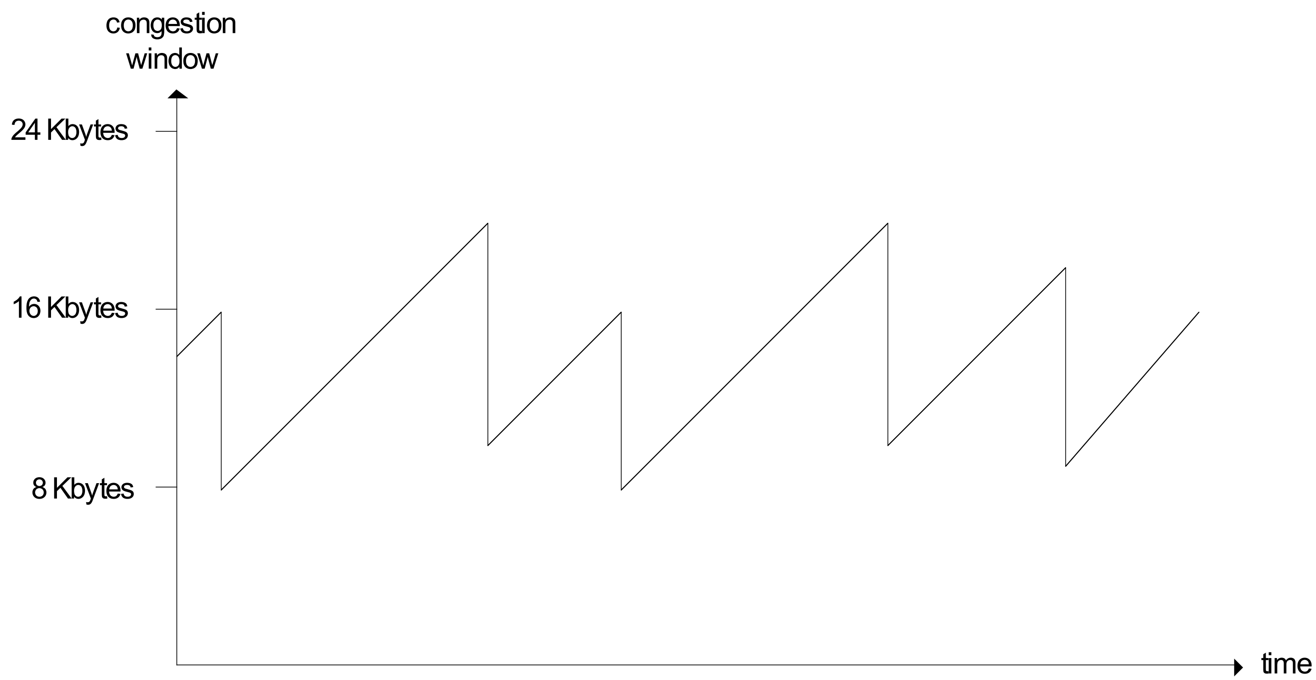
- Slow Start figures out roughly the rate at which the network starts getting congested
- Congestion Avoidance continues to react to network condition
  - Probes for more bandwidth, increase cwnd if more bandwidth available
  - If congestion detected, aggressively cut back cwnd

  - How?

TCP Multiplicative Decrease & Additive increase (AIMD)

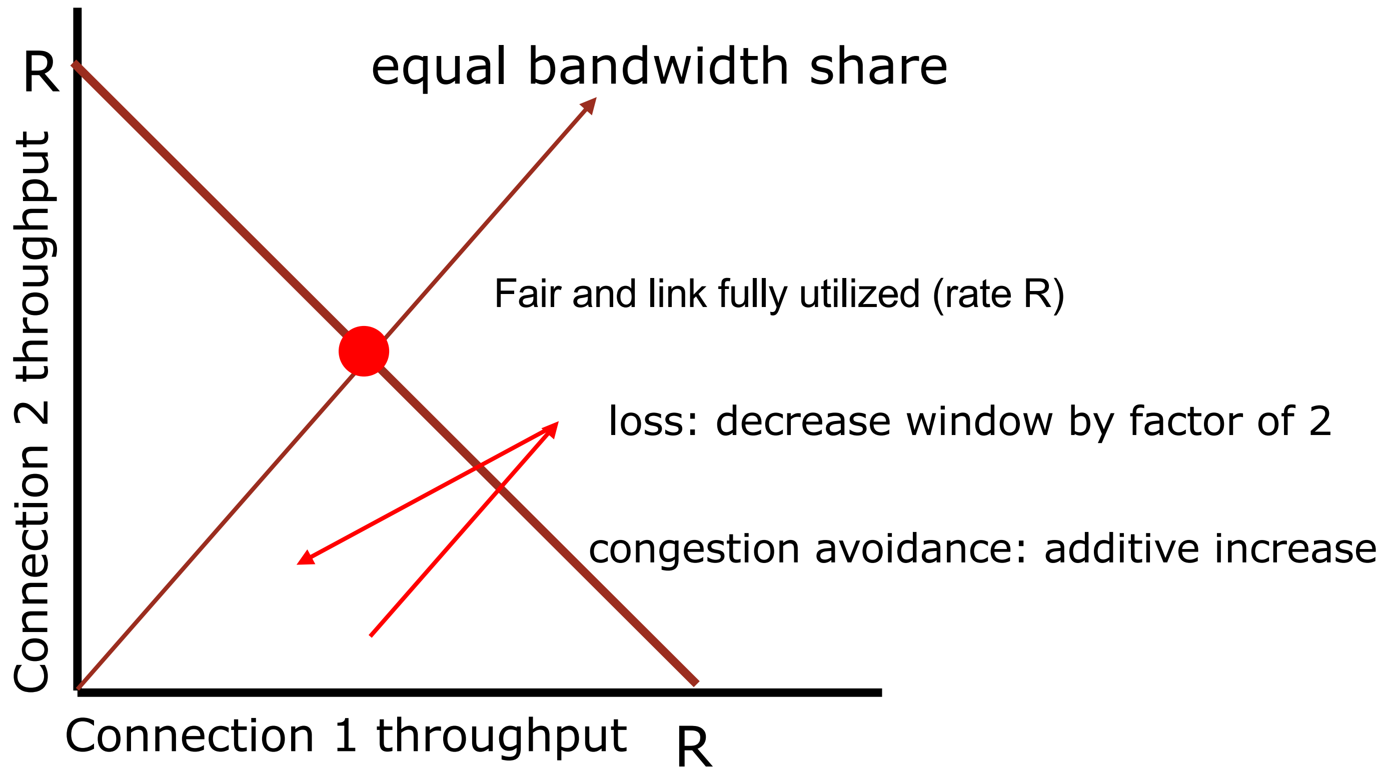
- multiplicative decrease: cut cwnd in half after loss event

additive increase: increase **cwnd** by 1 MSS every RTT in the absence of loss events: *probing*

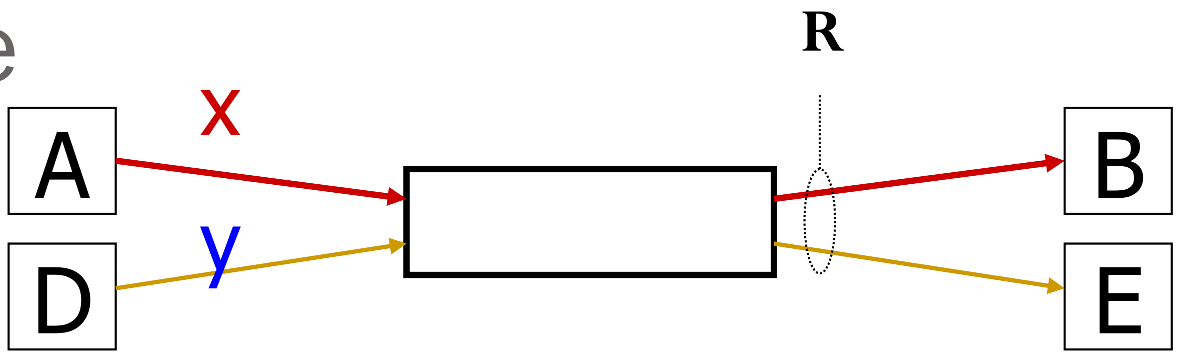


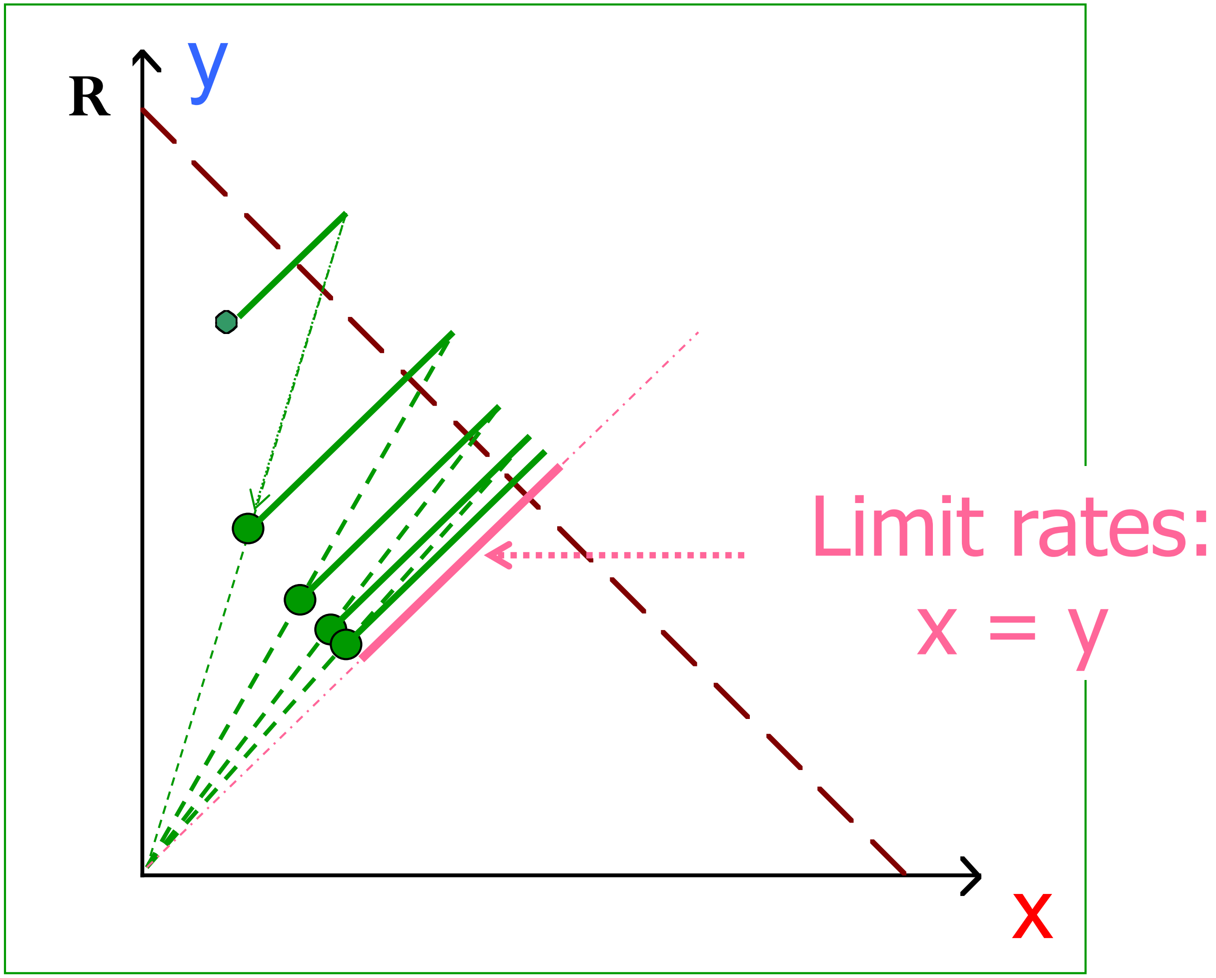
Why AIMD

- Two competing sessions
- Additive increase (AI) gives slope of 1, as throughout increases
- multiplicative decrease (MD) decreases throughput proportionally

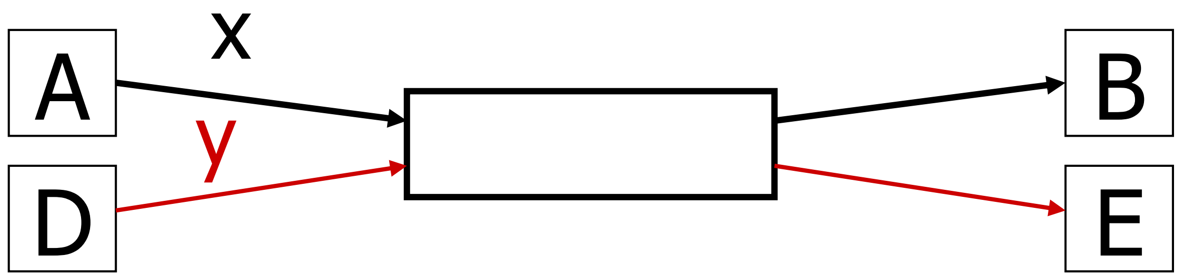


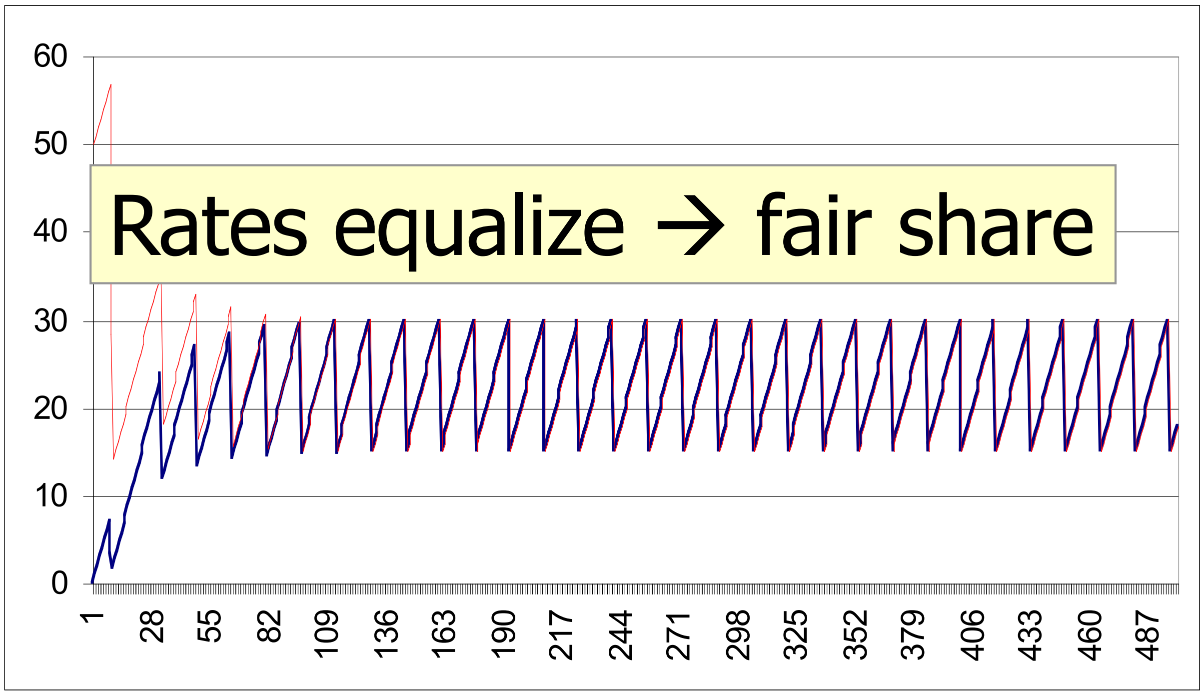
AIMD Example





AIMD Sharing Dynamics

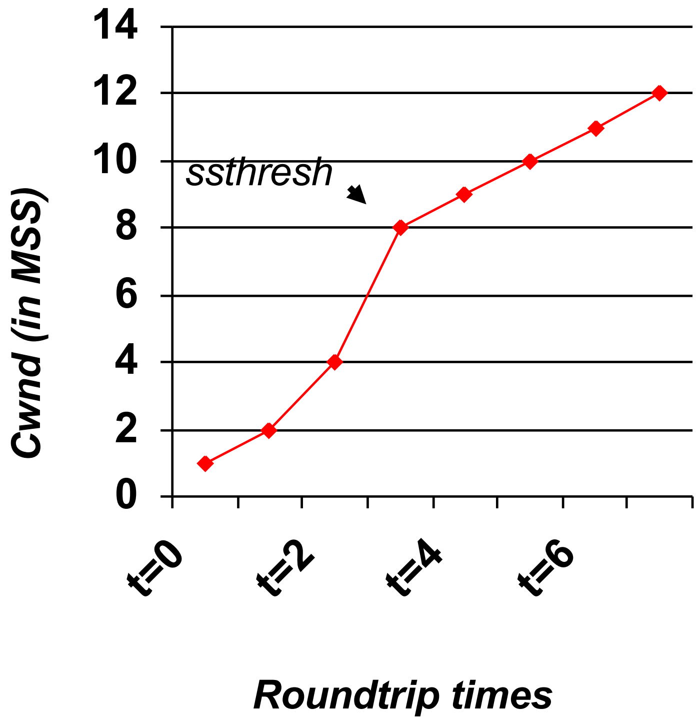


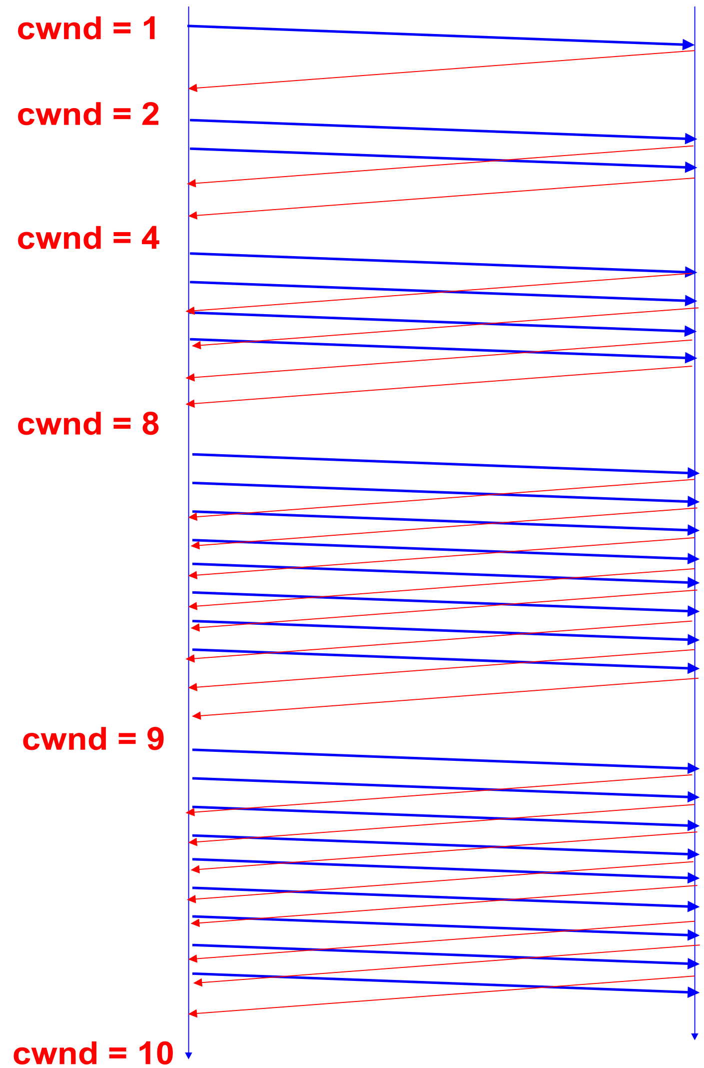


- No congestion → rate increases by one packet/RTT every RTT
- Congestion → decrease rate by factor 2

Example of Slow Start + Congestion Avoidance

- Assume that ss_thresh = 8MSS





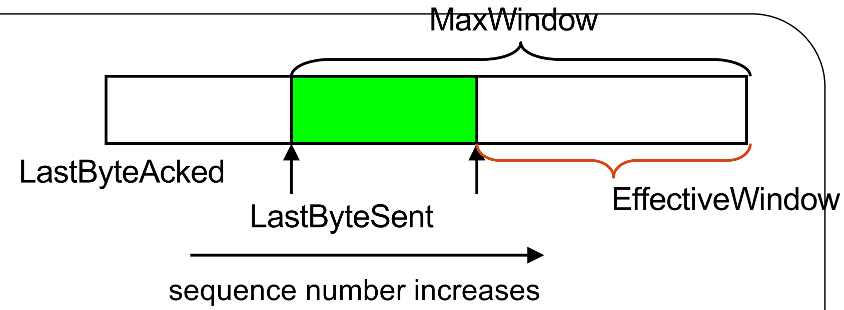
Responses to Congestion (Loss)

- There are algorithms developed for TCP to respond to congestion
  - TCP Tahoe
  - TCP Reno
- and many more:
  - TCP Vegas (research: use timing of ACKs to avoid loss)
  - TCP SACK (future deployment: selective ACK)

TCP Reno

- Upon timeout, cut ss_thresh by ½ and cwnd = 1MSS
  - Go to slow start phase
- **Fast retransmit and fast recovery** mechanism
  - Upon receiving 3 duplicate ACKs, retransmit the presumed lost segment ("fast retransmit")
  - But do not enter slow-start. Instead enter congestion avoidance ("fast recovery")

Fast retransmit algorithm:




```
event: ACK received, with ACK field value of y
        if (y > LastByteAcked+1) {
            LastByteAcked = y-1
            if (there are currently not-yet-acknowledged segments)
                start timer
          }
        else {
             increment count of dup ACKs received for y
             if (count of dup ACKs received for y = 3) {
                 resend segment with sequence number y
              }
```

a duplicate ACK for already ACKed segment

fast retransmit

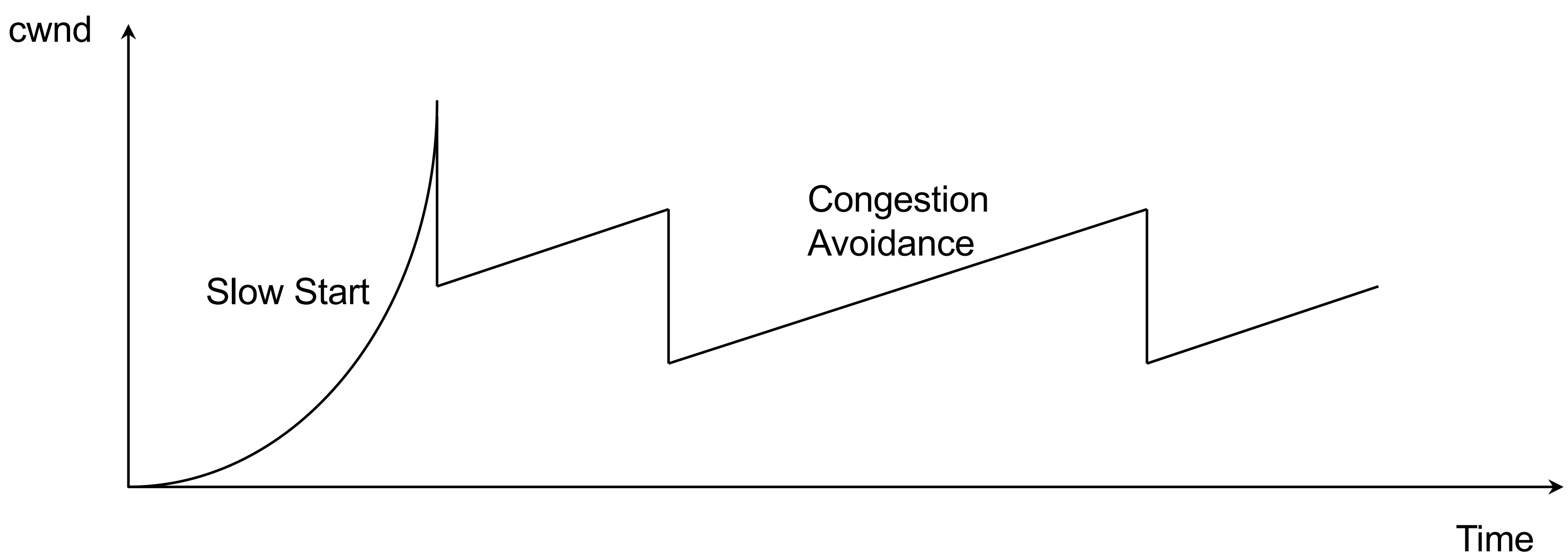
Fast Recovery

- After a fast-retransmit
  - cwnd = cwnd/2 (vs. 1 in Tahoe)
  - ss_thresh = cwnd
  - i.e. starts congestion avoidance at new cwnd
    - Not slow start from cwnd = 1MSS

- After a timeout
  - ss_thresh = cwnd/2
  - cwnd = 1MSS
  - Do slow start

Fast Retransmit and Fast Recovery

- Slow start only once per session (if no timeouts)
- In steady state, cwnd oscillates around the ideal window size.



Summary (1)

State	Event	TCP Reno	
		Sender Action	**Comment**
Slow Start (SS)	ACK receipt for previously unacked data	cwnd = cwnd + MSS, If (cwnd > Threshold) set state to “Congestion Avoidance”	Resulting in a doubling of cwnd every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	cwnd = cwnd +MSS * (MSS/cwnd)	Additive increase, resulting in increase of cwnd by 1 MSS every RTT
SS or CA	Loss event detected by triple duplicate ACK	ss_threshold = cwnd /2, cwnd = ss_threshold, Set state to “Congestion Avoidance”	Fast recovery, implementing multiplicative decrease. cwnd will not drop below 1 MSS.
SS or CA	Timeout	ss_threshold = cwnd /2, cwnd = 1 MSS, Set state to “Slow Start”	Enter slow start
SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	cwnd and ss_thresh not changed

Summary (2)

Distinguish algorithms & protocols!

- Multiplexing and demultiplexing
  - Port number
- Error detection/recovery
  - Internet checksum (inclusion of pseudo header)
  - Stop-n-Wait, Go-back-N, selective repeat
- Connection establishment/termination
- Flow control
- Congestion control
  - Sliding window
  - AIMD
  - Slow start
  - Fast retransmit & recovery