CS 4390: Transport Layer

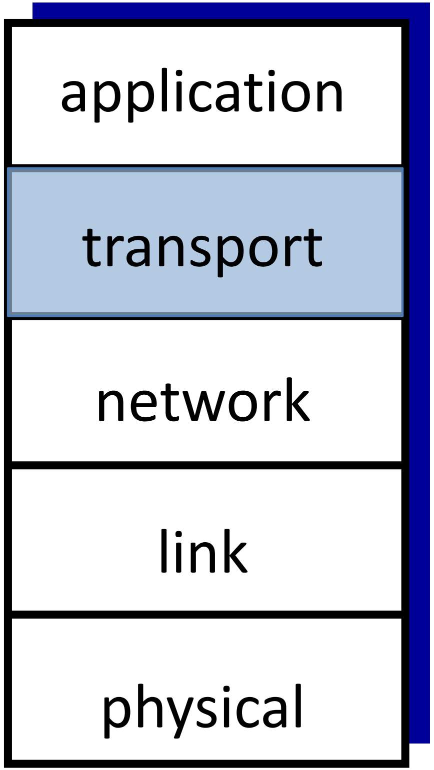
Shuang Hao
University of Texas at Dallas
Fall 2023

Outline

- Transport layer
  - Multiplexing

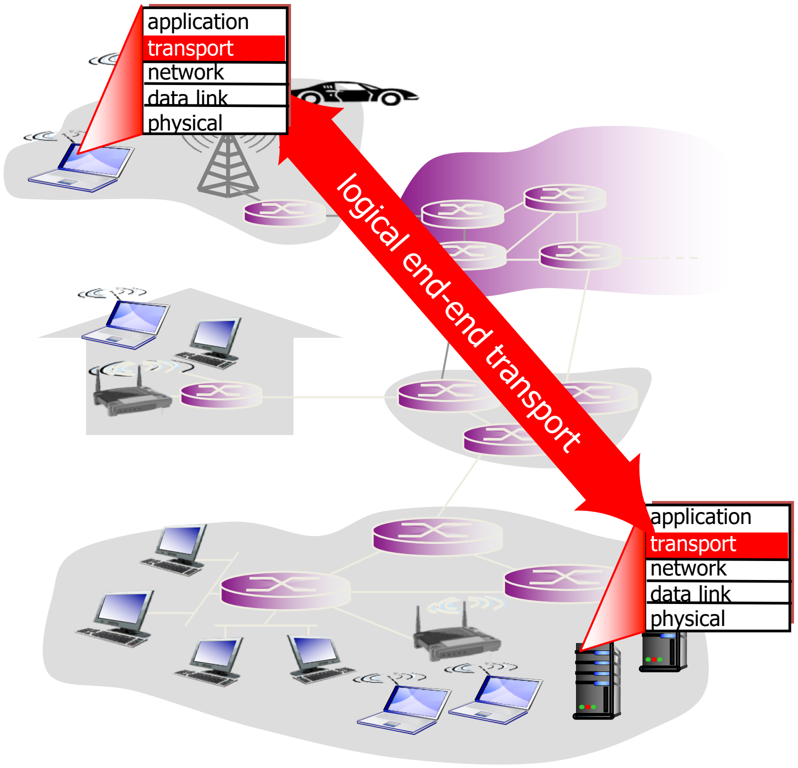
Transport Layer

- Our goals
  - Understand principles behind transport layer services:
    - multiplexing, demultiplexing
    - reliable data transfer
    - flow control
    - congestion control
  - Learn about Internet transport layer protocols:
    - UDP: connectionless transport
    - TCP: connection-oriented reliable transport



Transport Services and Protocols

- Provide logical communication between app processes running on different hosts
- Transport protocols run in end systems
  - send side: breaks app messages into segments, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- More than one transport protocol available to apps
  - Internet: TCP and UDP



Transport vs. Network Layer

- Network layer
  - Logical communication between hosts
- Transport layer
  - Logical communication between processes
  - relies on, enhances, network layer services

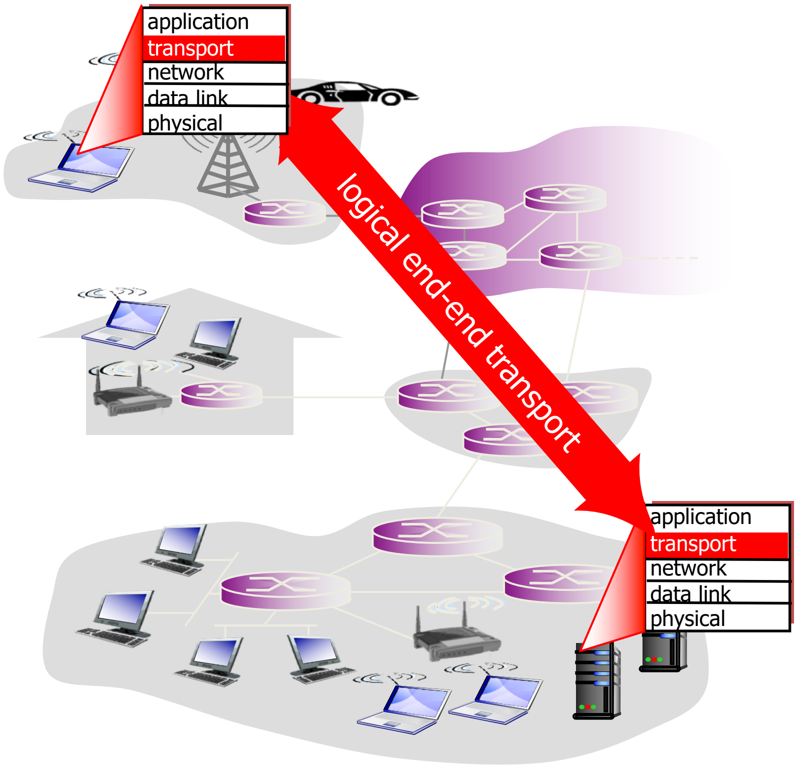
household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

Internet Transport Layer Protocols

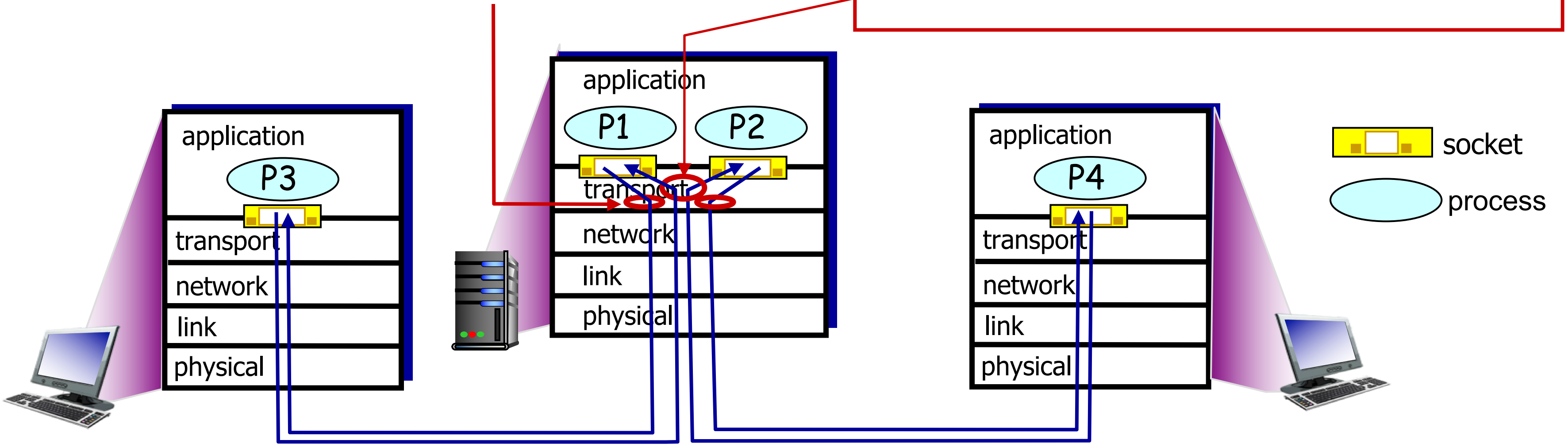
- Reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- Unreliable, unordered delivery (UDP)
  - extension of "best-effort" IP
- Services not available:
  - delay guarantees
  - bandwidth guarantees



Multiplexing/demultiplexing

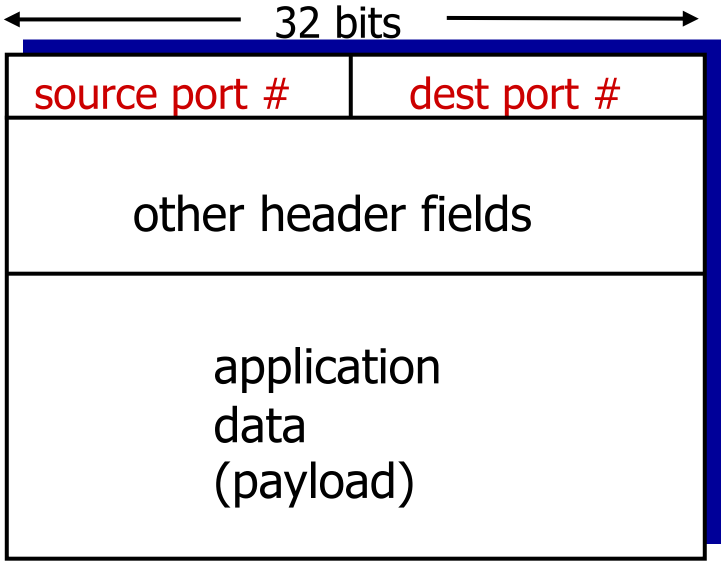
multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver:
use header info to deliver received segments to correct socket



How Demultiplexing Works

- Host receives IP datagrams
  - Each datagram has source, destination IP address
  - Each datagram carries one transport-layer segment
  - Each segment has source, destination port number
- Host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

Connectionless Demultiplexing

- *Create* socket has host-local port #:

```
DatagramSocket mySocket1 = new
DatagramSocket(12534);
```

- *When creating datagram to send into UDP socket, must specify*
  - *destination IP address*
  - *destination port #*

- When host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

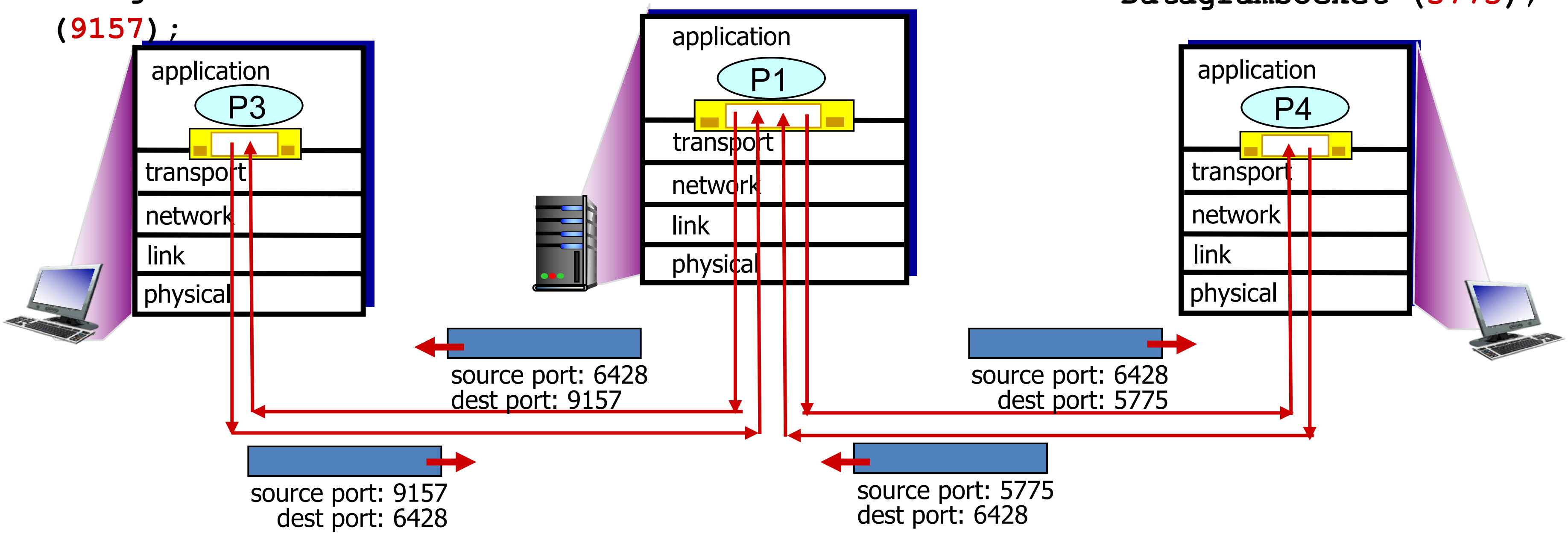
IP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

Connectionless Demux: Example

```
DatagramSocket serverSocket
 = new DatagramSocket
(6428);
```

```
DatagramSocket
 mySocket2 = new
 DatagramSocket
 (9157);
```

```
DatagramSocket
 mySocket1 = new
 DatagramSocket (5775);
```



Recap Where We're At

- Transport layer
  - Multiplexing

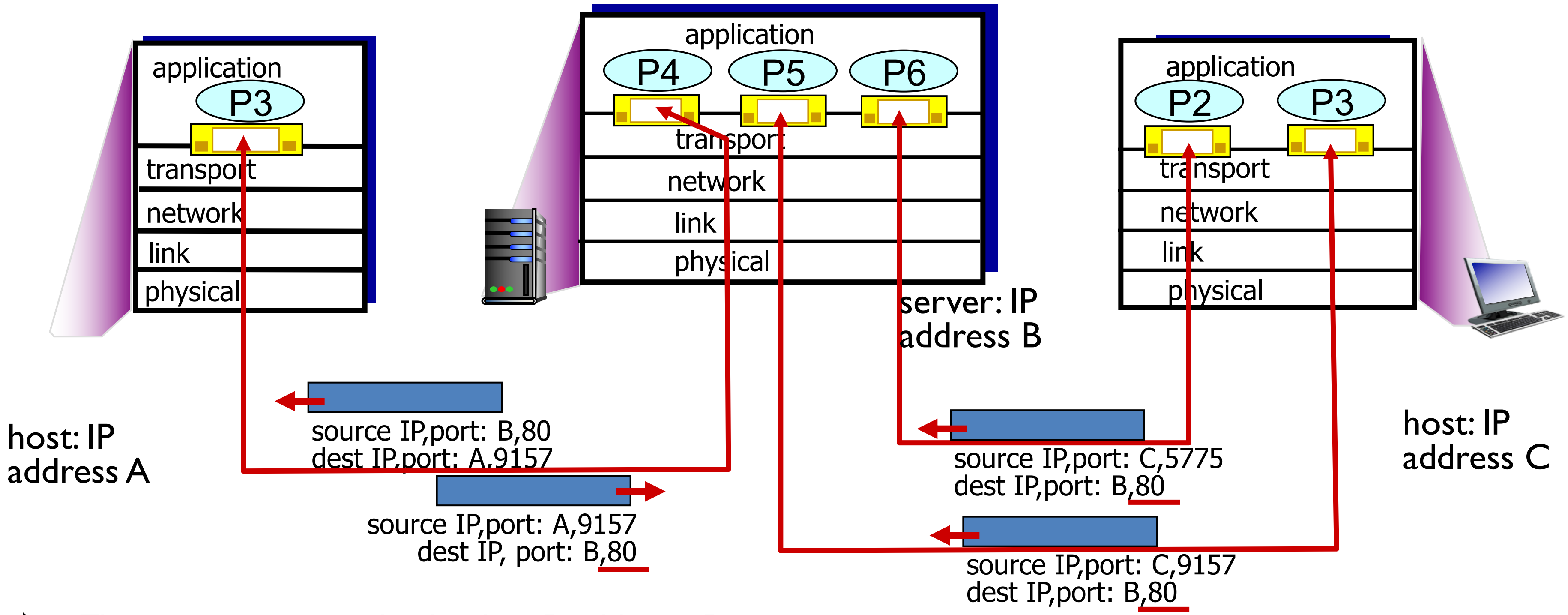
Outline

- Transport layer
  - Multiplexing (continue)
- UDP
- TCP

Connection-oriented Demux

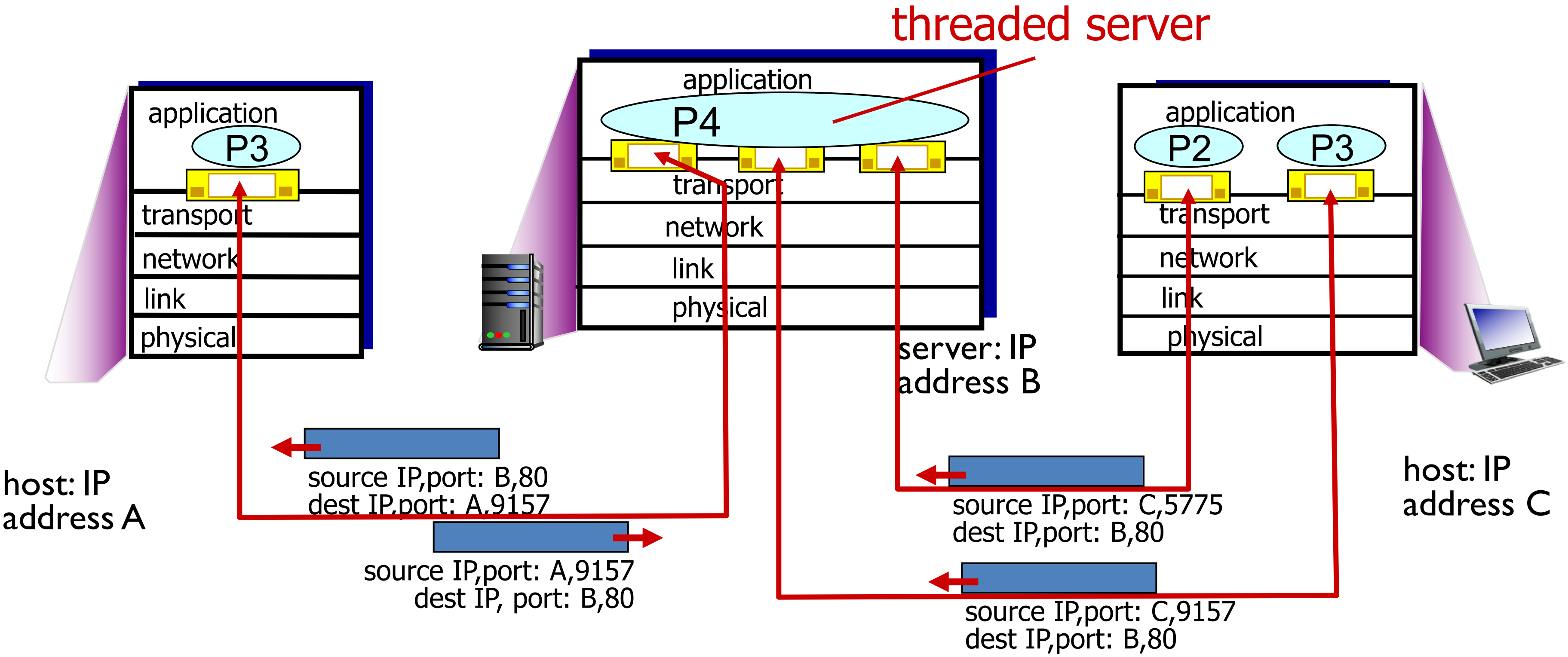
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- Demux
  - Receiver uses all four values to direct segment to appropriate socket
- Server host may support many simultaneous TCP sockets:
  - Each socket identified by its own 4-tuple
  - For example, web servers have different sockets for each connecting client

Connection-oriented Demux: Example



- Three segments, all destined to IP address: B, dest port: 80 are demultiplexed to different sockets

Connection-oriented Demux: Example



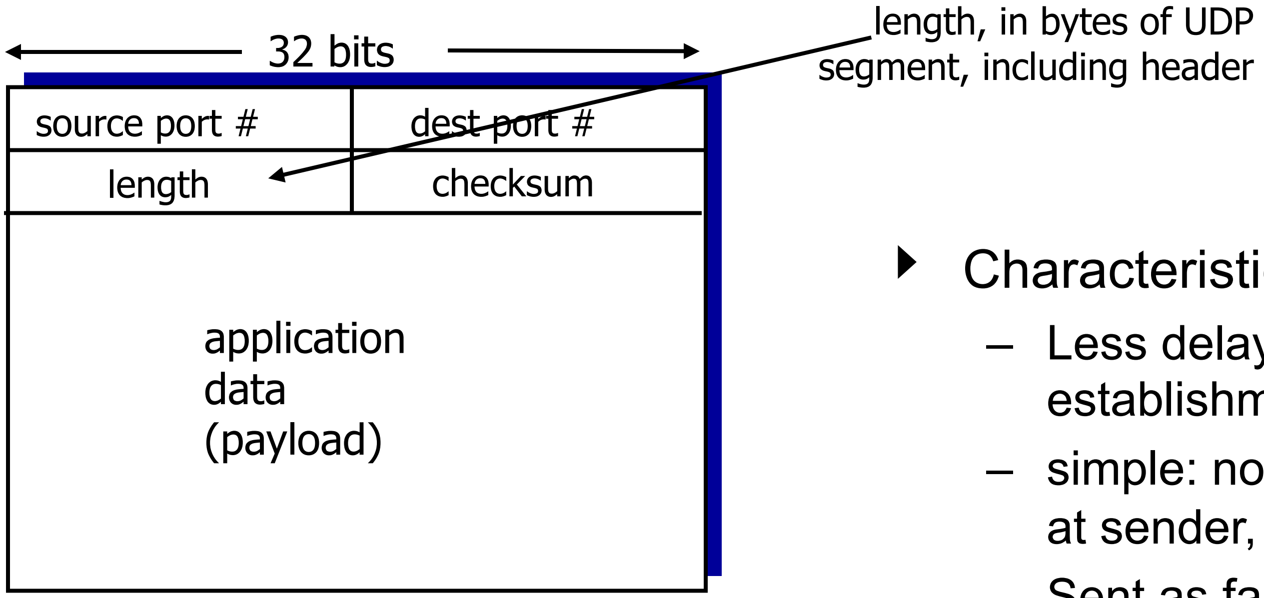
UDP: User Datagram Protocol (RFC 768)

- Connectionless:
  - No handshaking between UDP sender, receiver
  - Each UDP segment handled independently of others
  - “Best effort” service

- UDP use
  - Streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS

- Reliable transfer over UDP:
  - Add reliability at application layer
  - Application-specific error recovery

UDP: Segment Structure



UDP segment format

- Characteristics of UDP
  - Less delay: no connection establishment
  - simple: no connection state at sender, receiver
  - Sent as fast as desired: do not consider congestion

UDP Checksum

- Goal: detect “errors” (e.g., flipped bits) in transmitted segment

- Sender
  - Treat segment contents, including header fields, as sequence of 16-bit integers
  - Checksum: the 1s complement of the sum of segment contents
  - Sender puts checksum value into UDP checksum field

- Receiver
  - Compute checksum of received segment
  - Check if computed checksum equals checksum field value:
    - NO - error detected
    - YES - no error detected.

Internet Checksum: Example

- Example: add two 16-bit integers

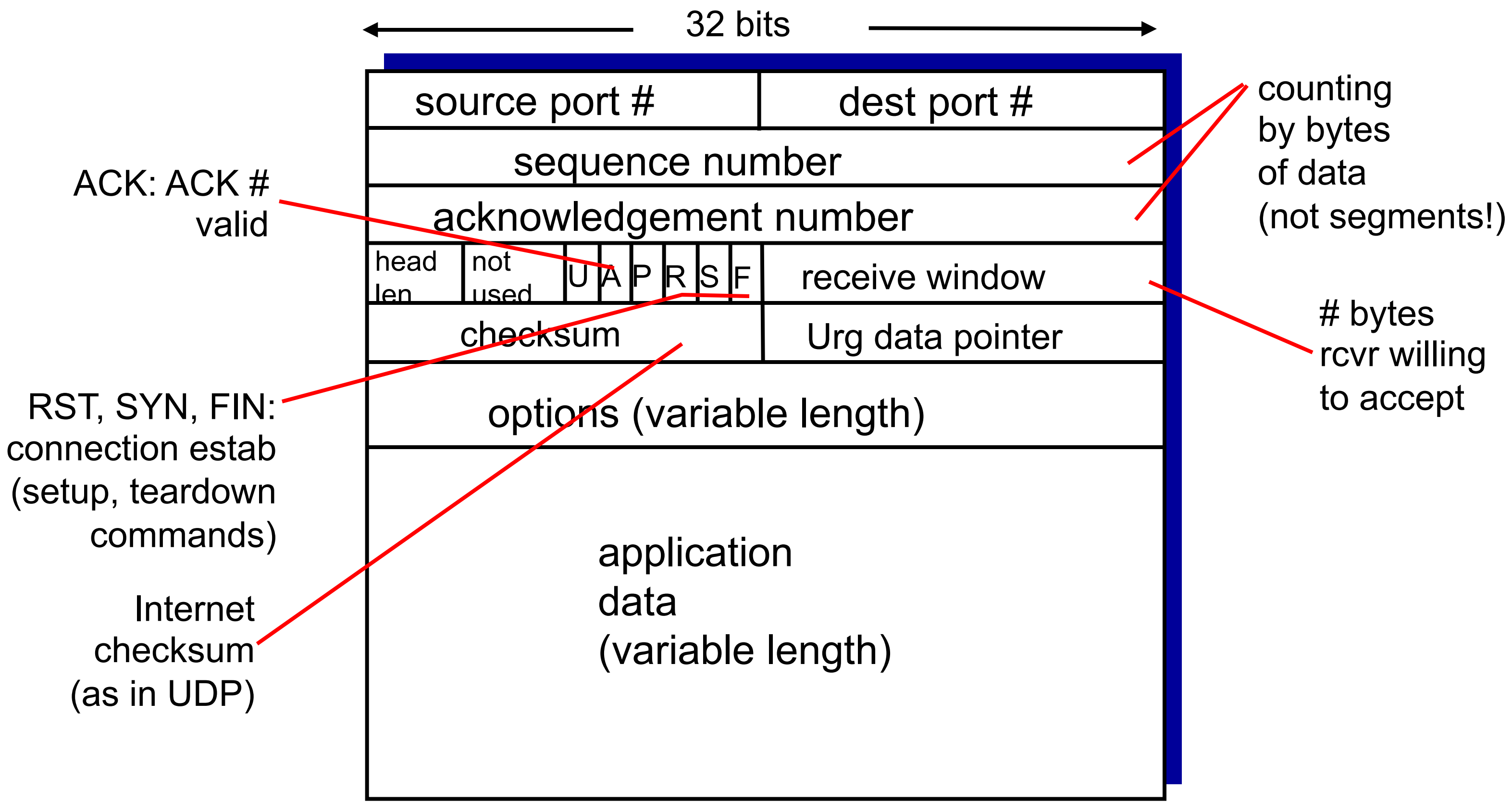
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

TCP: Overview (RFCs 793,1122,1323, 2018, 2581)

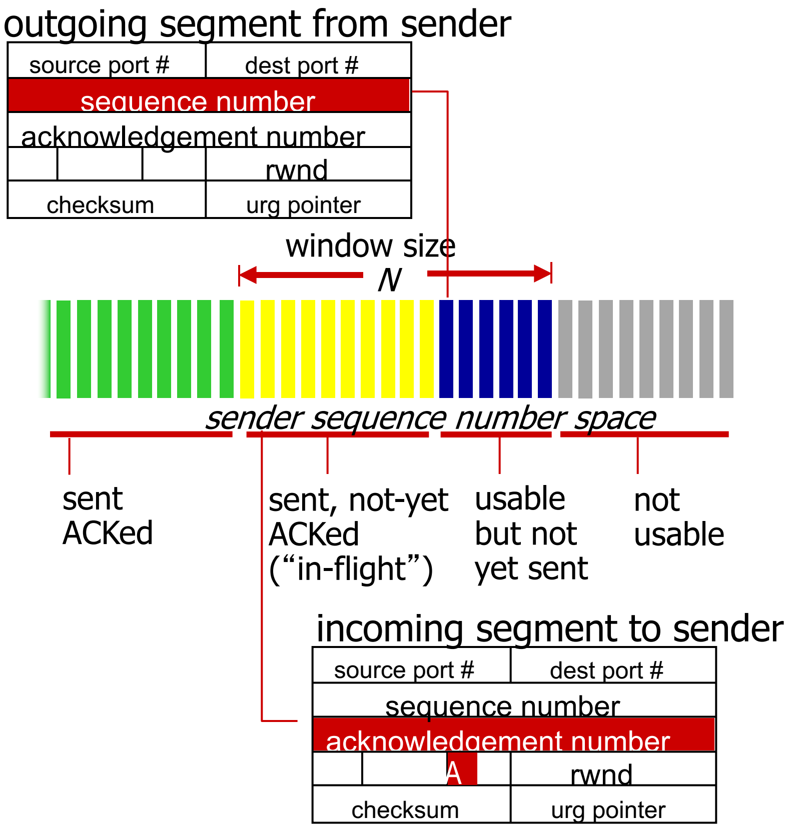
- Transmission Control Protocol
  - Reliable, in-order byte steam
  - Connection-oriented
    - Handshaking (exchange of control msgs) inits sender, receiver state before data exchange
  - Flow controlled
    - Sender will not overwhelm receiver

TCP Segment Structure

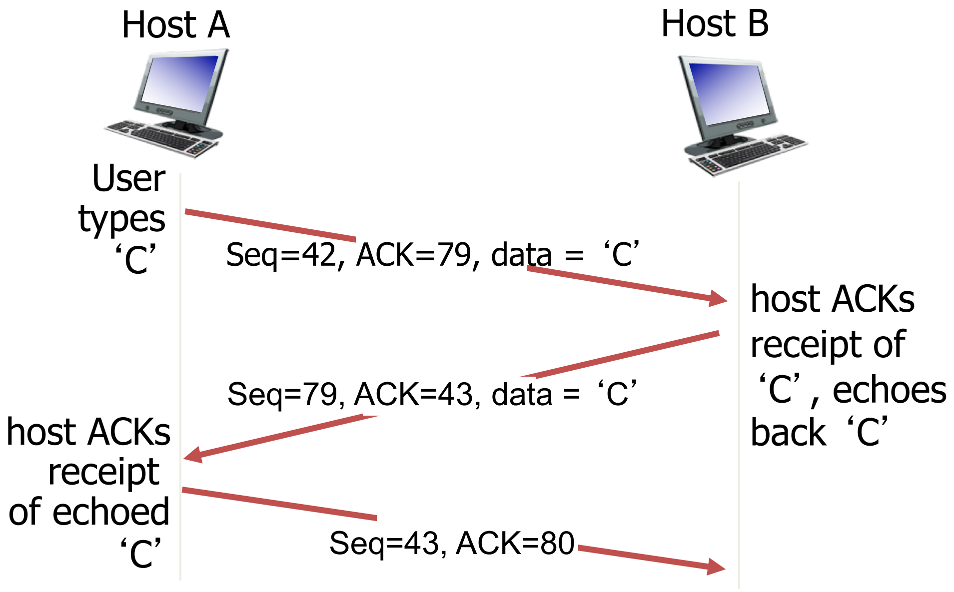


TCP Sequence Numbers, Acks

- Sequence numbers
  - Byte stream "number" of first byte in segment's data
- Acknowledgements
  - seq # of next byte expected from other side
  - cumulative ACK



TCP Sequence Numbers, Acks



TCP Round Trip Time, Timeout

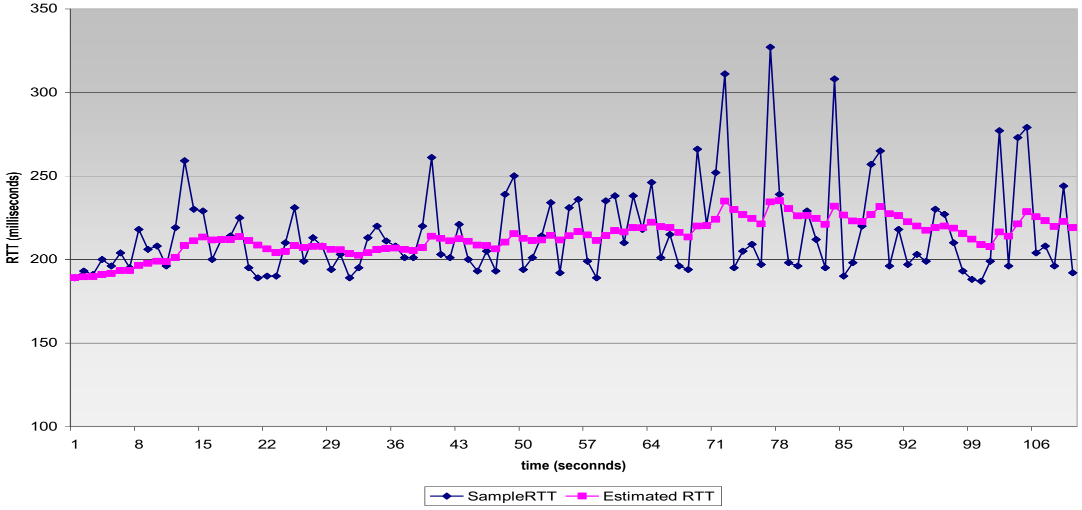
- Q: how to set TCP timeout value?
  - longer than RTT
    - but RTT varies
  - too short: premature timeout, unnecessary retransmissions
  - too long: slow reaction to segment loss

- 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

TCP Round Trip Time, Timeout

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

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.12$



TCP Round Trip Time, Timeout

- Timeout interval: EstimatedRTT plus “safety margin”
  - Large variation in EstimatedRTT -> larger safety margin

```
DevRTT = (1-β)*DevRTT +
                β*|SampleRTT-EstimatedRTT|
                (typically, β = 0.25)
```

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT

“safety margin”

Recap Where We're At

- Transport layer
  - Multiplexing (continue)
- UDP
- TCP

Outline

- TCP
  - Retransmission
  - Flow Control
  - Connection Management
  - Congestion Control

TCP Reliable Data Transfer

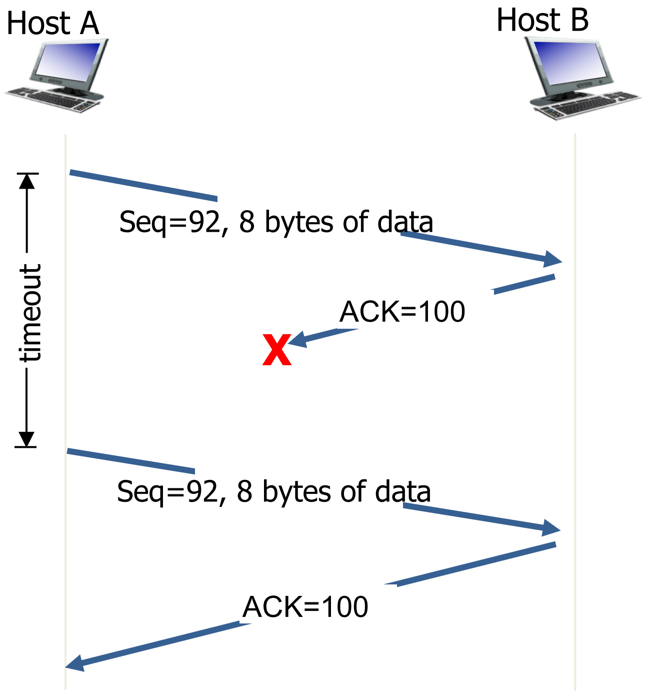
- TCP creates reliable data transfer service on top of IP’s unreliable service
  - Pipelined segments
  - Cumulative acks
  - Single retransmission timer
- Retransmissions triggered by:
  - Timeout events
  - Duplicate acks
- Let’s initially consider simplified TCP sender
  - Ignore duplicate acks
  - Ignore flow control, congestion control

TCP Sender Events

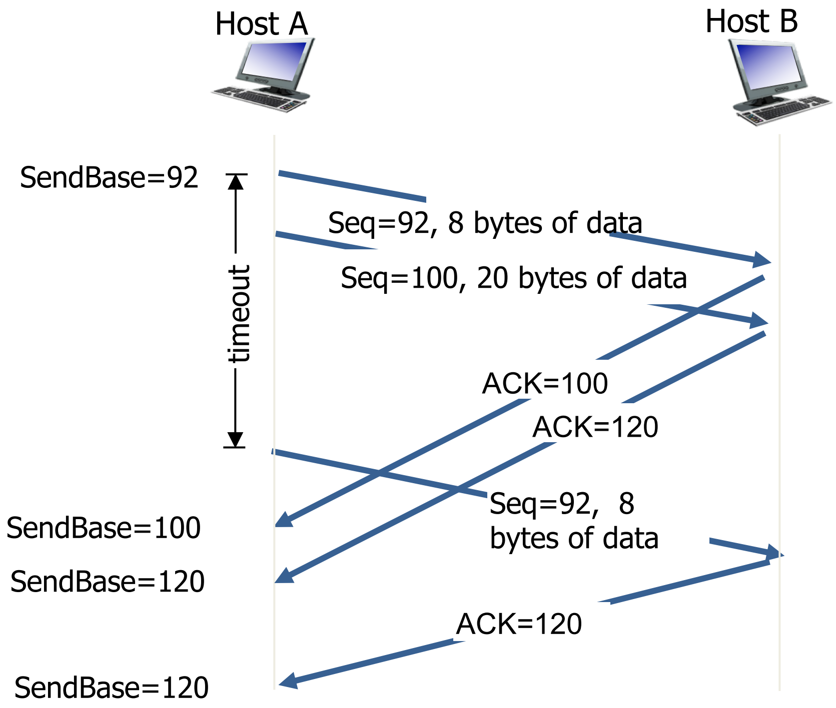
- Data received from app
  - create segment with seq #
  - seq # is byte-stream number of first data byte in segment
  - start timer if not already running
    - expiration interval: TimeOutInterval

- Timeout
  - retransmit segment that caused timeout
  - restart timer
- ack received
  - if ack acknowledges previously unacked segments
    - update what is known to be ACKed
    - start timer if there are still unacked segments

TCP: Retransmission Scenarios

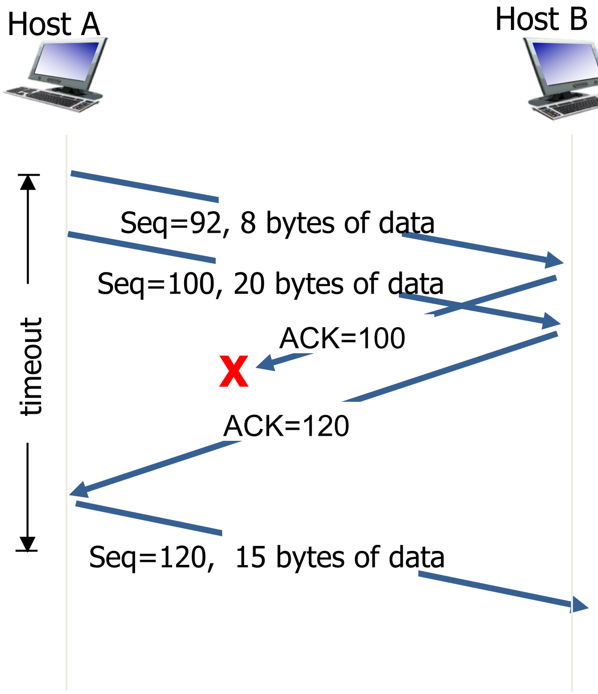


lost ACK scenario



premature timeout

TCP: Retransmission Scenarios

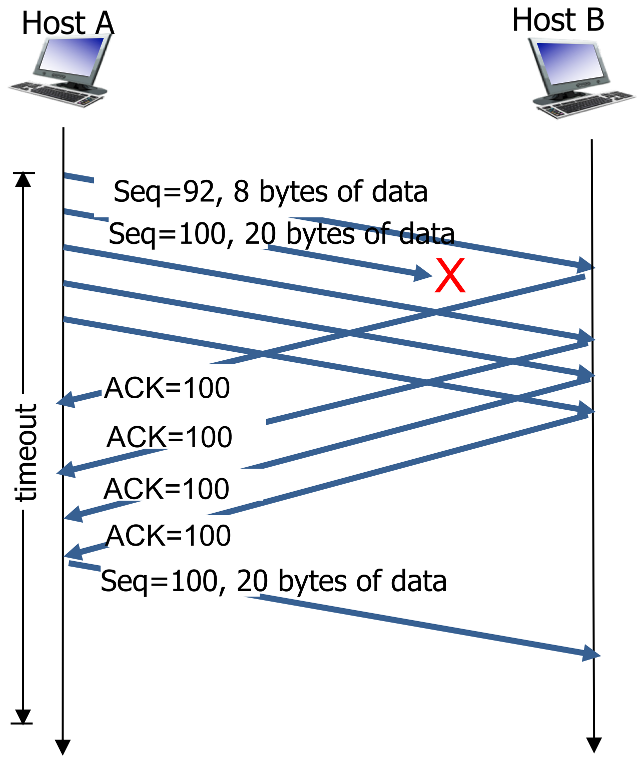


cumulative ACK

TCP Fast Retransmit

- Time-out period often relatively long
  - Long delay before resending lost packet
- Detect lost segments via duplicate ACKs
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs
- TCP fast retransmit
  - If sender receives 3 ACKs for same data
    - (“triple duplicate ACKs”), resend unacked segment with smallest seq #
    - likely that unacked segment lost, so don’t wait for timeout

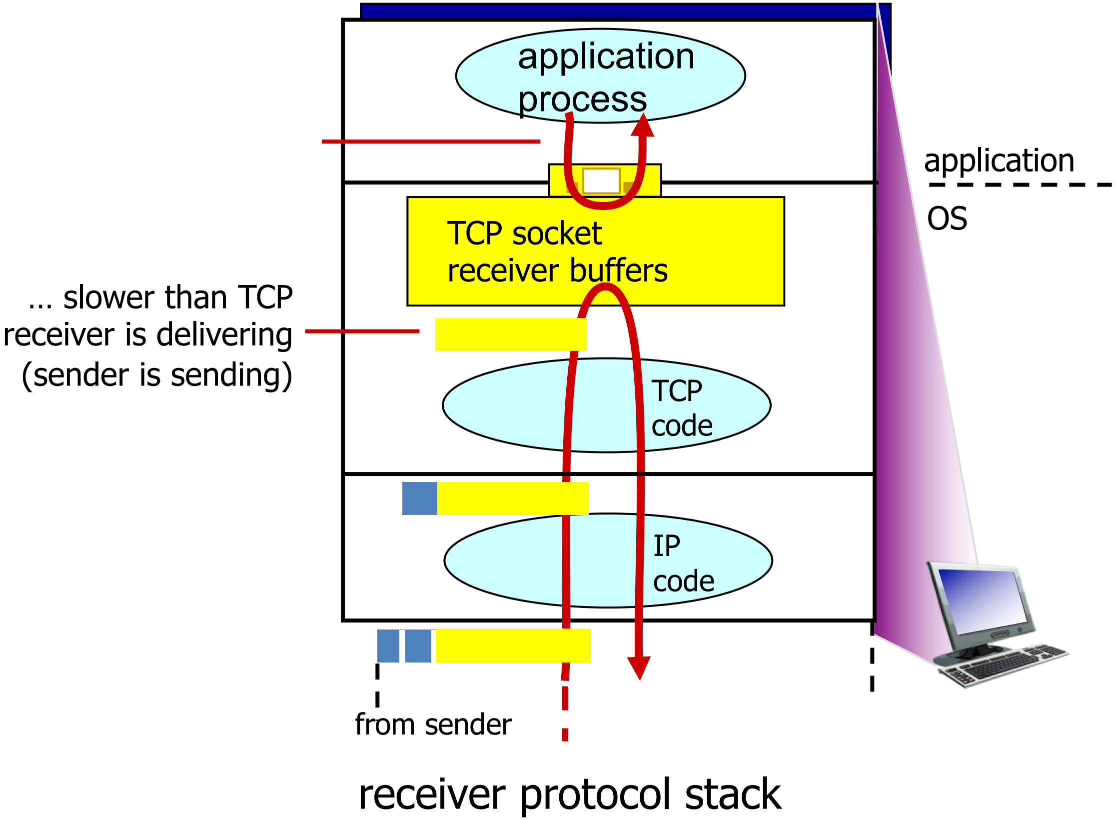
TCP Fast Retransmit



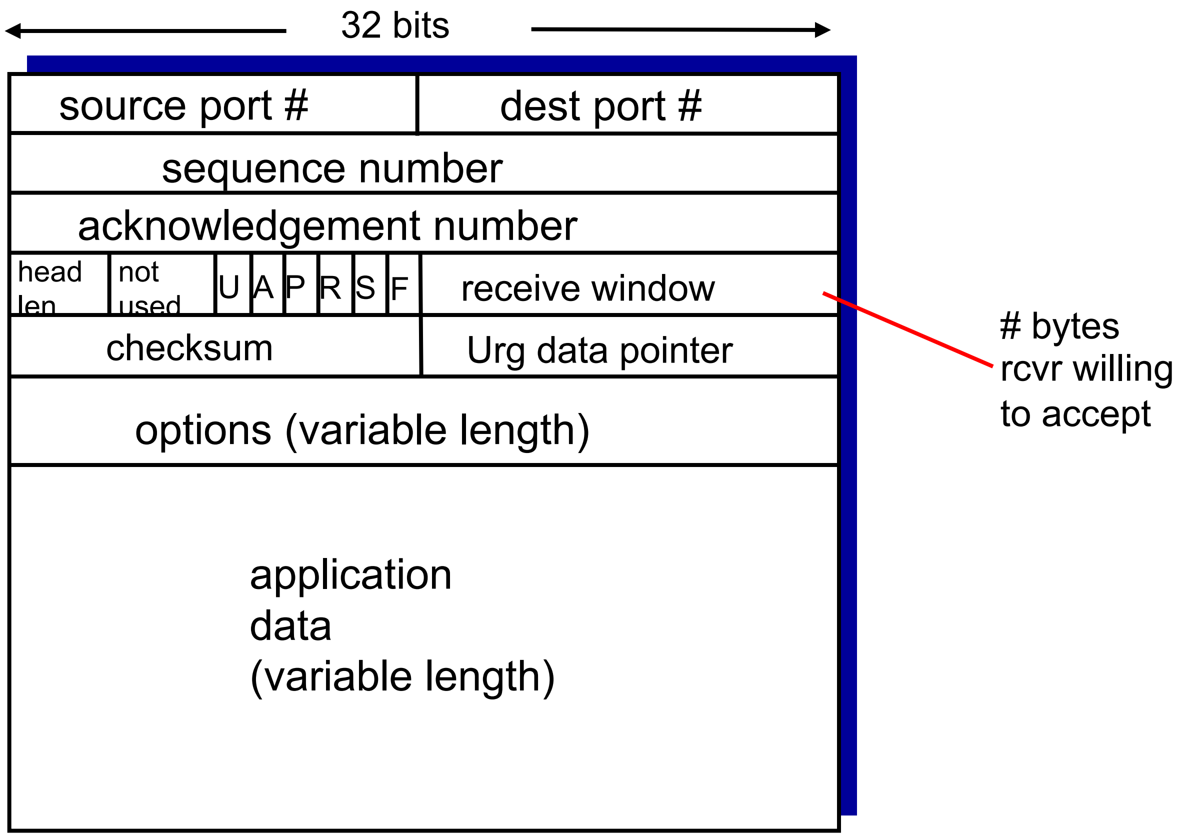
fast retransmit after sender receipt of triple duplicate ACK

TCP Flow Control

- Flow control
  - receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

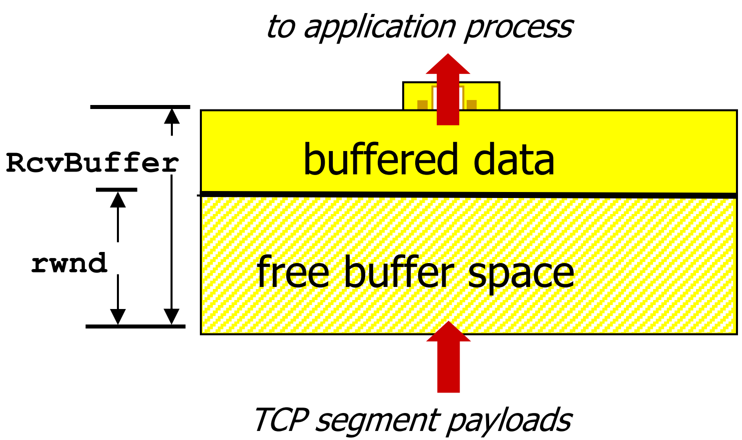


TCP Segment Structure



TCP Flow Control

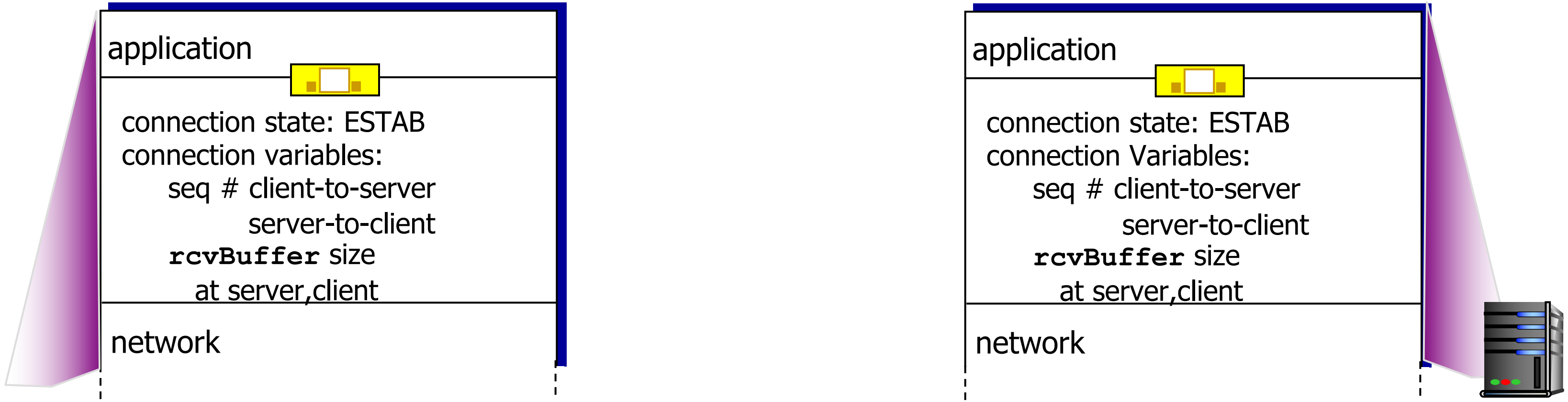
- Receiver "advertises" free buffer space by including rwnd value in TCP header of receiver-to-sender segments
- Sender limits amount of unacked ("in-flight") data to receiver's rwnd value
- Guarantees receive buffer will not overflow



receiver-side buffering

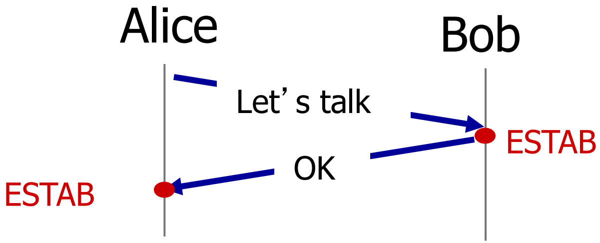
Connection Management

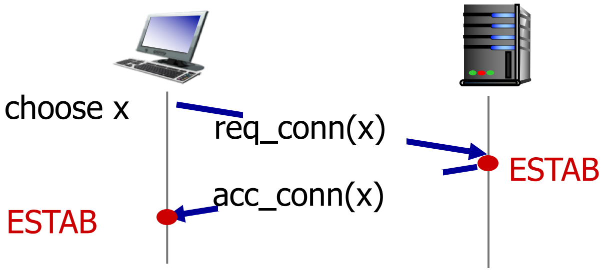
- Before exchanging data, sender/receiver “handshake”:
  – Agree to establish connection (each knowing the other willing to establish connection)
  – Agree on connection parameters



Agreeing to Establish A Connection

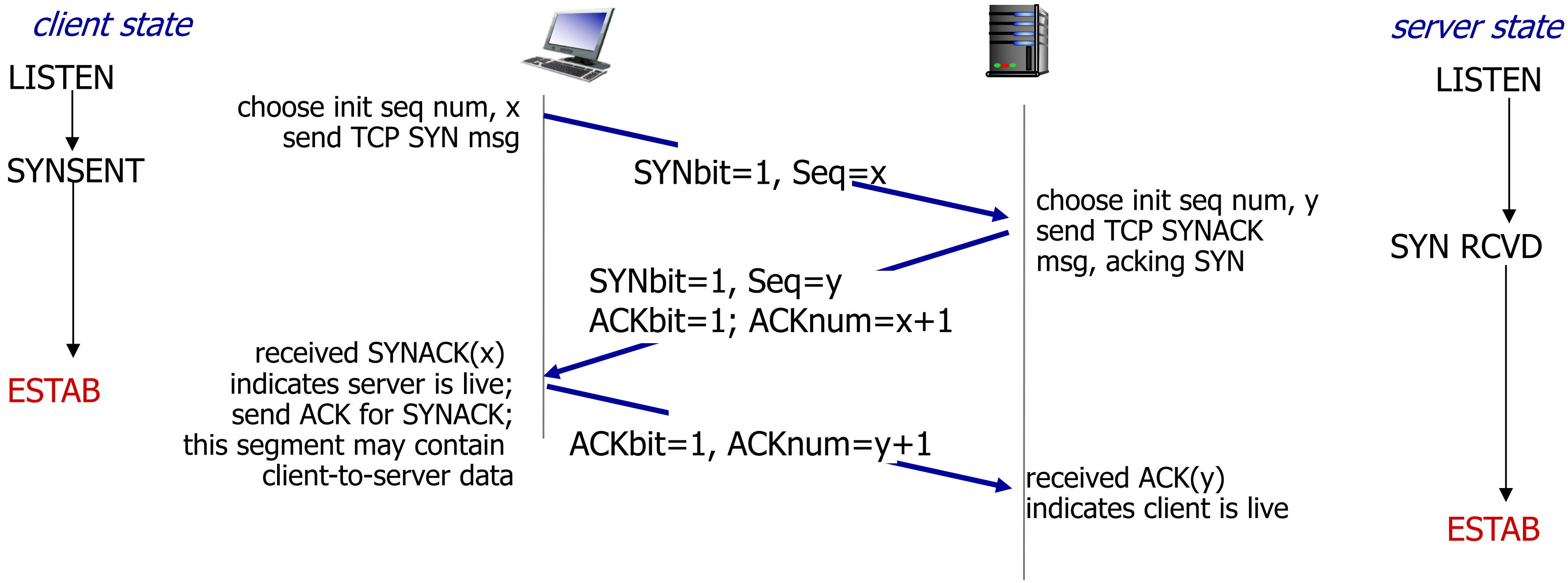
2-way handshake:





- Will 2-way handshake always work in network?
  - can't "see" other side

TCP 3-way Handshake

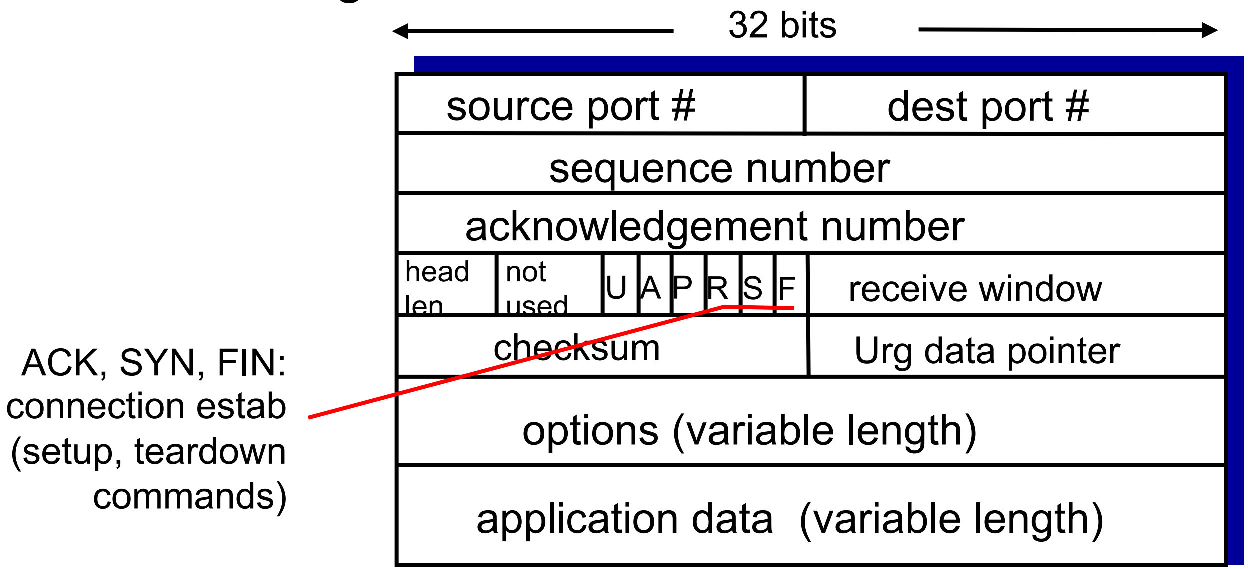


SYN-flooding Attack

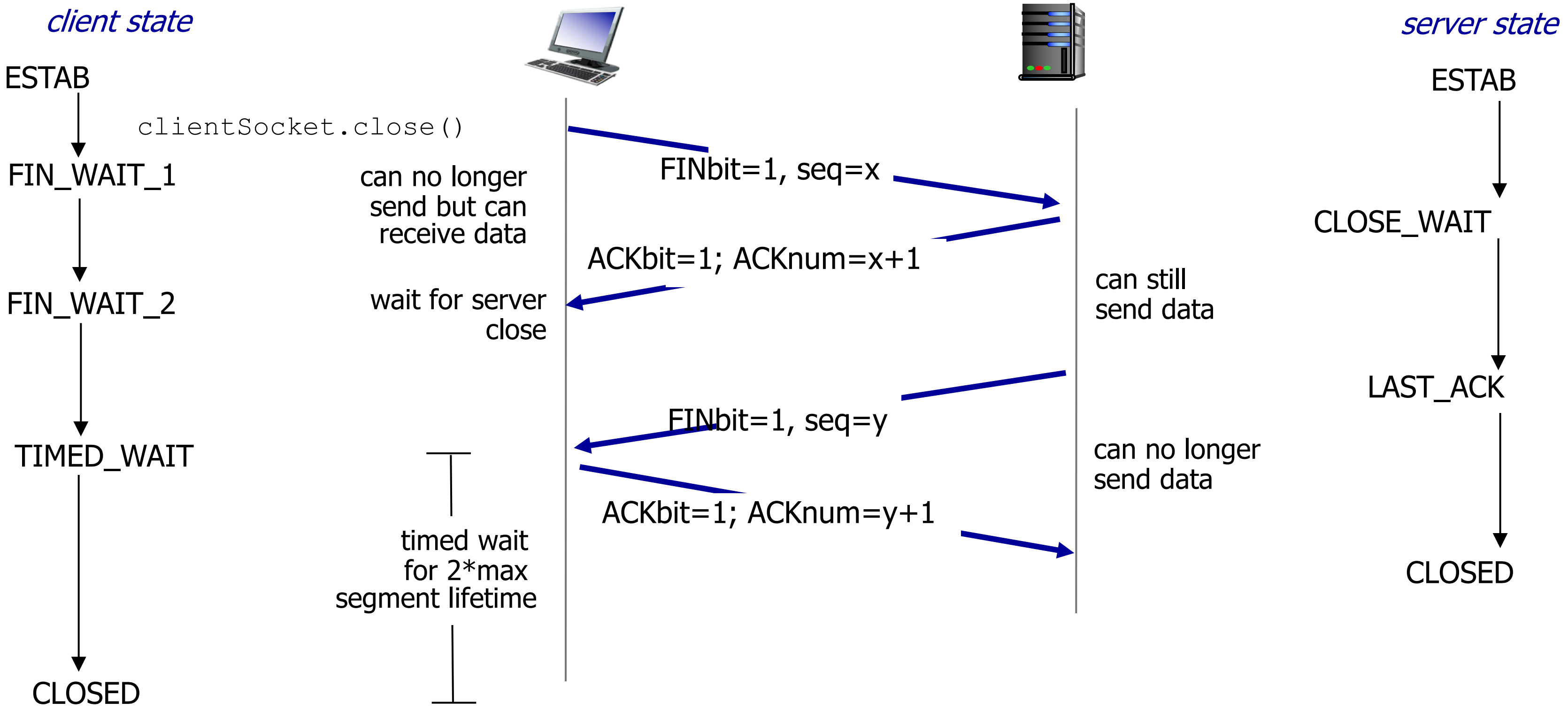
- Very common denial-of-service attack, aka Neptune
- Attacker starts handshake with SYN-marked segment
- Victim replies with SYN-ACK segment
- Attacker… stays silent
  - Note that the source IP of the attacker can be spoofed, since no final ACK is required
- A host can keep a limited number of TCP connections in half-open state. After that limit, it cannot accept any more connections

TCP: Closing A Connection

- Client, server each close their side of connection
  - Send TCP segment with FIN bit = 1
- Respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN



TCP: Closing A Connection



Principles of Congestion Control

- Congestion:
  - Informally: “too many sources sending too much data too fast for network to handle”
  - Different from flow control
  - Manifestations:
    - Lost packets (buffer overflow at routers)
    - Long delays (queueing in router buffers)

Recap Where We're At

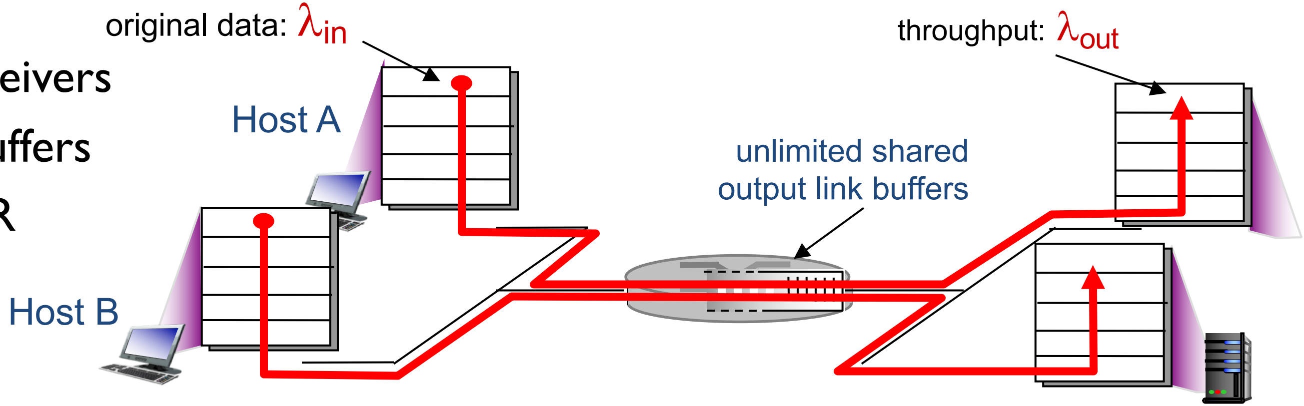
- TCP
  - Retransmission
  - Flow Control
  - Connection Management
  - Congestion Control

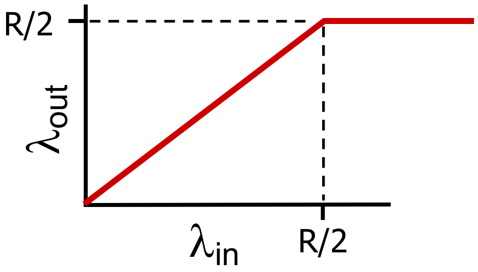
Outline

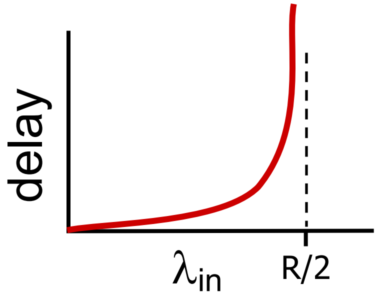
- TCP
  - Congestion Control

Causes/Costs of Congestion: Scenario 1

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





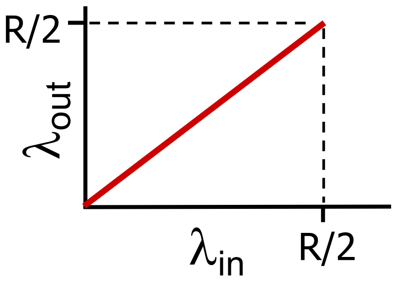


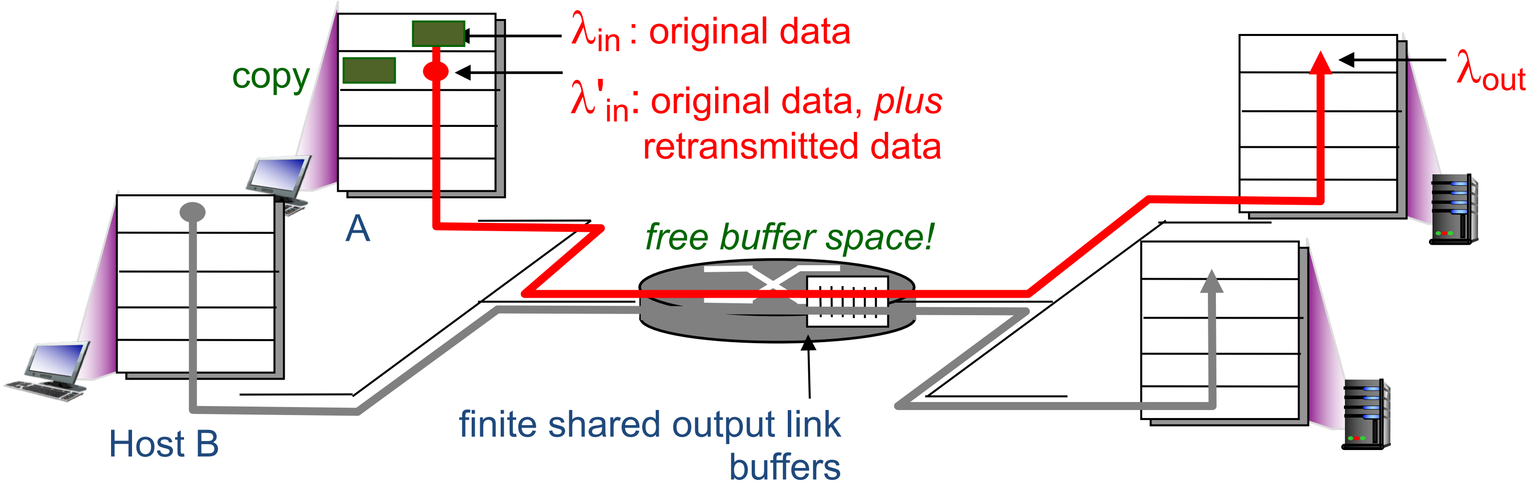
- maximum per-connection throughput: R/2

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

Causes/Costs of Congestion: Scenario 2

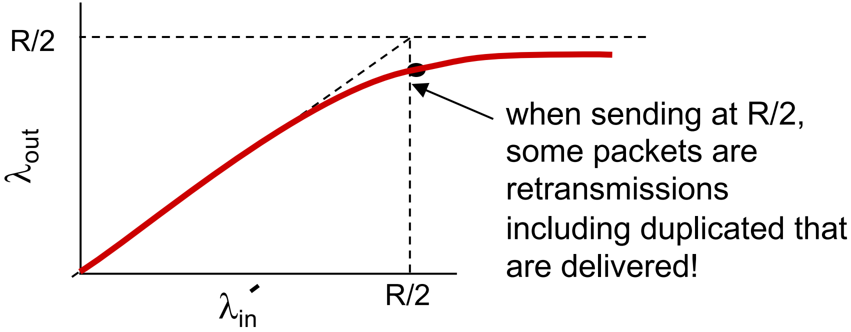
- Idealization: perfect knowledge
  - Sender sends only when router buffers available

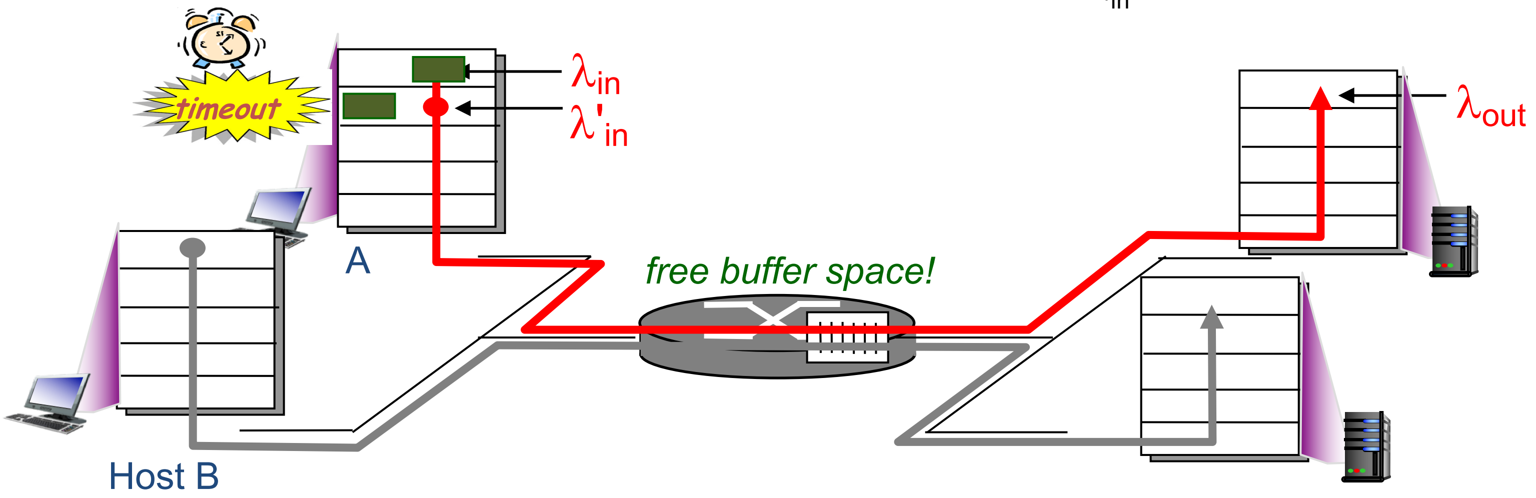




Causes/Costs of Congestion: Scenario 2

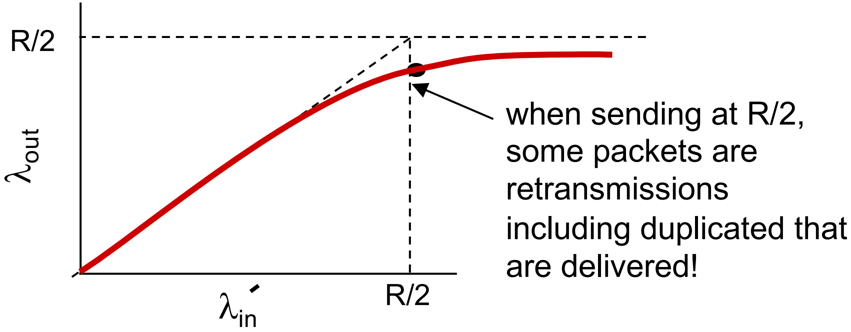
- Realistic: duplicates
  - Packets can be lost, dropped at router due to full buffers
  - Sender times out prematurely, sending two copies, both of which are delivered





Causes/Costs of Congestion: Scenario 2

- Realistic: duplicates
  - Packets can be lost, dropped at router due to full buffers
  - Sender times out prematurely, sending two copies, both of which are delivered

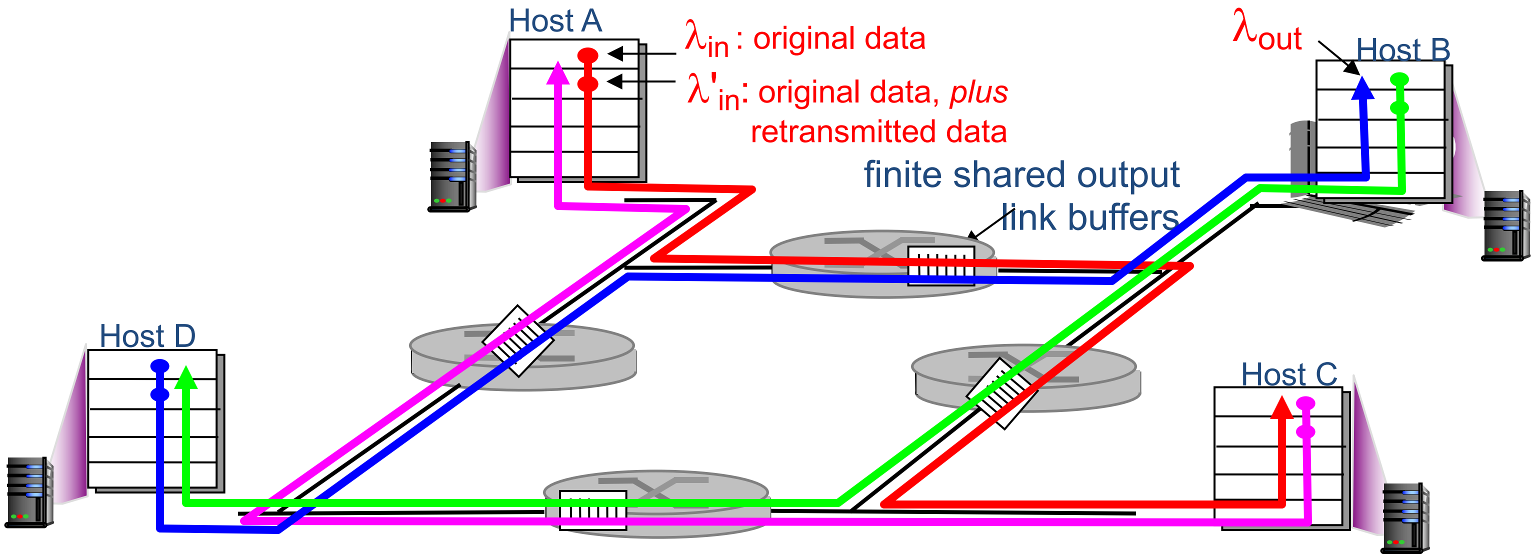


- "costs" of congestion:
  - More work (retrans) for given "throughput"
  - Unneeded retransmissions: link carries multiple copies of packages
    - Decreasing throughput

Causes/Costs of Congestion: Scenario 3

- Four senders
- Multihop paths
- Timeout/retransmit

- As red λ_{in}' increases, all arriving blue pkts at upper queue are dropped, blue throughput → 0



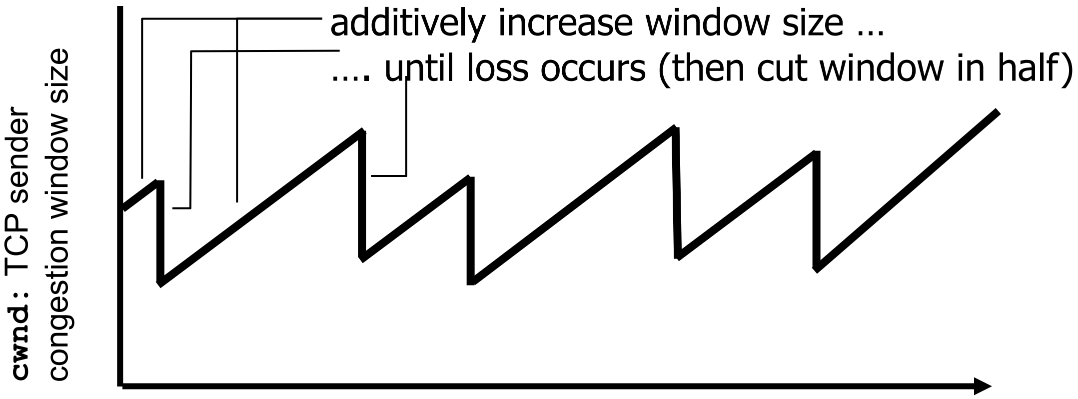
Approaches Towards Congestion Control

- End-end congestion control:
  - No explicit feedback from network
  - Congestion inferred from end-system observed loss, delay
  - Approach taken by TCP
- Network-assisted congestion control
  - Routers provide feedback to end systems
  - Single bit indicating congestion
  - Explicit rate for sender to send at

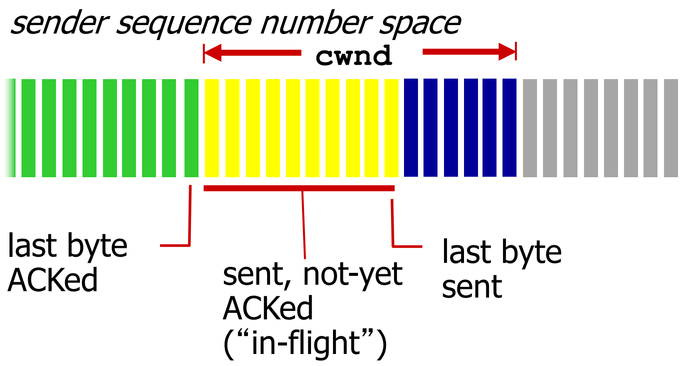
TCP: Additive Increase Multiplicative Decrease

- Approach: sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - Additive increase: increase cwnd by 1 maximum segment size (MSS) every RTT until loss detected
  - Multiplicative decrease: cut cwnd in half after loss

AIMD saw tooth behavior: probing for bandwidth



TCP Congestion Control



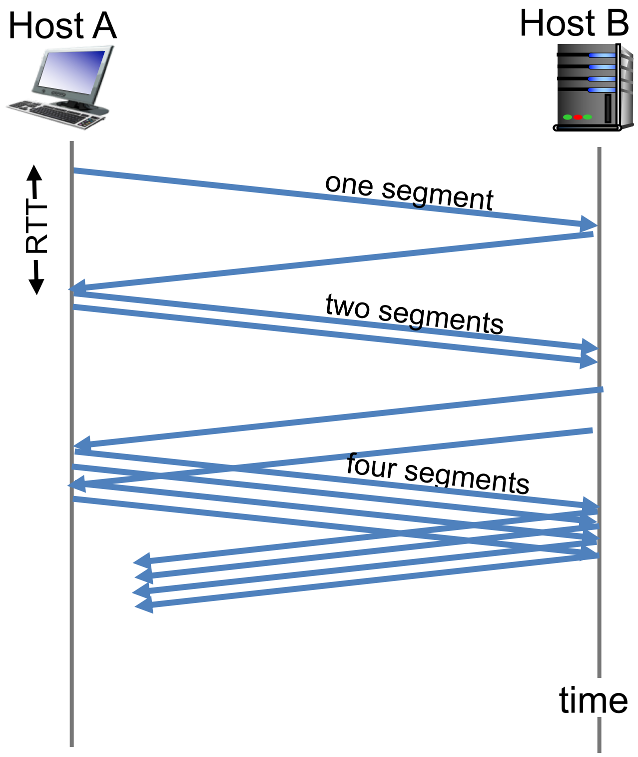
- Sender limits transmission:

$$\boxed{\texttt{LastByteSent-LastByteAcked} \leq \texttt{cwnd}}$$

- cwnd is dynamic, function of perceived network congestion

TCP Slow Start

- When connection begins, increase rate exponentially until first loss event:
  - initially cwnd = 1 MSS
  - double cwnd every RTT
  - done by incrementing cwnd for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast

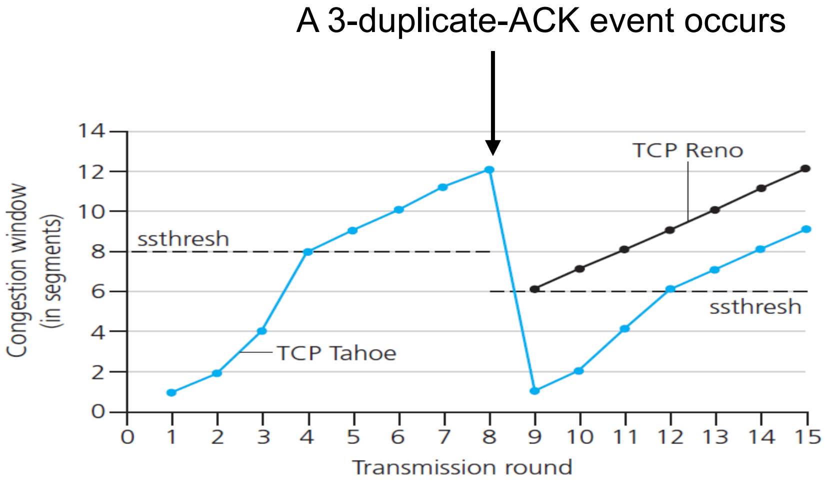


TCP: Detecting, Reacting to Loss

- Loss indicated by timeout:
  - cwnd set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
- Loss indicated by 3 duplicate ACKs: TCP RENO
  - duplicate ACKs indicate network capable of delivering some segments
  - cwnd is cut in half window then grows linearly
- TCP Tahoe always sets cwnd to 1 (timeout or 3 duplicate acks)

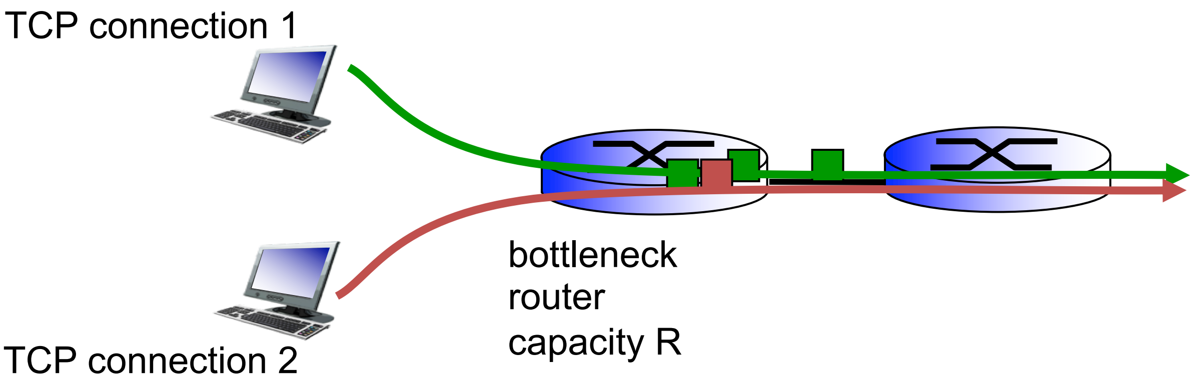
TCP: Switching from Slow Start to Congestion Avoidance

- When should the exponential increase switch to linear?
  - When cwnd gets to 1/2 of its value before timeout.
  - ssthresh (shorthand for “slow start threshold”) to cwnd/2



TCP Fairness

- Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP Fair?

- Two competing sessions:
  - Additive increase gives slope of 1, as throughout increases
  - Multiplicative decrease decreases throughput proportionally