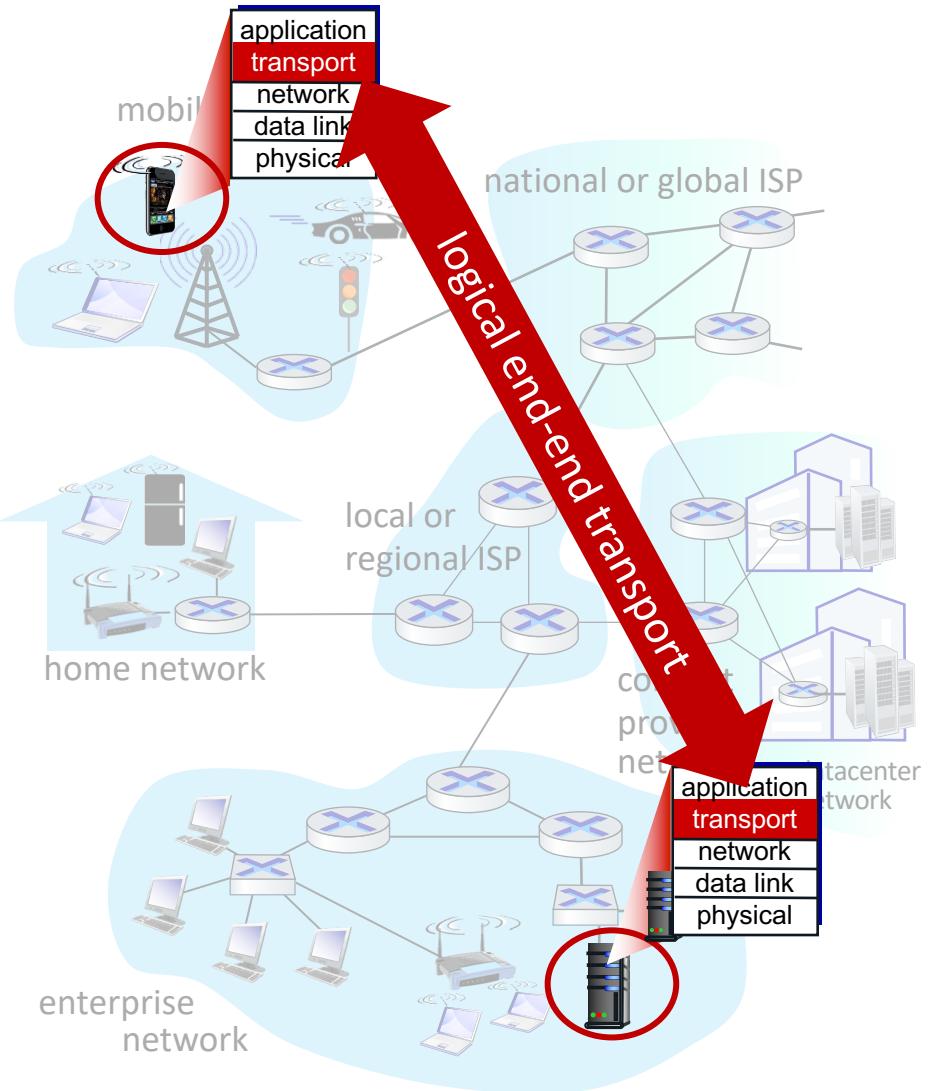


## Transport Layer Introduction



# Transport Layer Services and Protocols

- provide *logical communication* between application processes running on different hosts
- transport protocols actions in end systems:
  - sender: breaks application messages into *segments*, passes to network layer
  - receiver: reassembles segments into messages, passes to application layer
- two transport protocols available to Internet applications
  - TCP, UDP



# Transport Layer vs. Network Layer



© 1922. WM. EVANS.

THERE was an old woman who lived in a shoe,  
She had so many children, she didn't know what to do.  
She gave them some milk and nice butter bread,  
She kissed them all round and put them to bed.

## *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 Layer vs. Network Layer

- network layer: logical communication between *hosts*
- transport layer: logical communication between *processes*
  - relies on, enhances, network layer services

*household analogy:*

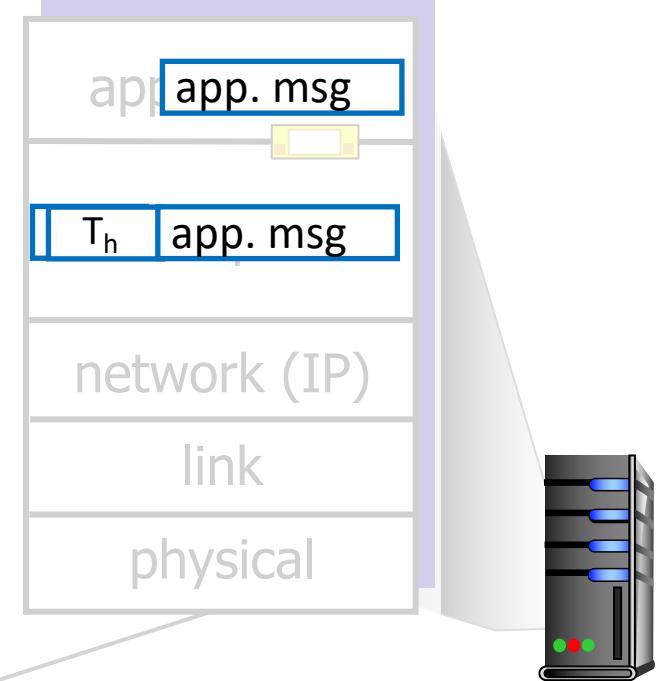
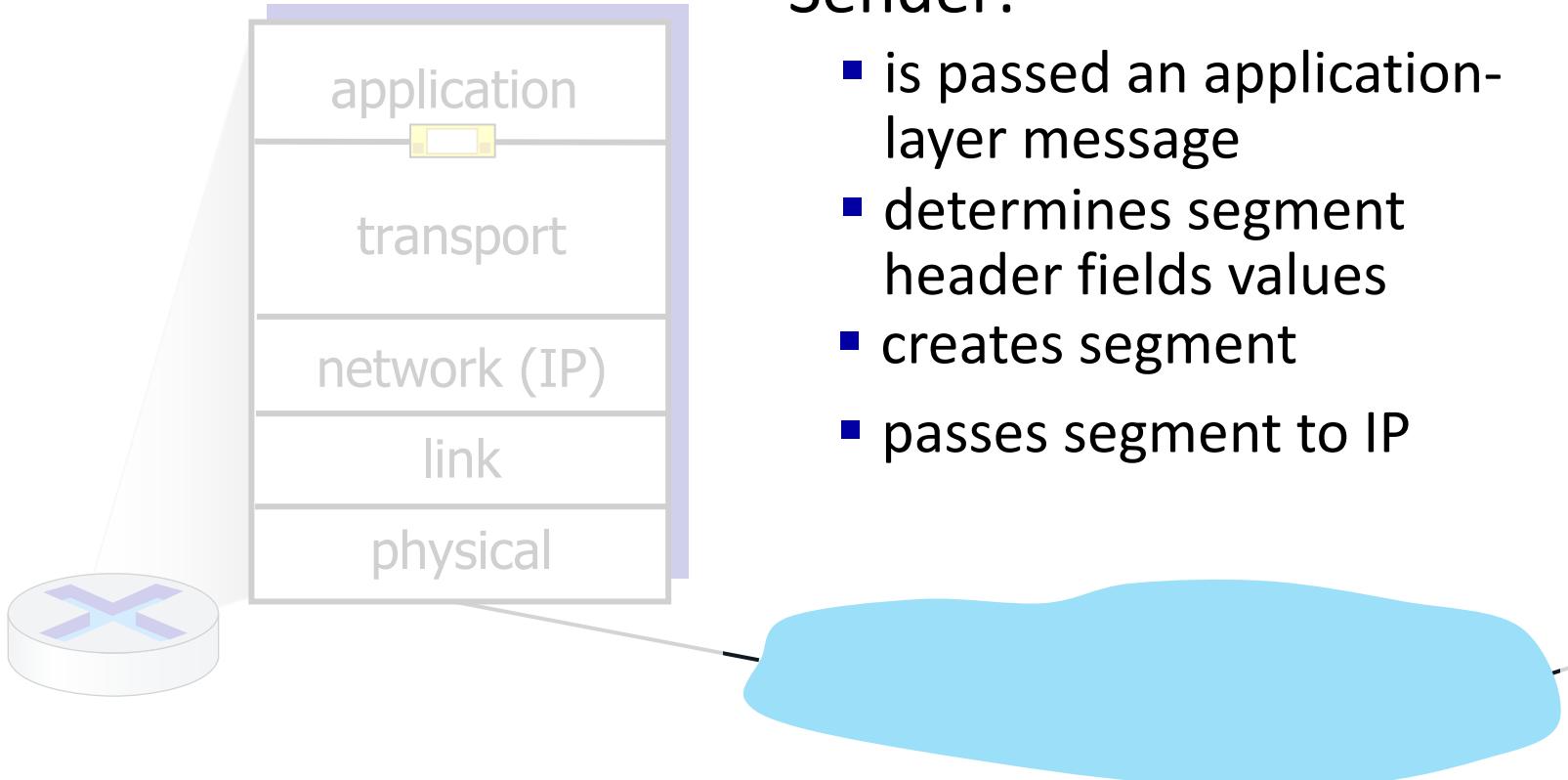
*12 kids in Ann's house sending letters to 12 kids in Bill's house:*

- hosts = houses
- processes = kids
- app messages = letters in envelopes

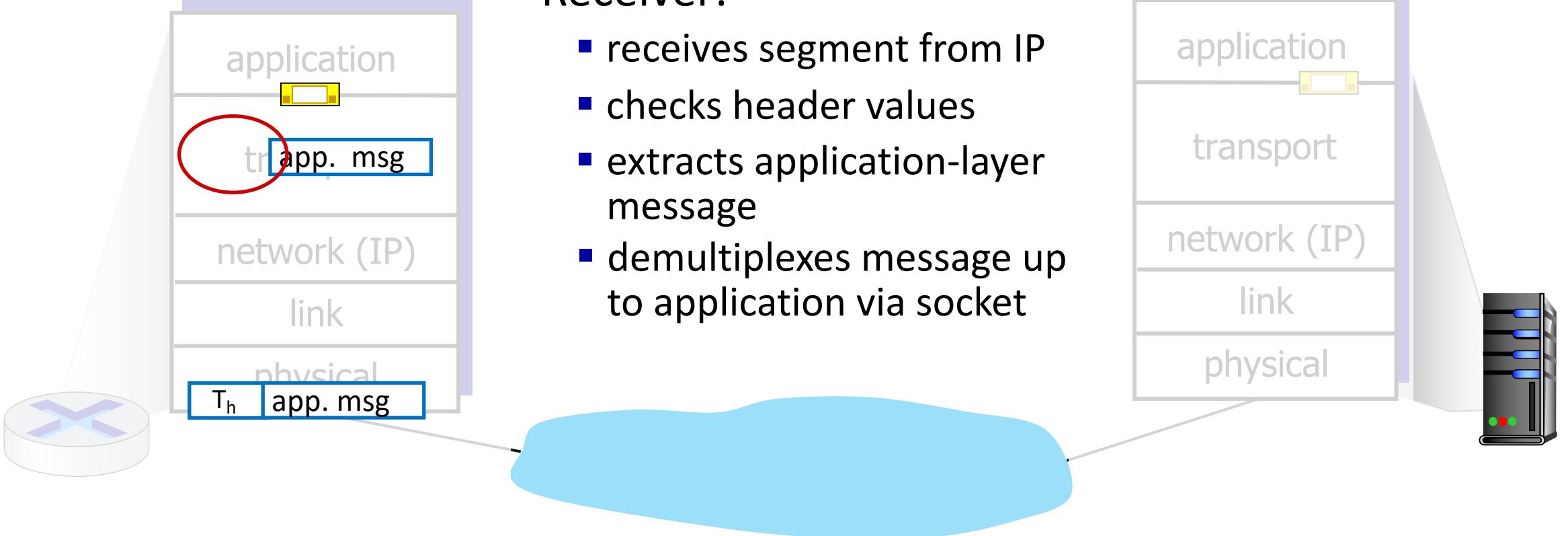
# Transport Layer Action

## Sender:

- is passed an application-layer message
- determines segment header fields values
- creates segment
- passes segment to IP

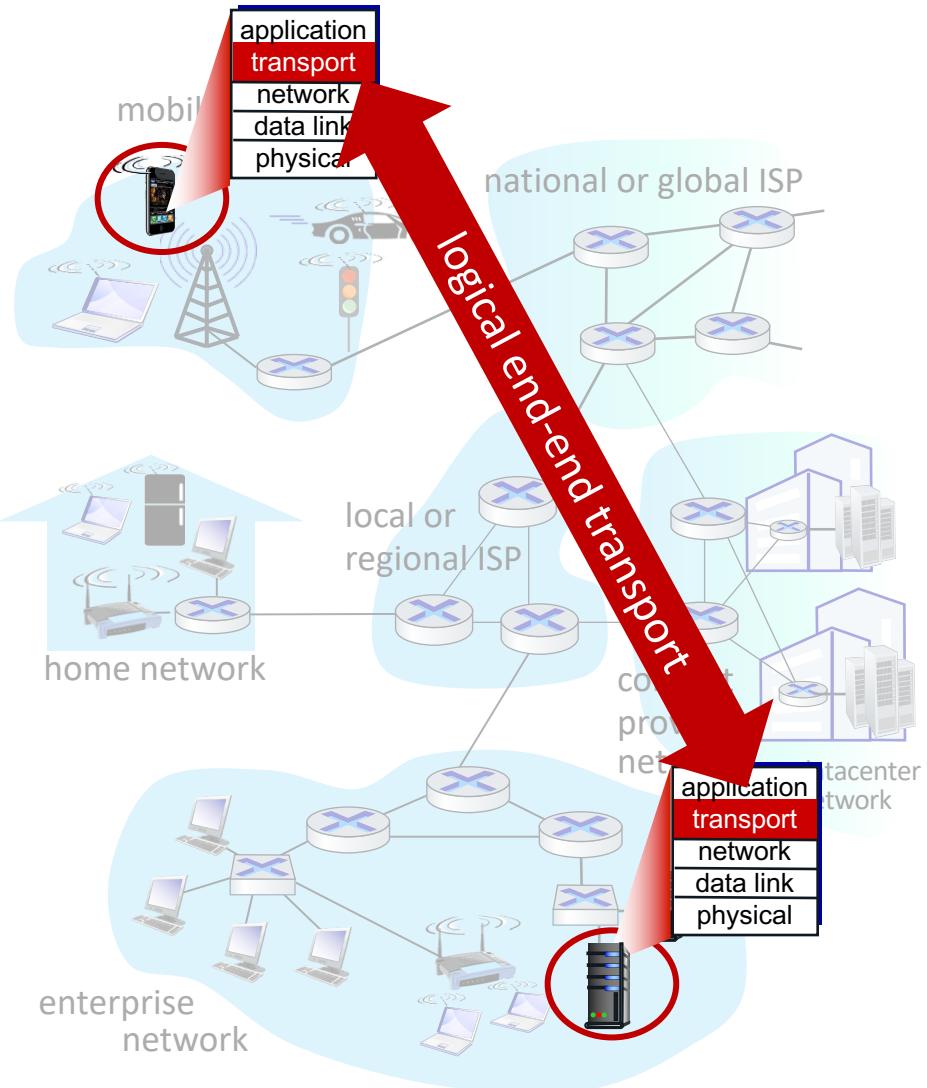


# Transport Layer Action

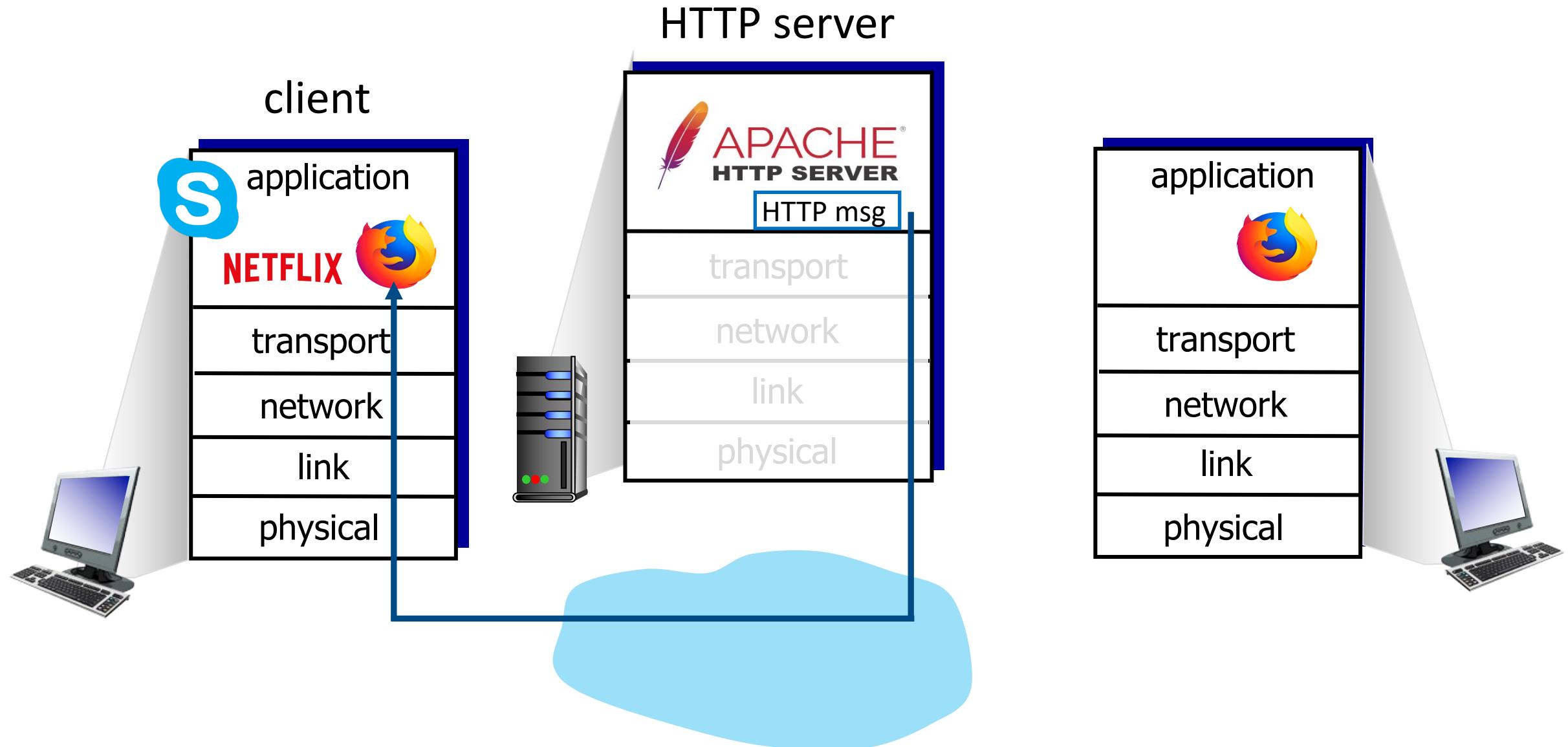


# Two Principal Internet Transport Protocols

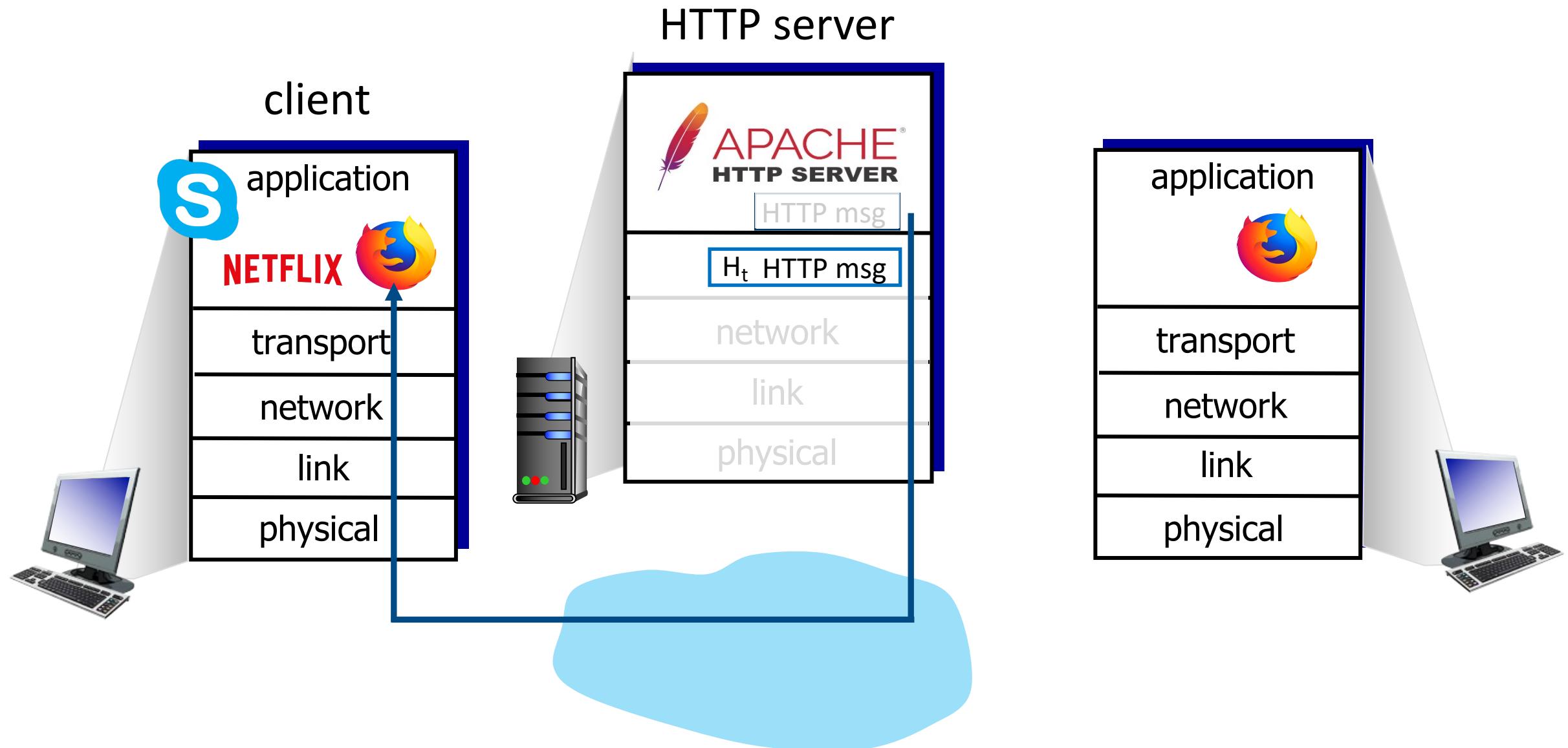
- **TCP:** Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



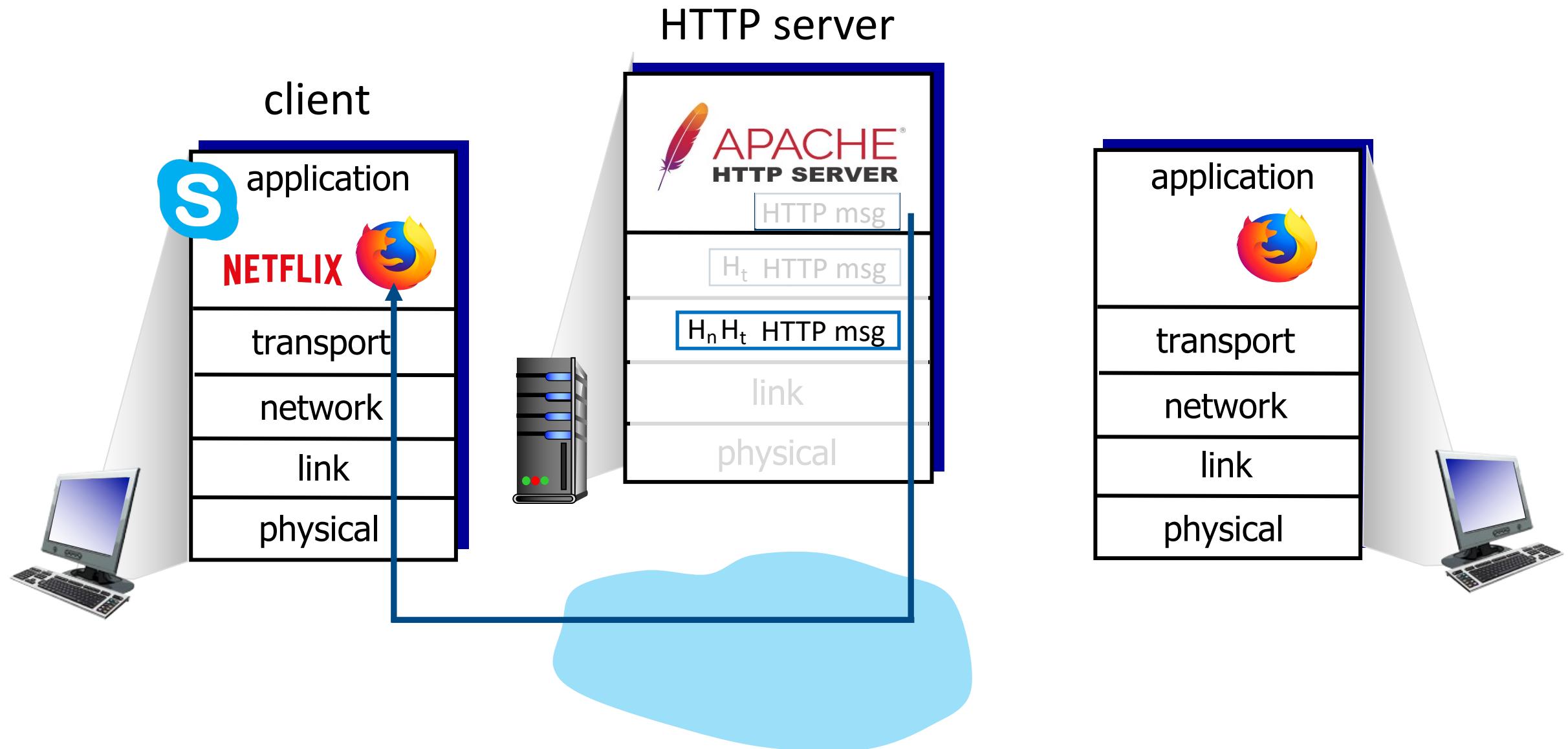
# Multiplexing and Demultiplexing Motivation



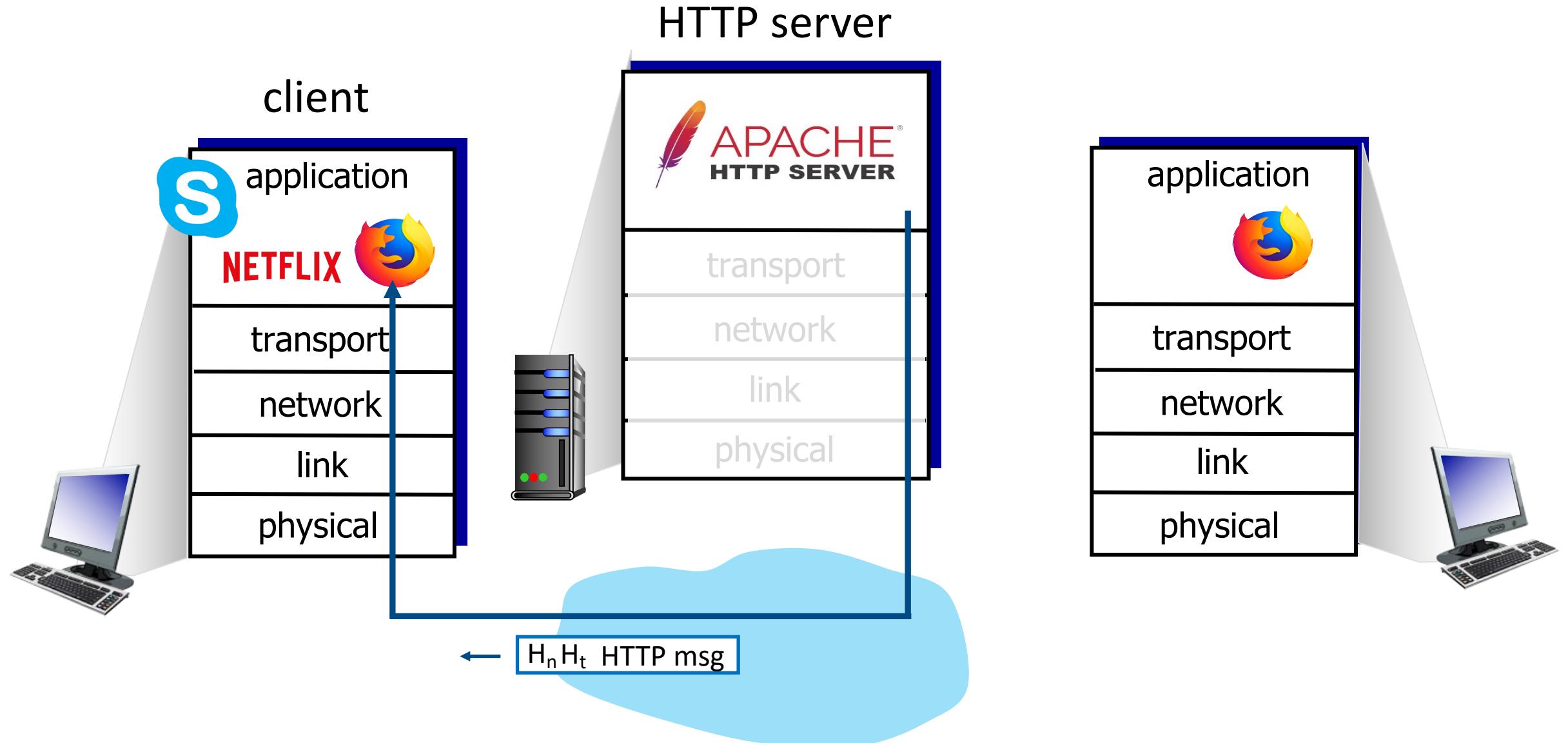
# Multiplexing and Demultiplexing Motivation



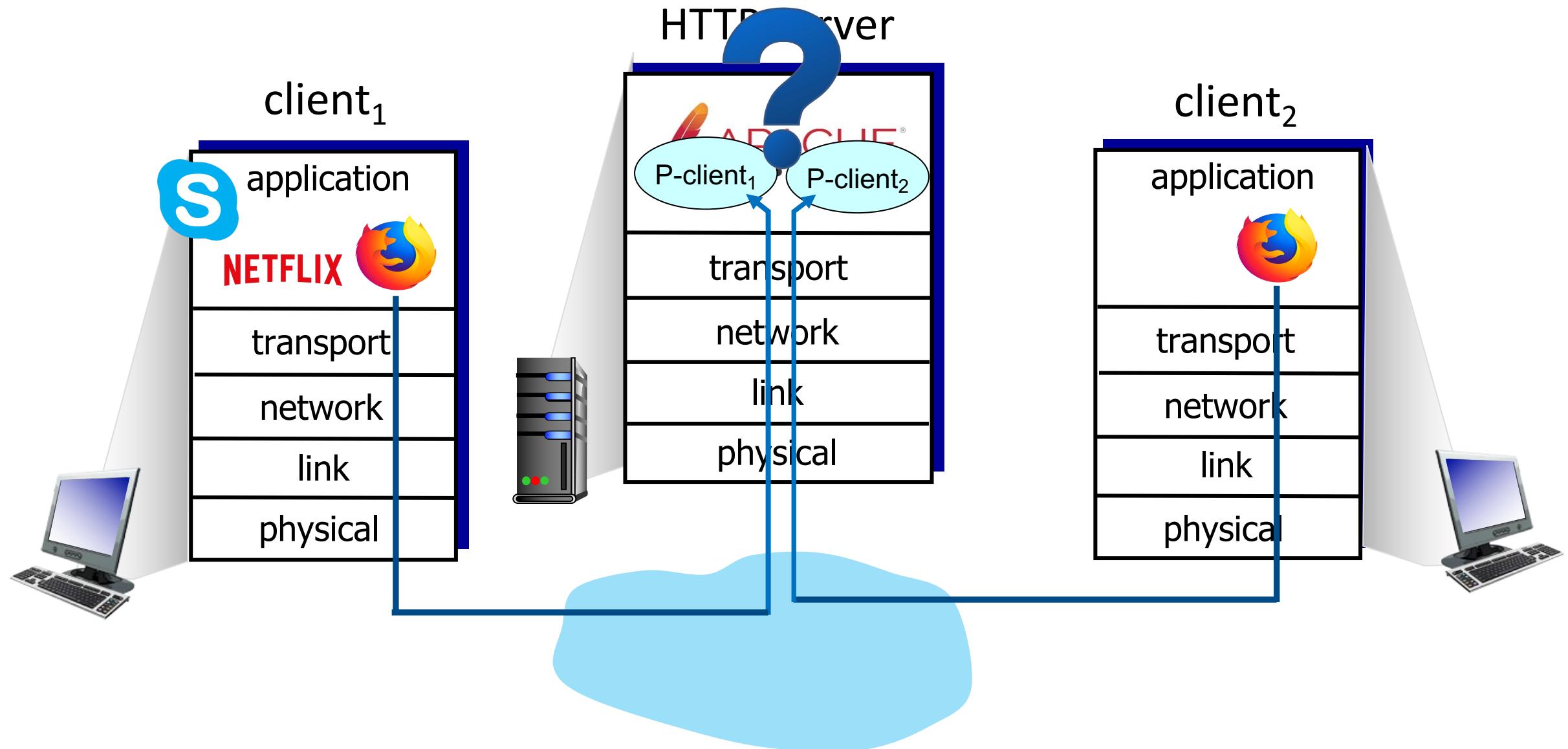
# Multiplexing and Demultiplexing Motivation



# Multiplexing and Demultiplexing Motivation



# Multiplexing and Demultiplexing Motivation



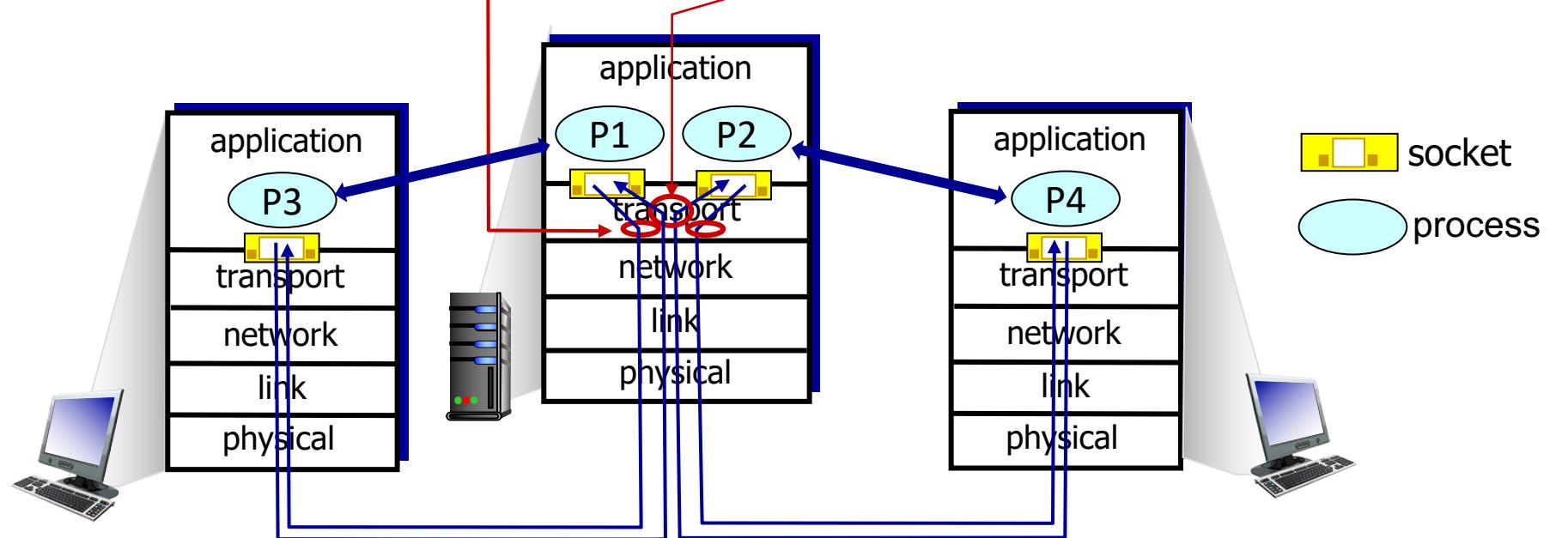
# Multiplexing and Demultiplexing

*multiplexing at sender:*

handle data from multiple sockets, add transport header (later used for demultiplexing)

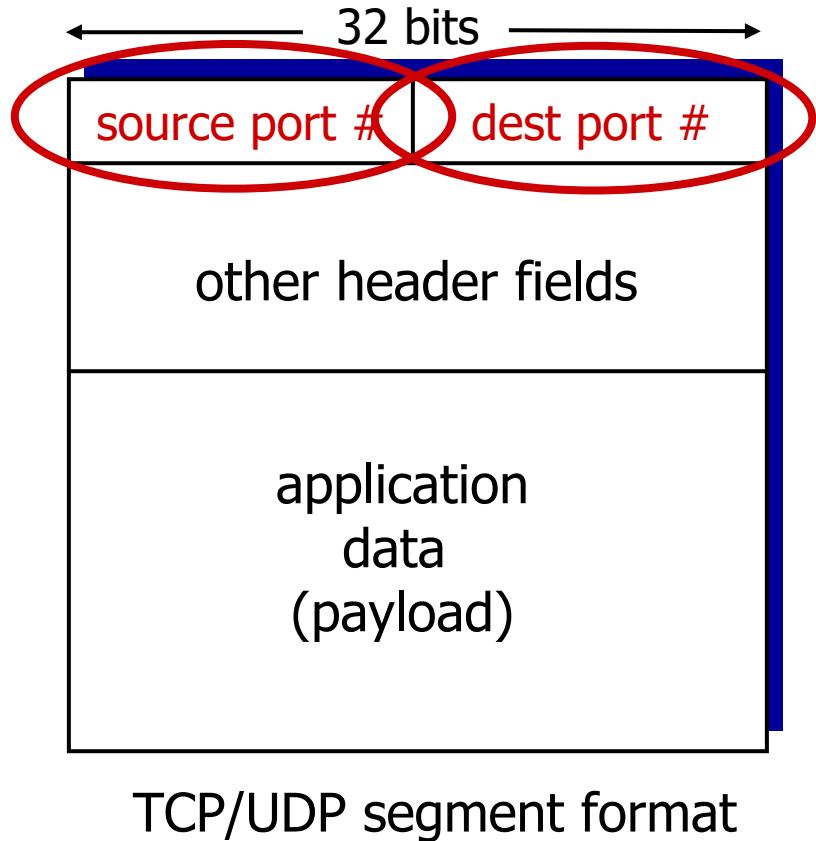
*demultiplexing at receiver:*

use header info to deliver received segments to correct socket



# How Demultiplexing Works

- host receives IP datagrams
  - each datagram has source IP address, 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



# Connectionless Demultiplexing

*Recall:*

- when creating socket, must specify *host-local* port #:
- when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

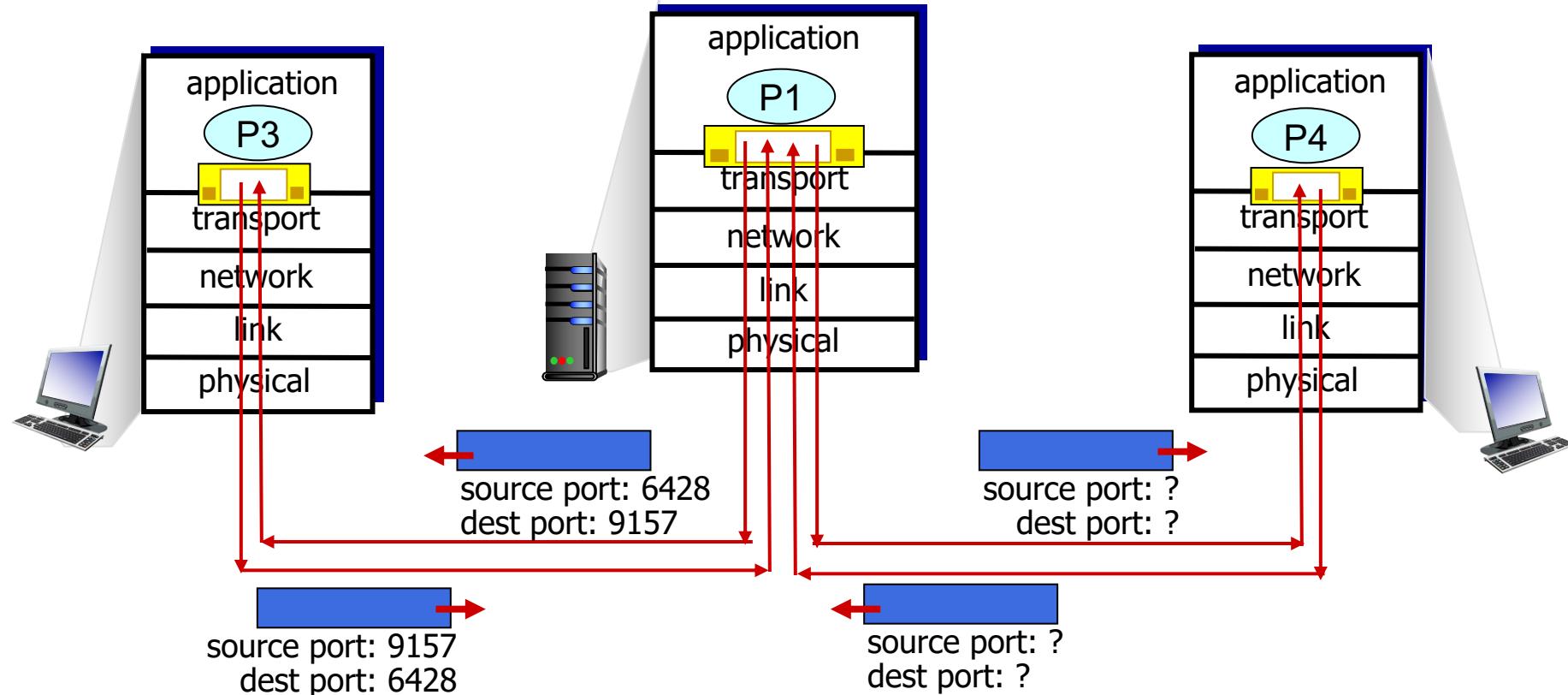
when receiving host receives *UDP* segment:

- checks destination port # in segment
- directs UDP segment to socket with that port #



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

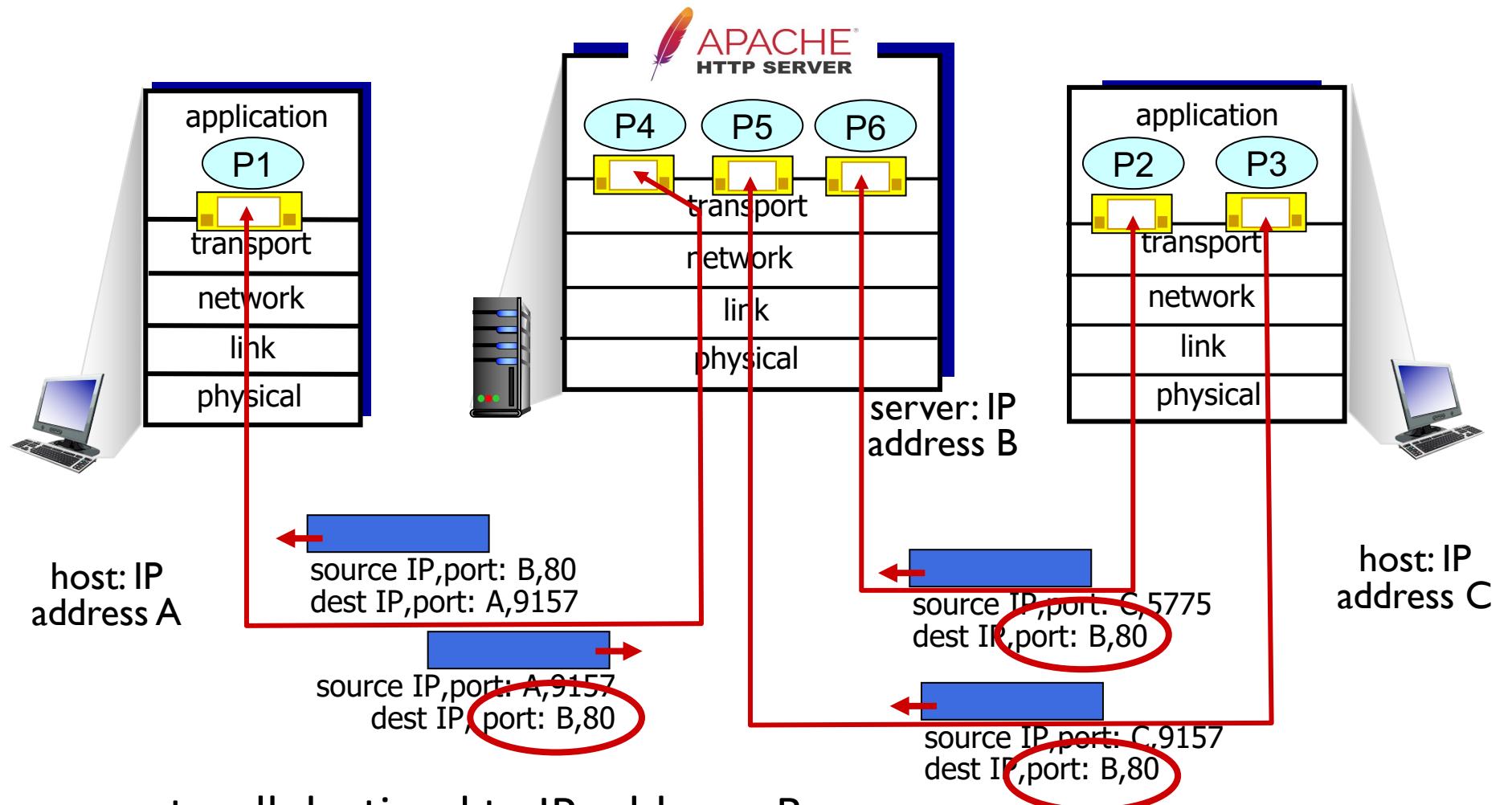
# Connectionless Demultiplexing: Example



# Connection-Oriented Demultiplexing

- TCP socket identified by **4-tuple**:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses *all four values (4-tuple)* to direct segment to appropriate socket
- server may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
  - each socket associated with a different connecting client

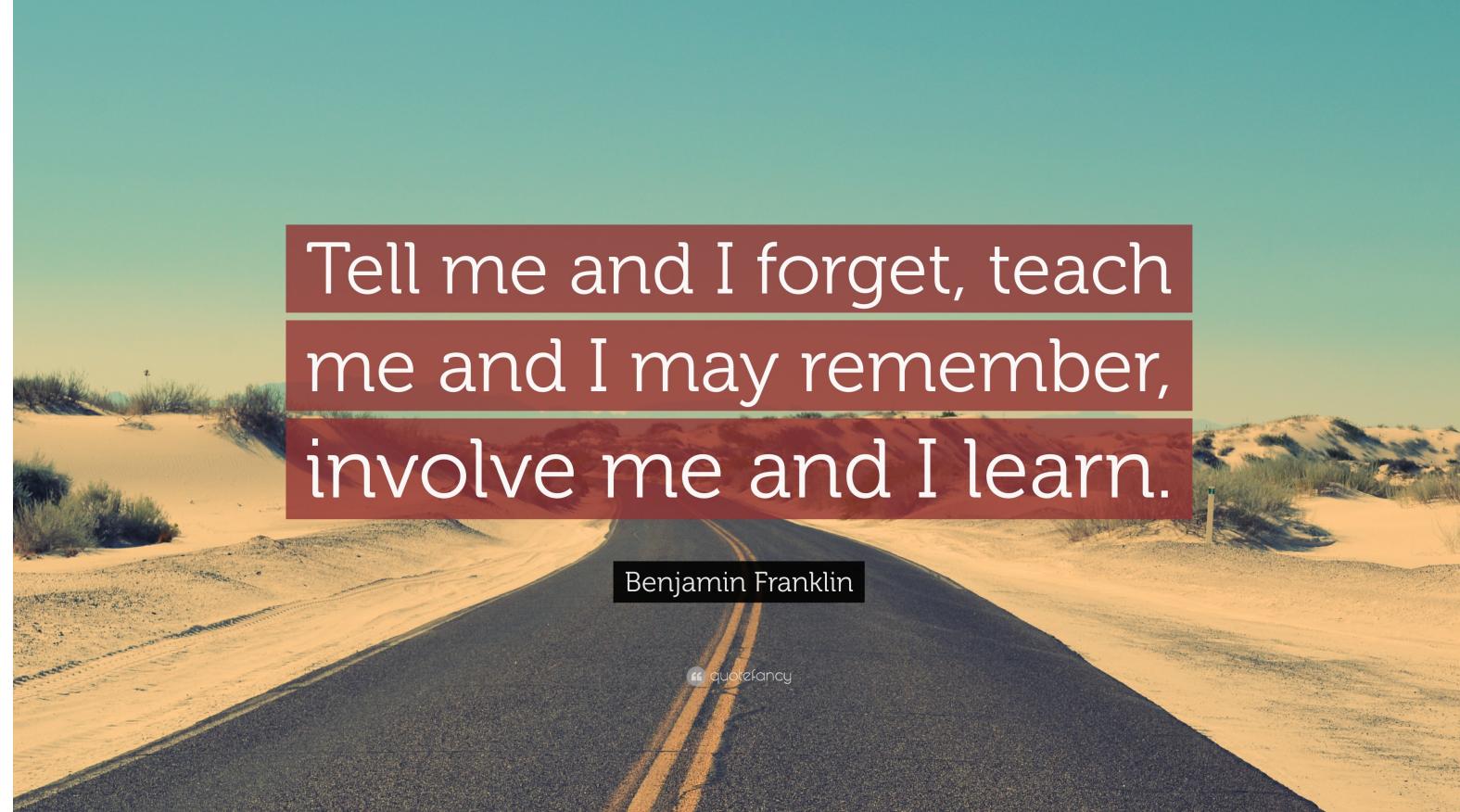
# Connection-Oriented Demultiplexing: Example



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

# Multiplexing and Demultiplexing Summary

- Multiplexing, demultiplexing: based on segment, datagram header field values
- **UDP:** demultiplexing using destination port number (only)
- **TCP:** demultiplexing using 4-tuple: source and destination IP addresses, and port numbers
- Multiplexing/demultiplexing happen at *all* layers



Tell me and I forget, teach  
me and I may remember,  
involve me and I learn.

Benjamin Franklin

" quotefancy

# User Datagram Protocol (UDP)



# UDP: User Datagram Protocol

- “no frills,” “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 (which can add RTT delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control
  - UDP can blast away as fast as desired!
  - can function in the face of congestion

# UDP: User Datagram Protocol

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
  - HTTP/3
- if reliable transfer needed over UDP (e.g., HTTP/3):
  - add needed reliability at application layer
  - add congestion control at application layer

# UDP: User Datagram Protocol (RFC 768)

INTERNET STANDARD  
RFC 768 J. Postel  
ISI  
28 August 1980

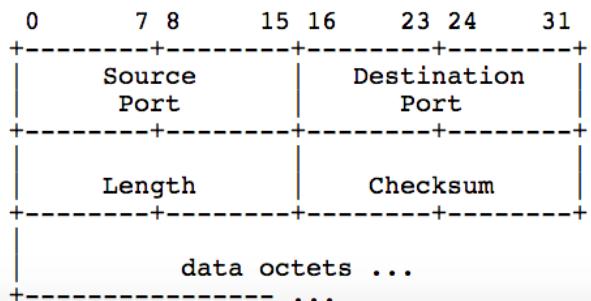
## User Datagram Protocol

### Introduction

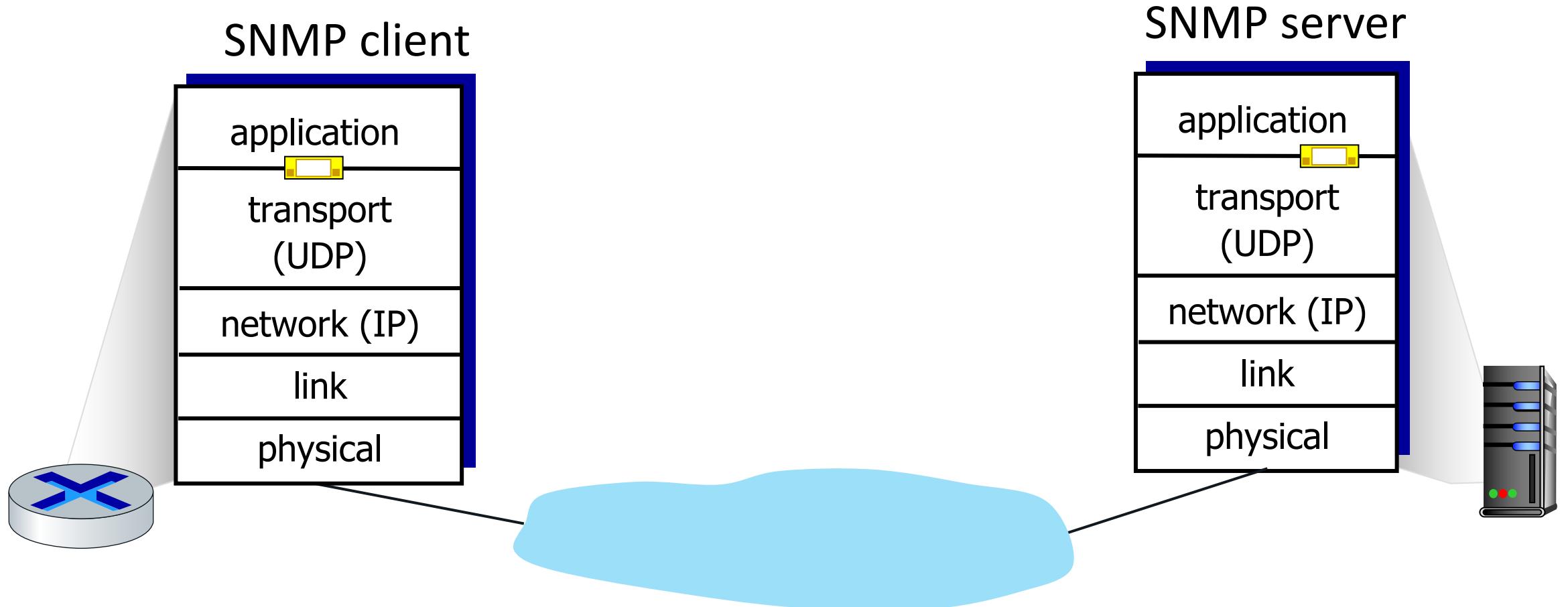
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

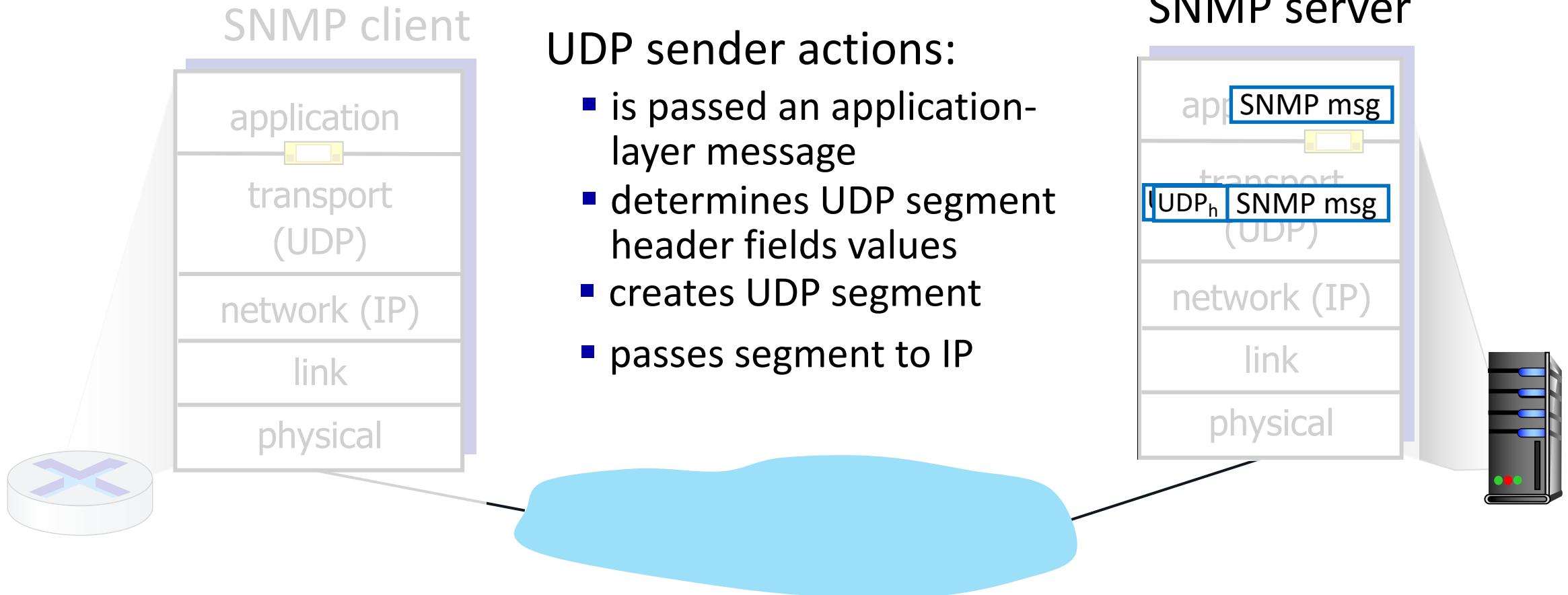
### Format



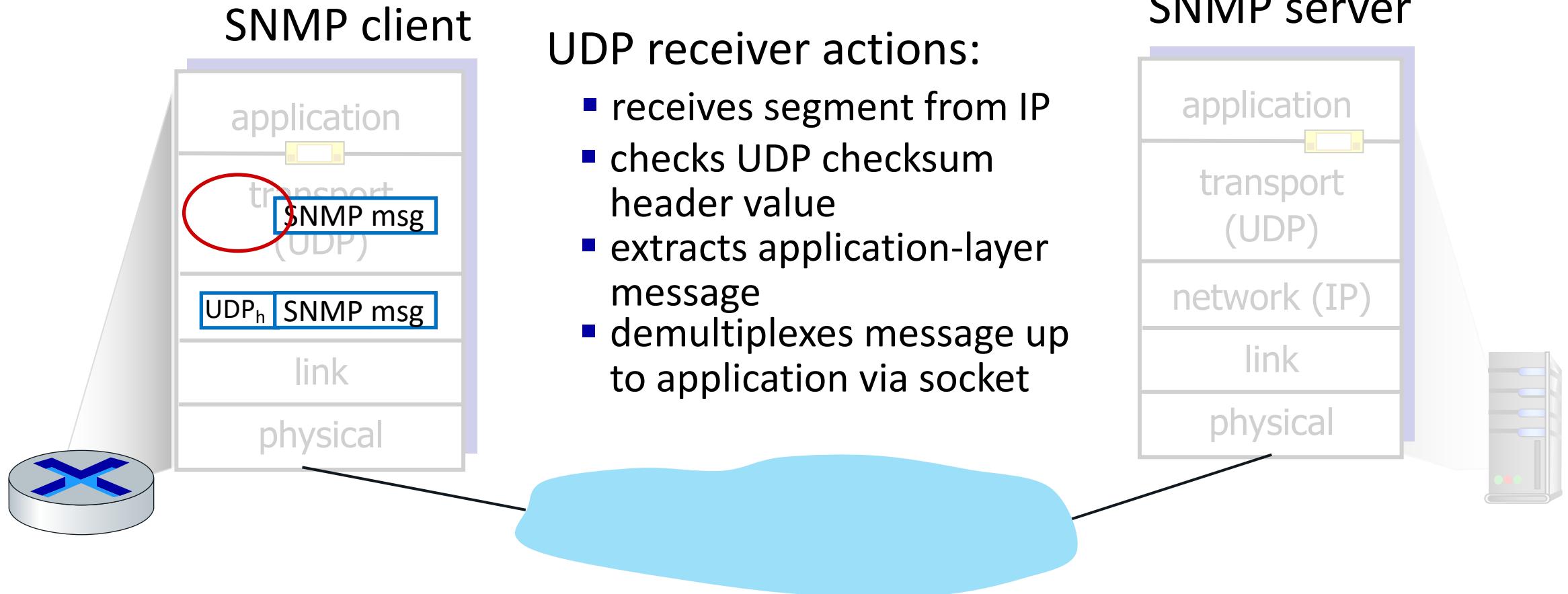
# UDP: Transport Layer Actions



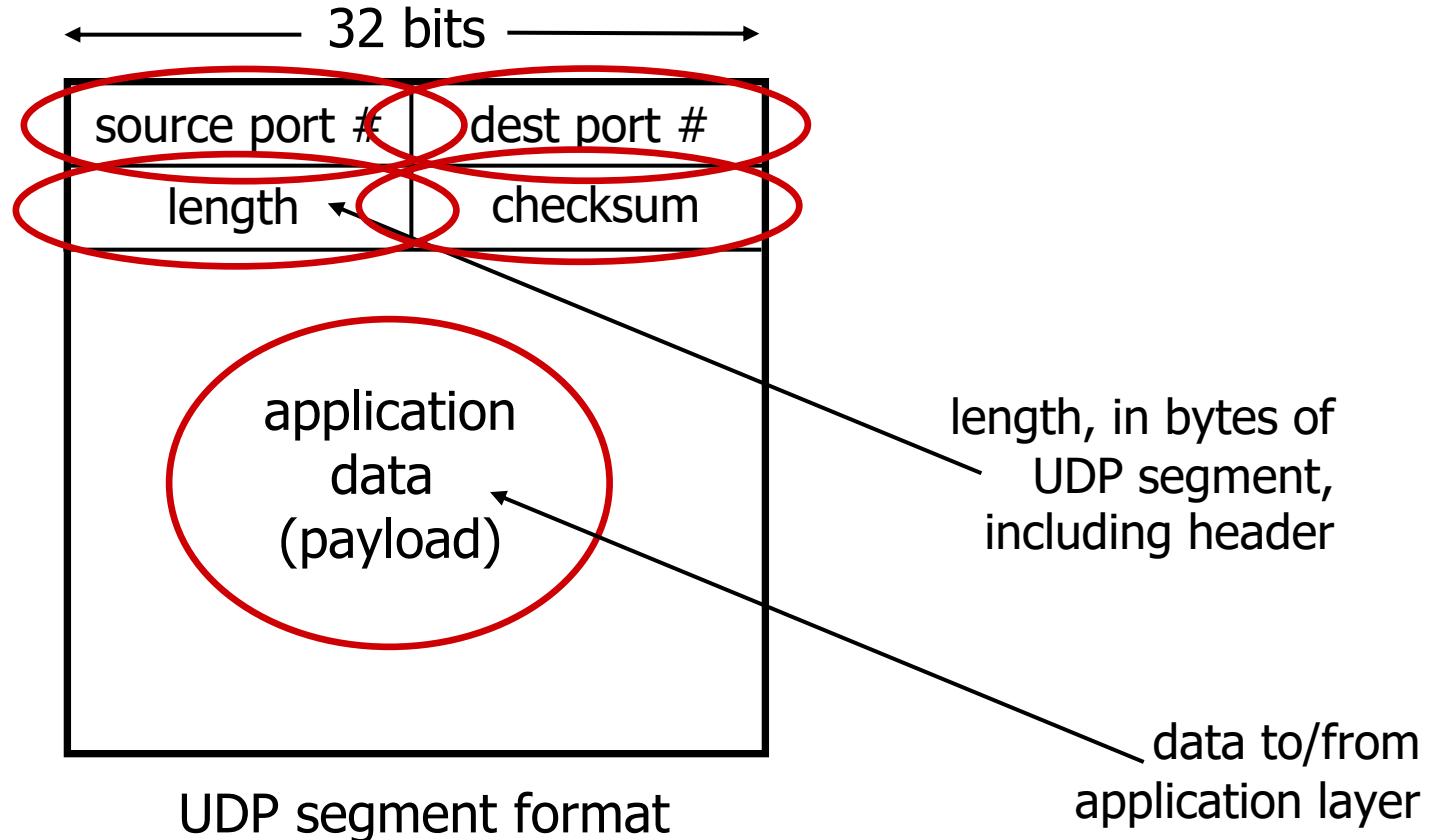
# UDP: Transport Layer Actions



# UDP: Transport Layer Actions



# UDP Segment Header



**Goal:** detect errors (*i.e.*, flipped bits) in transmitted segment

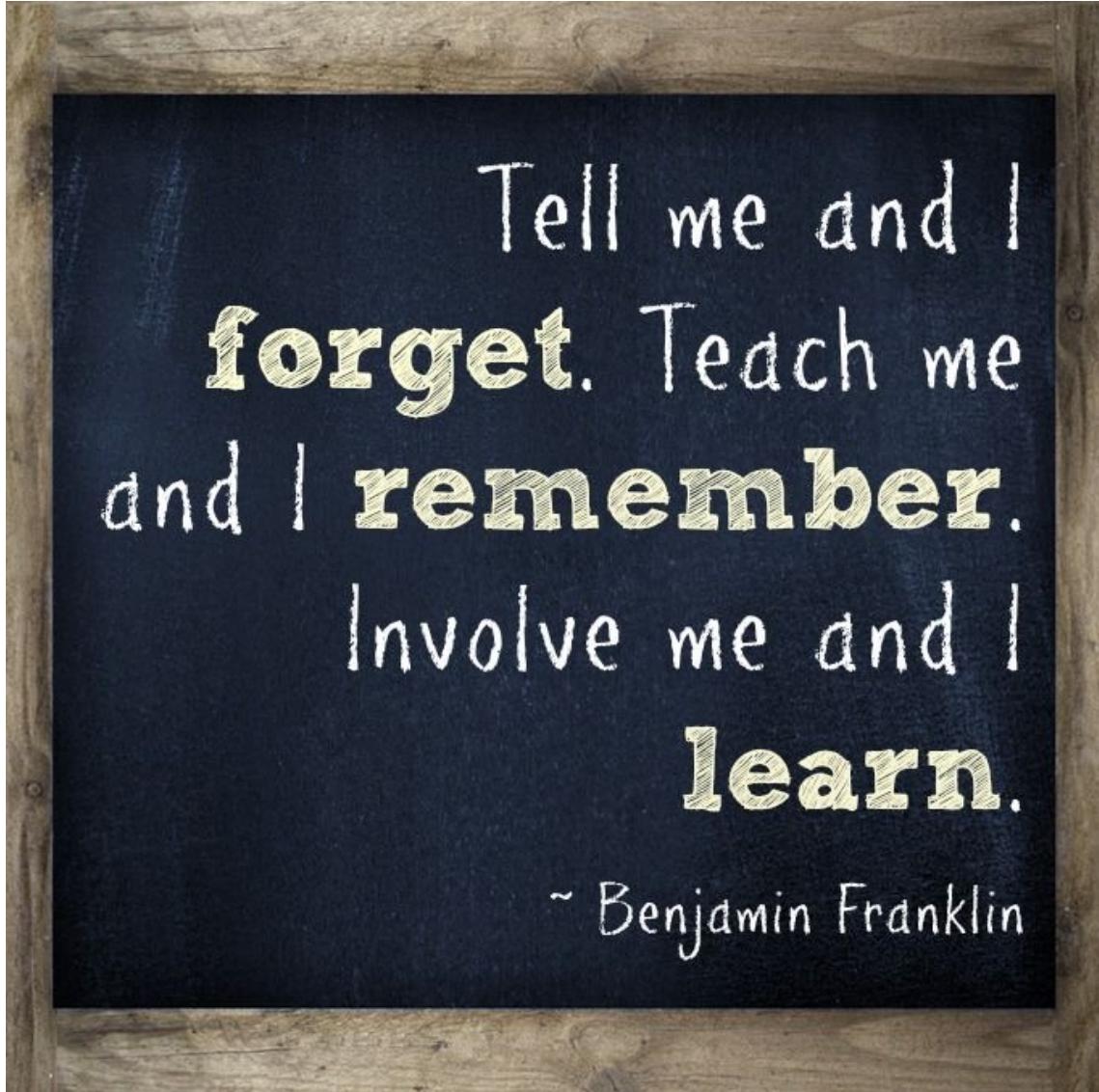
## sender:

- treat contents of UDP segment (including UDP header fields and IP addresses) as sequence of 16-bit integers
- **checksum:** addition (one's complement sum) of segment content
- checksum value put into UDP checksum field

## receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - Not equal - error detected
  - Equal - no error detected. *But maybe errors nonetheless?* More later ....

- “no frills” protocol:
  - segments may be lost, delivered out of order
  - best effort service: “send and hope for the best”
- UDP has its plusses:
  - no setup/handshaking needed (no RTT incurred)
  - can function when network service is compromised
  - helps with reliability (checksum)
- build additional functionality on top of UDP in application layer (e.g., HTTP/3)



## Principles of Reliable Data Transfer

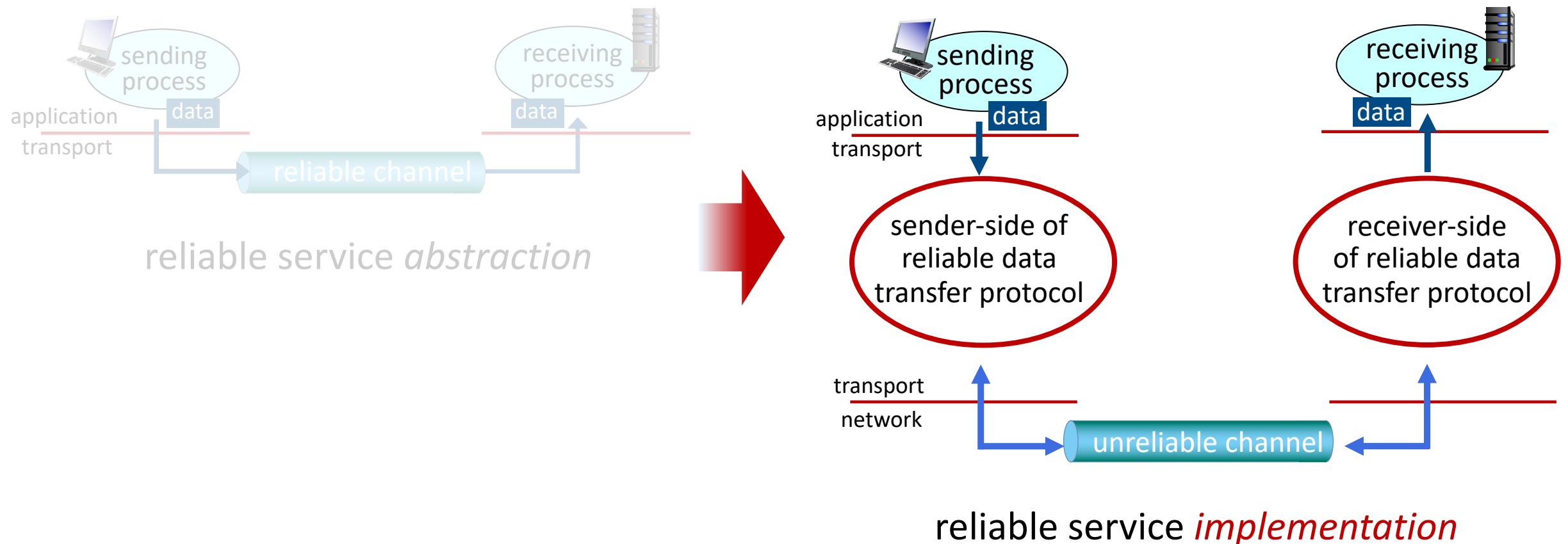


# Principles of Reliable Data Transfer



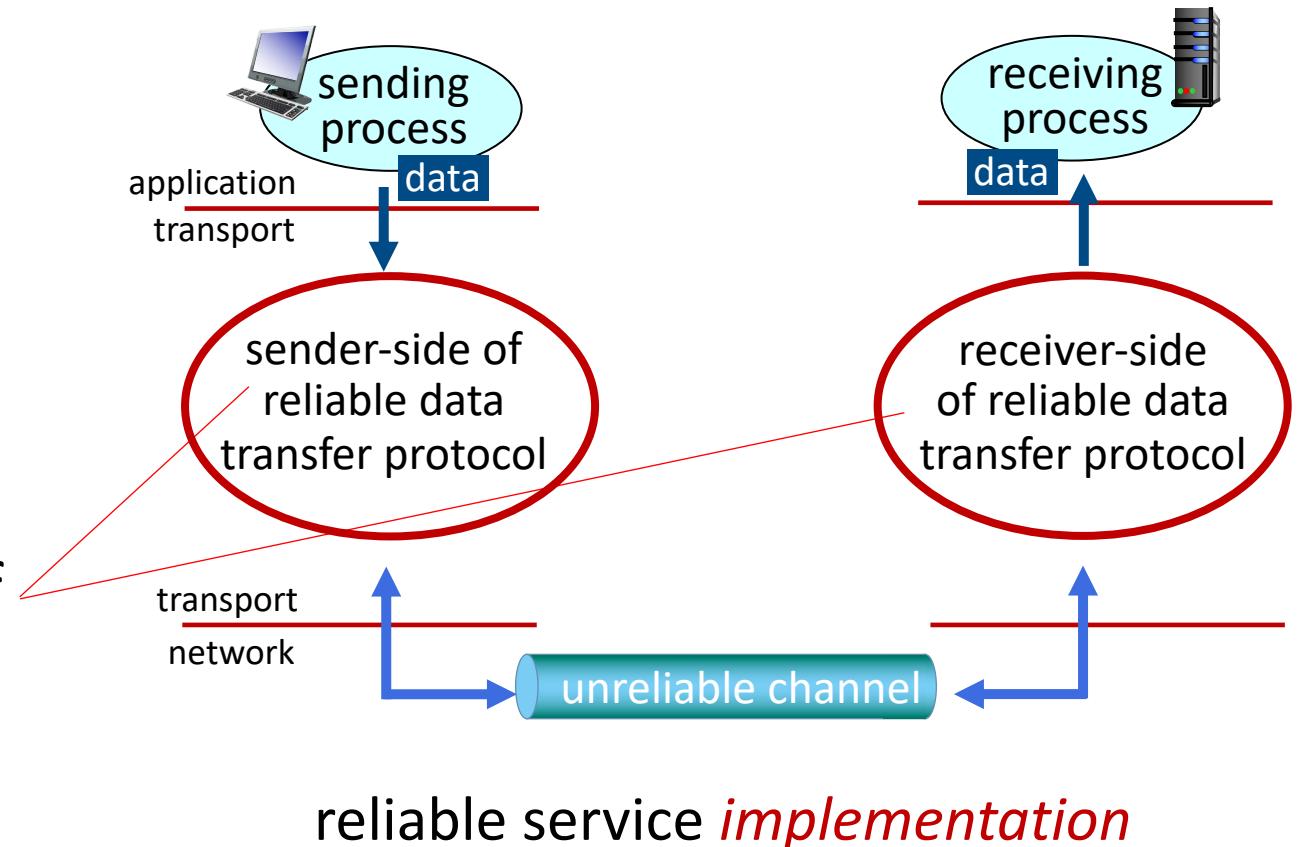
reliable service *abstraction*

# Principles of Reliable Data Transfer



# Principles of Reliable Data Transfer

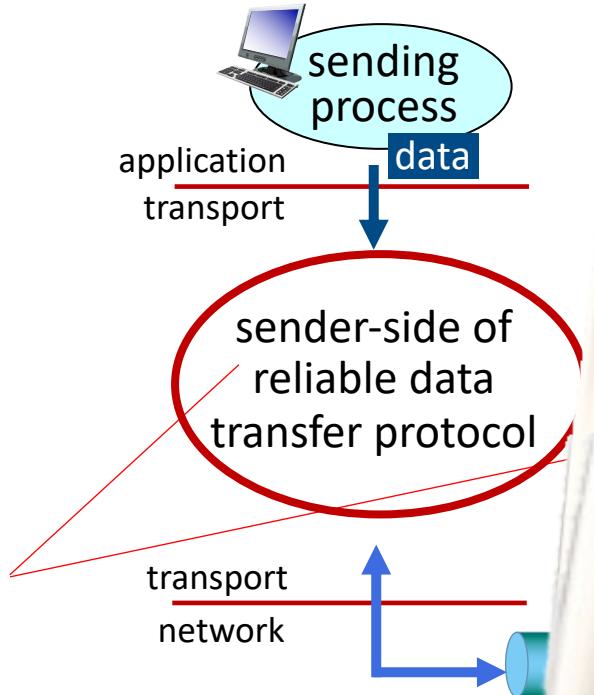
Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)



# Principles of Reliable Data Transfer

Sender, receiver do *not* know the “state” of each other, e.g., was a message received?

- unless communicated via a message



reliable service *implementation*

# Reliable Data Transfer (RDT3.0)

*New channel assumption:* underlying channel can also *lose* packets (data, ACKs)

- checksum, sequence #s, ACKs, retransmissions will be of help ...  
but not quite enough

*Q:* How do *humans* handle lost sender-to-receiver words in conversation?

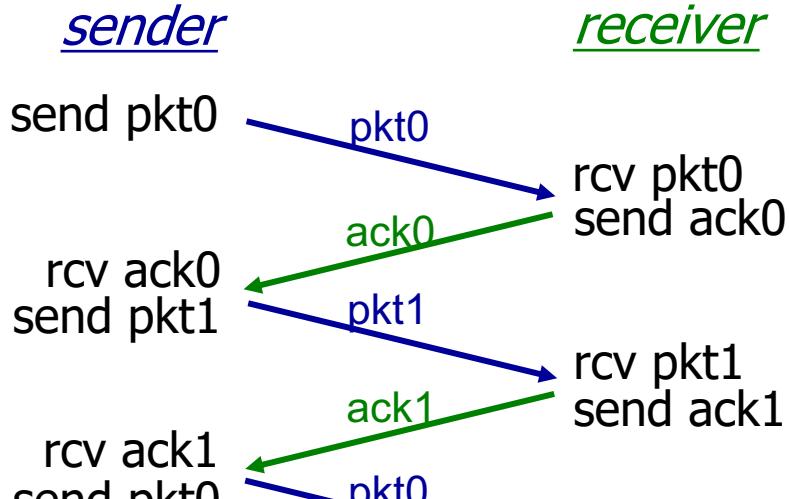
# Reliable Data Transfer (RDT3.0)

*Approach:* sender waits “reasonable” amount of time for ACK

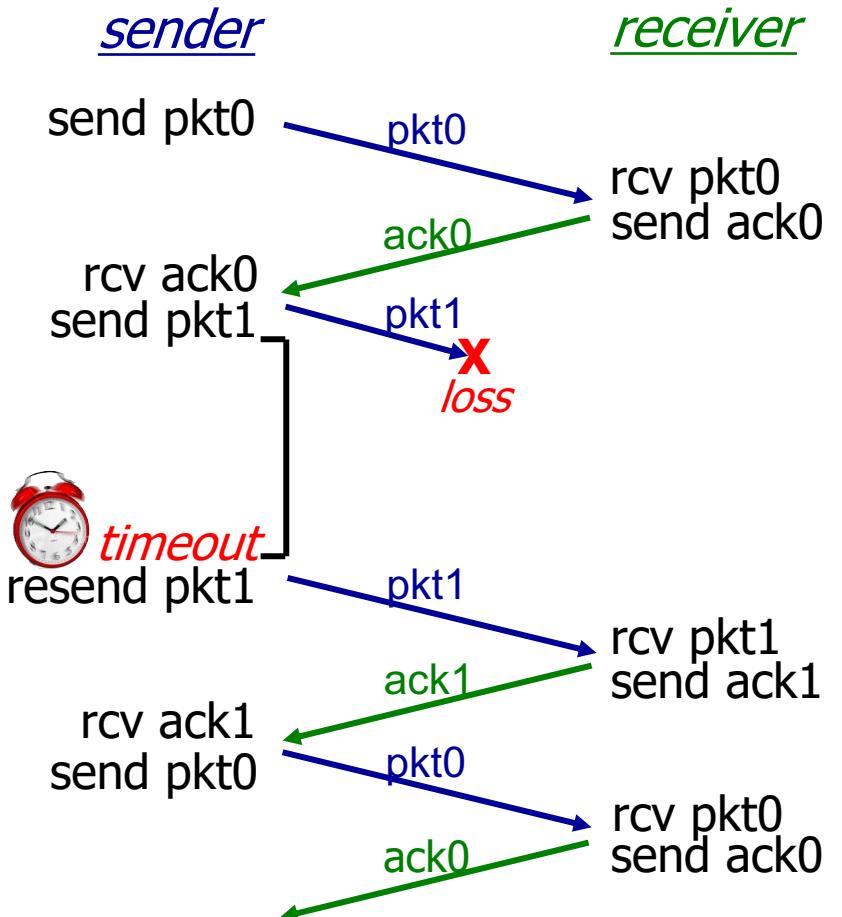
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq #s already handles this!
  - receiver must specify seq # of packet being ACKed
- use countdown timer to interrupt after “reasonable” amount of time



# RDT3.0 in Action

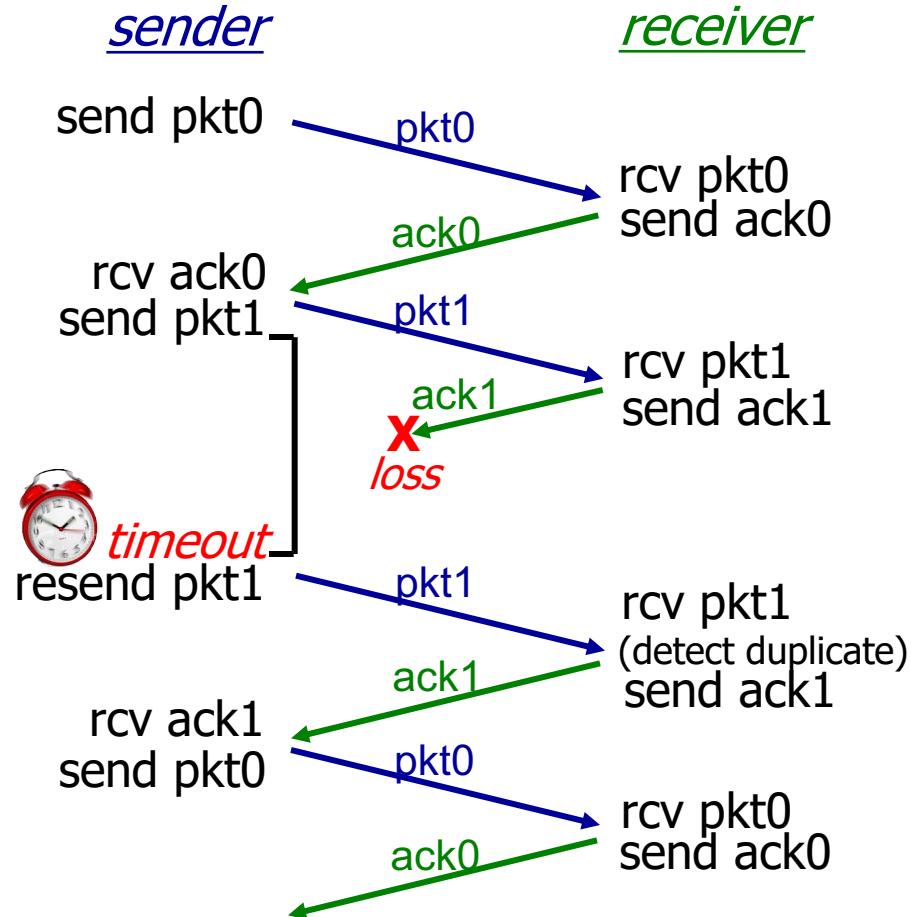


(a) no loss

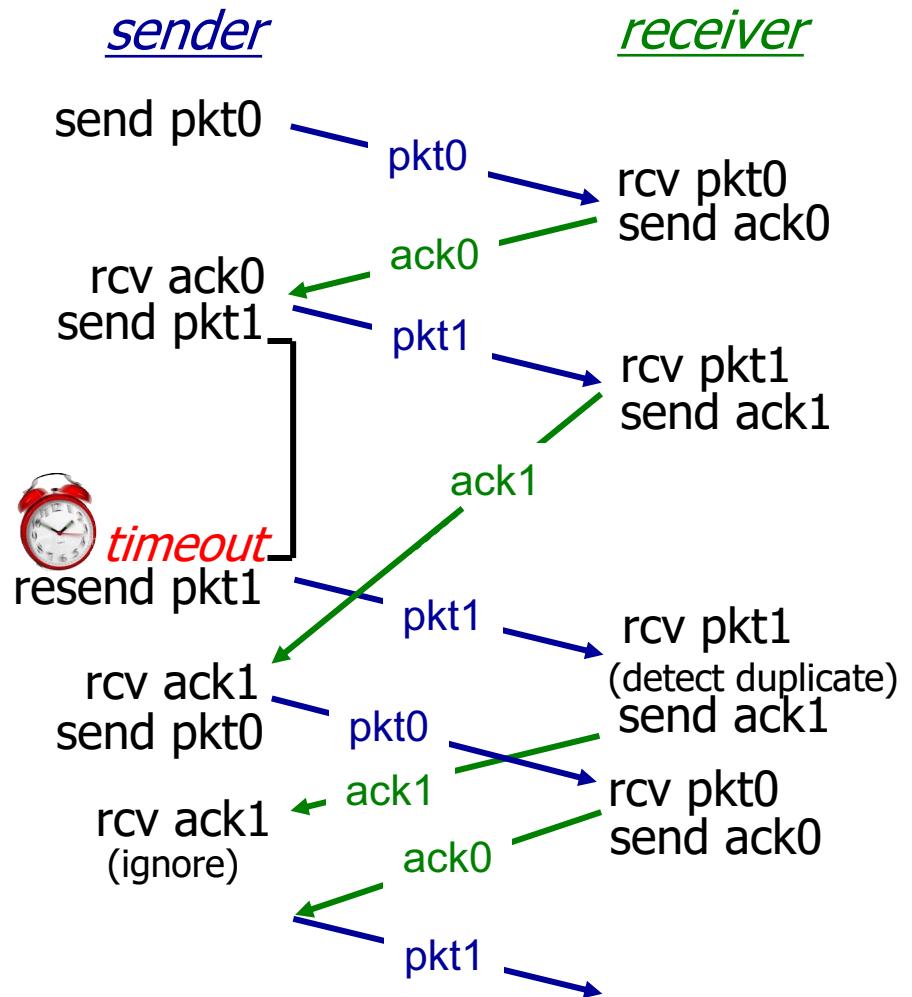


(b) packet loss

# RDT3.0 in Action



(c) ACK loss

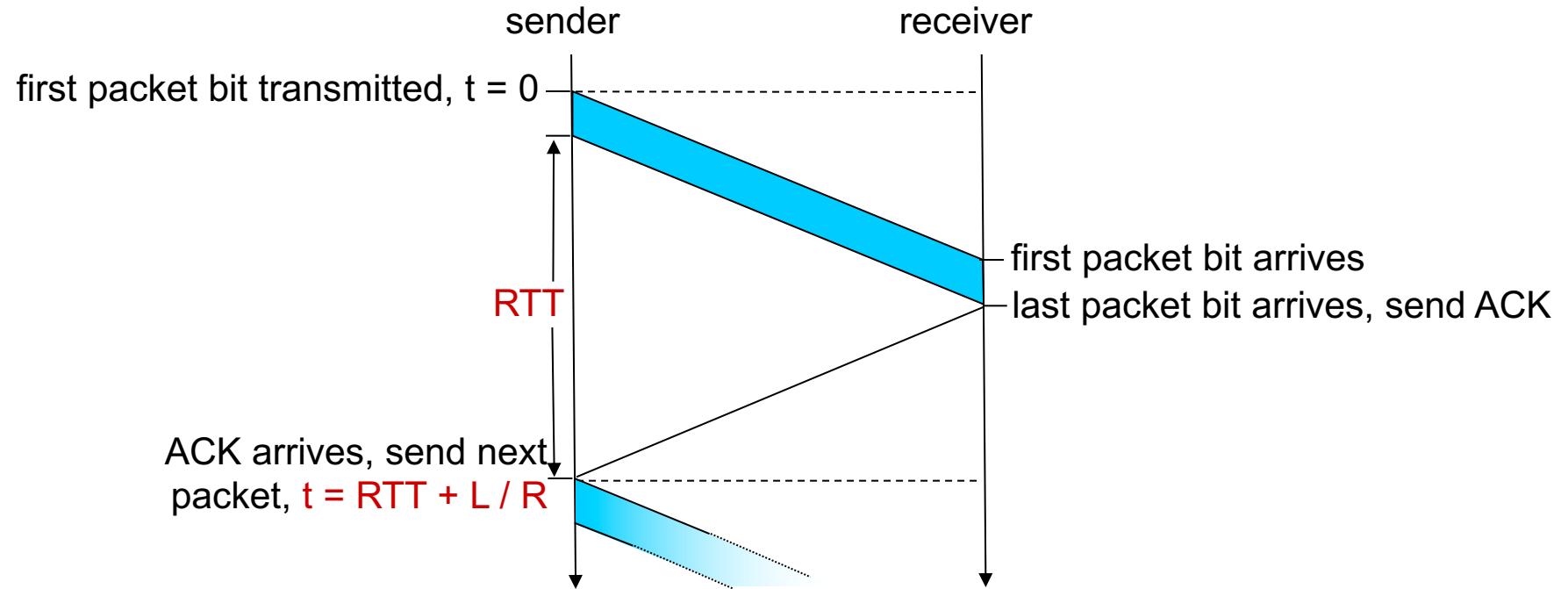


(d) premature timeout/ delayed ACK

# Performance of RDT3.0 (stop-and-wait)

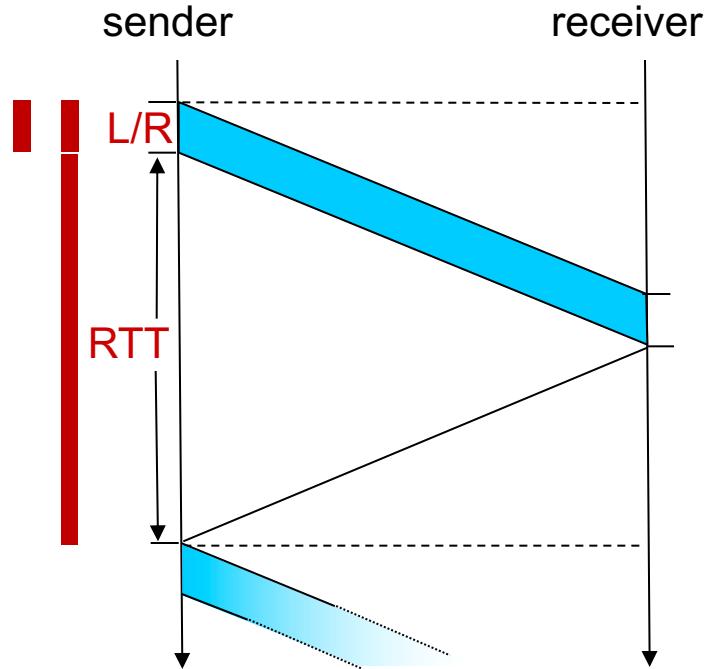
- $U_{sender}$ : *utilization* – fraction of time sender busy sending
- example: 1 Gbps link, 15 millis prop. delay, 8000 bit packet
  - time to transmit packet into channel:  
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

# Performance of RDT3.0 (stop-and-wait)



# Performance of RDT3.0 (stop-and-wait)

$$\begin{aligned} U_{\text{sender}} &= \frac{L / R}{RTT + L / R} \\ &= \frac{.008}{30.008} \\ &= 0.00027 \end{aligned}$$

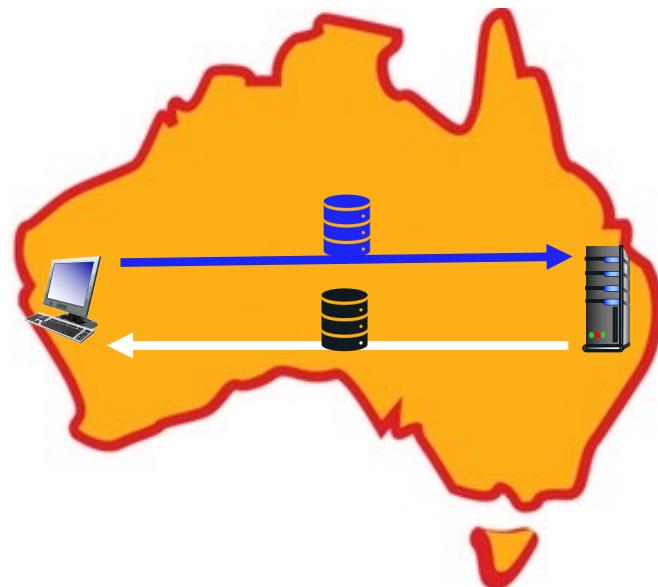


- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

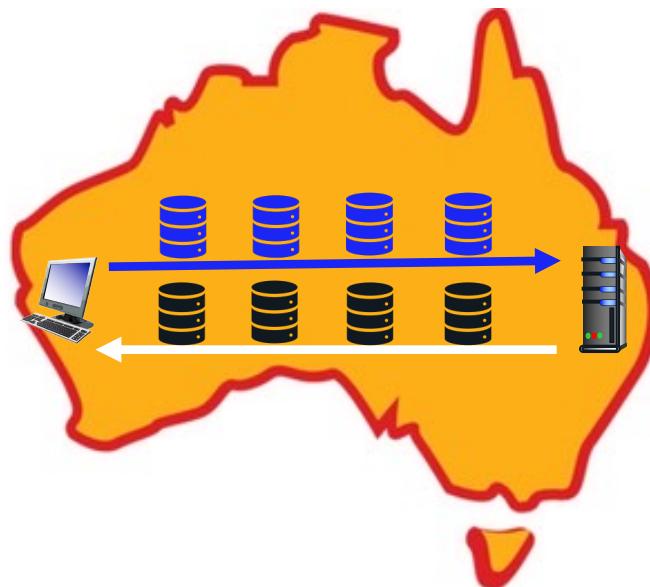
# RDT3.0: Pipelined Protocol Operation

**pipelining:** sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

- range of sequence numbers must be increased
- buffering at sender and/or receiver

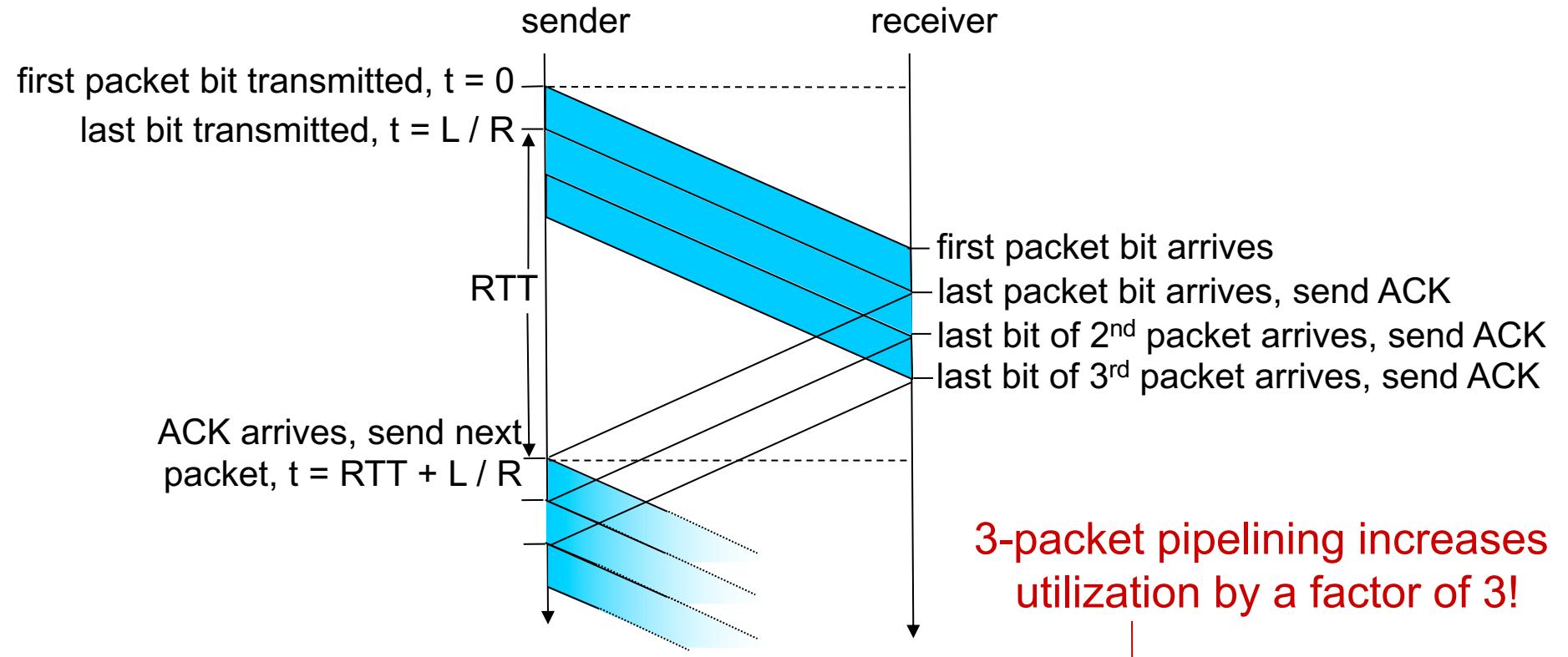


(a) a stop-and-wait protocol in operation

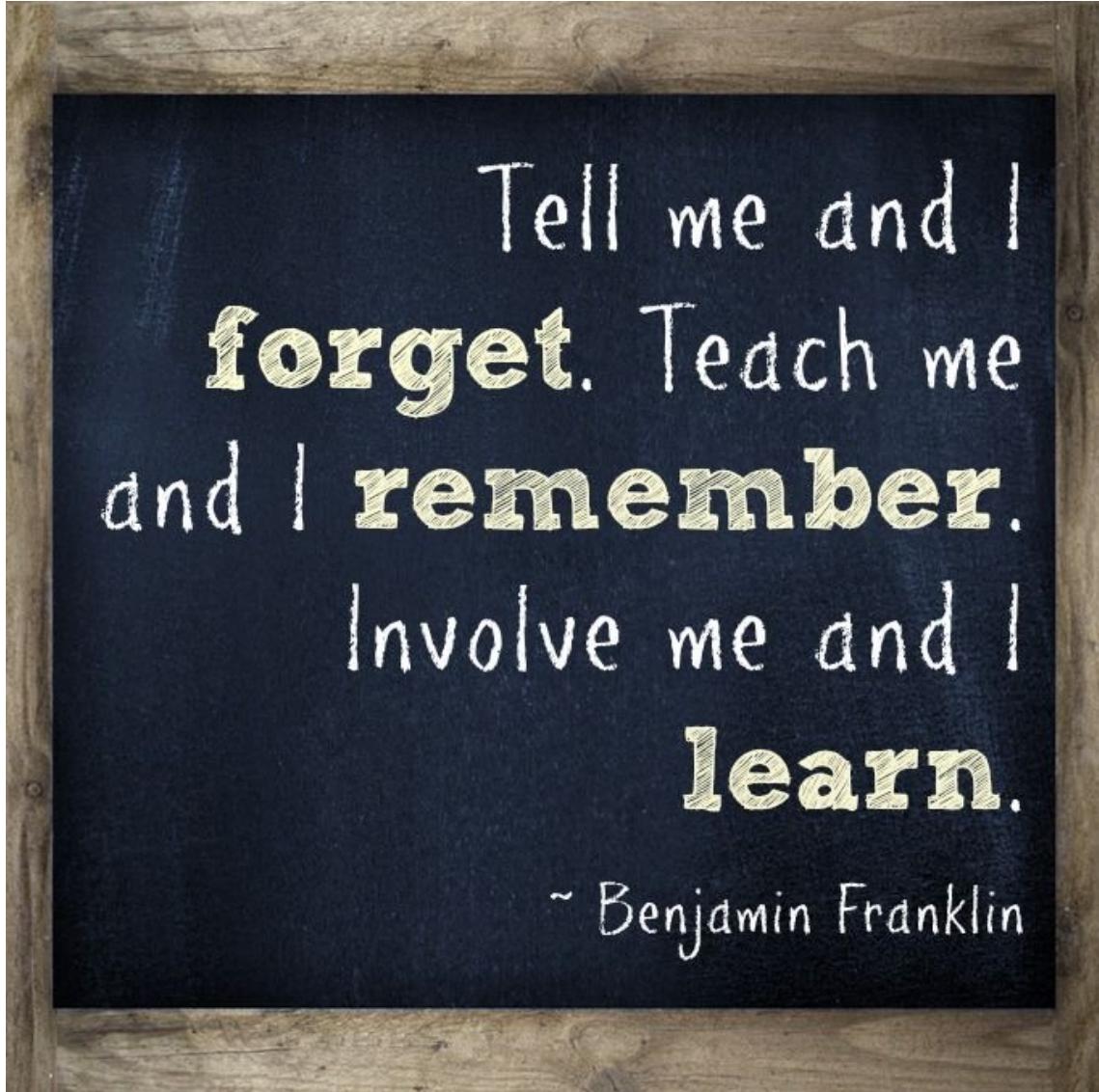


(b) a pipelined protocol in operation

# Pipelining Increases Utilization



$$U_{\text{sender}} = \frac{3L / R}{RTT + 3L / R} = \frac{.024}{30.024} = 0.000799$$

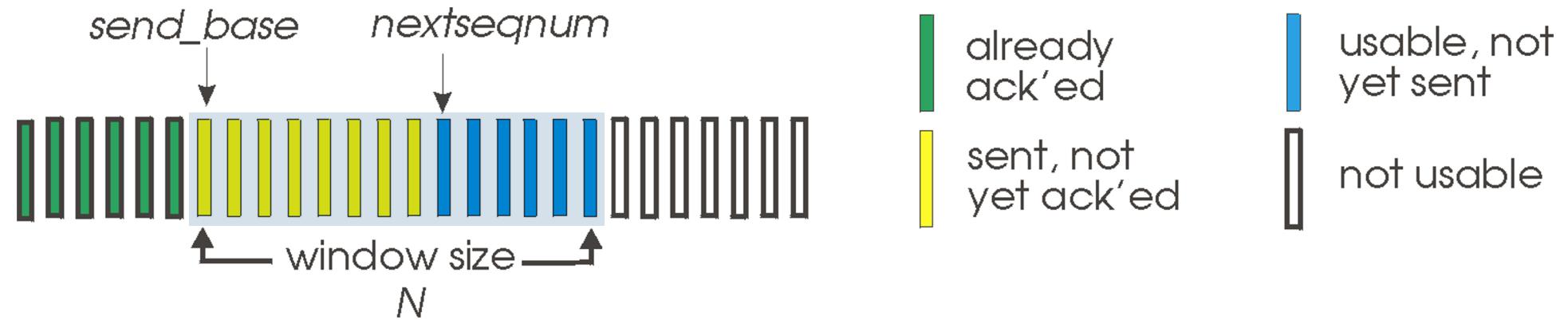


## Pipelined RDT Protocols



# Go-Back-N: Sender

- sender: “window” of up to N, consecutive transmitted but unACKed pkts
  - k-bit seq # in pkt header

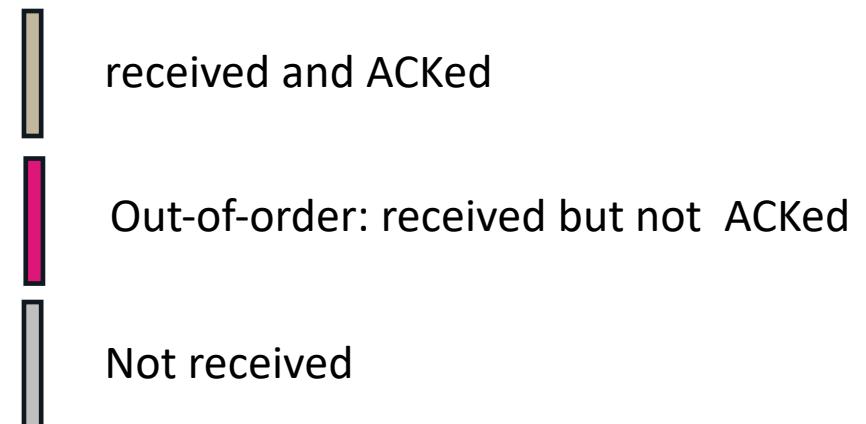
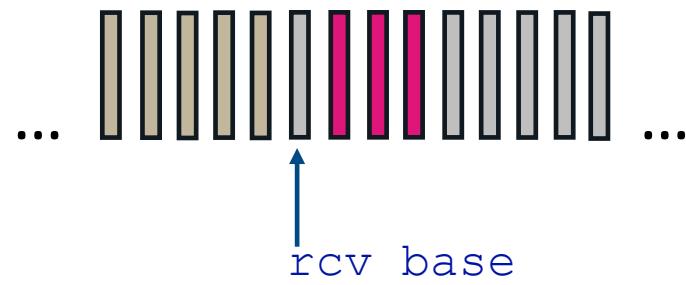


- **cumulative ACK**:  $\text{ACK}(n)$ : ACKs all packets up to, including seq #  $n$ 
  - on receiving  $\text{ACK}(n)$ : move window forward to begin at  $n+1$
- timer for oldest in-flight packet
- $\text{timeout}(n)$ : retransmit packet  $n$  and all higher seq # packets in window

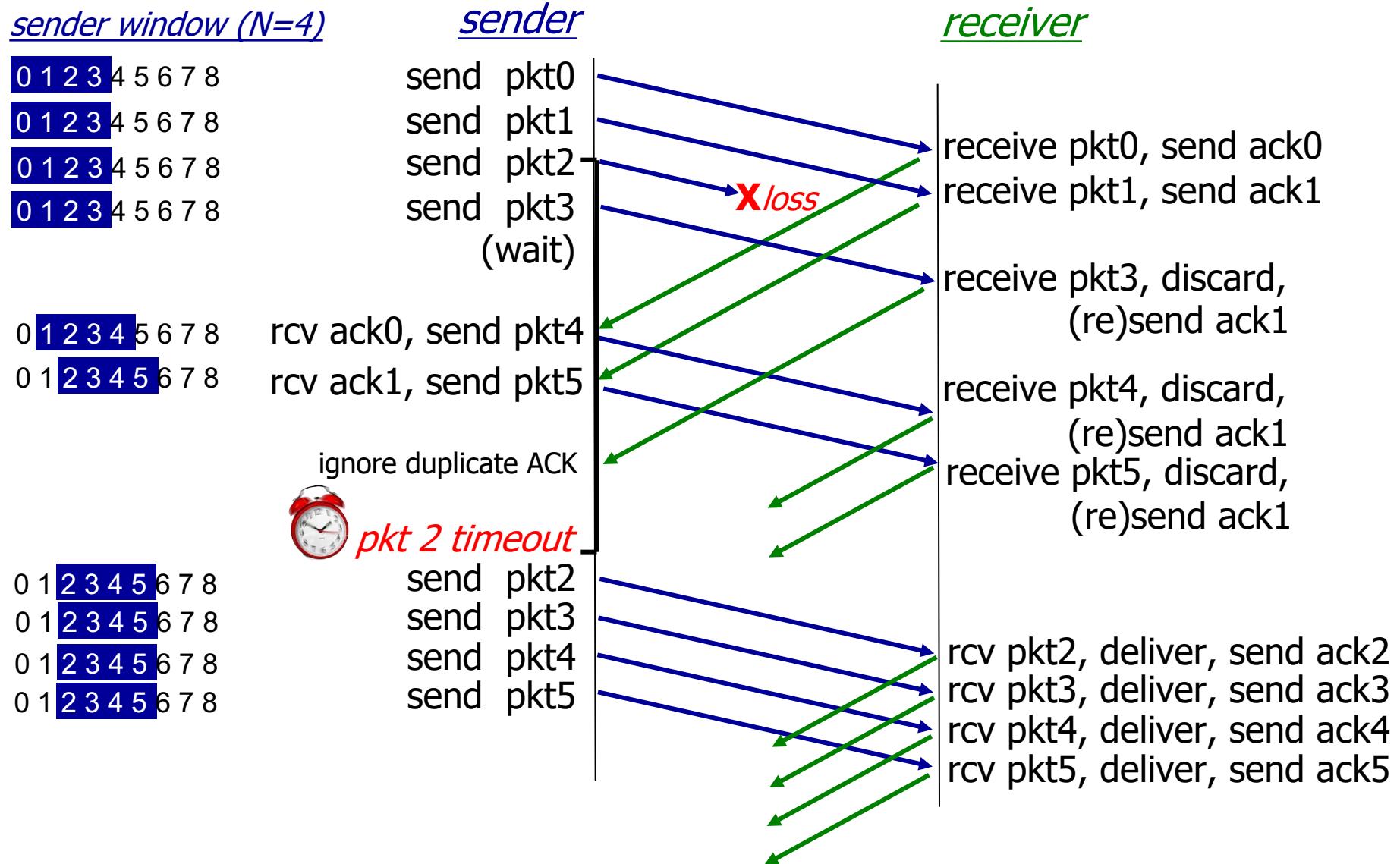
# Go-Back-N: Receiver

- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base`
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

Receiver view of sequence number space:



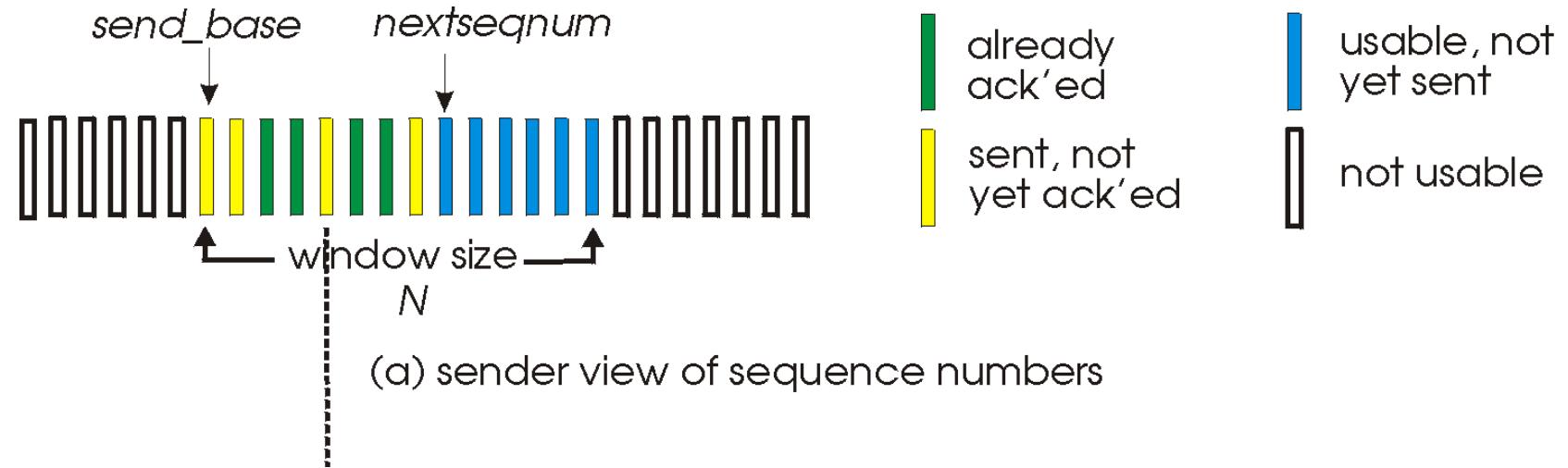
# Go-Back-N: In Action



# Selective Repeat

- receiver *individually* acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer
- sender times-out/retransmits individually for unACKed packets
  - sender maintains timer for each unACKed pkt
- sender window
  - $N$  consecutive seq #s
  - limits seq #s of sent, unACKed packets

# Selective Repeat: Sender and Receiver Window



# Selective Repeat: Sender and Receiver

## sender

data from above:

- if next available seq # in window, send packet

**timeout( $n$ ):**

- resend packet  $n$ , restart timer

**ACK( $n$ ) in [sendbase,sendbase+N]:**

- mark packet  $n$  as received
- if  $n$  smallest unACKed packet, advance window base to next unACKed seq #

## receiver

packet  $n$  in [rcvbase, rcvbase+N-1]

- send ACK( $n$ )
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

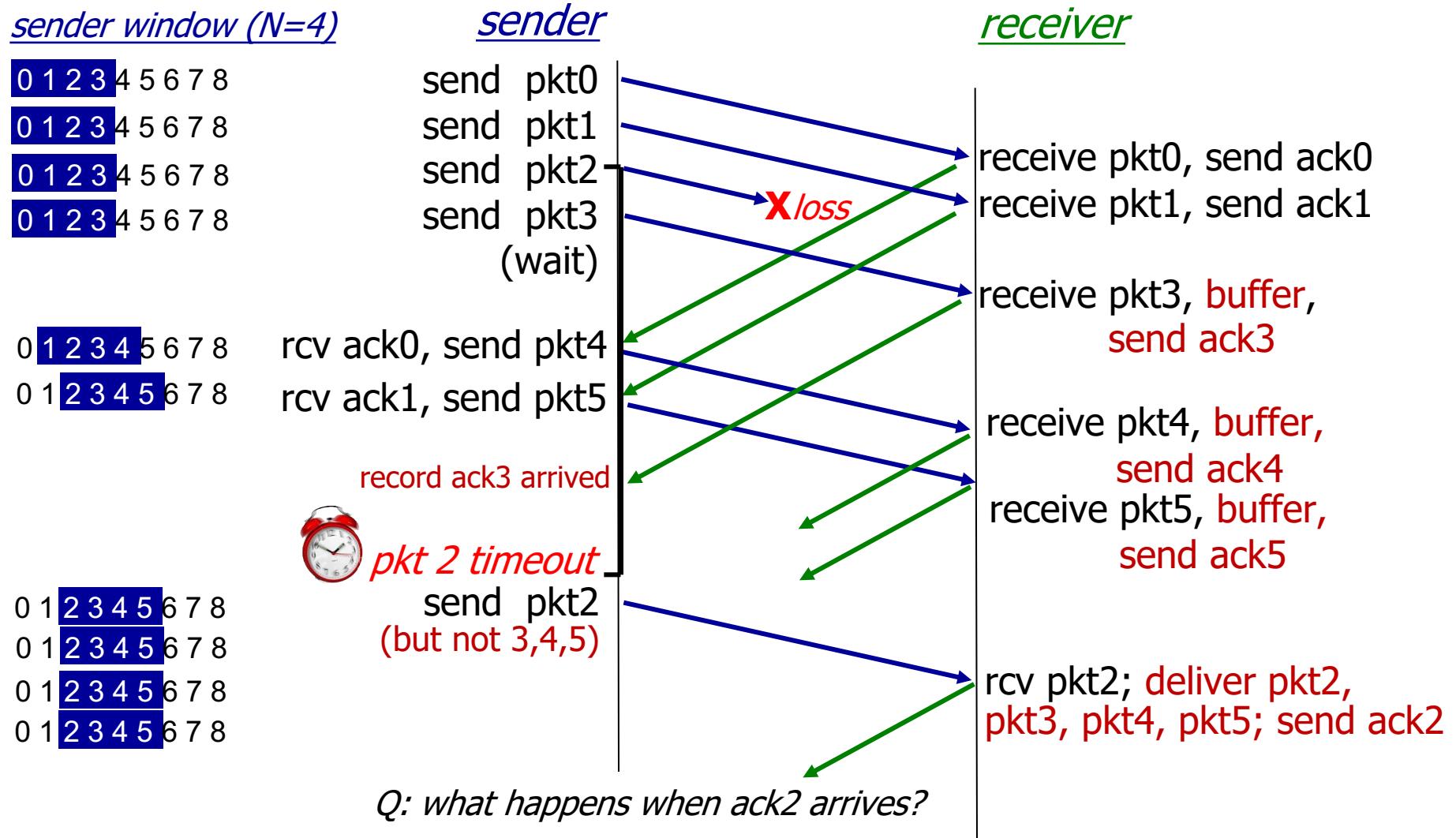
packet  $n$  in [rcvbase-N,rcvbase-1]

- ACK( $n$ )

**otherwise:**

- ignore

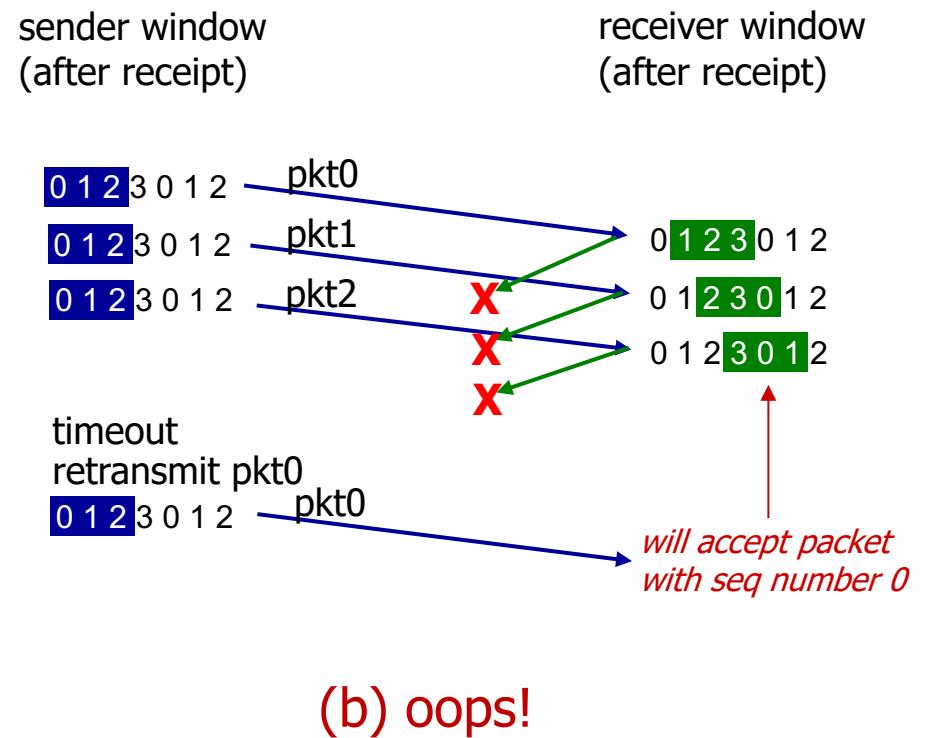
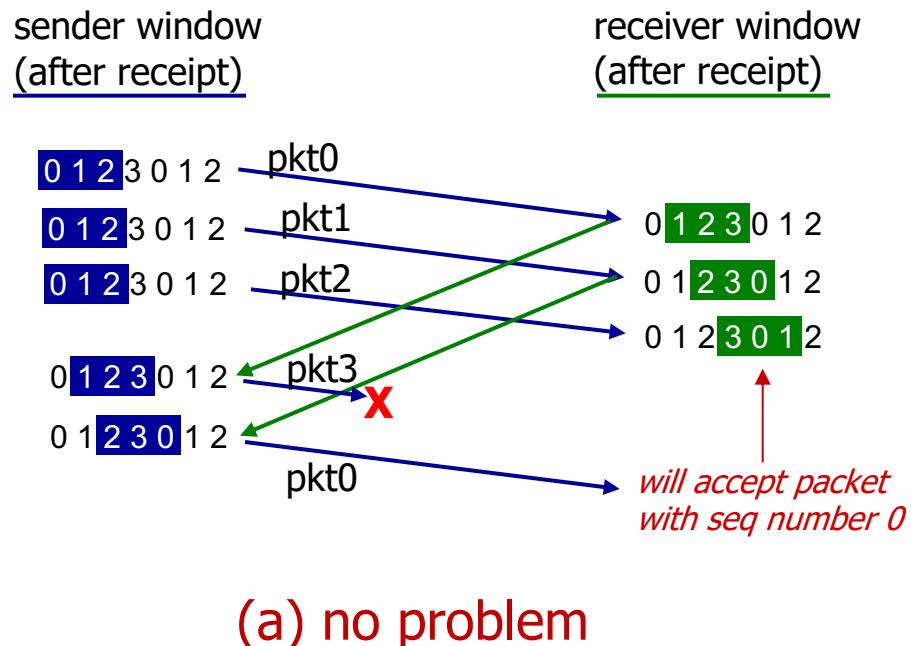
# Selective Repeat: In Action



# Selective Repeat: A Dilemma

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3

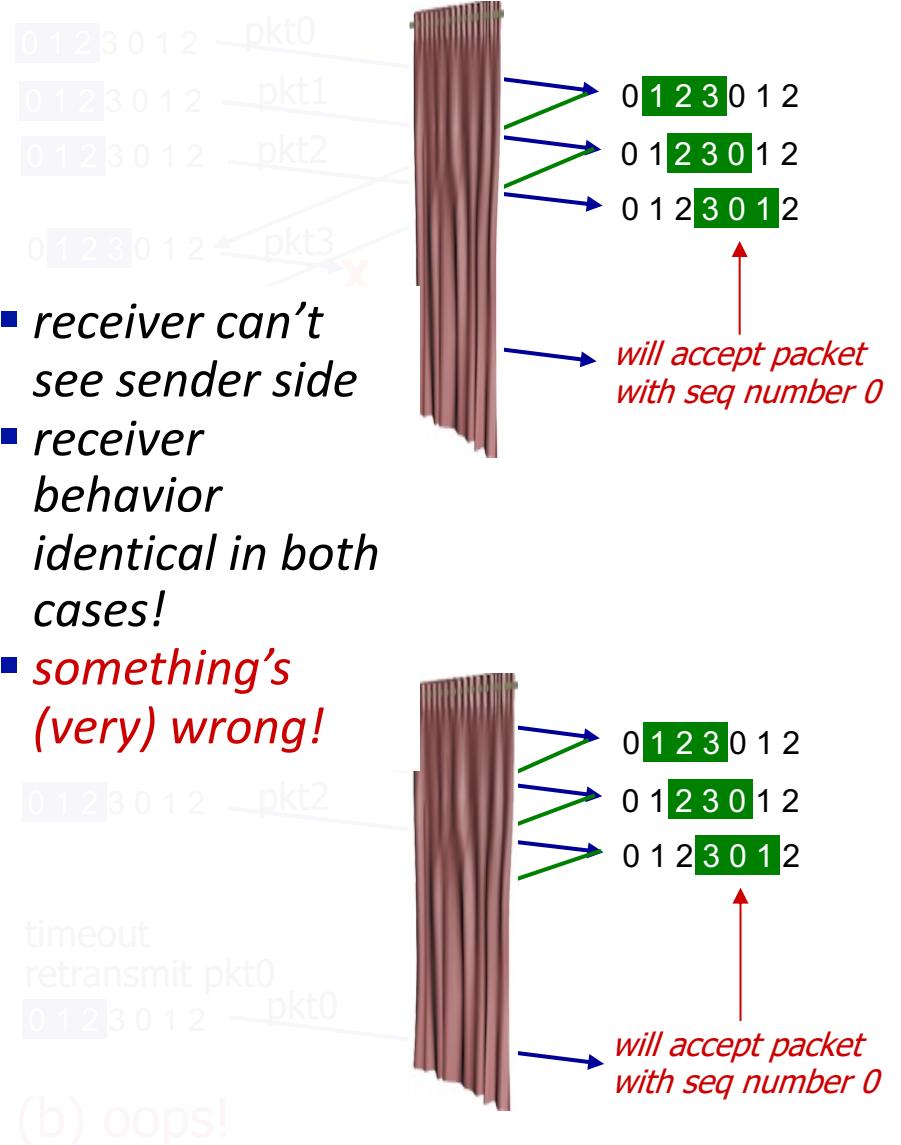


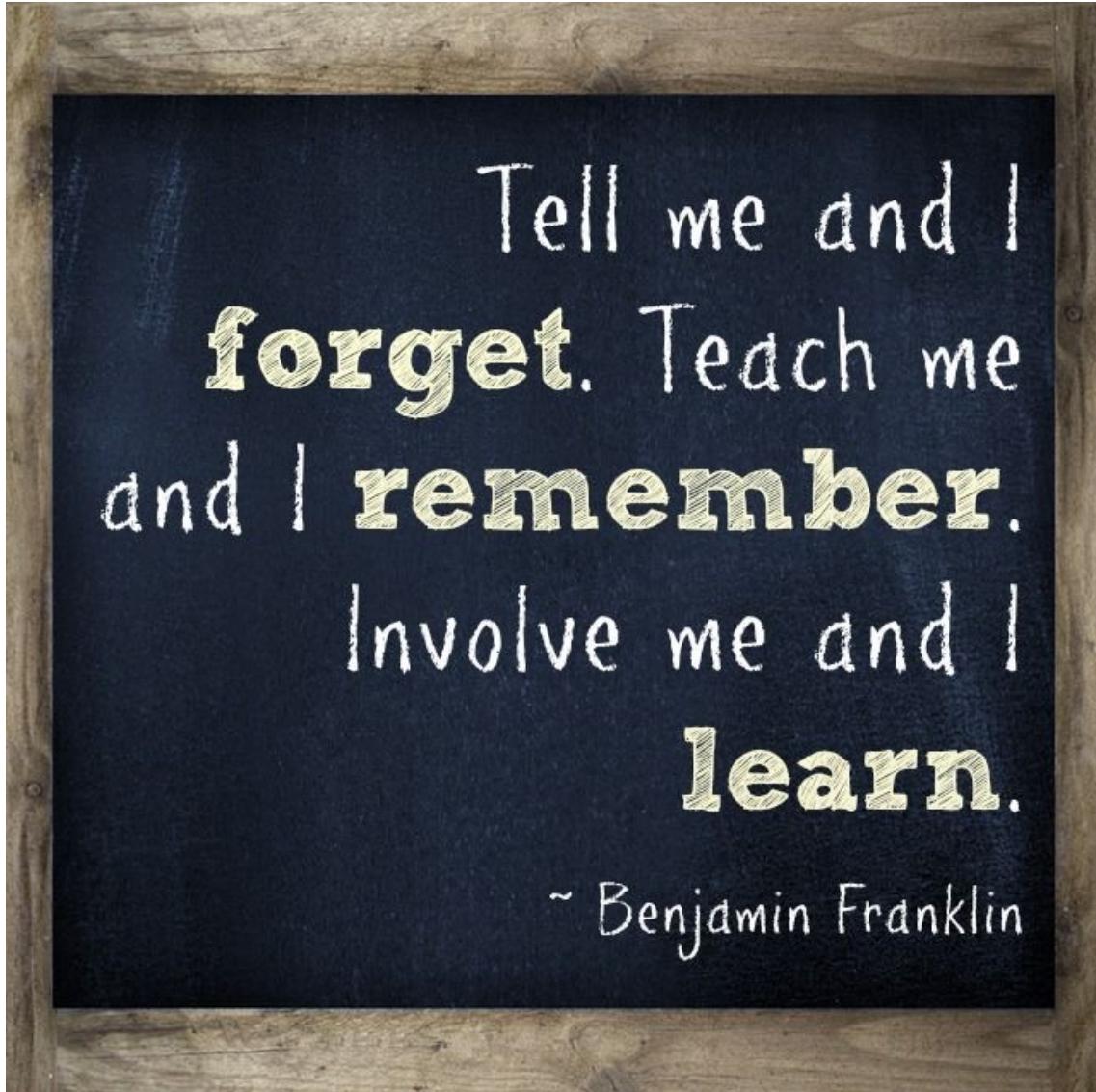
# Selective Repeat: A Dilemma

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3

**Q:** what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?





# Transmission Control Protocol (TCP Part 1)



- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam*:
  - no “message boundaries”
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- cumulative ACKs
- pipelining:
  - TCP congestion and flow control set window size
- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

# TCP Segment Structure

ACK: seq # of next expected byte; A bit: this is an ACK

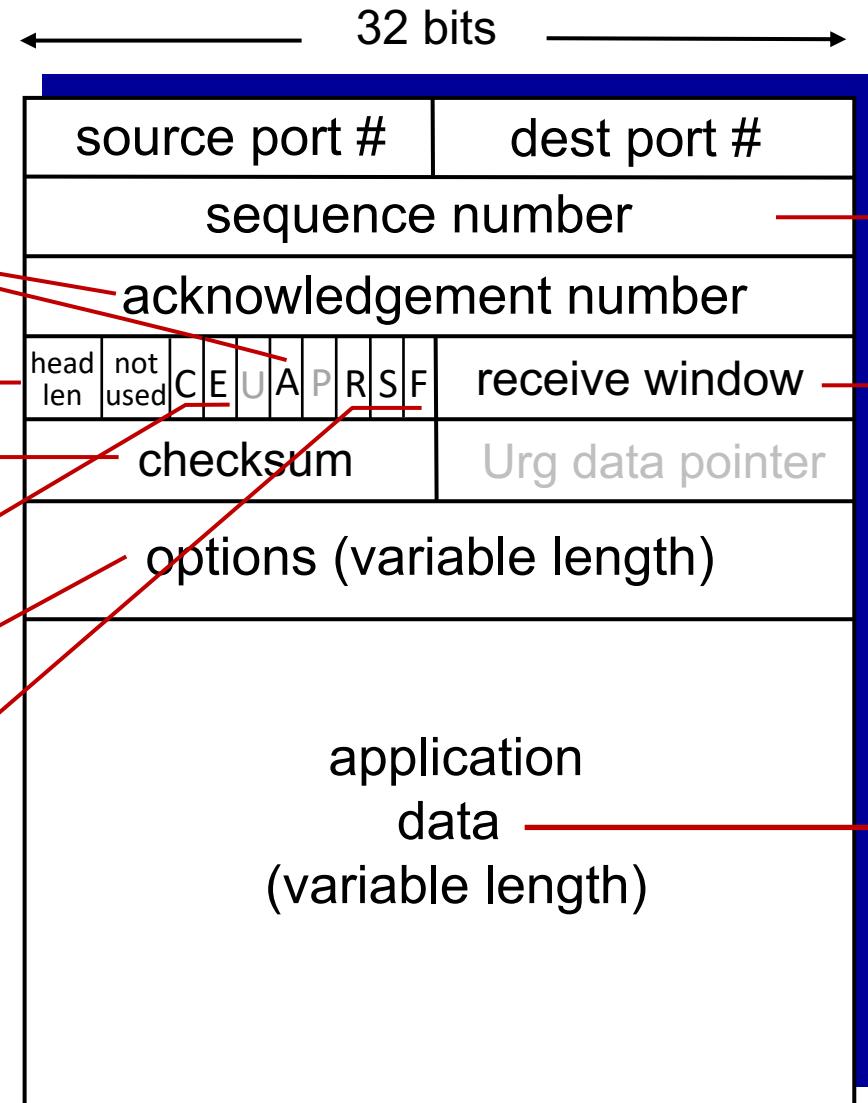
length (of TCP header)

Internet checksum

C, E: congestion notification

TCP options

RST, SYN, FIN: connection management



segment seq #: counting bytes of data into bytestream (not segments!)

flow control: # bytes receiver willing to accept

data sent by application into TCP socket

# TCP Sequence Number and ACKs

## *Sequence numbers:*

- byte stream “number” of first byte in segment’s data

## *Acknowledgements:*

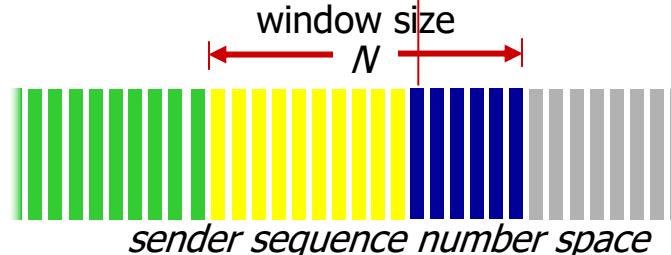
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn’t say, - up to implementor

outgoing segment from sender

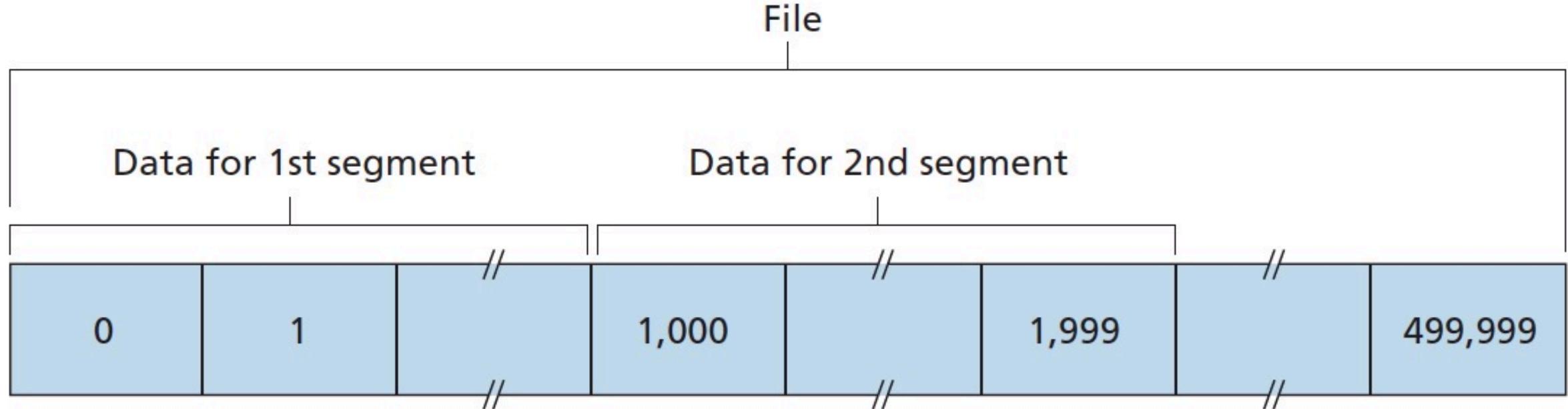
source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer



outgoing segment from receiver

source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer

# TCP Sequence Number and ACKs



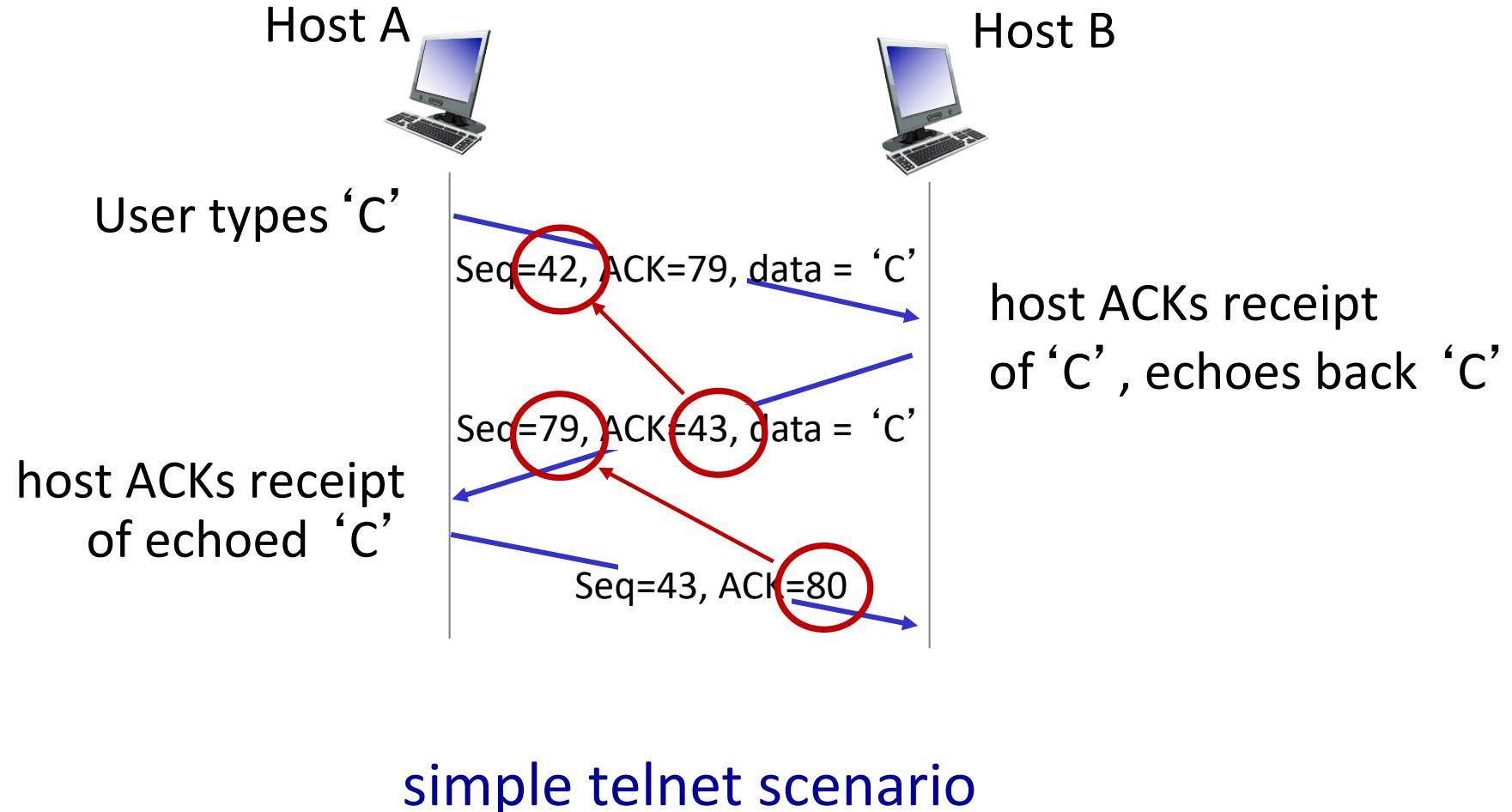
## *Sequence numbers:*

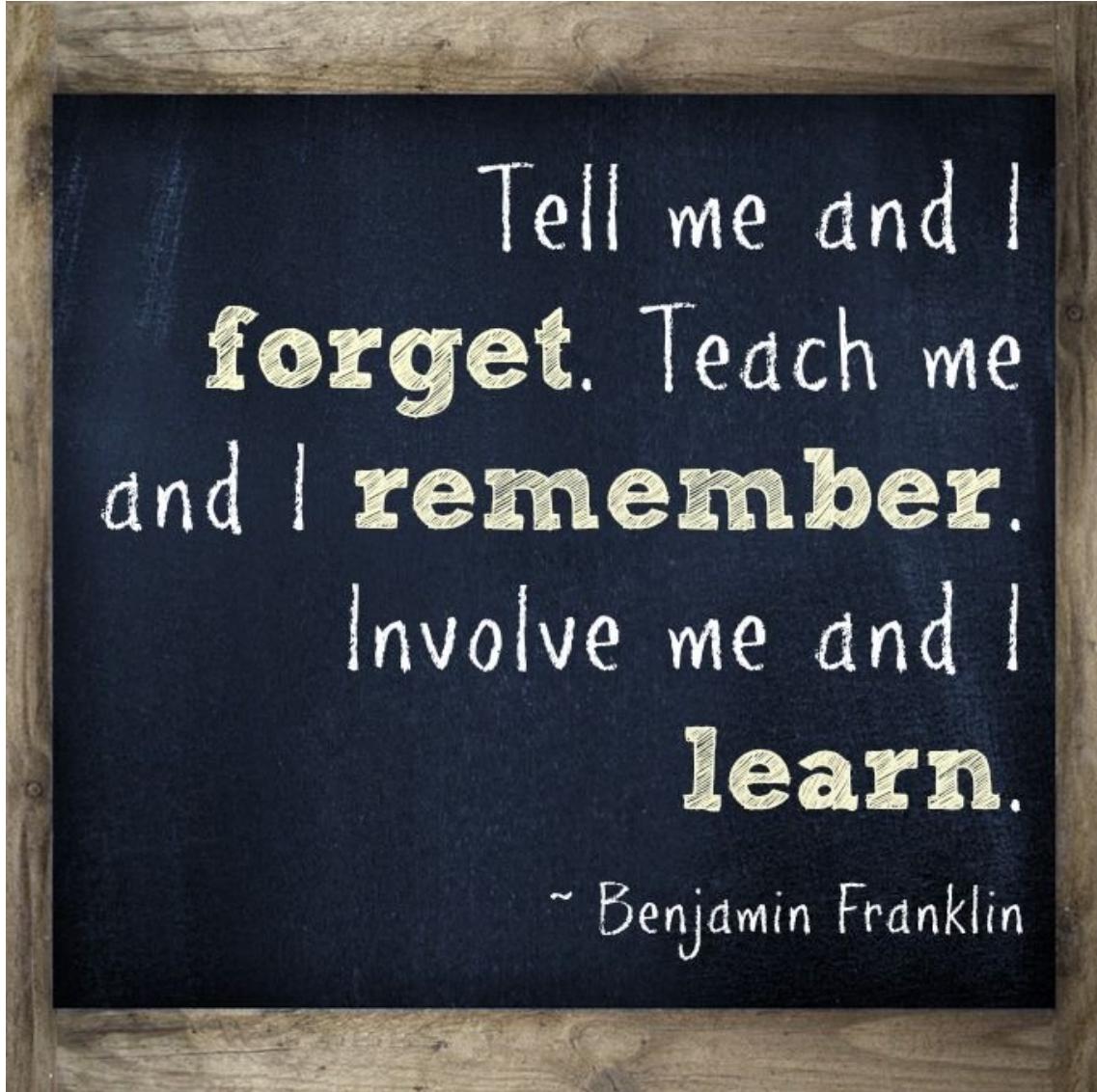
- byte stream “number” of first byte in segment’s data
- random initial sequence

## *Acknowledgements:*

- seq # of next byte expected from other side
- cumulative ACK

# TCP Sequence Number and ACKs





## Transmission Control Protocol (TCP Part 2)



# TCP Round Trip Time and 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

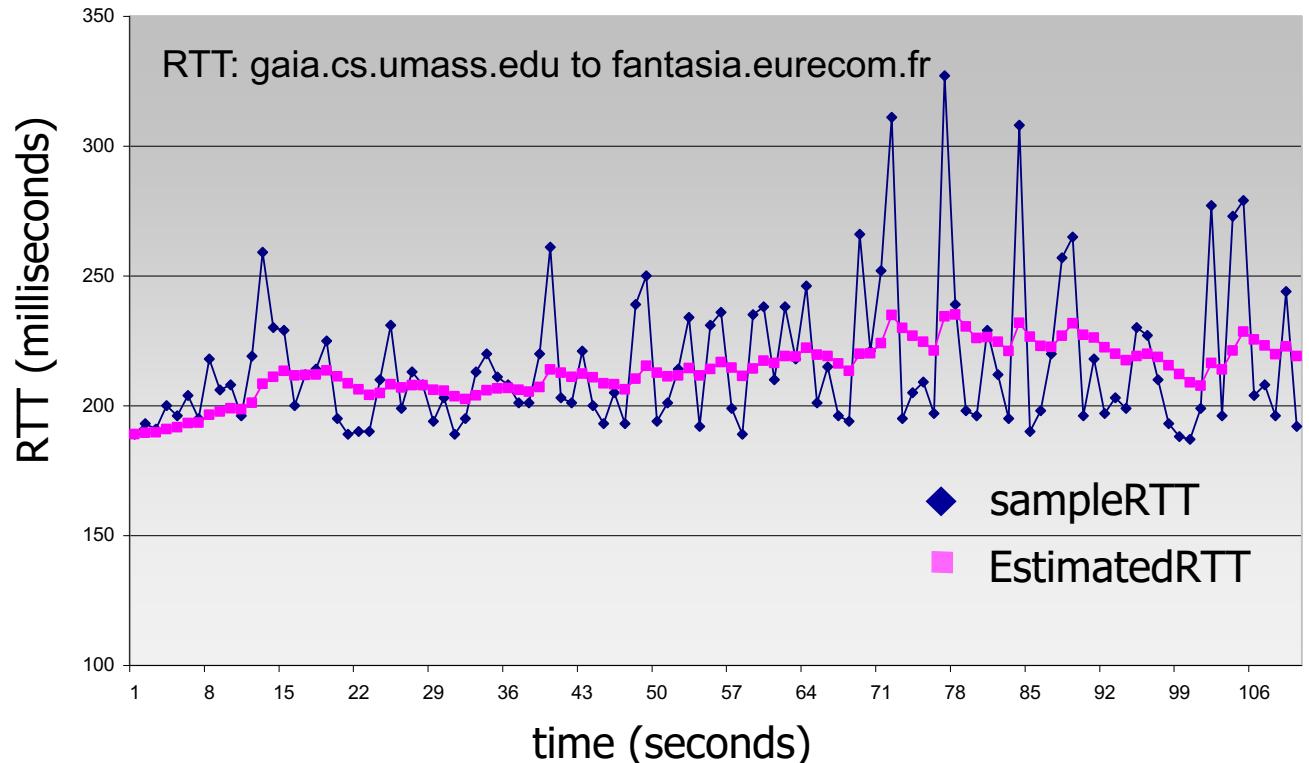
**Q:** how to estimate RTT?

- **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 and Timeout

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

- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value:  $\alpha = 0.125$



# TCP Round Trip Time and Timeout

- timeout interval: **EstimatedRTT** plus “safety margin”
  - large variation in **EstimatedRTT**: want a larger safety margin

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$



estimated RTT

“safety margin”

- **DevRTT**: EWMA of **SampleRTT** deviation from **EstimatedRTT**:

$$\text{DevRTT} = (1-\beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$$

(typically,  $\beta = 0.25$ )

*event: data received from application*

- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unACKed segment
  - expiration interval:  
**TimeOutInterval**

*event: timeout*

- retransmit segment that caused timeout
- restart timer

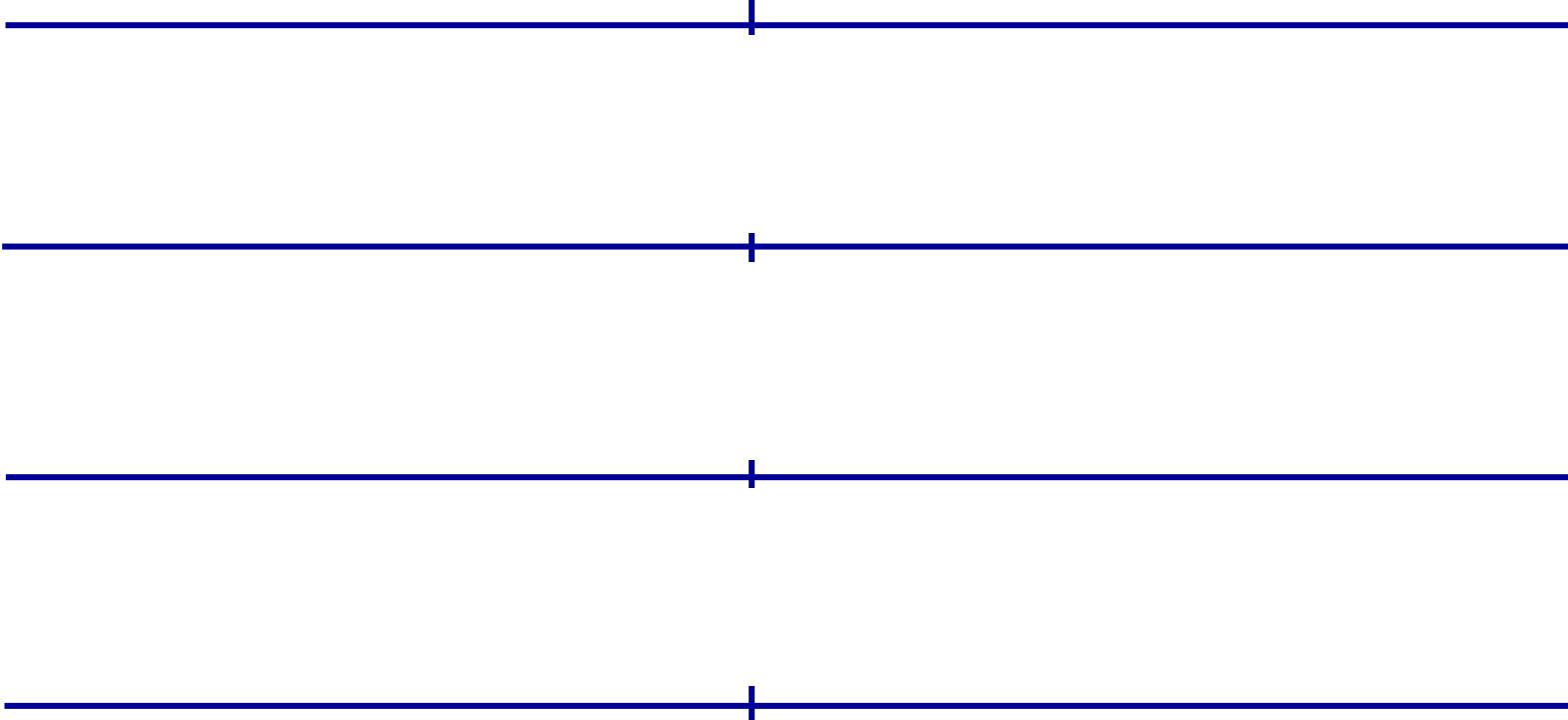
*event: ACK received*

- if ACK acknowledges previously unACKed segments
  - update what is known to be ACKed
  - start timer if there are still unACKed segments

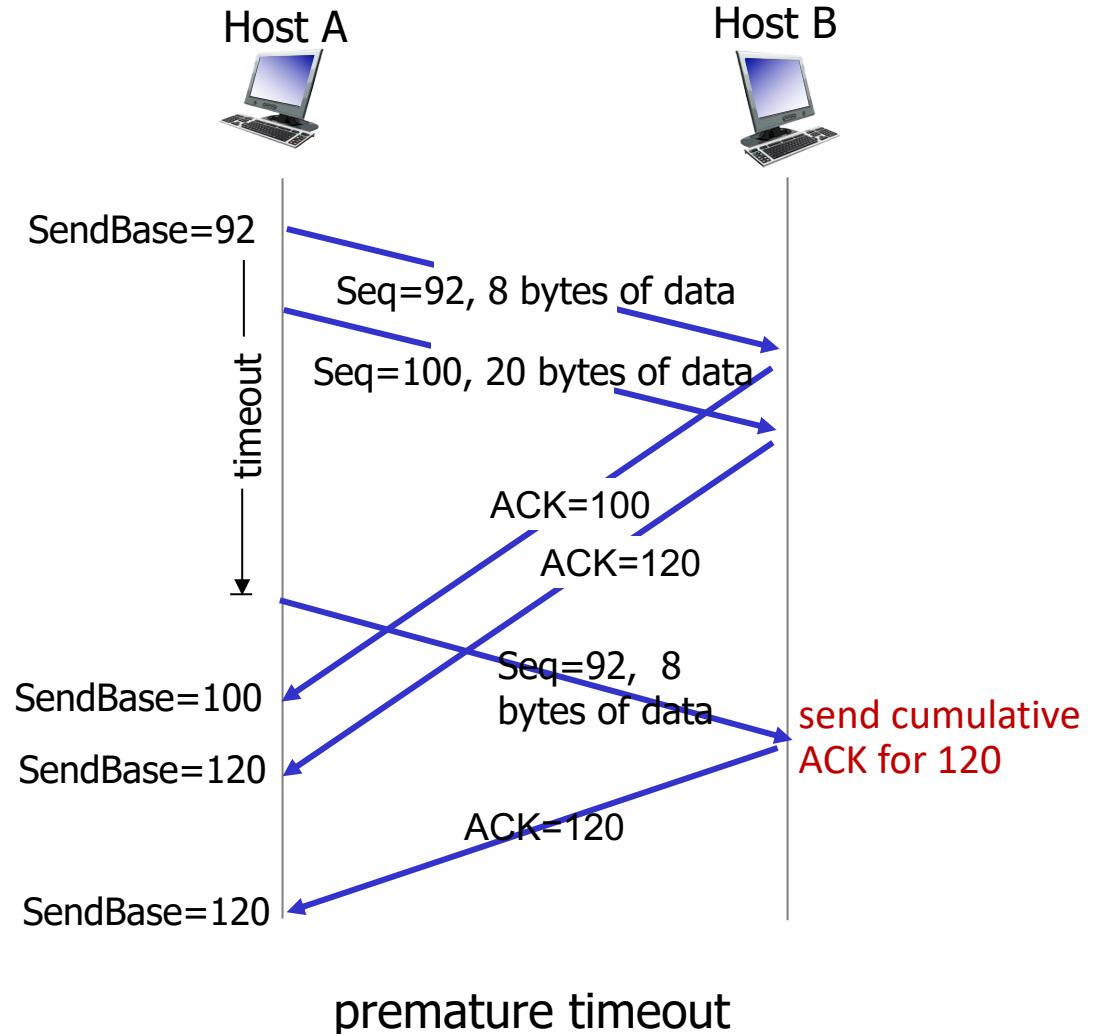
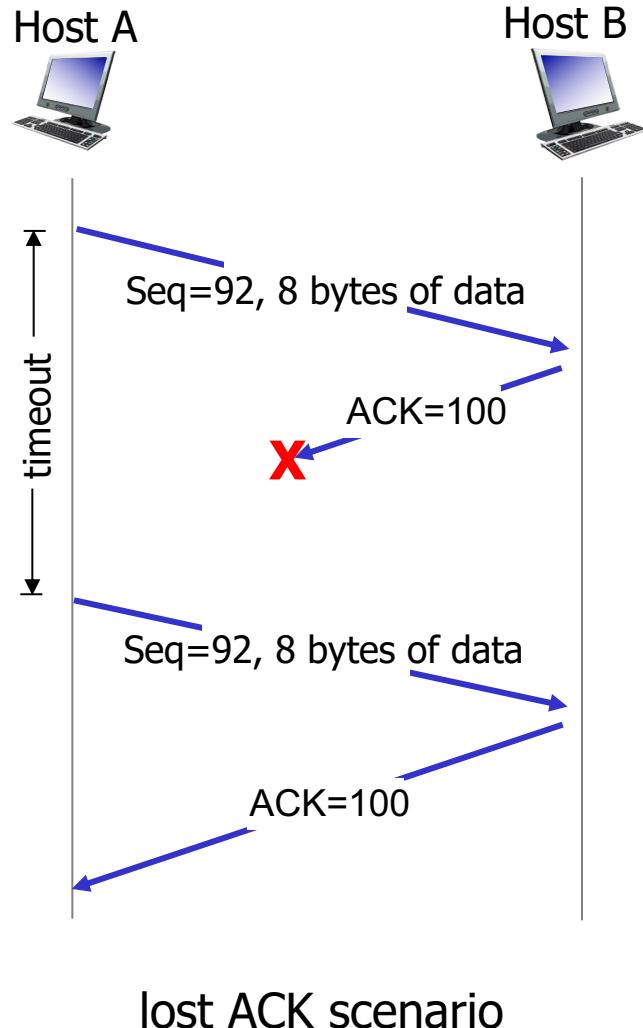
# TCP Receiver: ACK Generation

*Event at receiver*

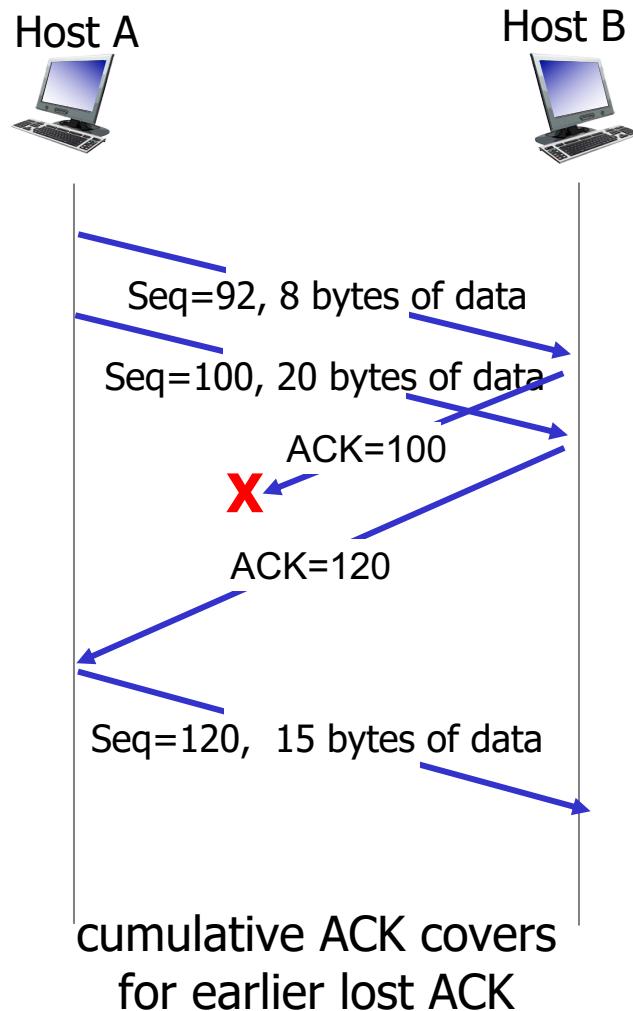
*TCP receiver action*



# TCP: Retransmission Scenarios



# TCP: Retransmission Scenarios



# TCP: Retransmission Scenarios

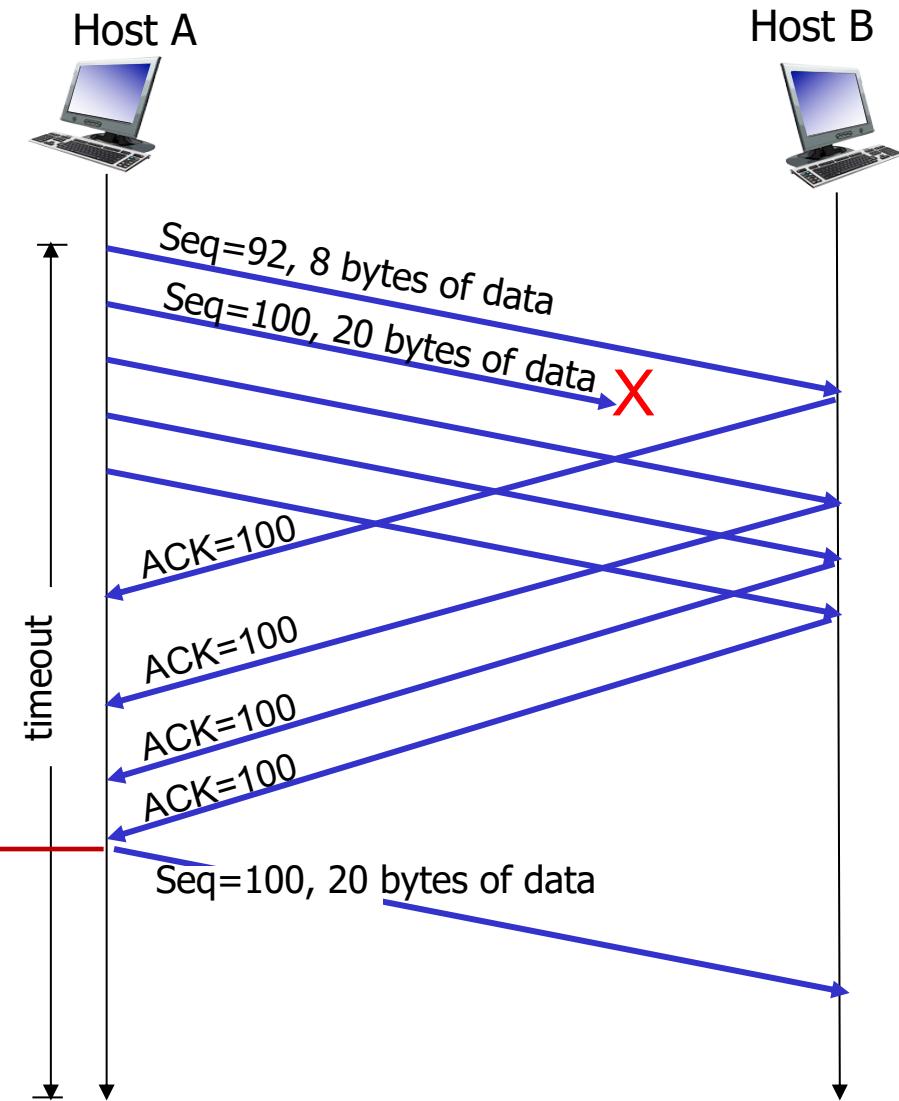
## *TCP fast retransmit*

if sender receives 3 additional ACKs for same data (“triple duplicate ACKs”), resend unACKed segment with smallest seq #

- likely that unACKed segment lost, so don’t wait for timeout

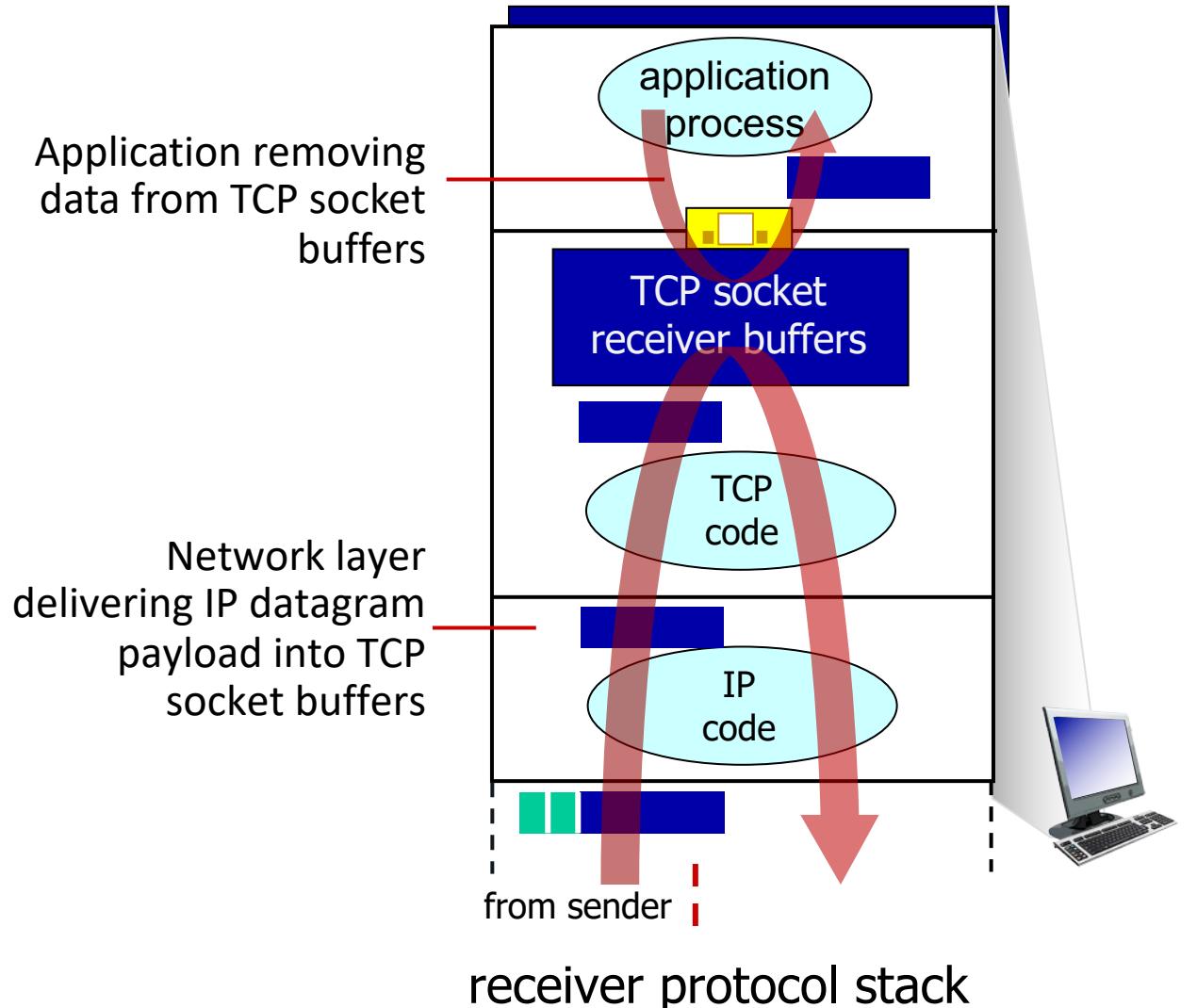


Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!



# TCP: Flow Control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



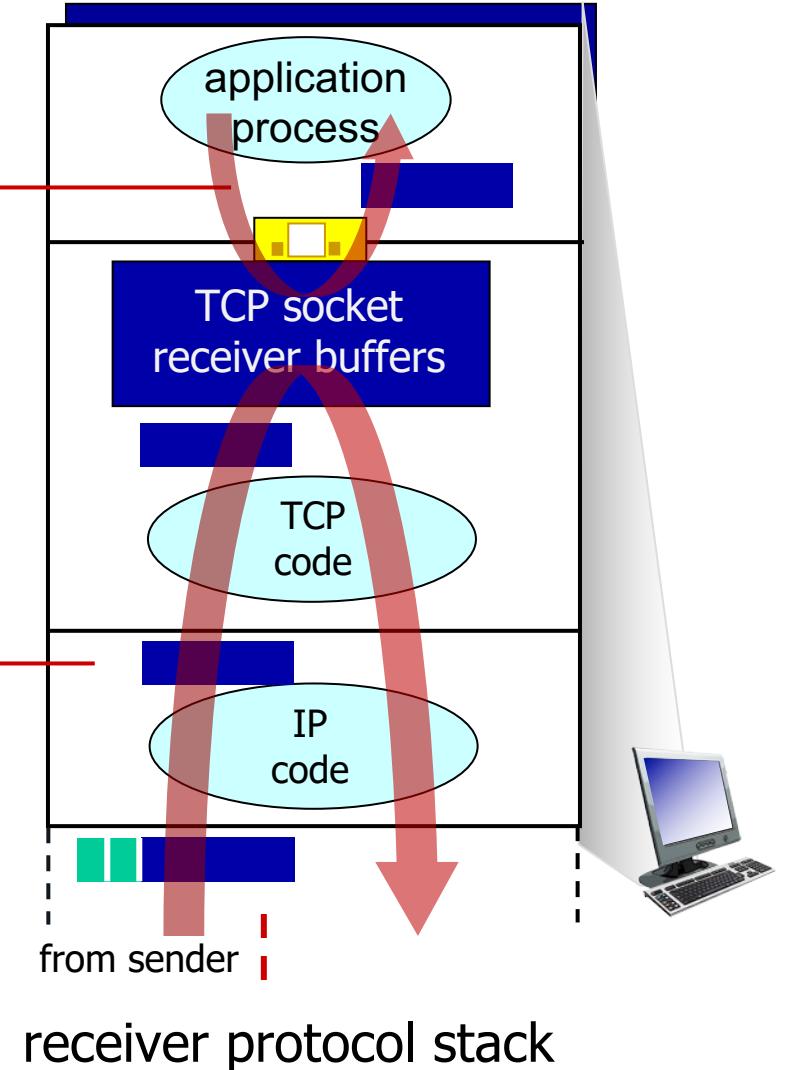
# TCP: Flow Control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



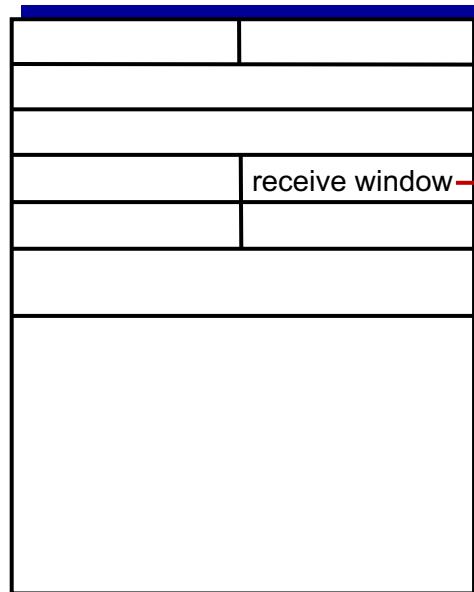
Application removing data from TCP socket buffers

Network layer delivering IP datagram payload into TCP socket buffers

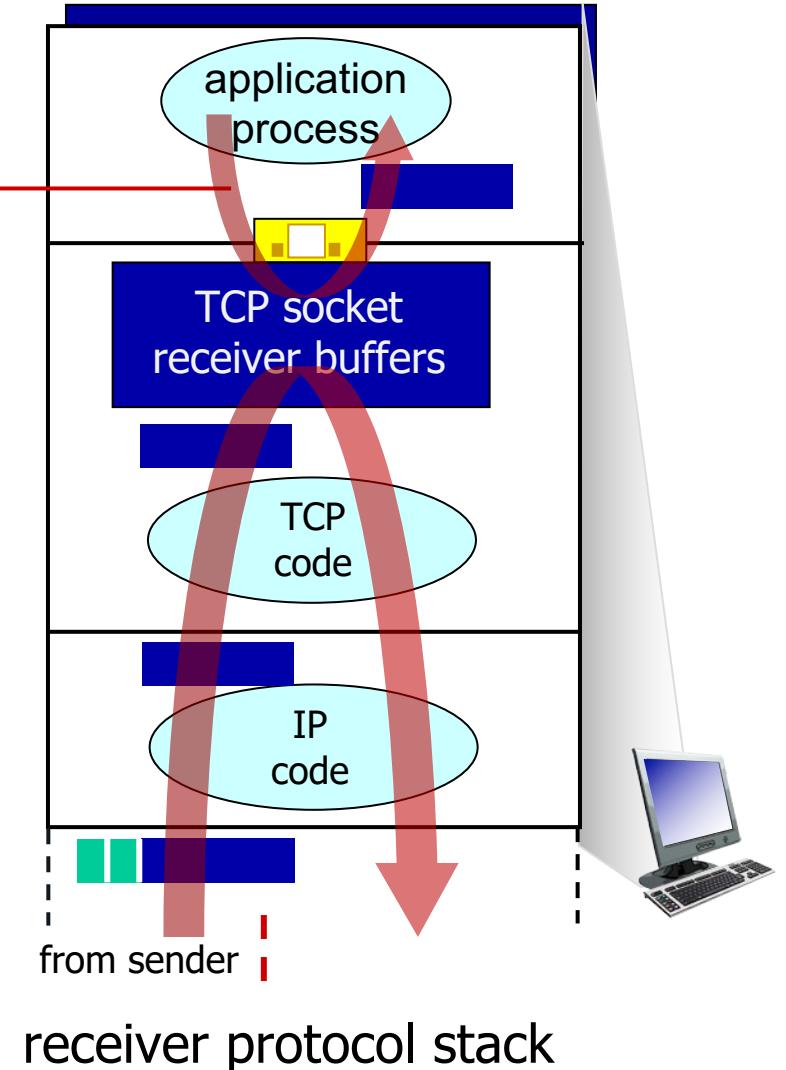


# TCP: Flow Control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



Application removing data from TCP socket buffers



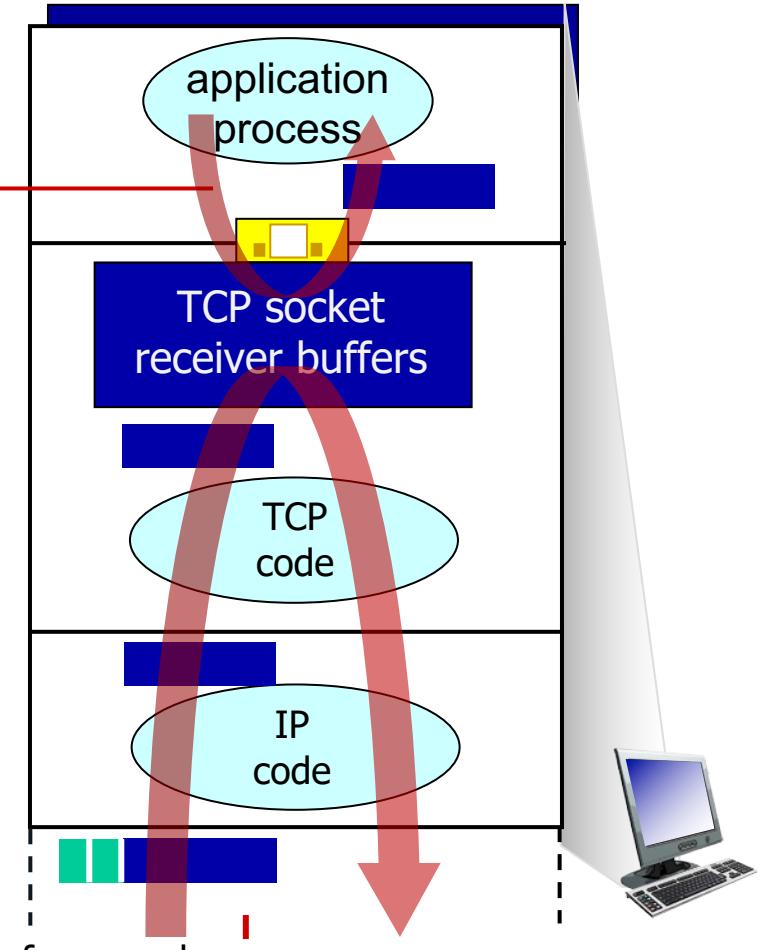
# TCP: Flow Control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

## flow control

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

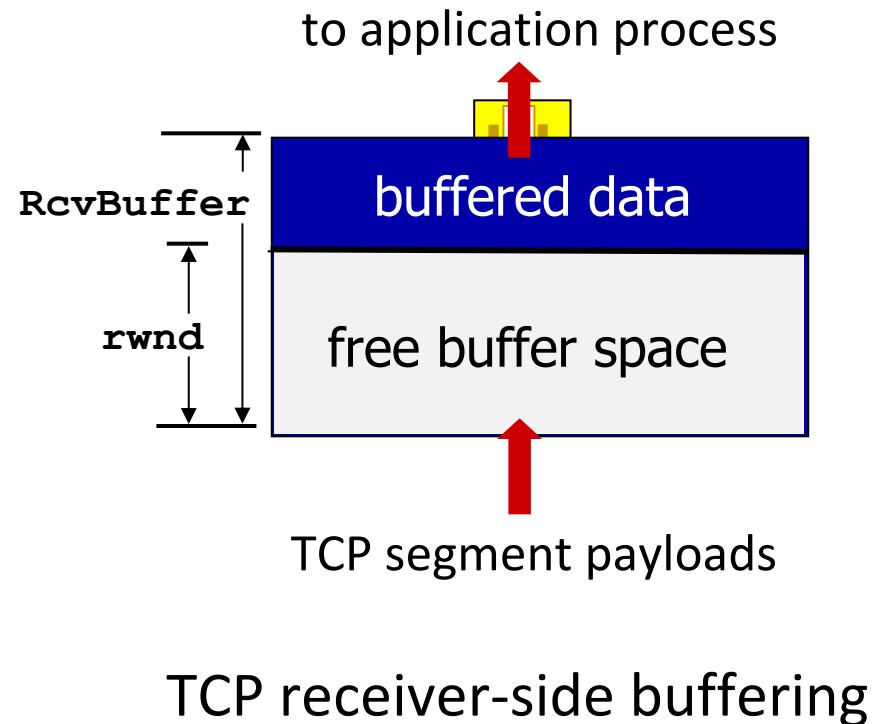
Application removing data from TCP socket buffers



receiver protocol stack

# TCP: Flow Control

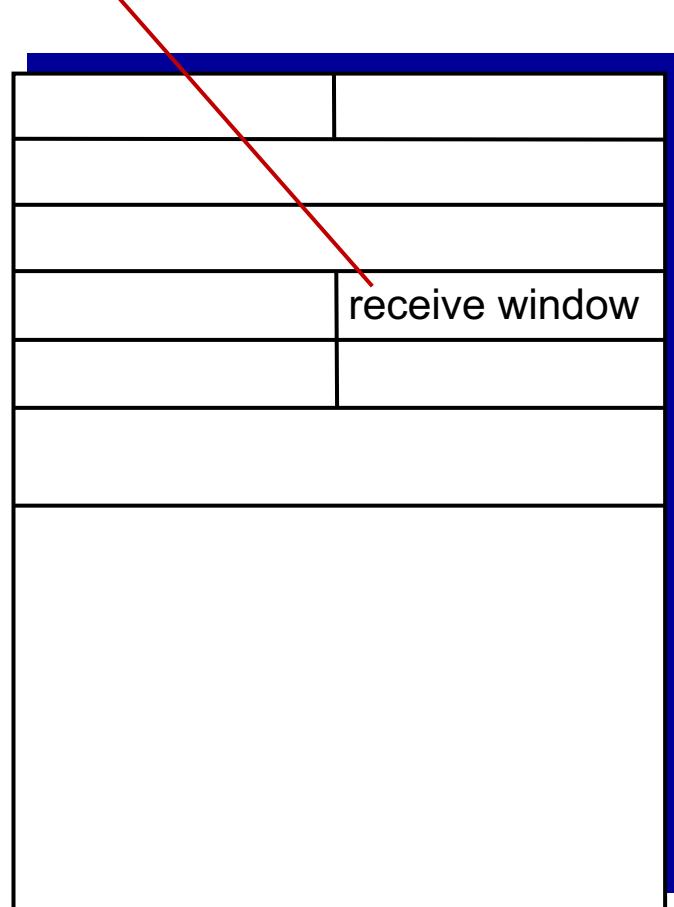
- TCP receiver “advertises” free buffer space in **rwnd** field in TCP header
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unACKed (“in-flight”) data to received **rwnd**
- guarantees receive buffer will not overflow



# TCP: Flow Control

- TCP receiver “advertises” free buffer space in **rwnd** field in TCP header
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unACKed (“in-flight”) data to received **rwnd**
- guarantees receive buffer will not overflow

flow control: # bytes receiver willing to accept

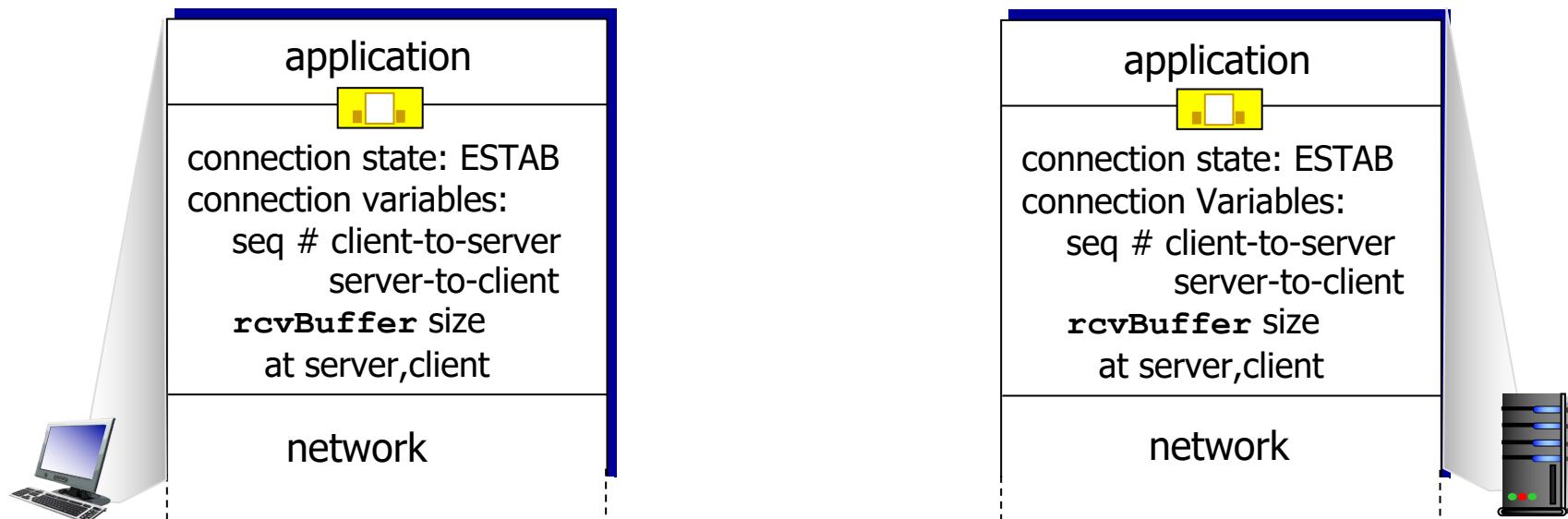


TCP segment format

# TCP Connection Management

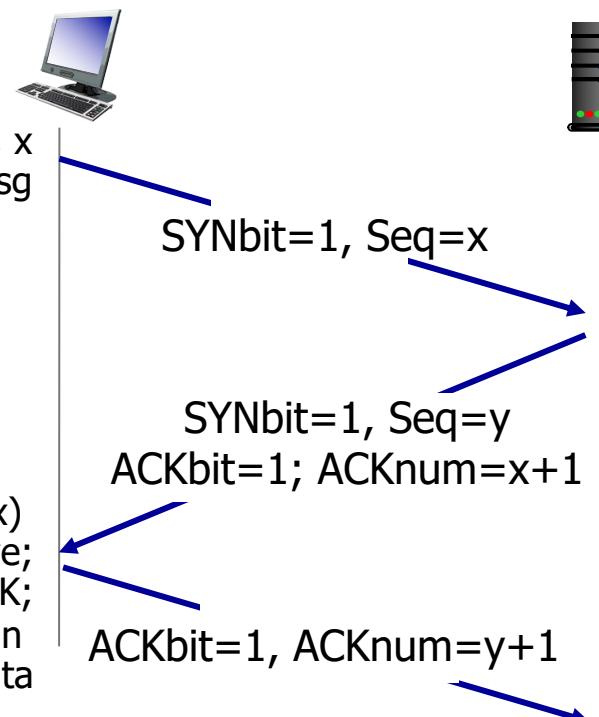
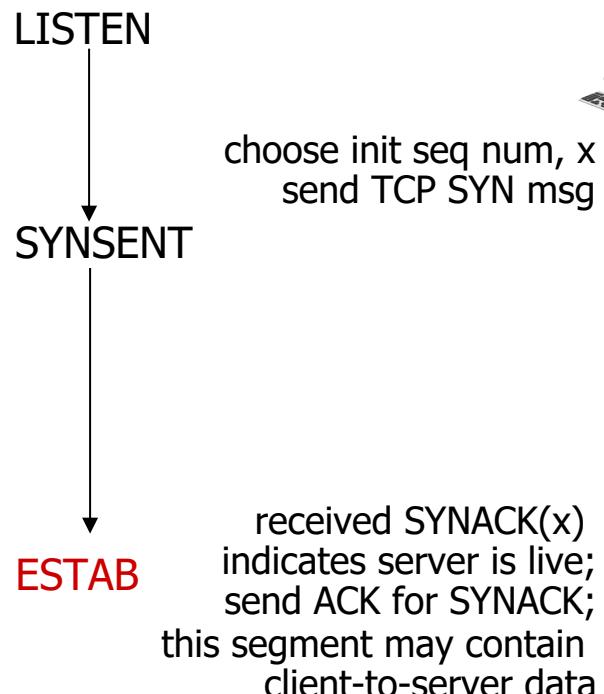
before exchanging data, sender/receiver “handshake”:

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters (e.g., starting seq #s)

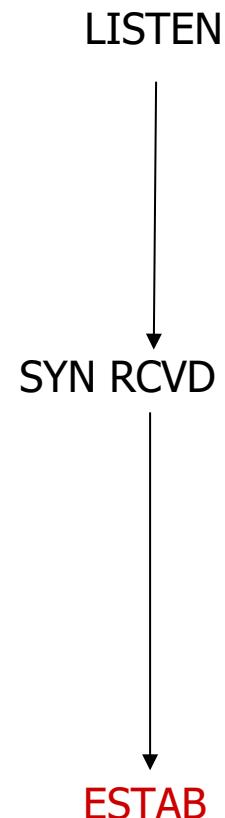


# TCP 3-Way Handshake

## Client State

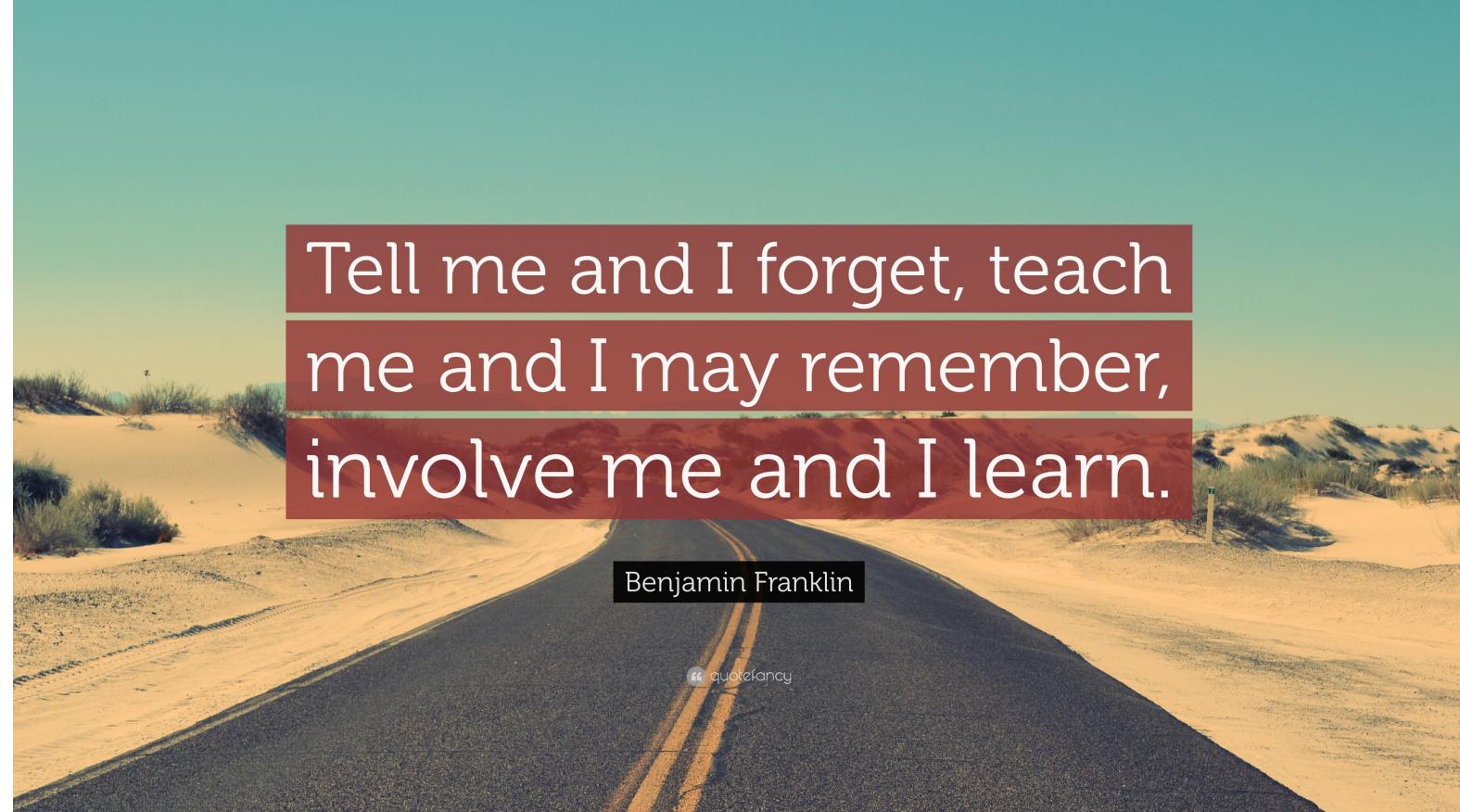


## Server State



# Closing A TCP 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
- simultaneous FIN exchanges can be handled



Tell me and I forget, teach  
me and I may remember,  
involve me and I learn.

Benjamin Franklin

“ quotefancy

## Principles of Congestion Control



# Principles of Congestion Control

## Congestion:

- informally: “too many sources sending too much data too fast for *network* to handle”
- manifestations:
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)
- different from flow control!
- a top-10 problem!



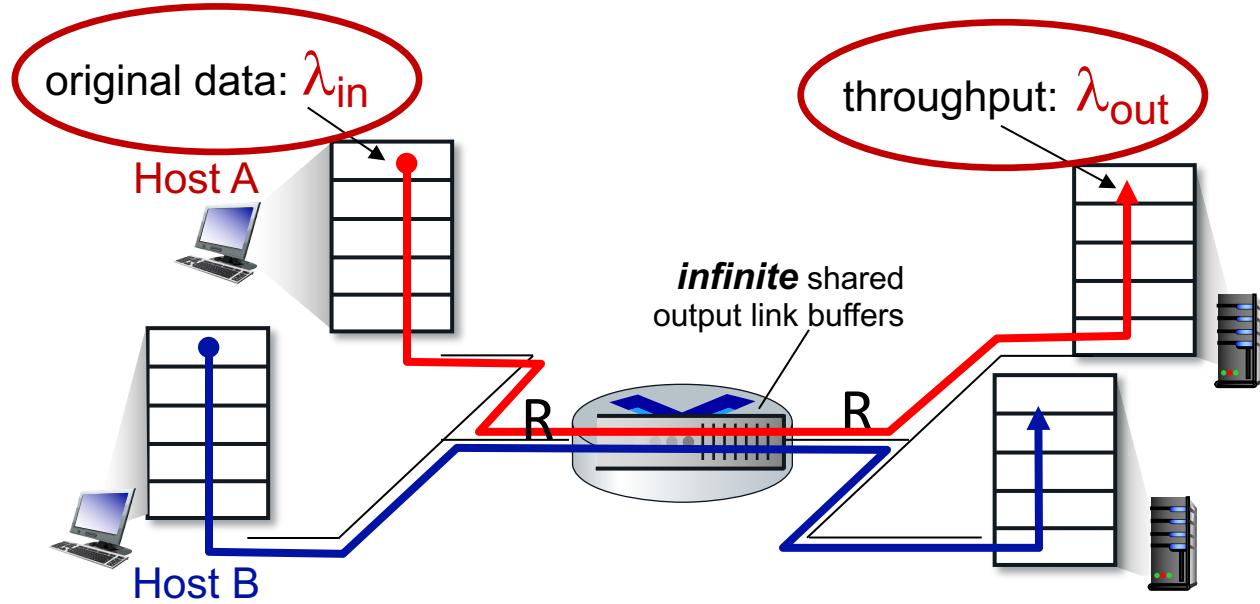
**congestion control:**  
too many senders,  
sending too fast

**flow control:** one sender  
too fast for one receiver

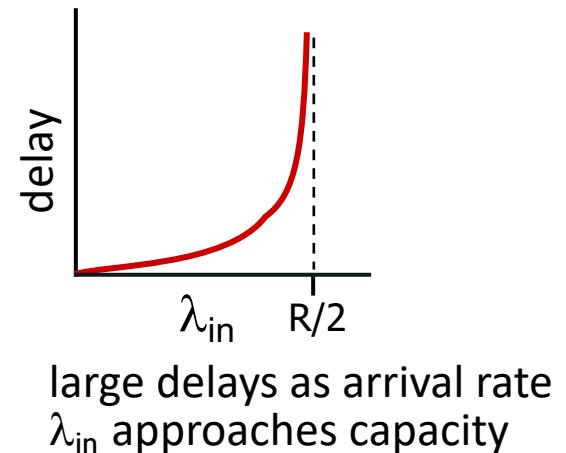
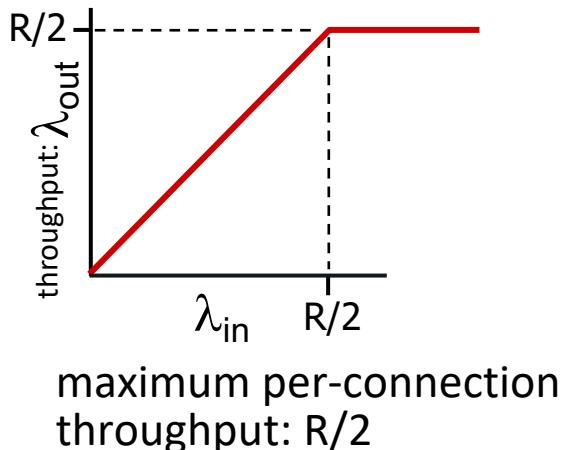
# Causes/Costs of Congestion: Ideal Scenario

Simplest scenario:

- one router, infinite buffers
- input, output link capacity:  $R$
- two flows
- no retransmissions needed



**Q:** What happens as arrival rate  $\lambda_{in}$  approaches  $R/2$ ?

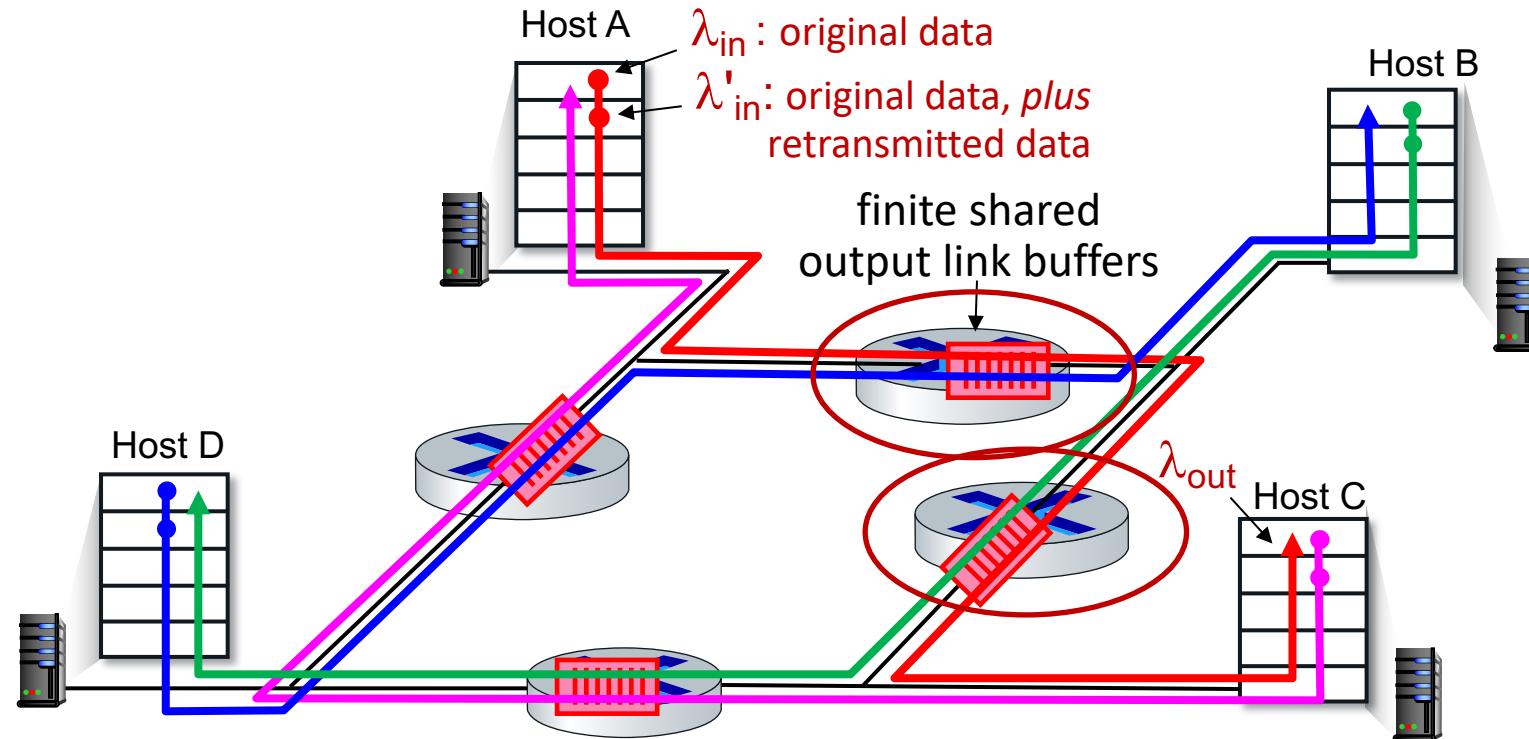


# Causes/Costs of Congestion: Practice

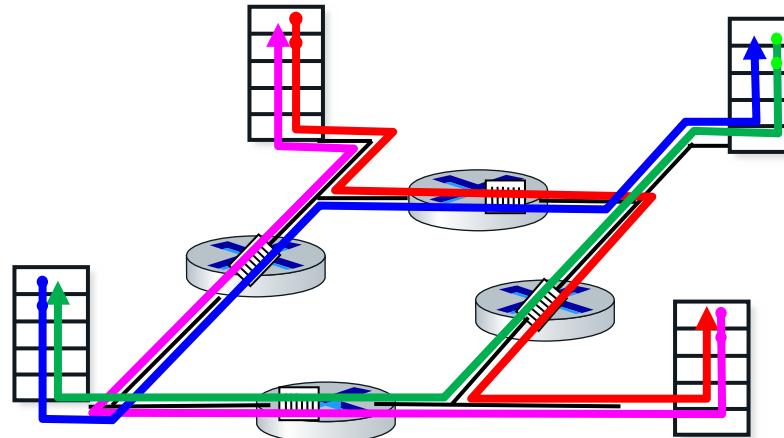
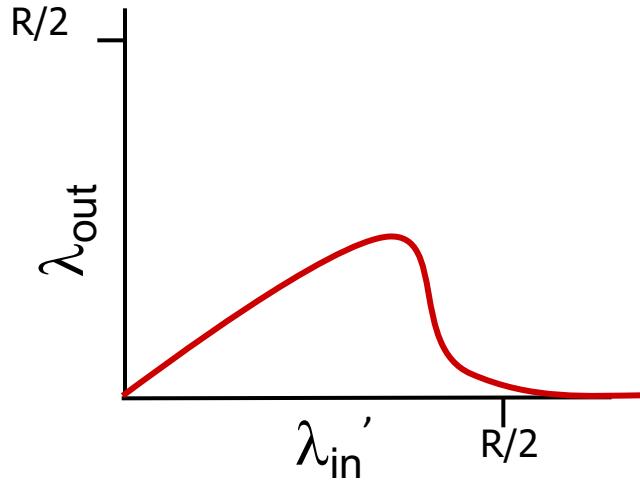
- four senders
- multi-hop paths
- timeout/retransmit

**Q:** what happens as  $\lambda_{in}$  and  $\lambda'_{in}$  increase ?

**A:** as red  $\lambda'_{in}$  increases, all arriving blue pkts at upper queue are dropped, blue throughput  $\rightarrow 0$



# Causes/Costs of Congestion: Practice



another “cost” of congestion:

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

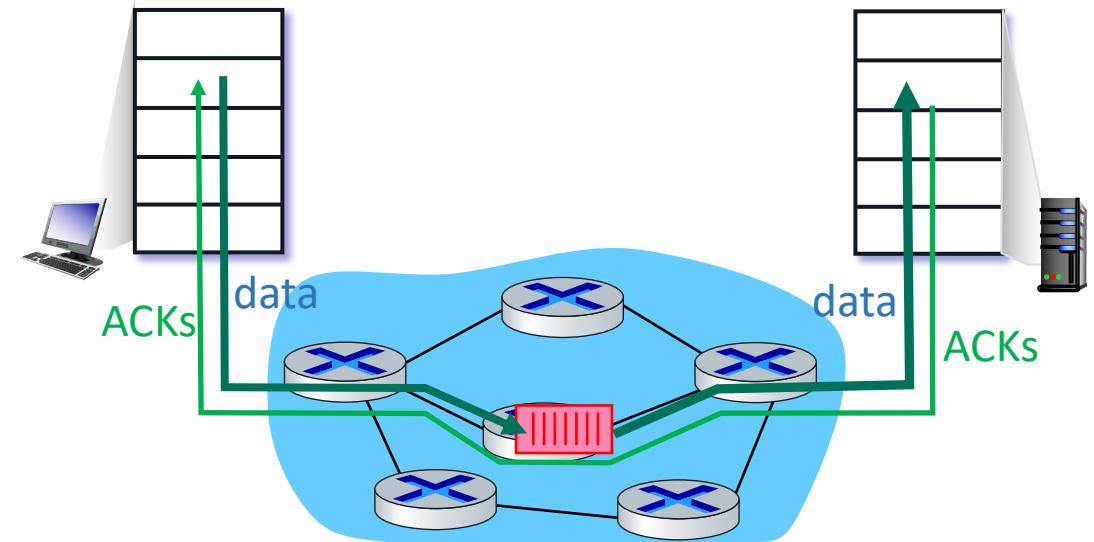
# Approaches Towards Congestion Control

## End-end congestion control:

no explicit feedback from network

congestion *inferred* from observed loss, delay

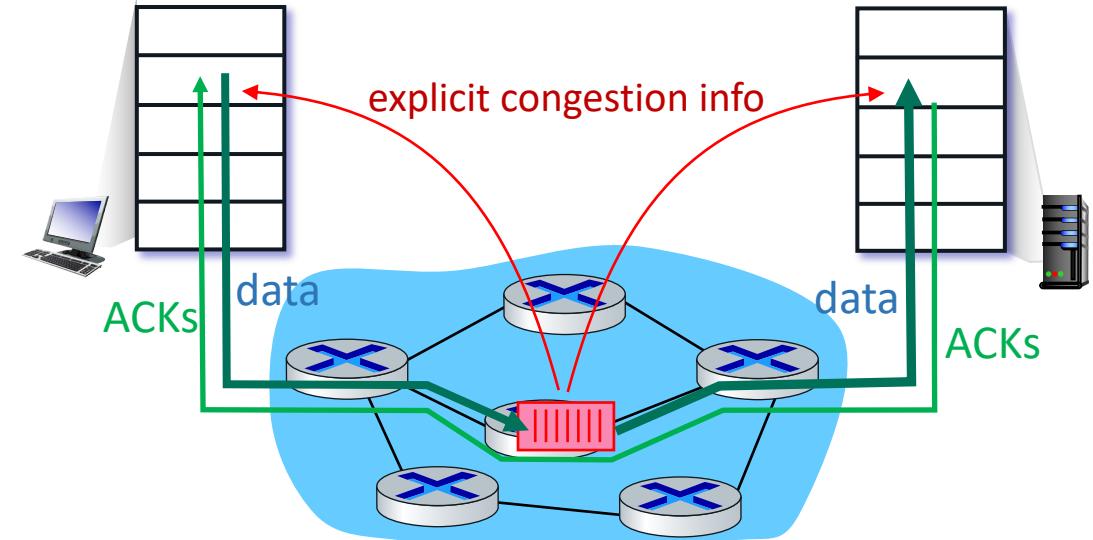
- approach taken by TCP

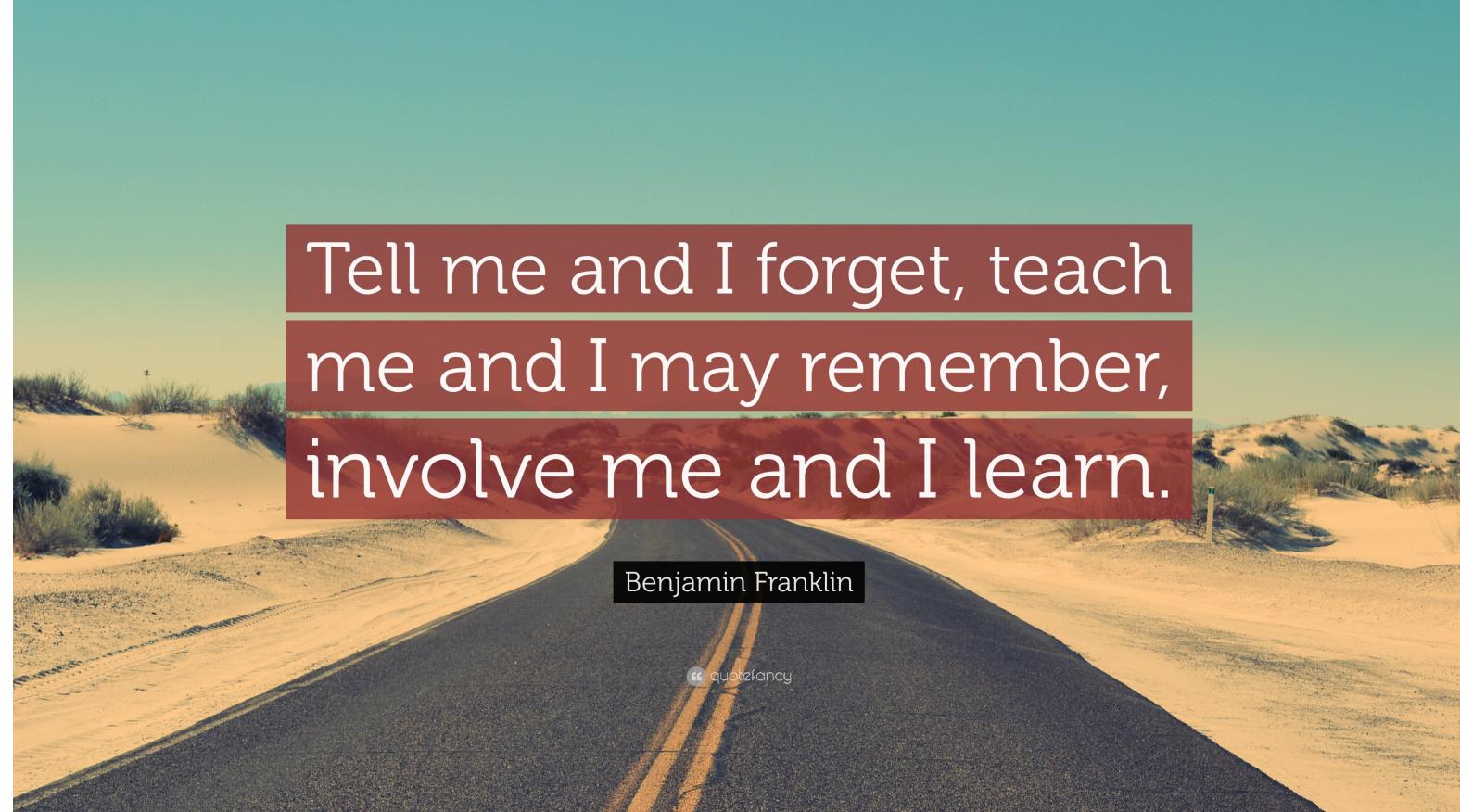


## Network-assisted congestion control:

- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate

TCP ECN, ATM, DECbit protocols





Tell me and I forget, teach  
me and I may remember,  
involve me and I learn.

Benjamin Franklin

“ quotefancy

# TCP Congestion Control



# TCP Congestion Control: AIMD

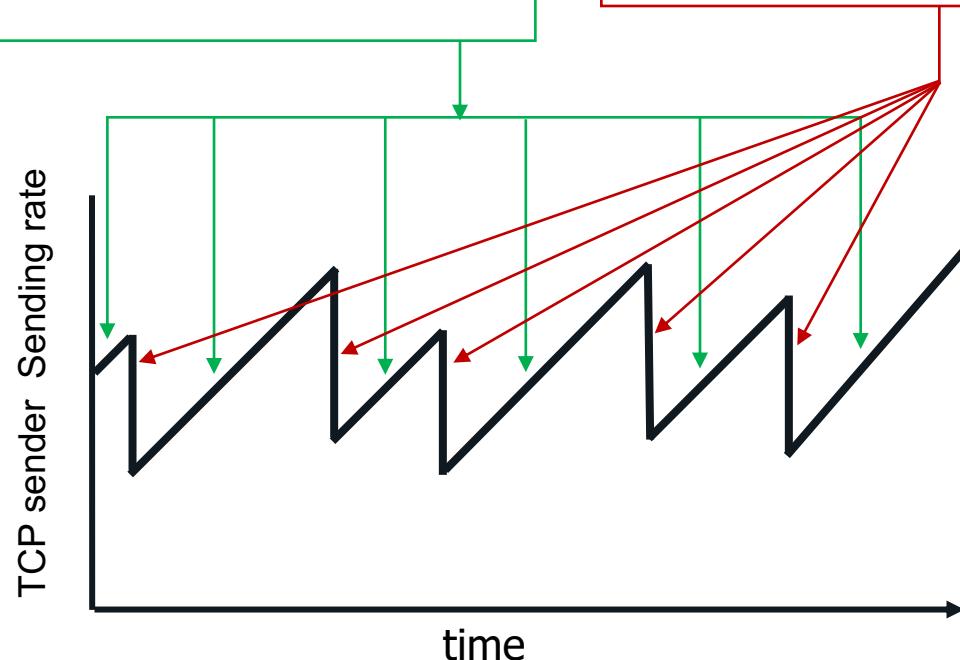
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

## Additive Increase

increase sending rate by 1 maximum segment size every RTT until loss detected

## Multiplicative Decrease

cut sending rate in half at each loss event



**AIMD** sawtooth behavior: *probing* for bandwidth

# TCP Congestion Control: AIMD

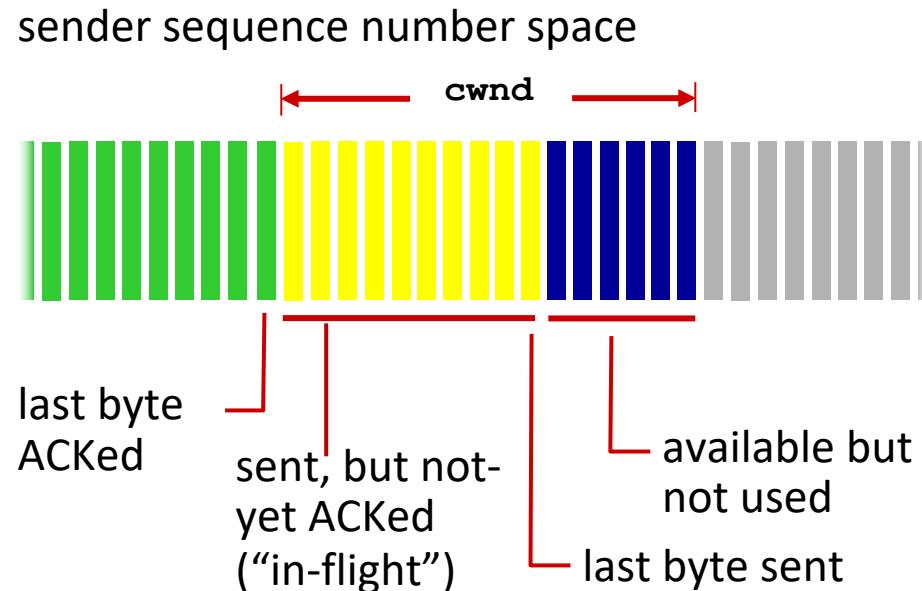
*Multiplicative decrease* detail: sending rate is

- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

Why AIMD?

- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - optimize congested flow rates network wide!
  - have desirable stability properties

# TCP Congestion Control: Details



TCP sending behavior:

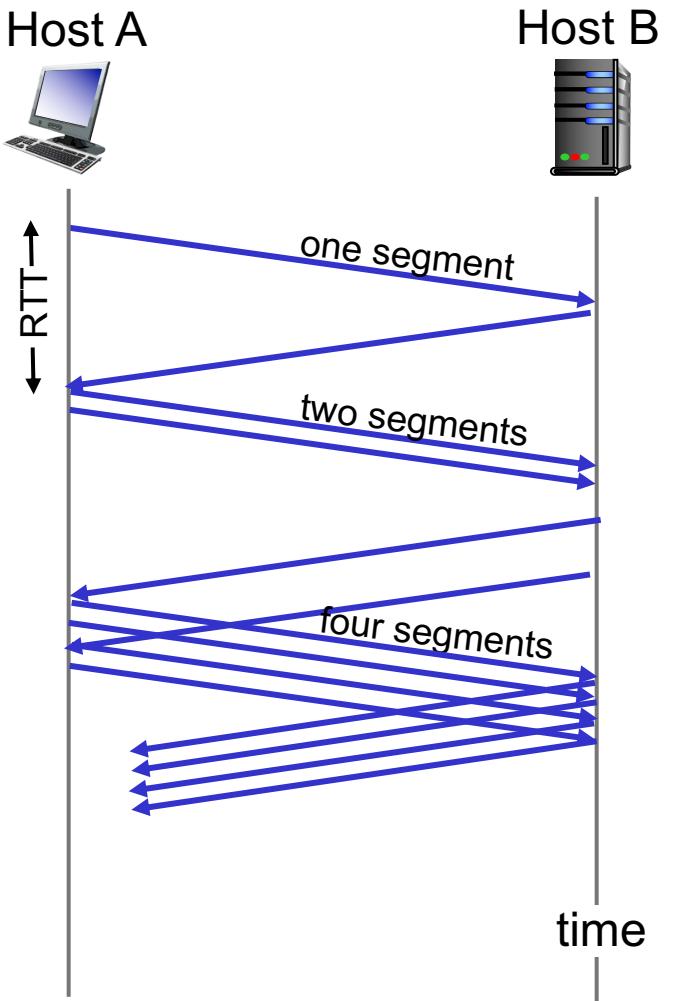
- *roughly*: send cwnd bytes, wait RTT for ACKS, then send more bytes

$$\text{TCP rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

- TCP sender limits transmission:  $\text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}$
- cwnd is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

# 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



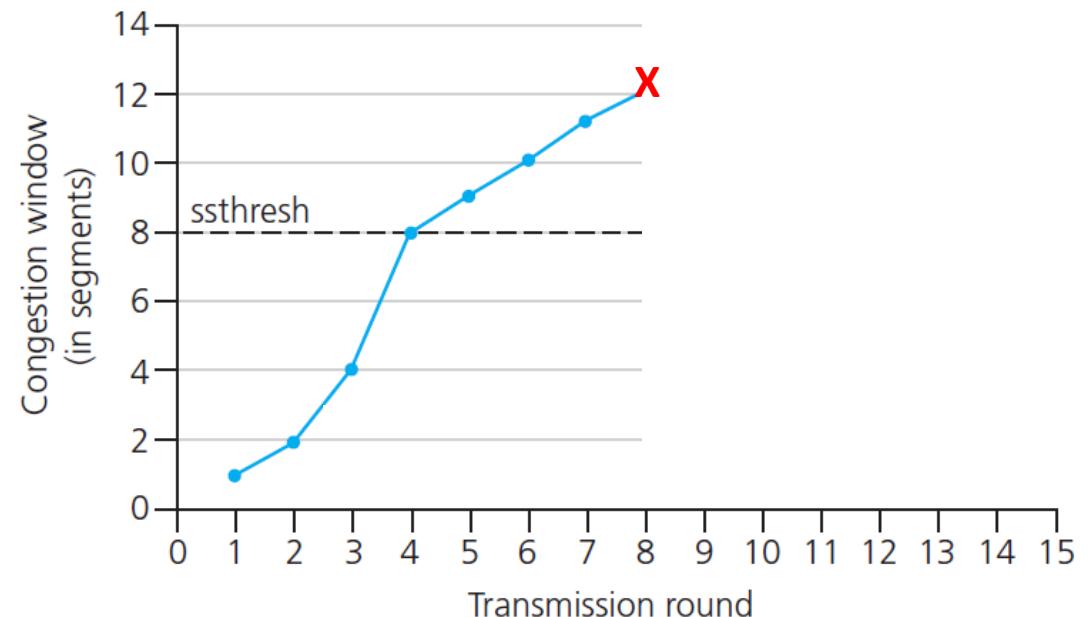
# TCP Slow Start to Congestion Avoidance

**Q:** when should the exponential increase switch to linear?

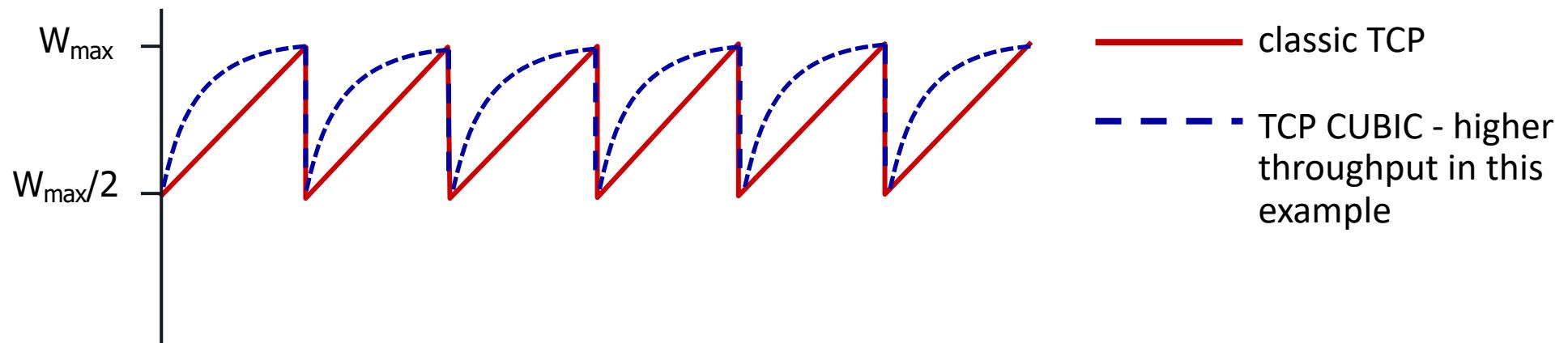
**A:** when **cwnd** gets to 1/2 of its value before timeout.

## Implementation:

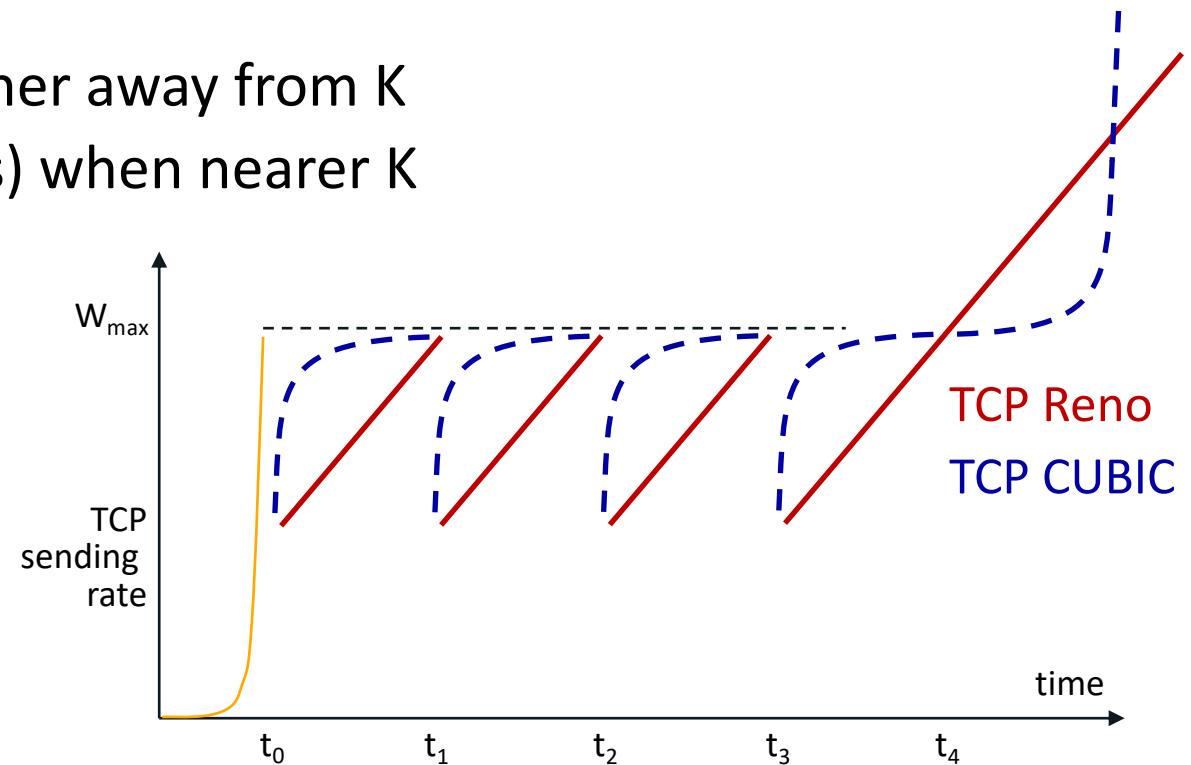
- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event



- Is there a better way than AIMD to “probe” for usable bandwidth?
- Insight/intuition:
  - $W_{\max}$ : sending rate at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn’t changed much
  - after cutting rate/window in half on loss, initially ramp to  $W_{\max}$  *faster*, but then approach  $W_{\max}$  more *slowly*



- K: point in time when TCP window size will reach  $W_{\max}$ 
  - K itself is tuneable
- increase W as a function of the *cube* of the distance between current time and K
  - larger increases when further away from K
  - smaller increases (cautious) when nearer K
- TCP CUBIC default in Linux, most popular TCP for popular Web servers



# TCP Network-Assisted Congestion Control

TCP deployments often implement *network-assisted* congestion control:

- two bits in IP header (ToS field) marked *by network router* to indicate congestion
  - *policy* to determine marking chosen by network operator
- congestion indication carried to destination
- destination sets ECE bit on ACK segment to notify sender of congestion
- involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)

