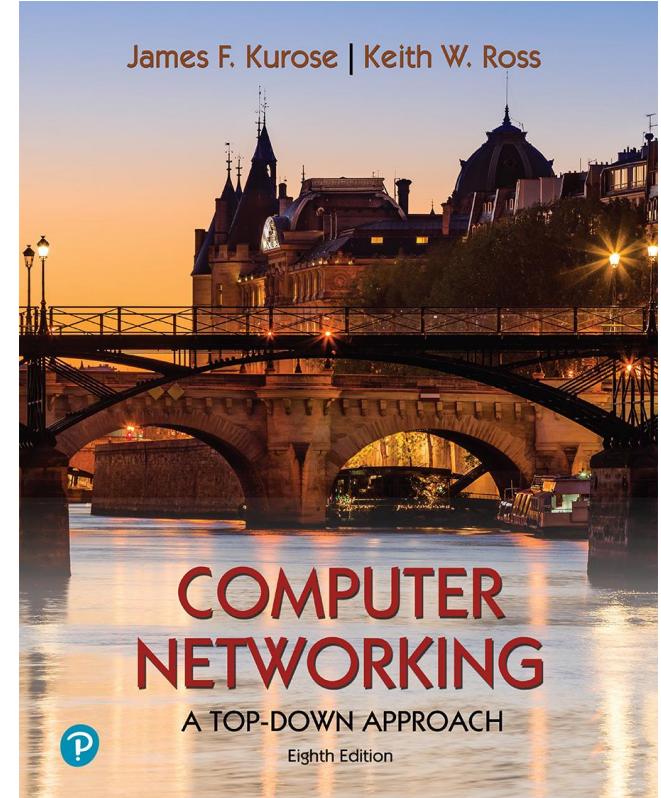


# Chapter 3

# Transport Layer



*Computer Networking: A  
Top-Down Approach*  
8<sup>th</sup> edition  
Jim Kurose, Keith Ross  
Pearson, 2020

# Transport layer: overview

*Our goal:*

- 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
    - TCP congestion control

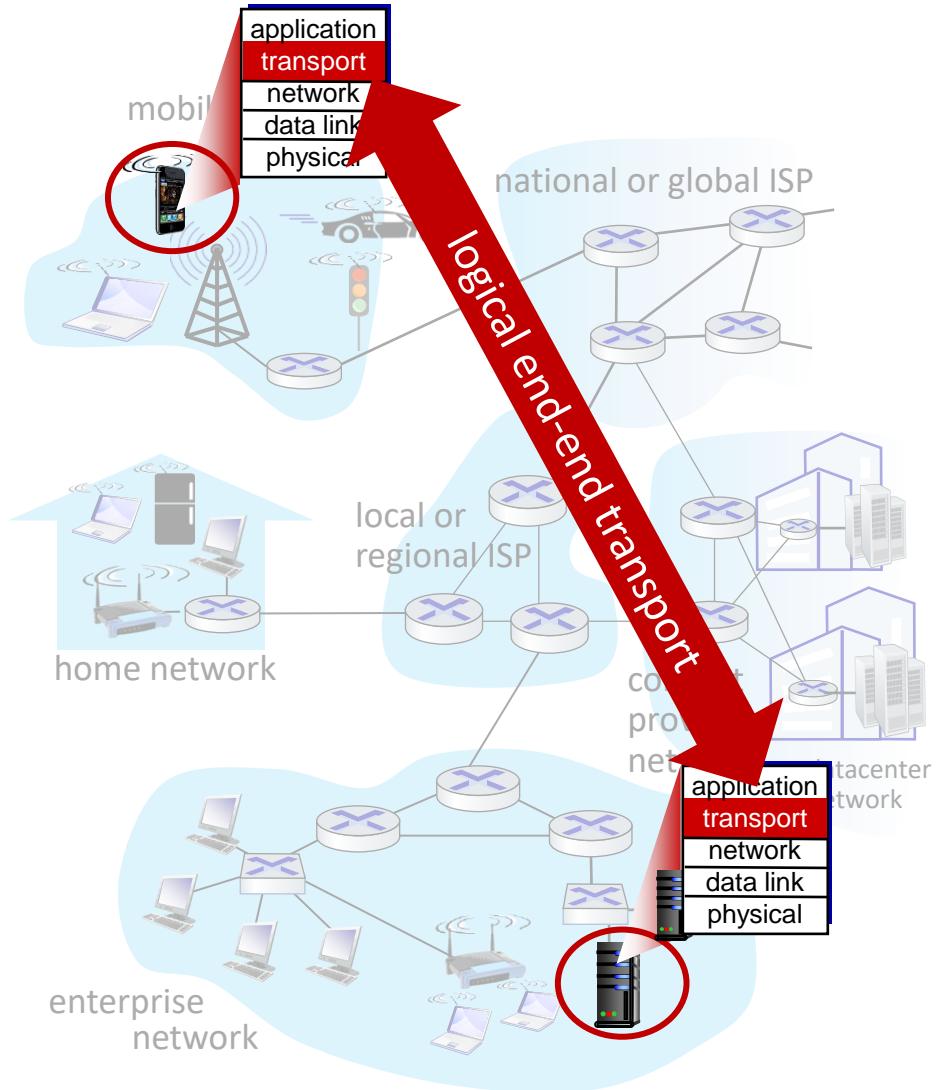
# Transport layer: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: **UDP**
- Principles of reliable data transfer
- Connection-oriented transport: **TCP**
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality



# Transport 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 vs. network layer services and protocols



*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 vs. network layer services and protocols

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

*household analogy:*

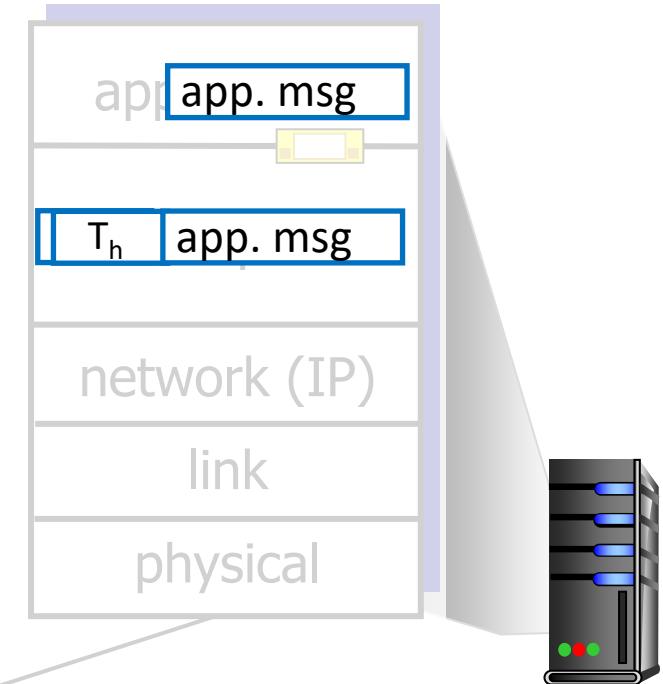
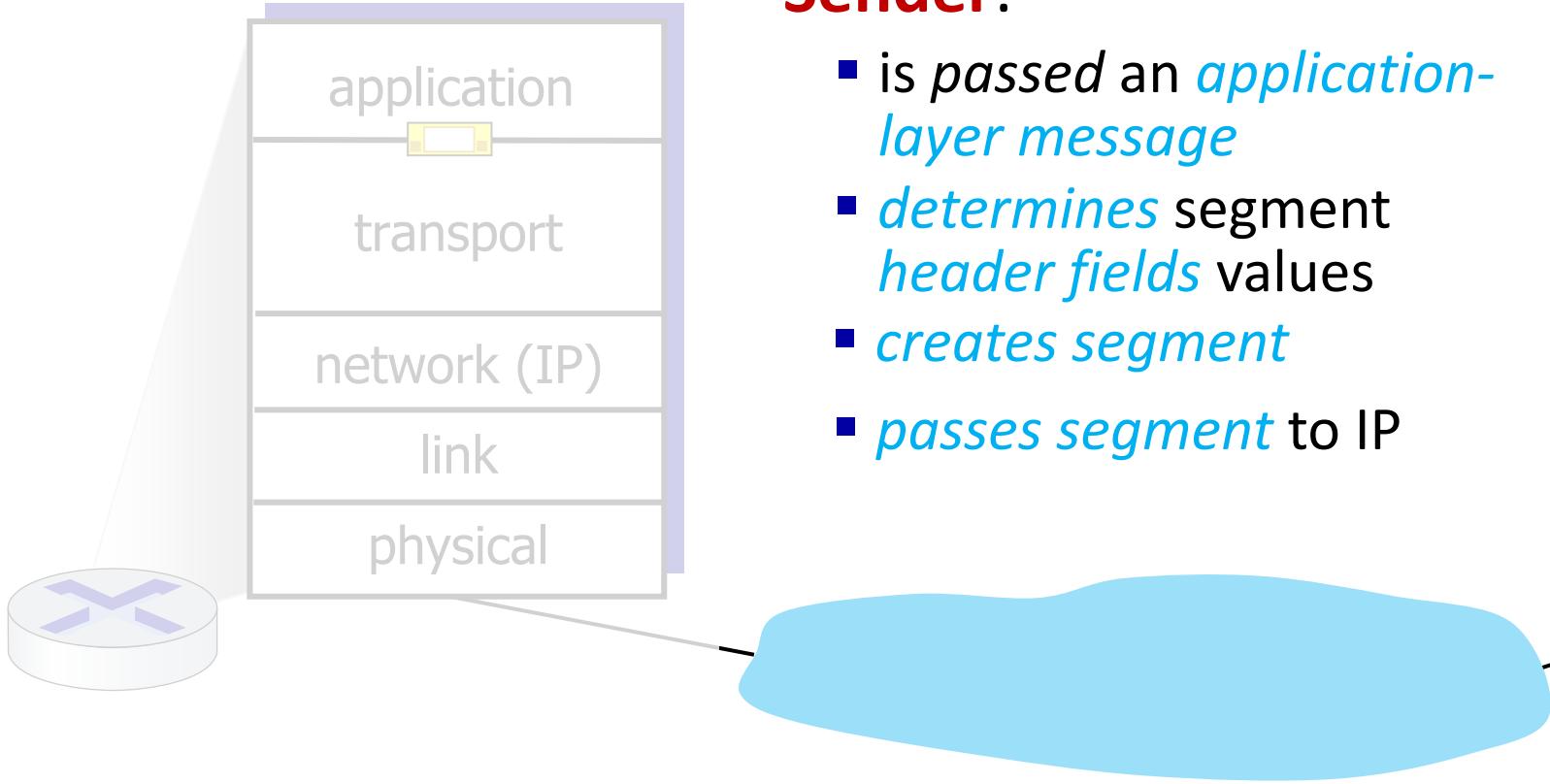
*12 kids in Ann's house sending letters to 12 kids in Bill's house:*

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

# Transport Layer Actions

## Sender:

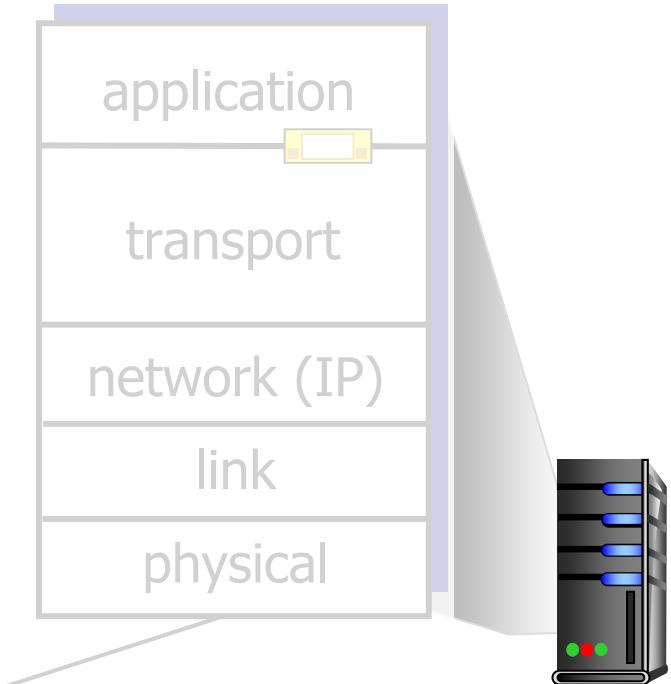
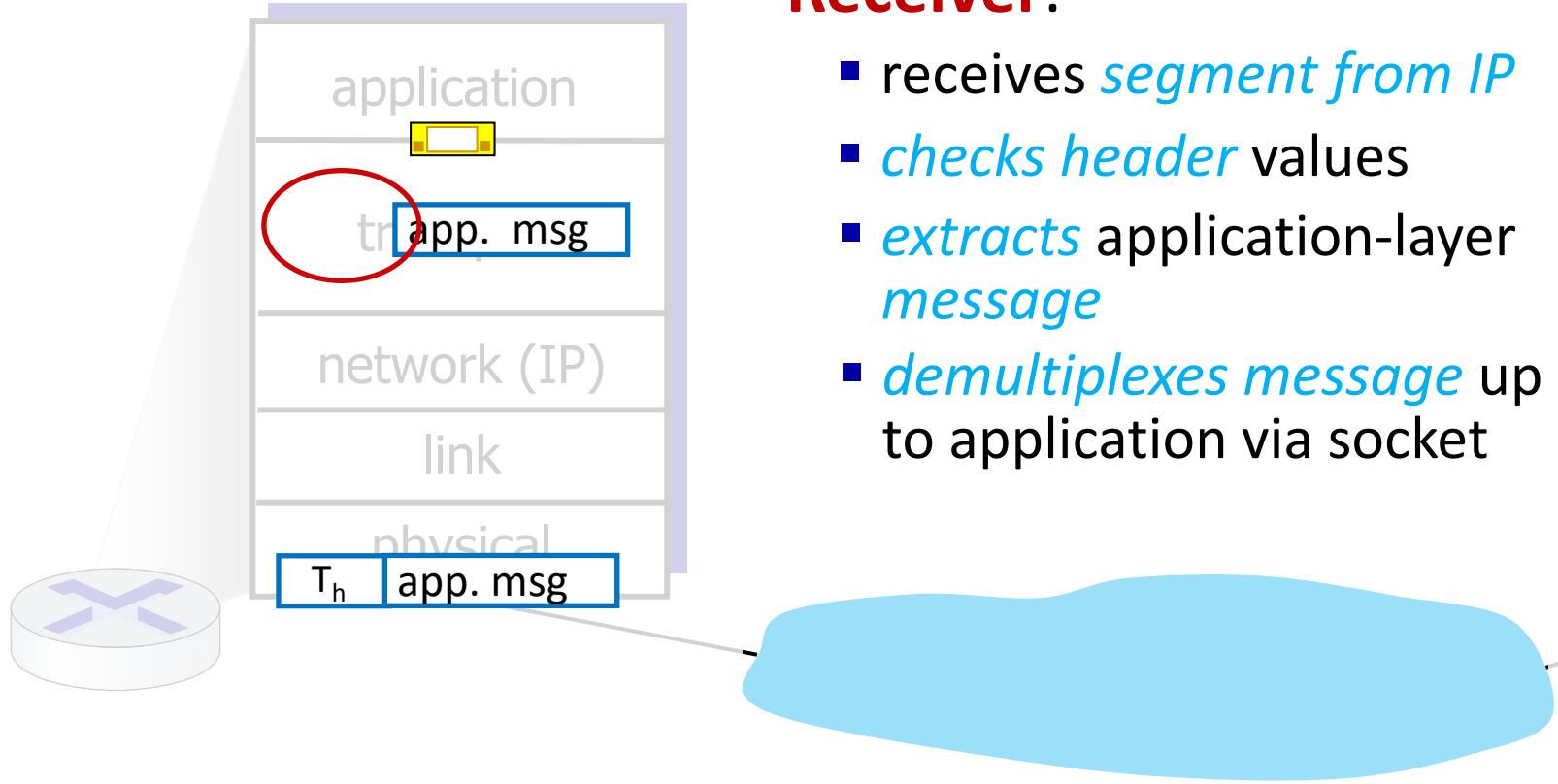
- is passed an *application-layer message*
- *determines segment header fields values*
- *creates segment*
- *passes segment* to IP



# Transport Layer Actions

## Receiver:

- receives *segment from IP*
- *checks header* values
- *extracts application-layer message*
- *demultiplexes message* up to application via socket



# Two principal Internet transport protocols

## ■ TCP: Transmission Control Protocol

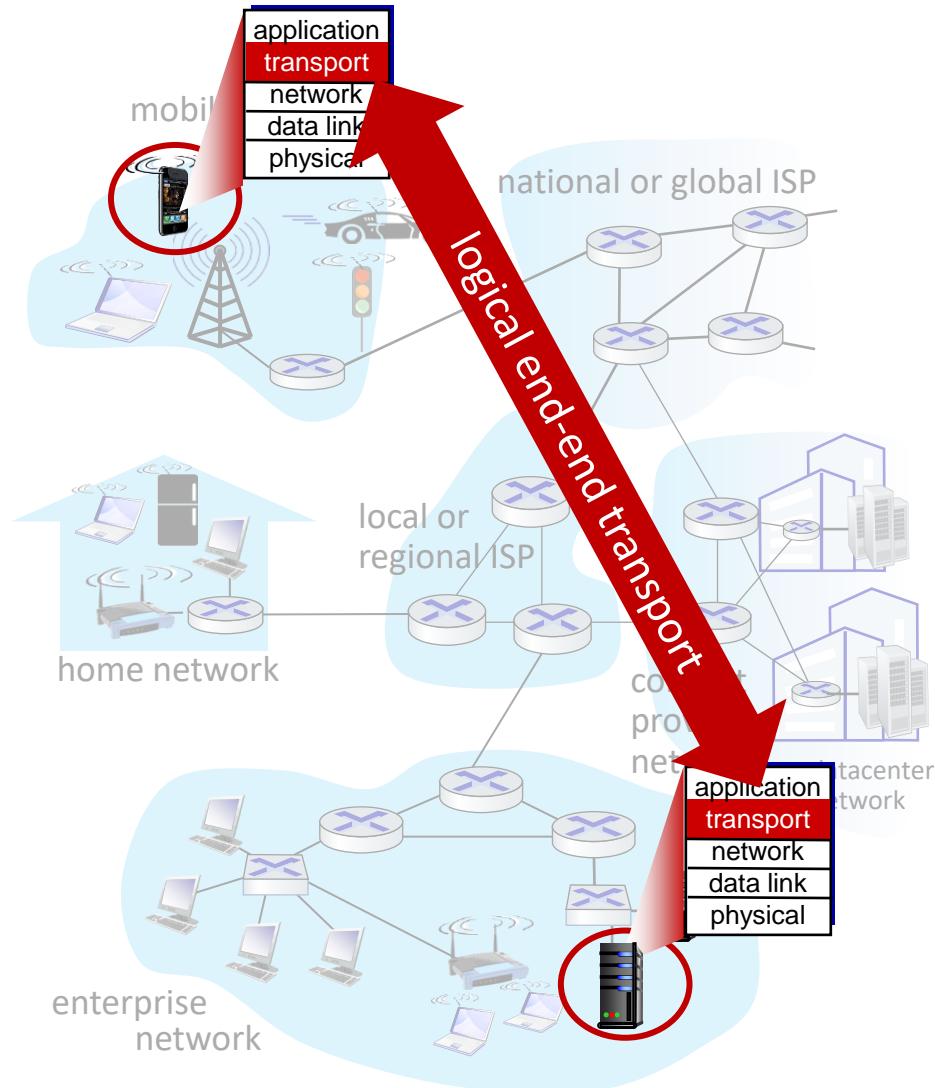
- reliable, in-order delivery
- congestion control
- flow control
- connection setup

## ■ UDP: User Datagram Protocol

- unreliable, unordered delivery
- no-frills extension of “best-effort” IP

## ■ *services not available:*

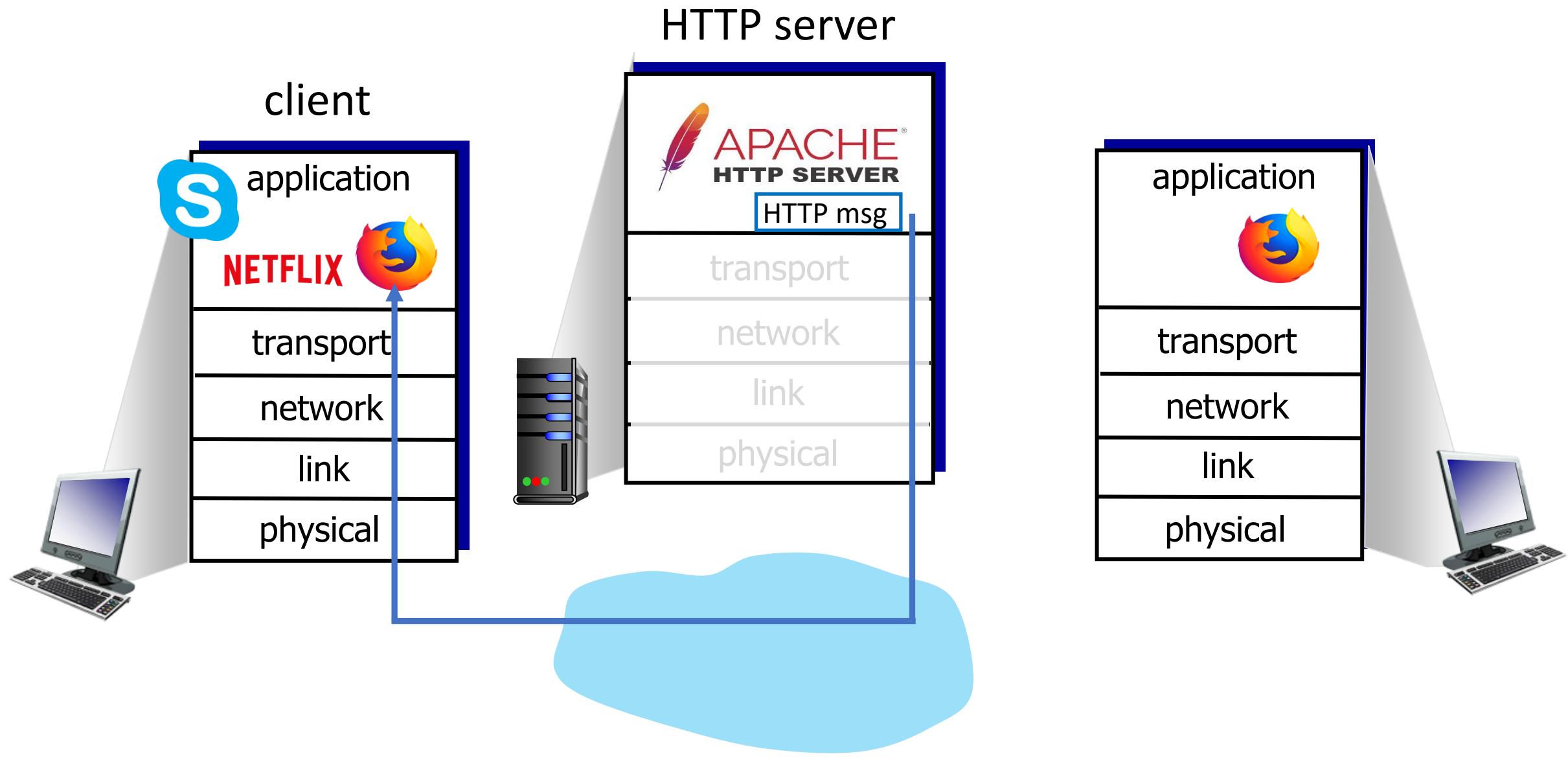
- *delay* guarantees
- *bandwidth* guarantees

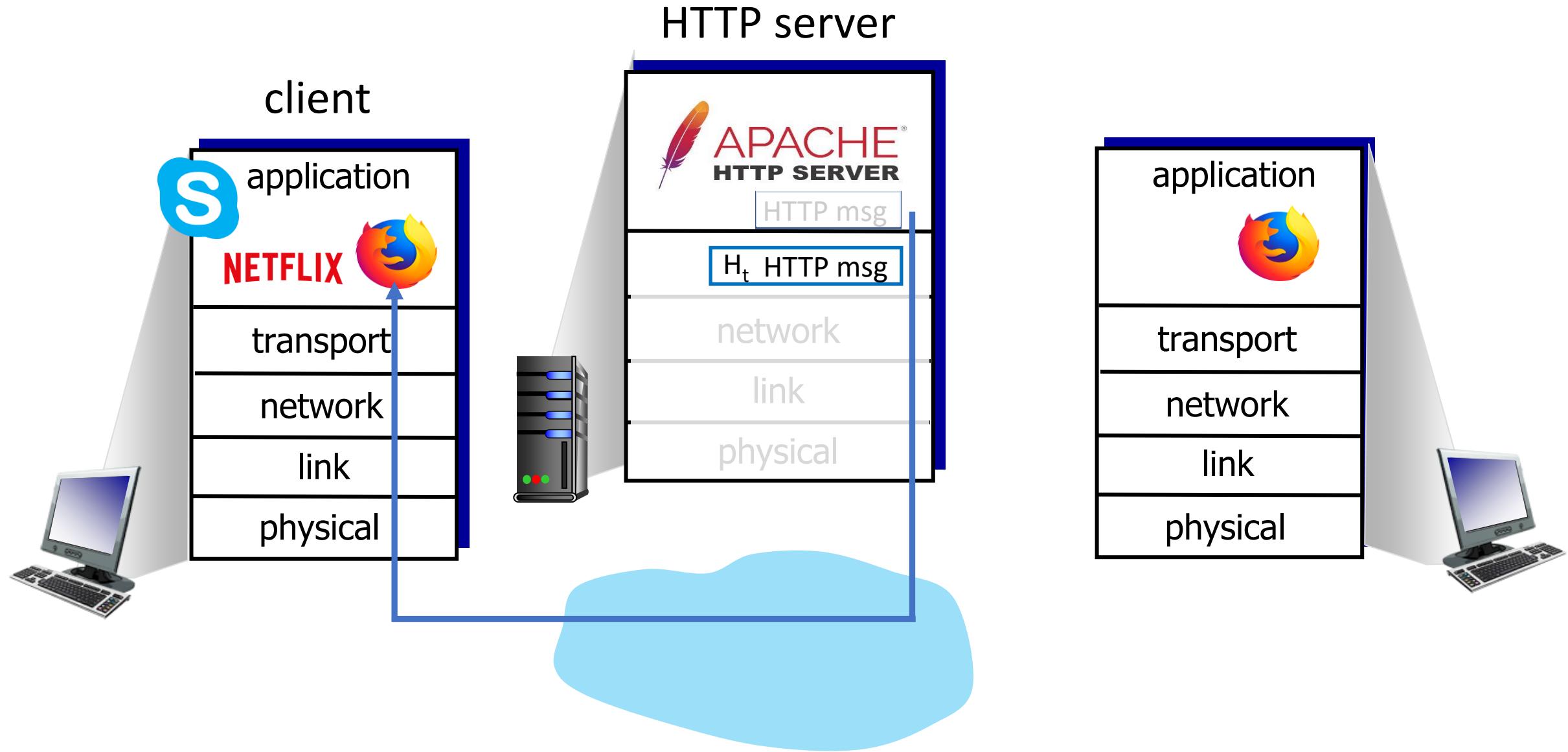


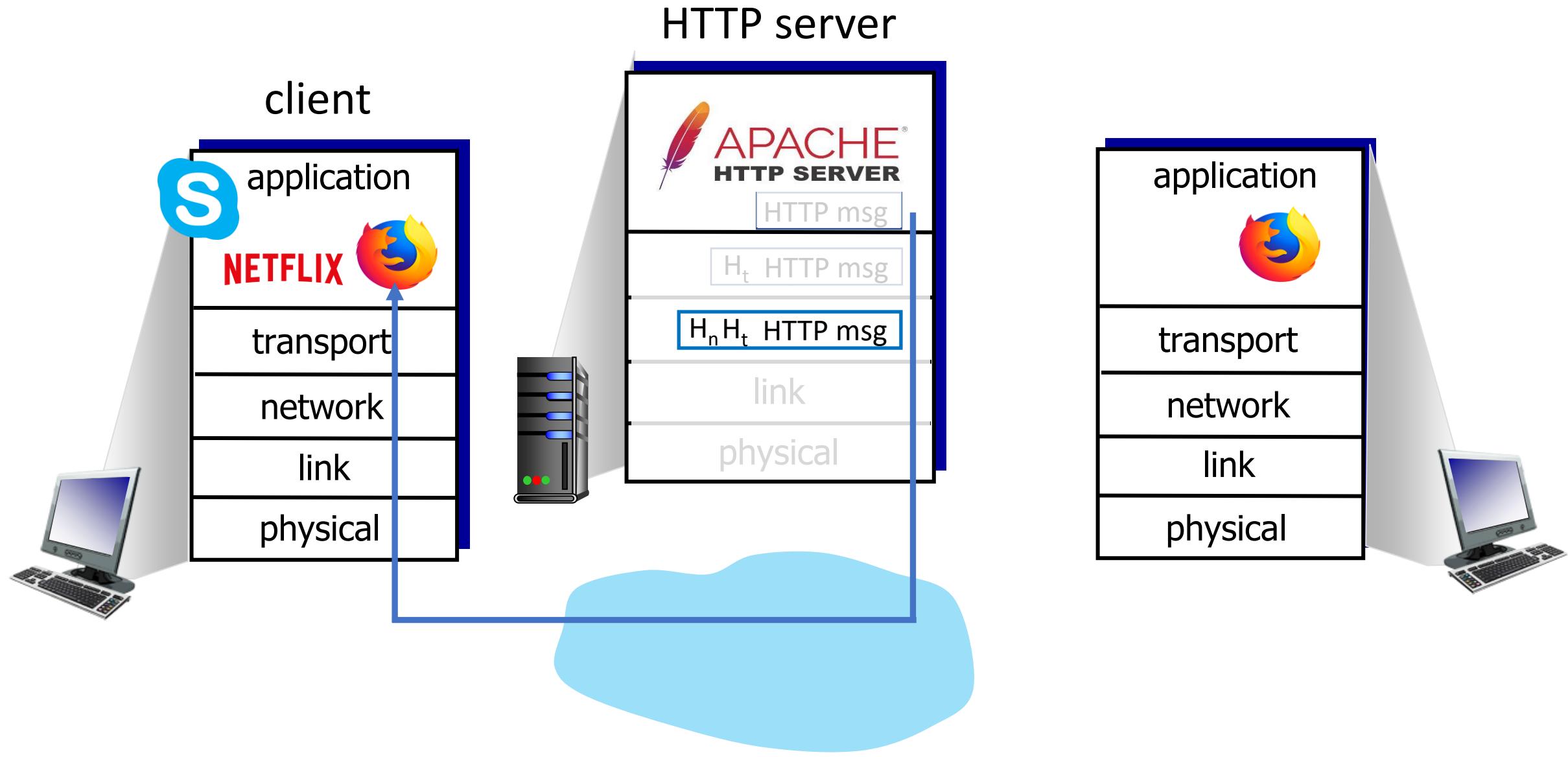
# Chapter 3: roadmap

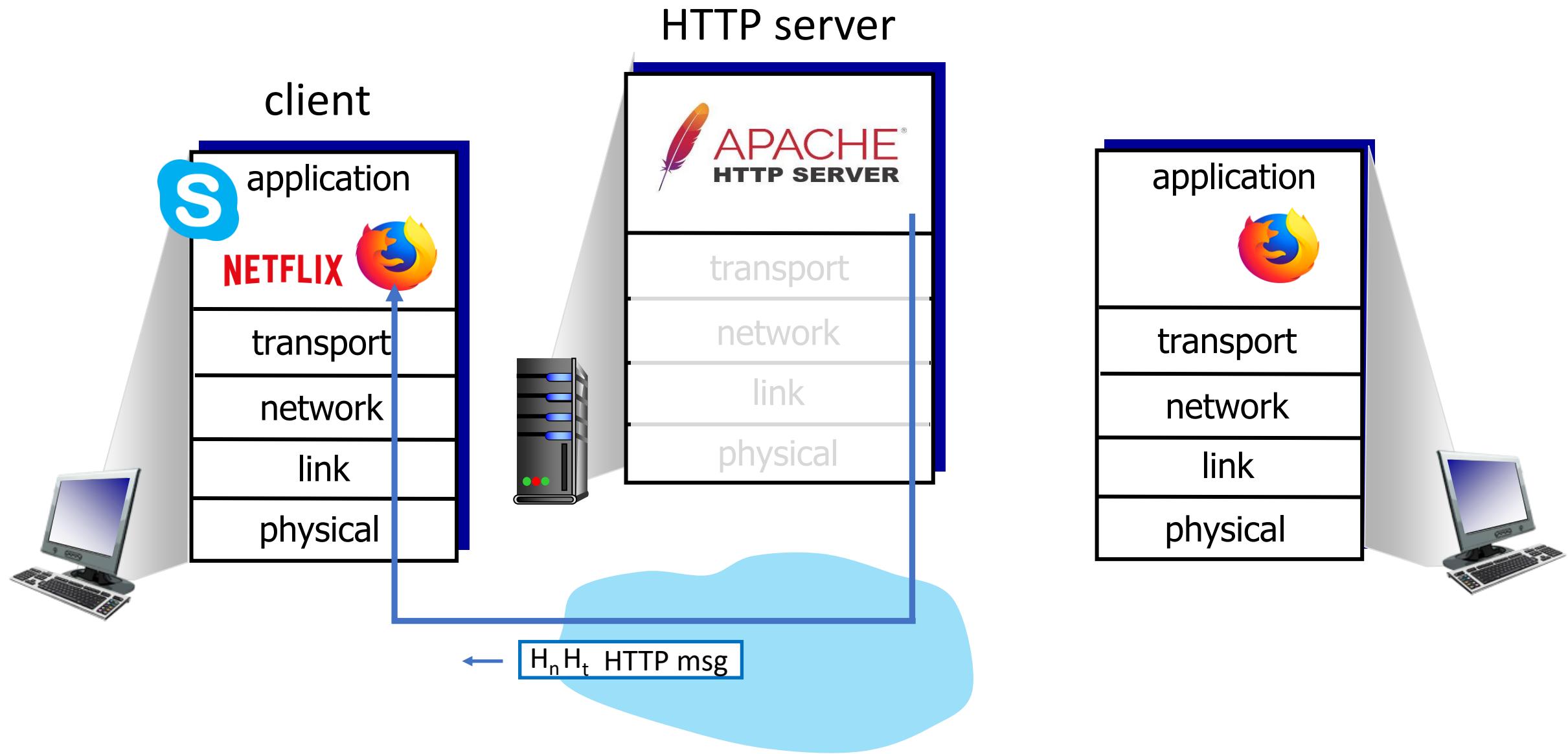
- Transport-layer services
- **Multiplexing and demultiplexing**
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

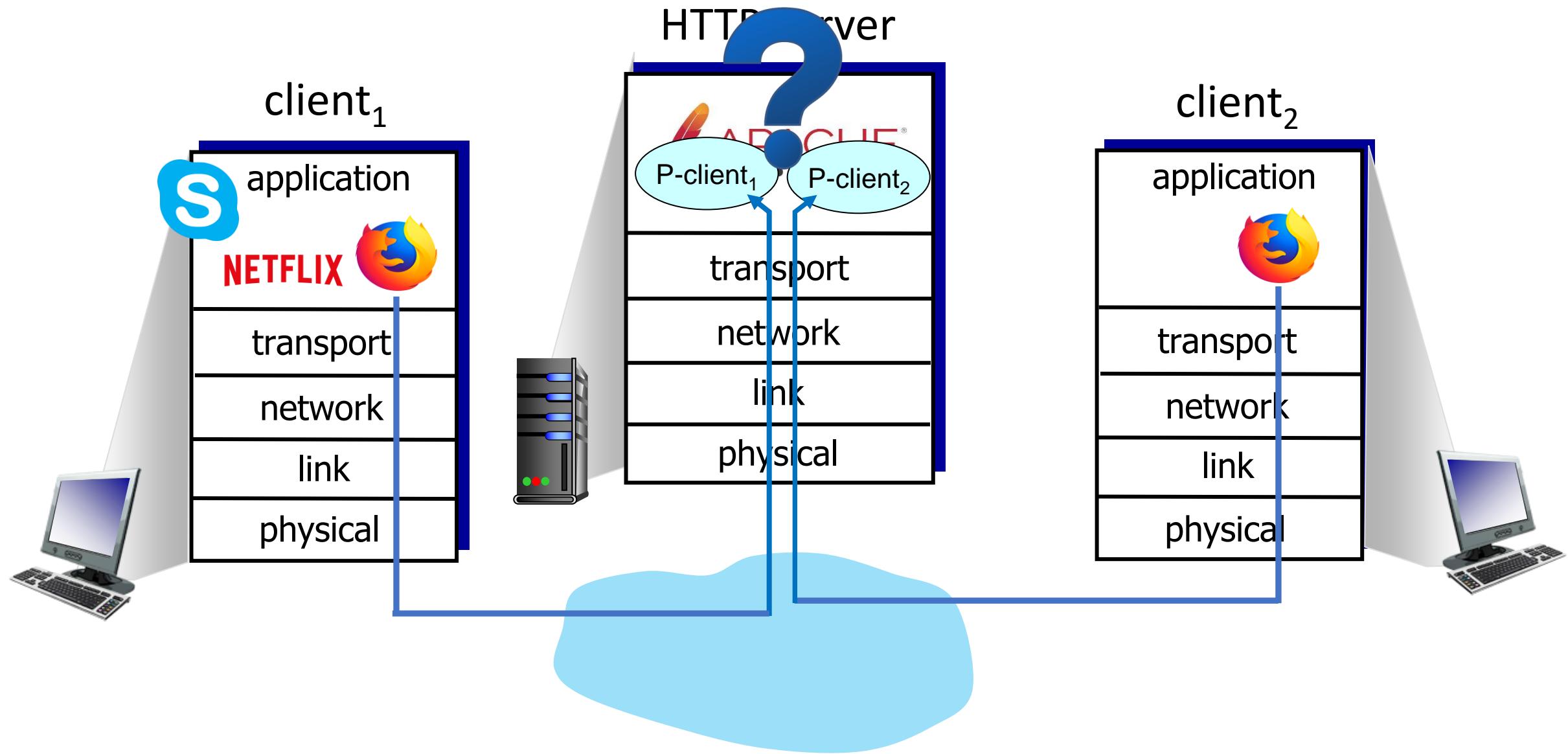












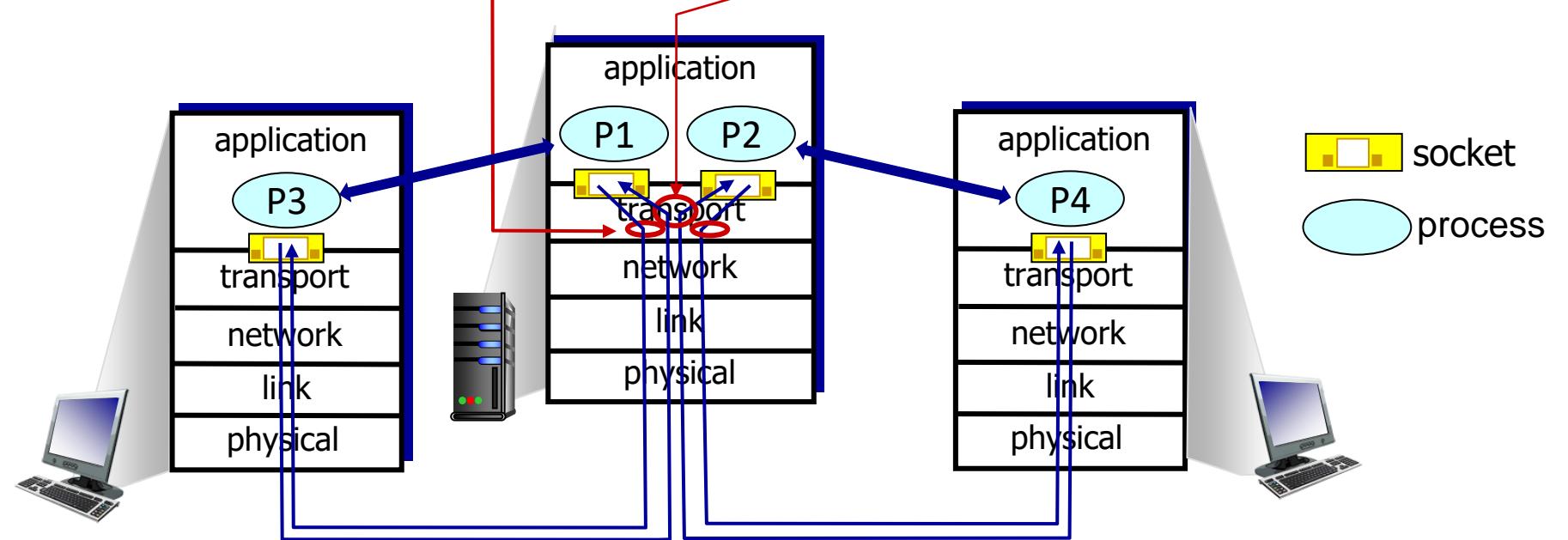
# Multiplexing/demultiplexing

*multiplexing at sender:*

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

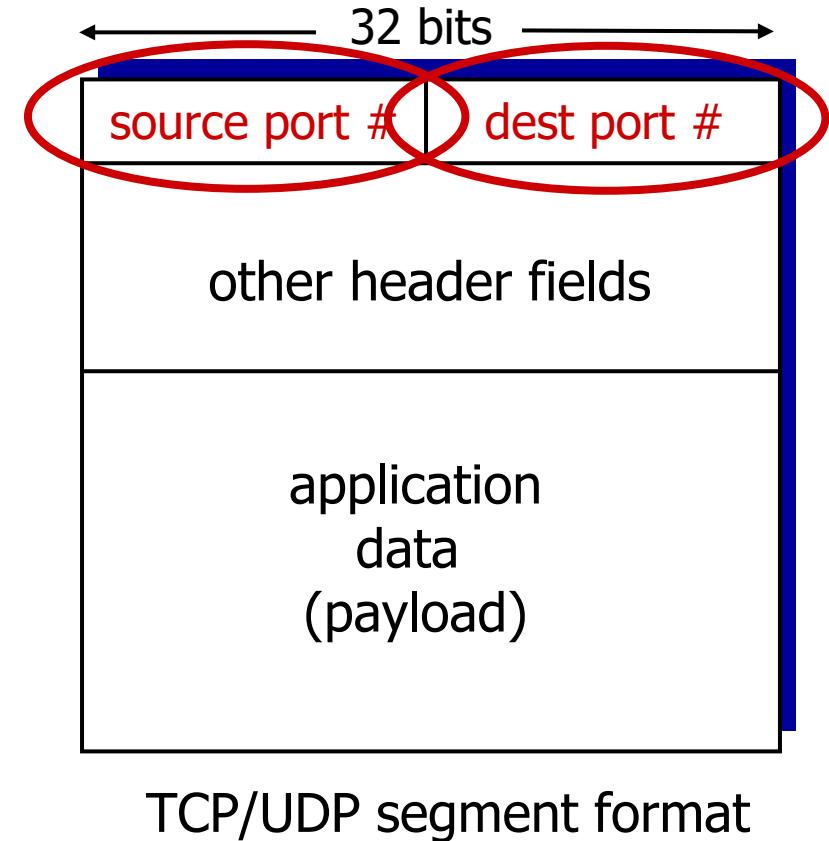
*demultiplexing at receiver:*

use header info to deliver received segments *to correct socket*



# How demultiplexing works

- host receives *IP datagrams*
  - each datagram has source 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*, host must specify *host-local port #*:

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

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

when receiving, host receives *UDP segment*:

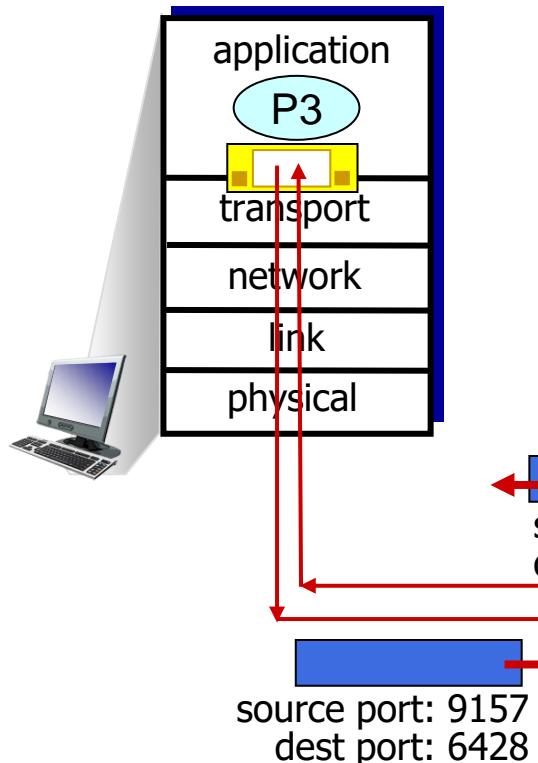
- checks *destination port #* in segment
- directs *UDP segment to socket* with that port #



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

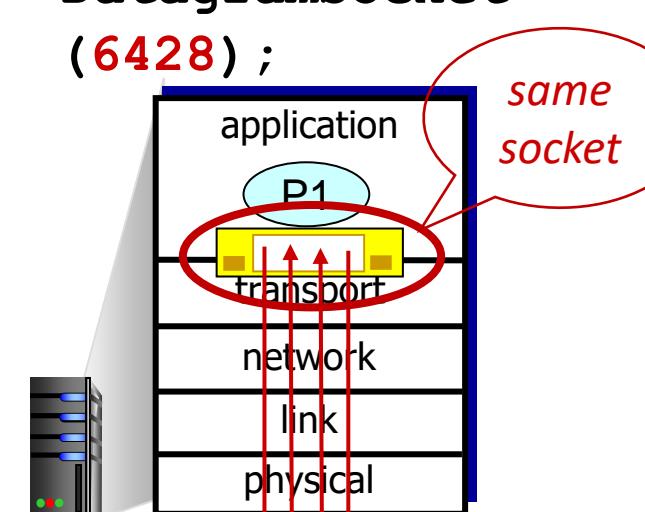
# Connectionless demultiplexing: an example

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

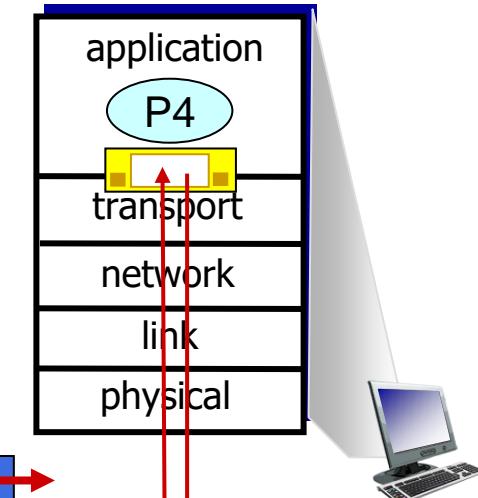


```
DatagramSocket  
serverSocket = new  
DatagramSocket  
(6428);
```

*same socket*



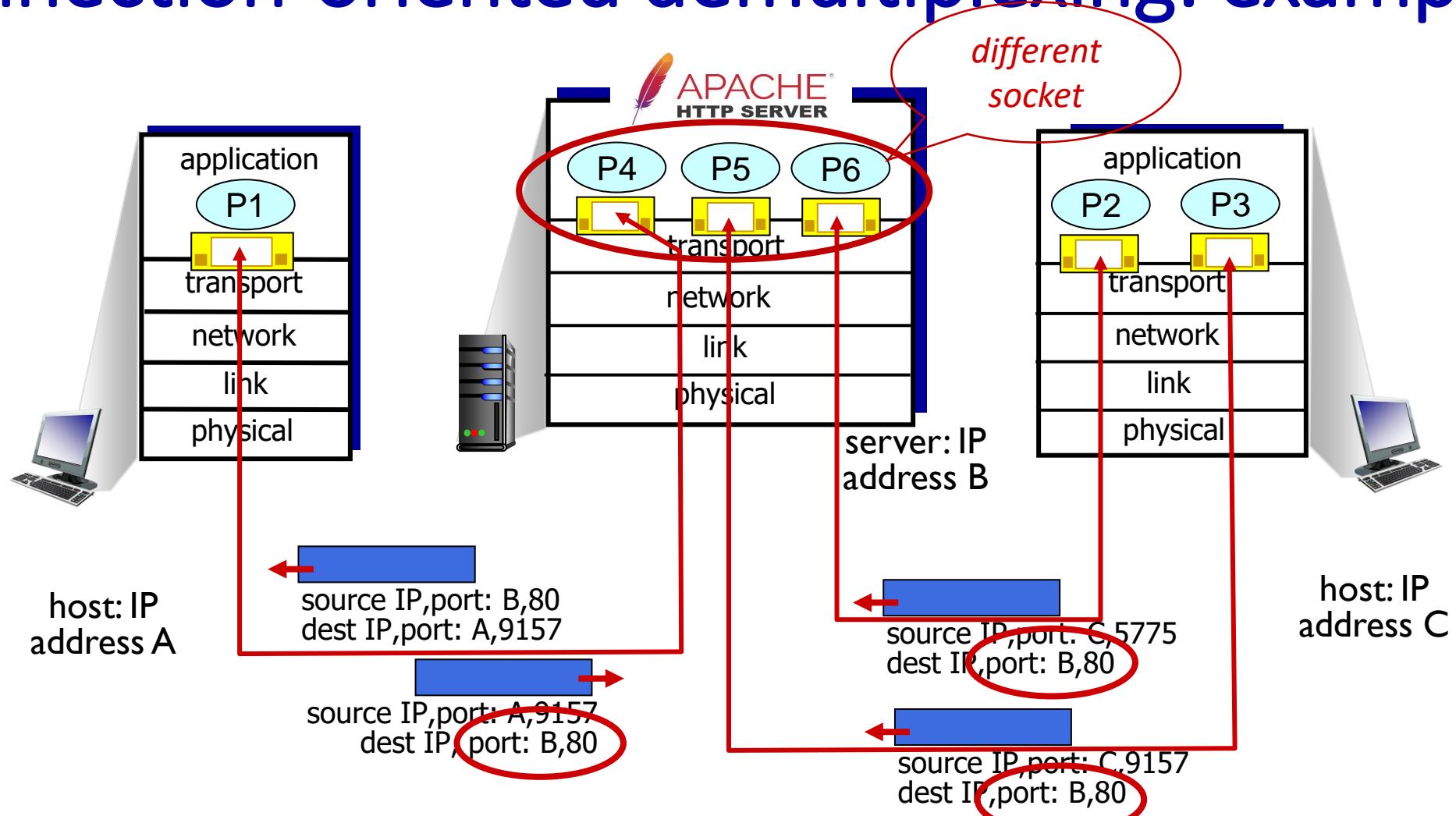
```
DatagramSocket mySocket1 =  
new DatagramSocket (5775);
```



# 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

# 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*

# Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- **Connectionless transport: UDP**
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality



# 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
- *overhead*: 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 used for:
  - 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

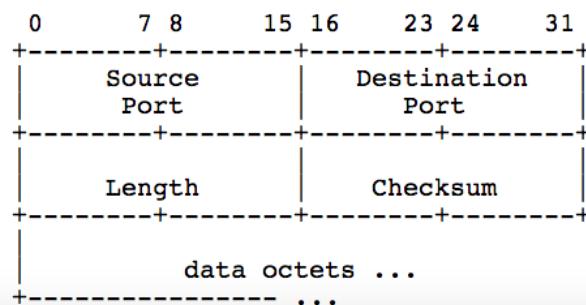
## User Datagram Protocol

### Introduction

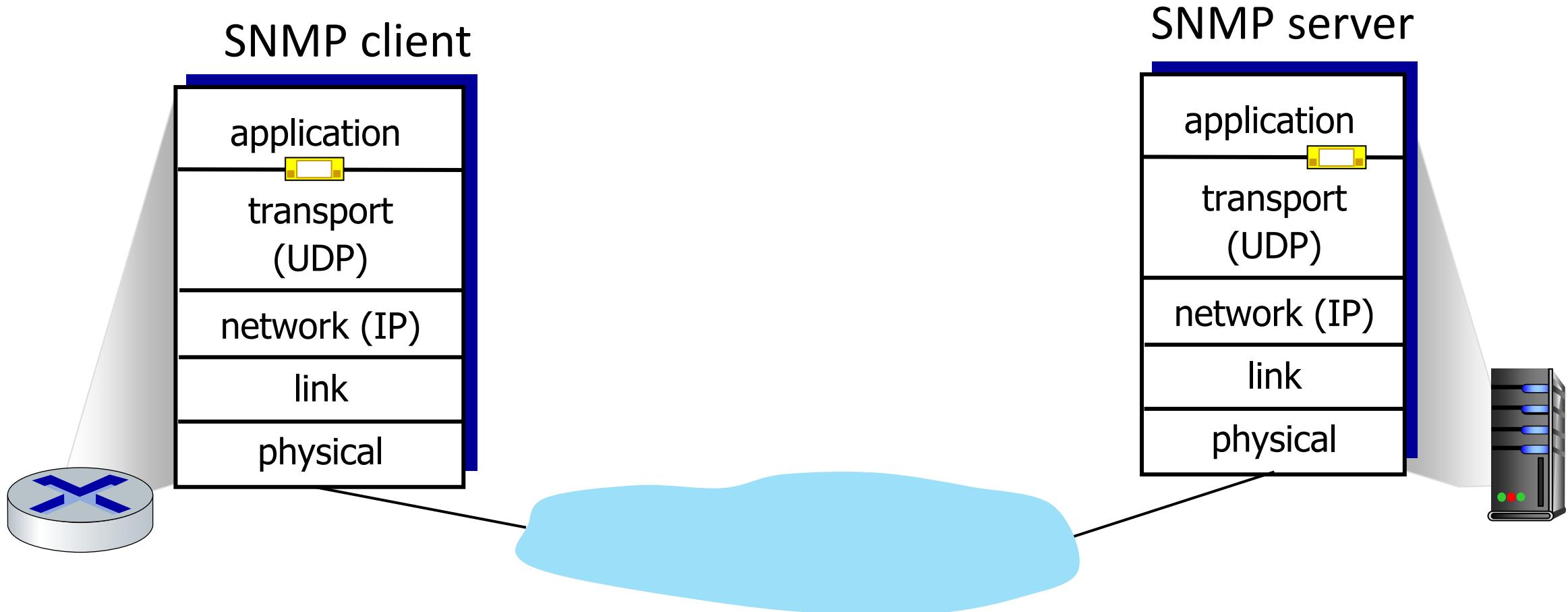
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

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].

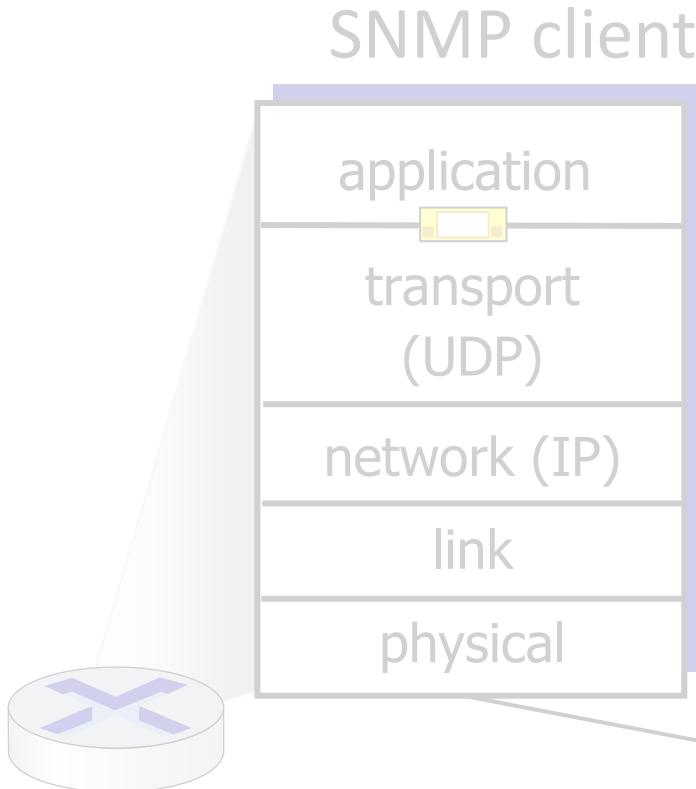
### Format



# UDP: Transport Layer Actions



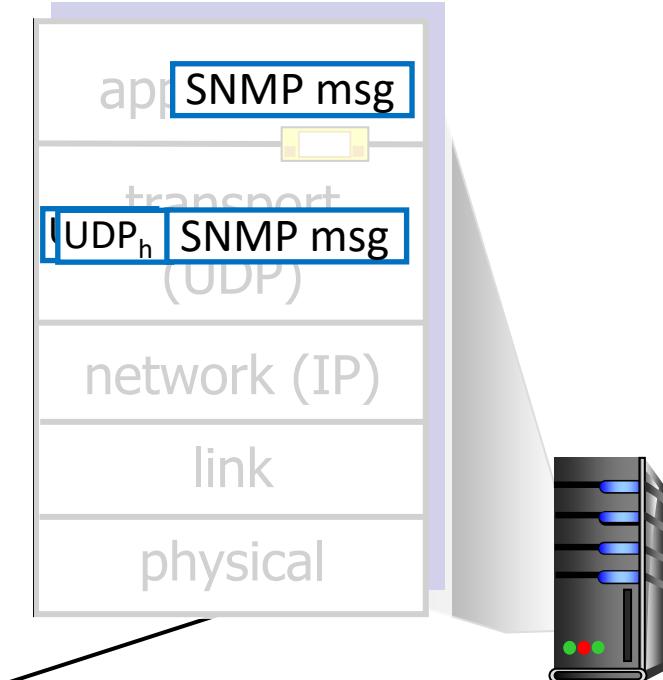
# UDP: Transport Layer Actions



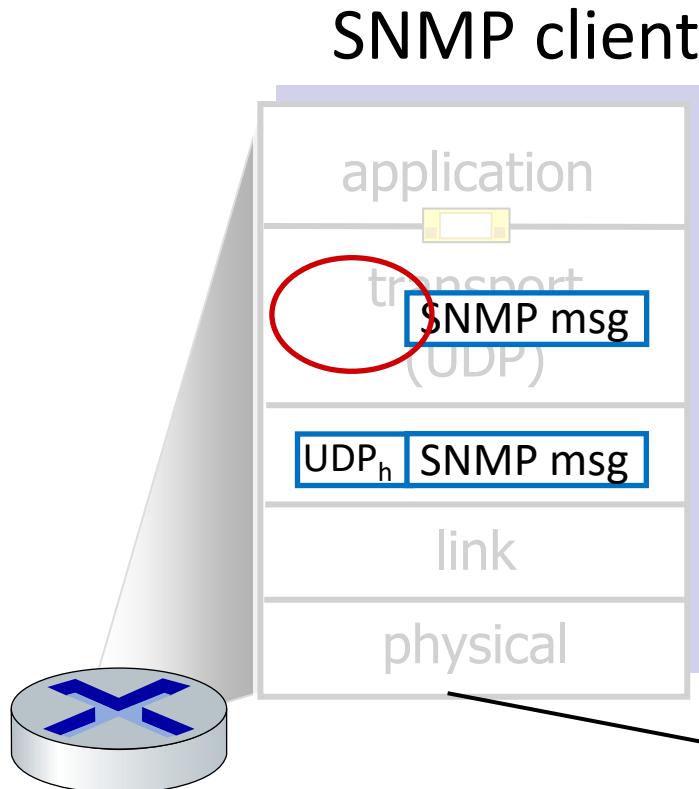
## *UDP sender* actions:

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

## SNMP server



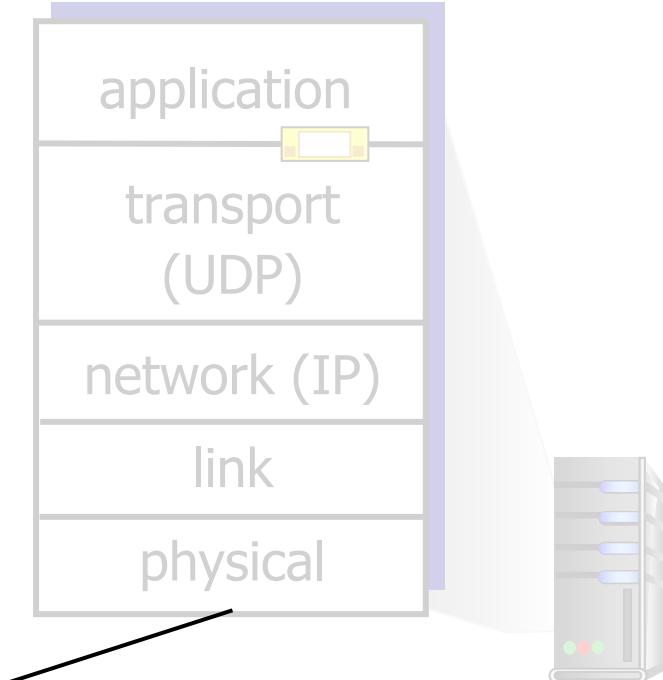
# UDP: Transport Layer Actions



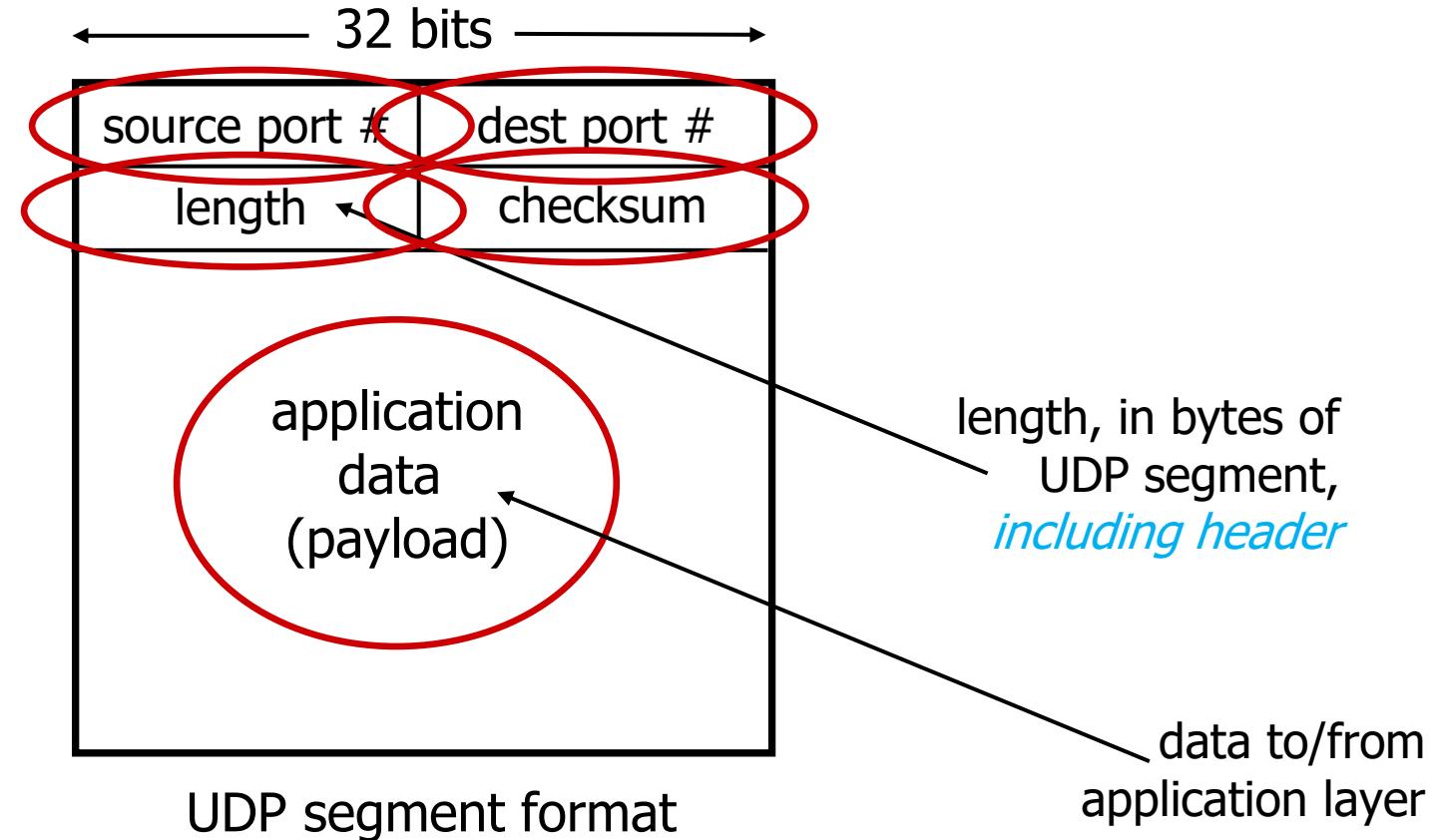
## *UDP receiver* actions:

- *receives* segment from IP
- *checks* UDP checksum header value
- *extracts* application-layer message
- *demultiplexes* message up to application via socket

## SNMP server

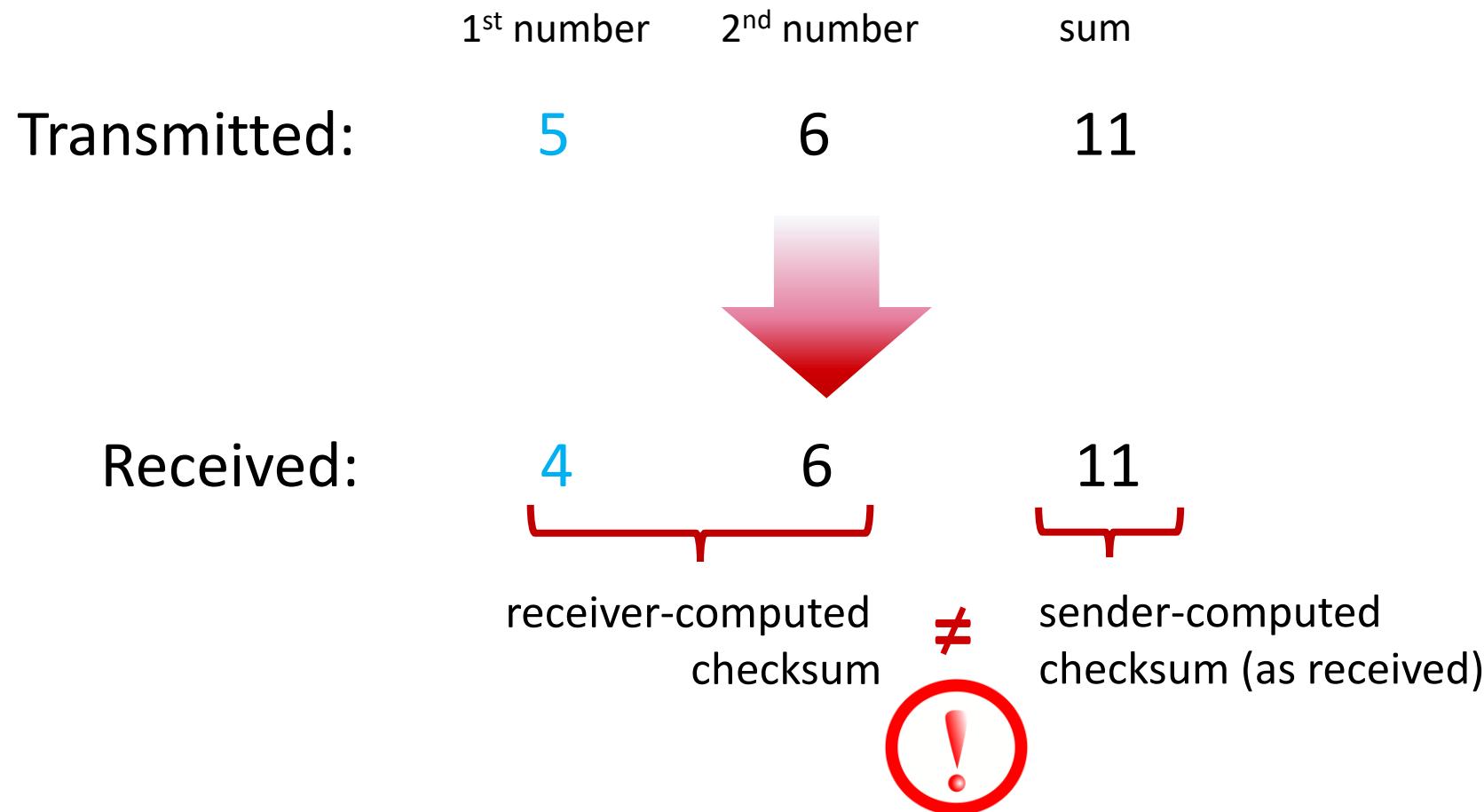


# UDP segment header



# UDP checksum

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



# Internet checksum

**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 ....

# Internet checksum: an 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
<hr/>																
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	0	1	1
<hr/>																
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

\* Check out the online interactive exercises for more examples: [http://gaia.cs.umass.edu/kurose\\_ross/interactive/](http://gaia.cs.umass.edu/kurose_ross/interactive/)

# Internet checksum: weak protection!

example: add two 16-bit integers

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
<hr/>																	
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	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1	0

Even though *numbers have changed* (bit flips), *no* change in checksum!

# Summary: UDP

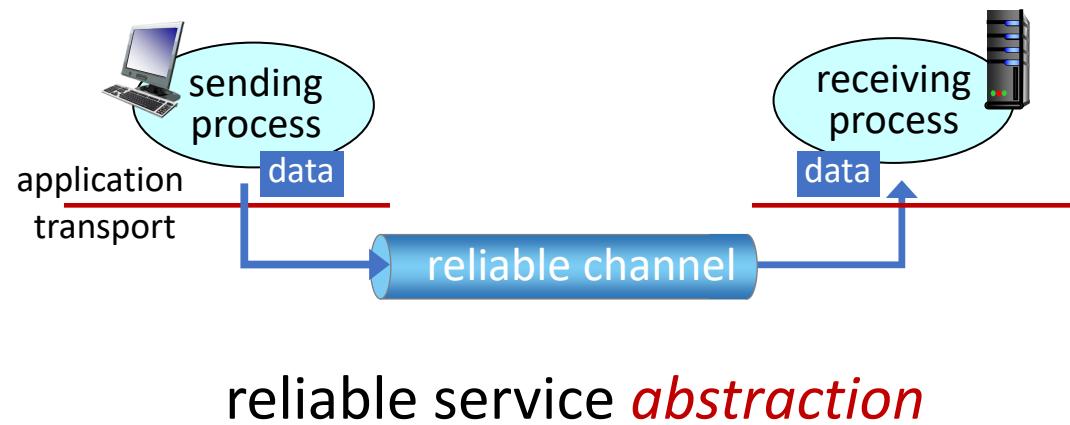
- “no frills” protocol:
  - segments may be *lost*, delivered *out of order*
  - *best effort* service: “send and hope for the best”
- UDP has its plusses:
  - *no setup*/handshaking needed (no RTT incurred)
  - can function when network service is compromised
  - helps with *reliability* (checksum)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

# Chapter 3: roadmap

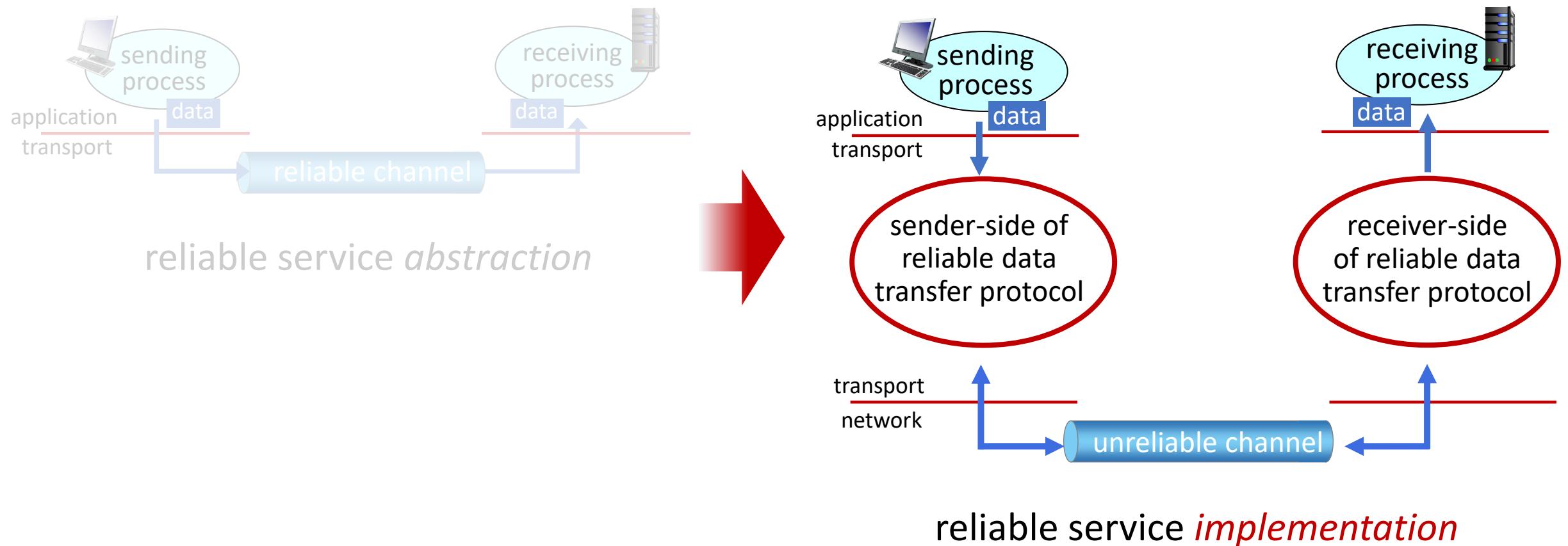
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- **Principles of reliable data transfer**
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality



# Principles of reliable data transfer

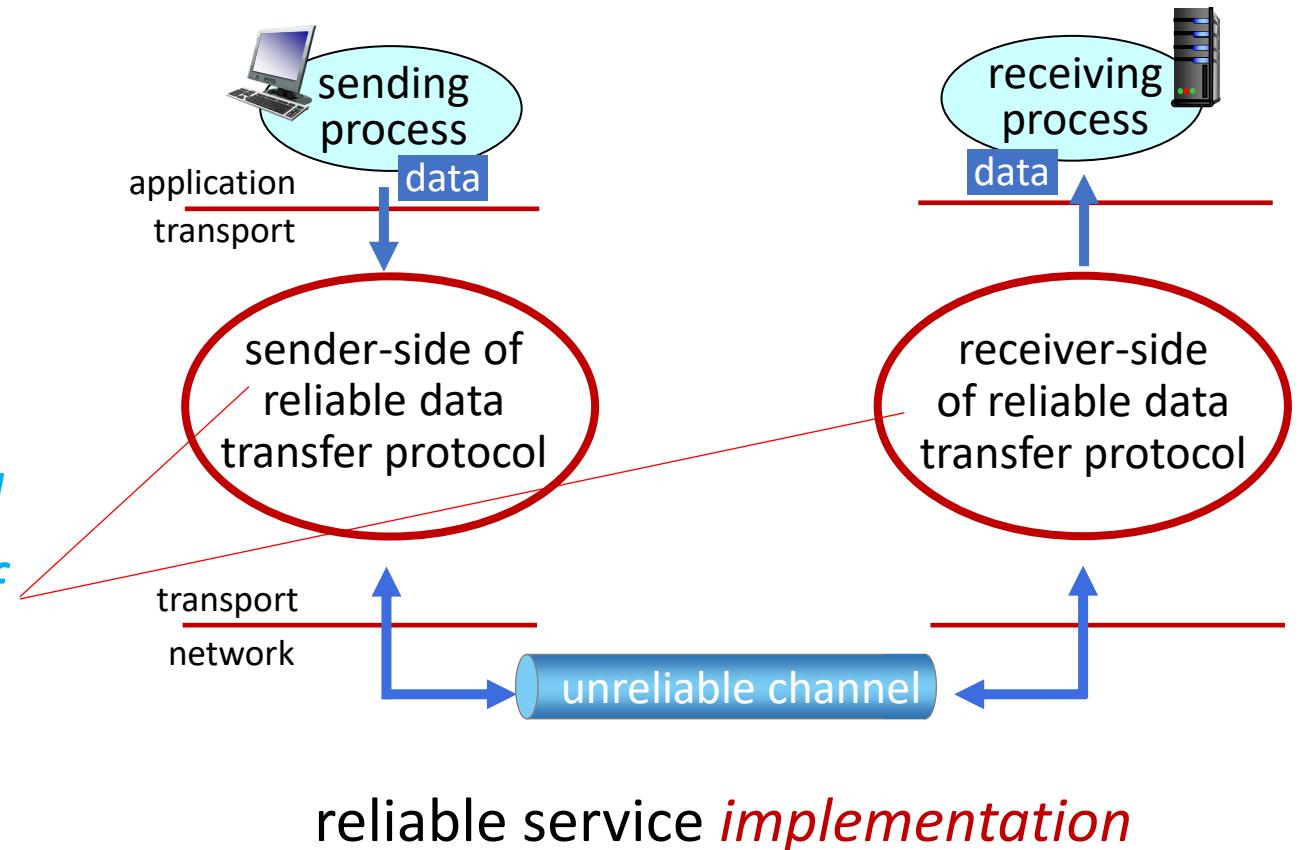


# Principles of reliable data transfer



# Principles of reliable data transfer

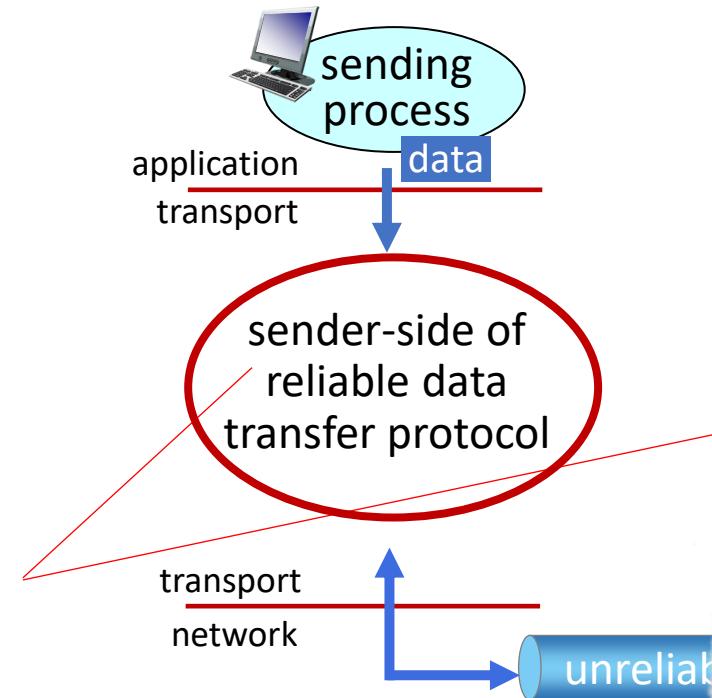
*Complexity* of reliable data transfer protocol will *depend* (strongly) *on characteristics of unreliable channel* (lose, corrupt, reorder data?)



# 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*

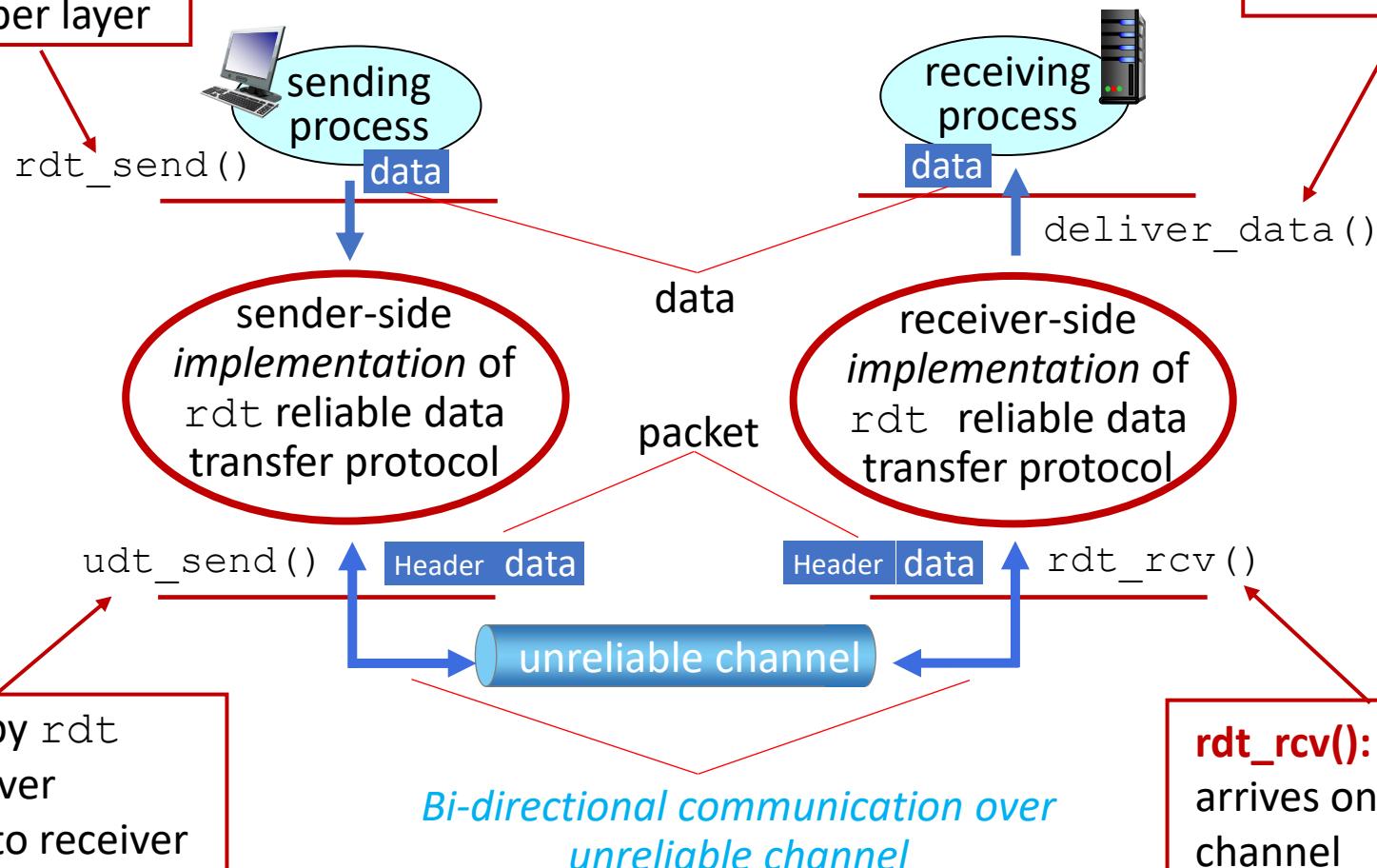


*reliable service implementation*



# Reliable data transfer protocol (rdt): interfaces

**rdt\_send()**: called from above, (e.g., by app.). Passed data to deliver to receiver upper layer



**udt\_send()**: called by rdt to transfer packet over unreliable channel to receiver

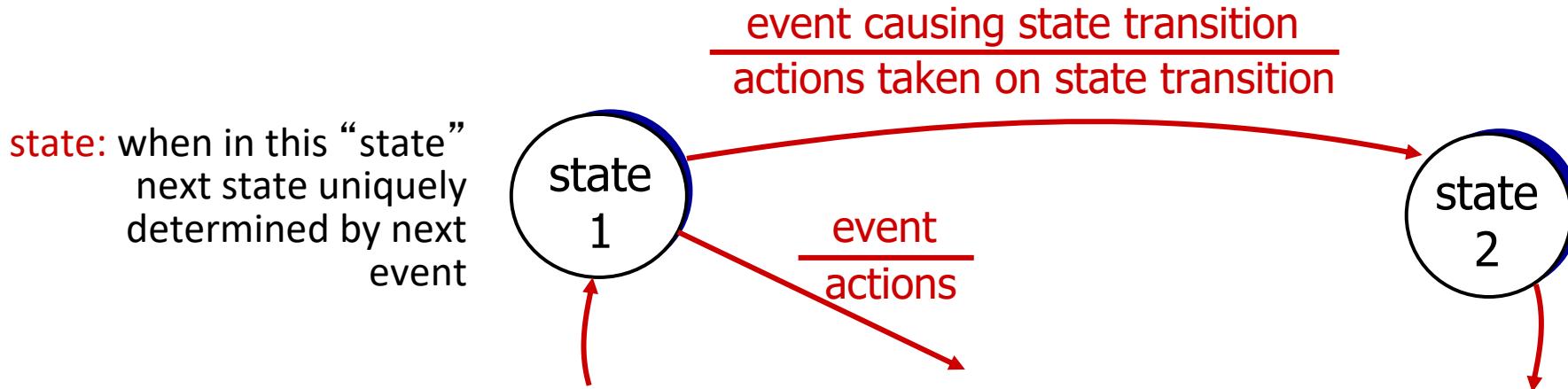
**deliver\_data()**: called by rdt to deliver data to upper layer

**rdt\_rcv()**: called when packet arrives on receiver side of channel

# Reliable data transfer: getting started

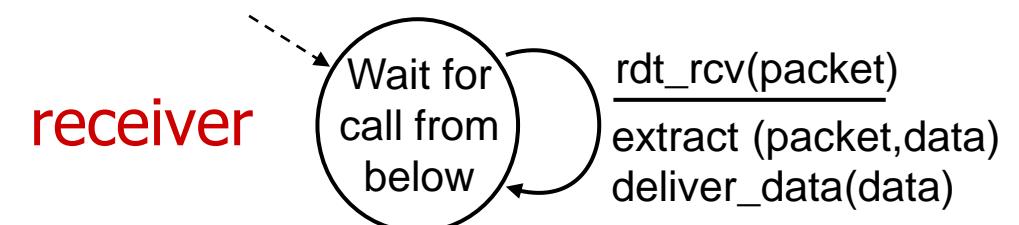
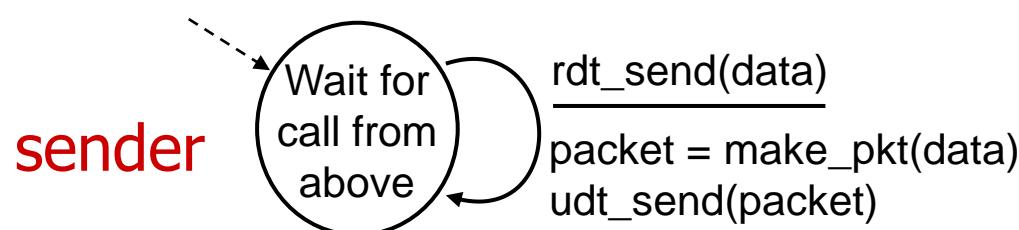
We will:

- *incrementally develop* sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow in both directions!
- use *finite state machines* (FSM) to specify sender, receiver



# rdt1.0: reliable transfer over a reliable channel

- underlying channel *perfectly reliable*
  - no bit errors
  - no loss of packets
- separate* FSMs for sender, receiver:
  - sender* sends data into underlying channel
  - receiver* reads data from underlying channel



# rdt2.0: channel with bit errors

- underlying channel *may flip bits in packet*
  - checksum (e.g., Internet checksum) to detect bit errors
- the question: *how to recover from errors?*

*How do humans recover from “errors” during conversation?*

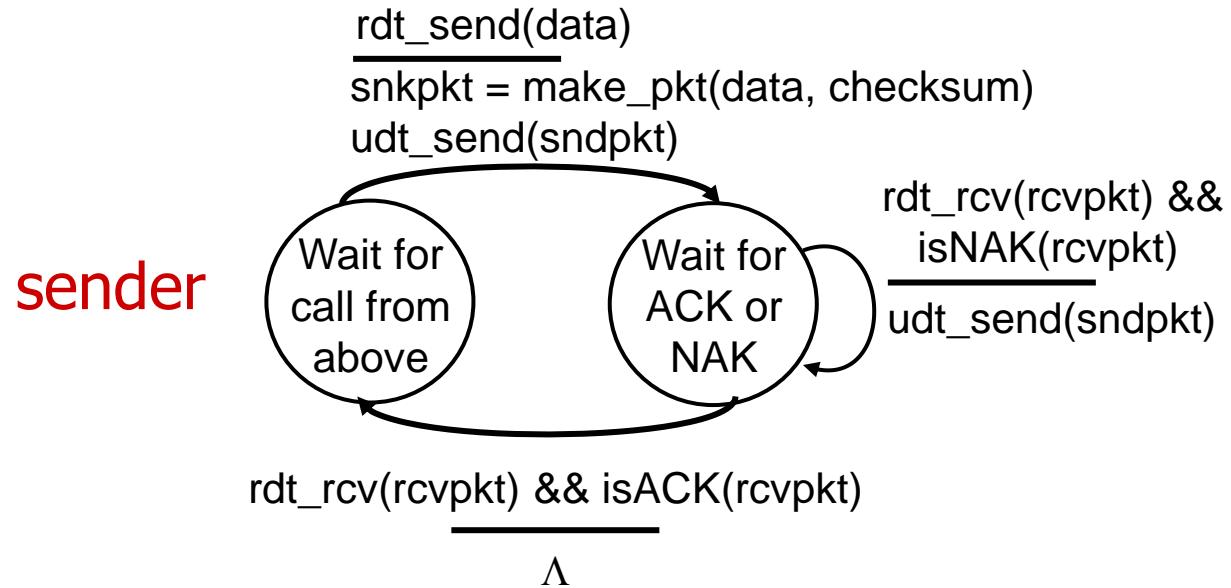
# rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- the question: how to recover from errors?
  - *acknowledgements (ACKs)*: receiver explicitly tells sender that pkt received *OK*
  - *negative acknowledgements (NAKs)*: receiver explicitly tells sender that pkt had *errors*
  - sender *retransmits* pkt on receipt of NAK

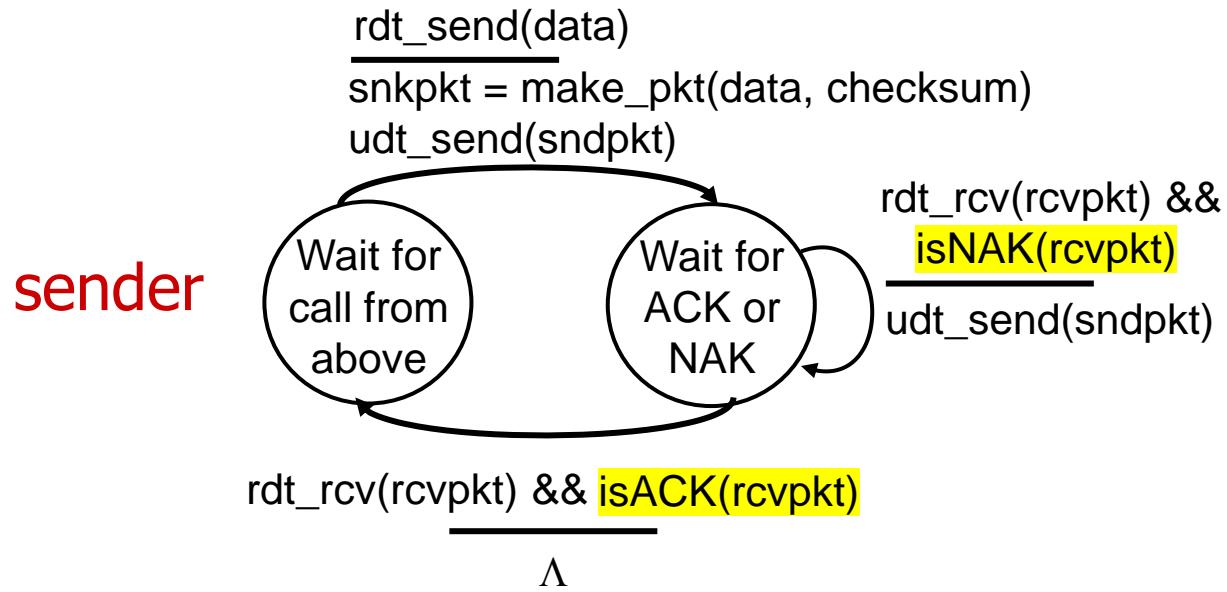
stop and wait

*sender sends one packet, then waits for receiver response*

# rdt2.0: FSM specifications



# rdt2.0: FSM specification

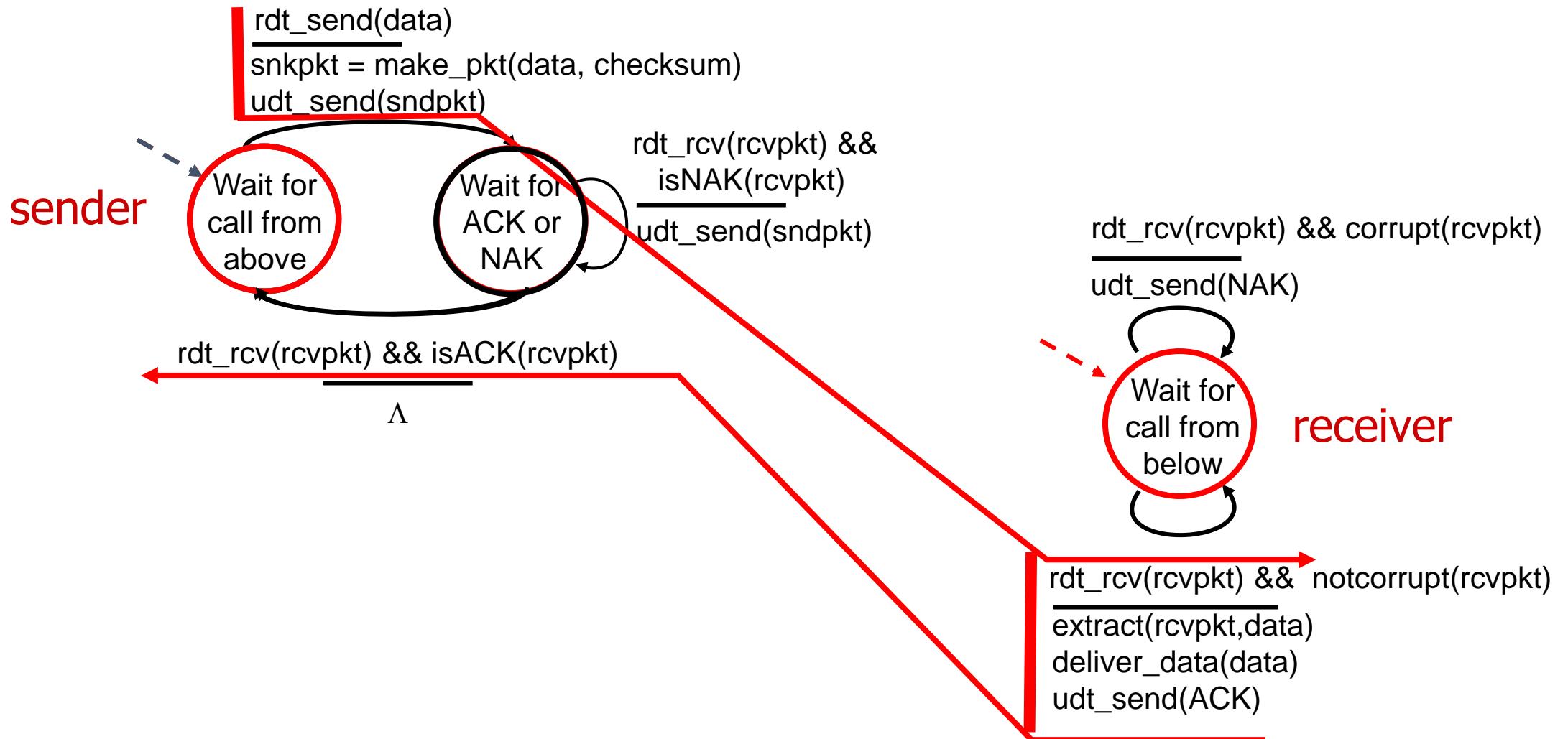


Note: “*state*” of receiver (did the receiver get my message correctly?) isn’t known to sender unless *somehow communicated from receiver to sender*

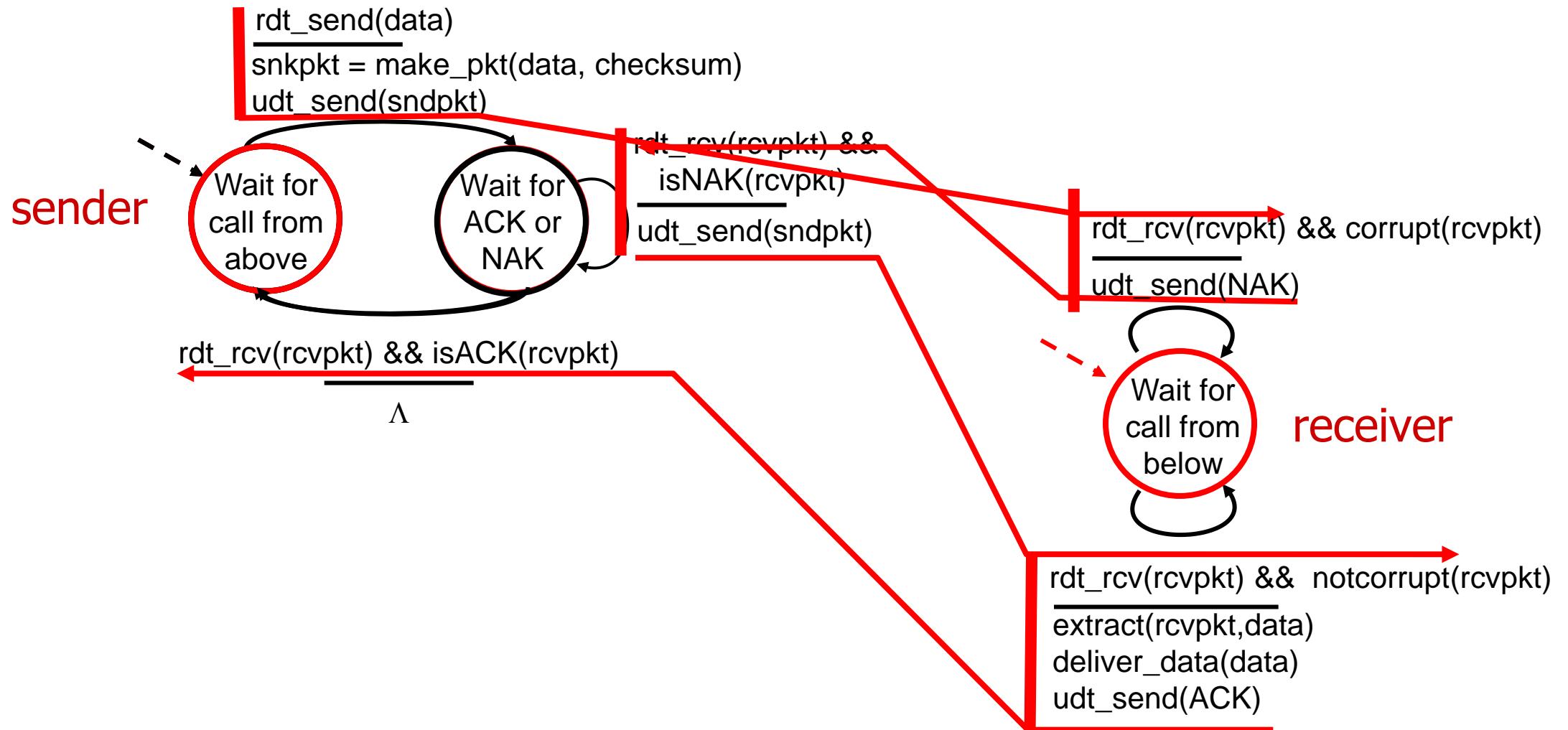
- that’s why we need a protocol!



# rdt2.0: operation with no errors



# rdt2.0: corrupted packet scenario



# rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: *possible duplicate*

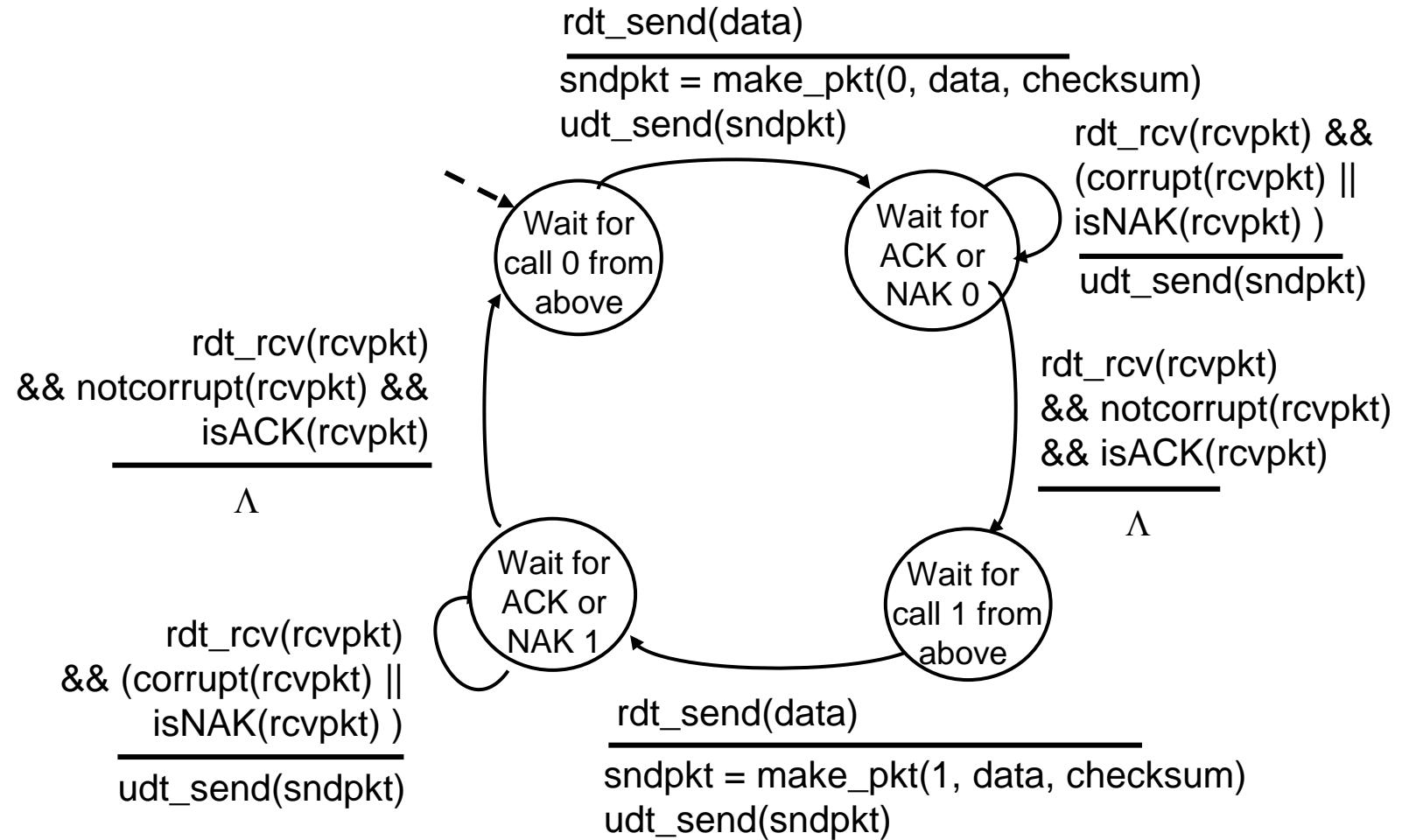
handling duplicates:

- sender *retransmits* current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver *discards* (doesn't deliver up) duplicate pkt

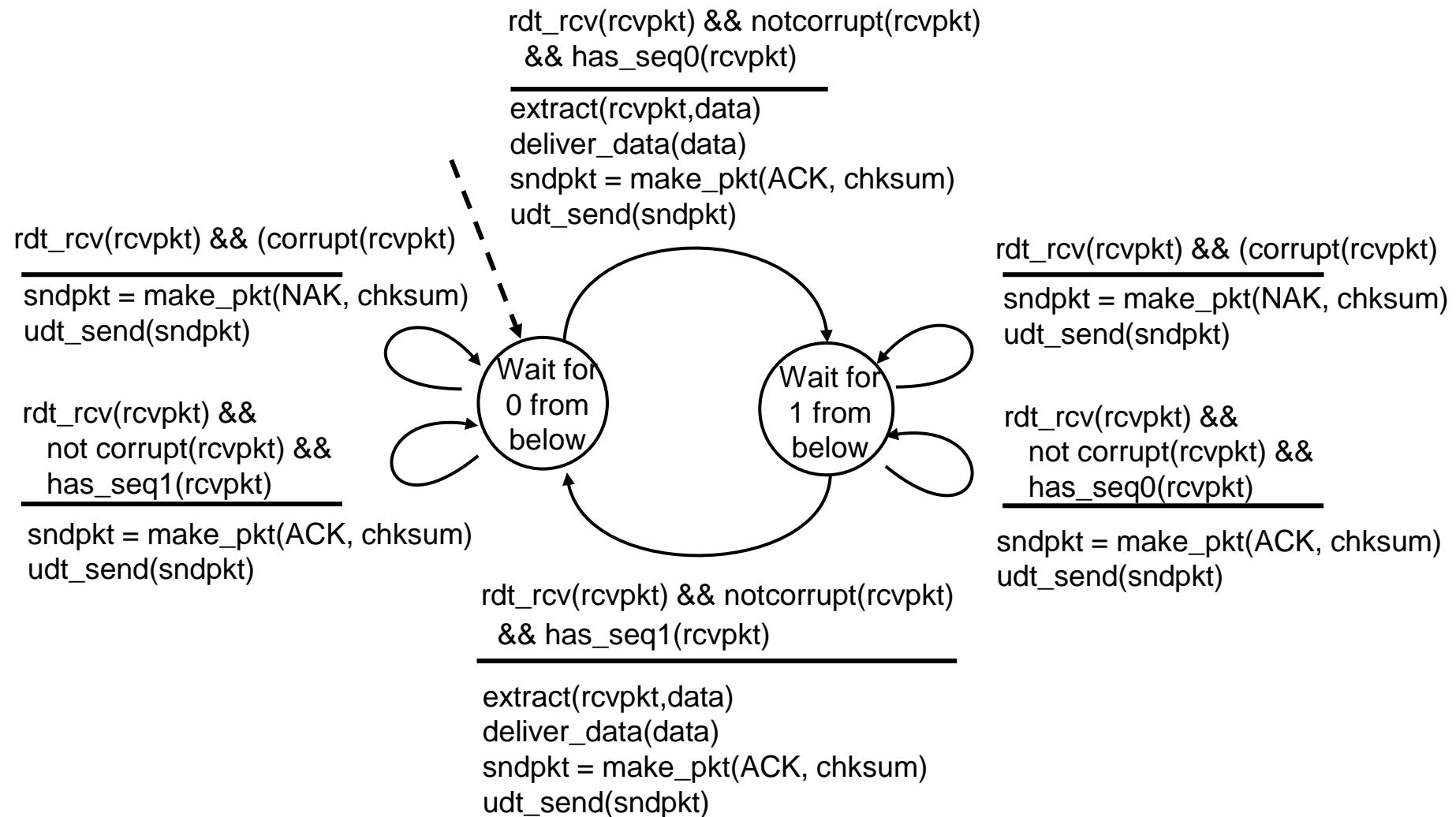
stop and wait

sender sends one packet, then waits for receiver response

# rdt2.1: sender, handling garbled ACK/NAKs



# rdt2.1: receiver, handling garbled ACK/NAKs



# rdt2.1: discussion

## sender:

- *seq #* added to pkt
- *two seq. #s (0,1) will suffice.*  
Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - *state* must “remember” whether “expected” pkt should have seq # of 0 or 1

## receiver:

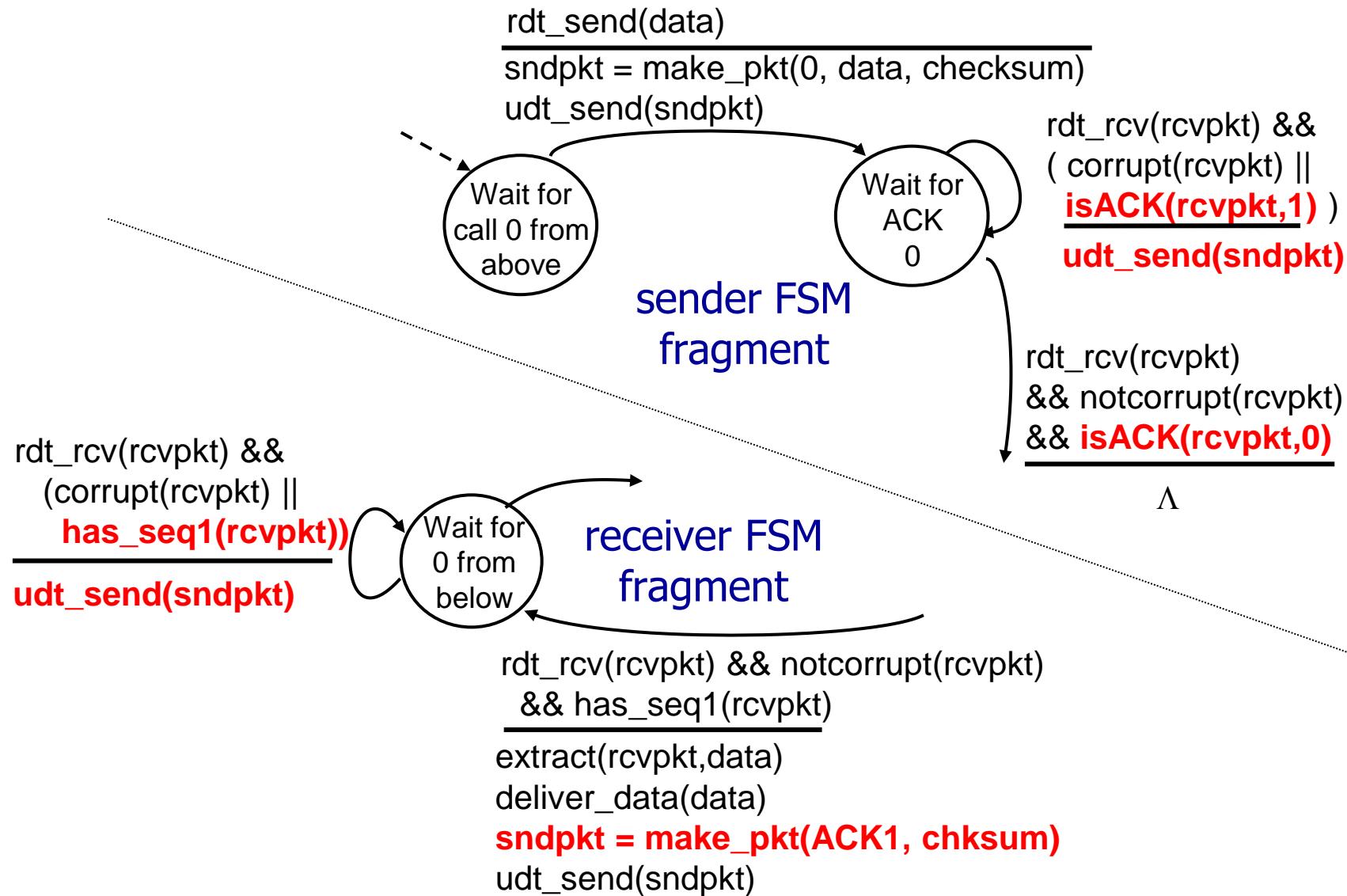
- must *check* if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not know* if its last *ACK/NAK received OK* at sender

# rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, *using ACKs only*
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- *duplicate ACK* at sender results in *same action as NAK*: retransmit current pkt

As we will see, TCP uses this approach to be NAK-free

# rdt2.2: sender, receiver fragments



# rdt3.0: channels with errors *and* loss

*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?

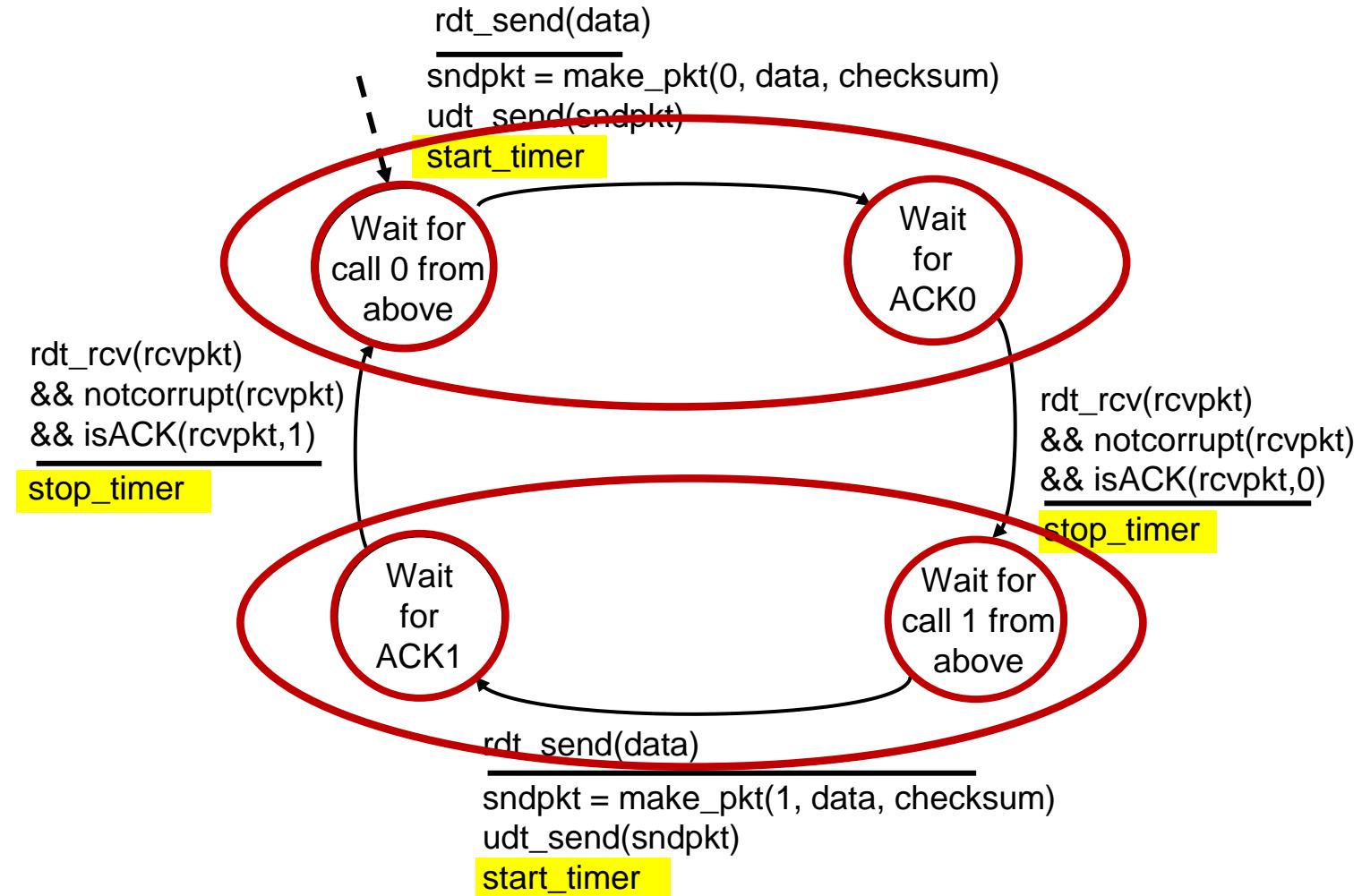
# rdt3.0: channels with errors *and* loss

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

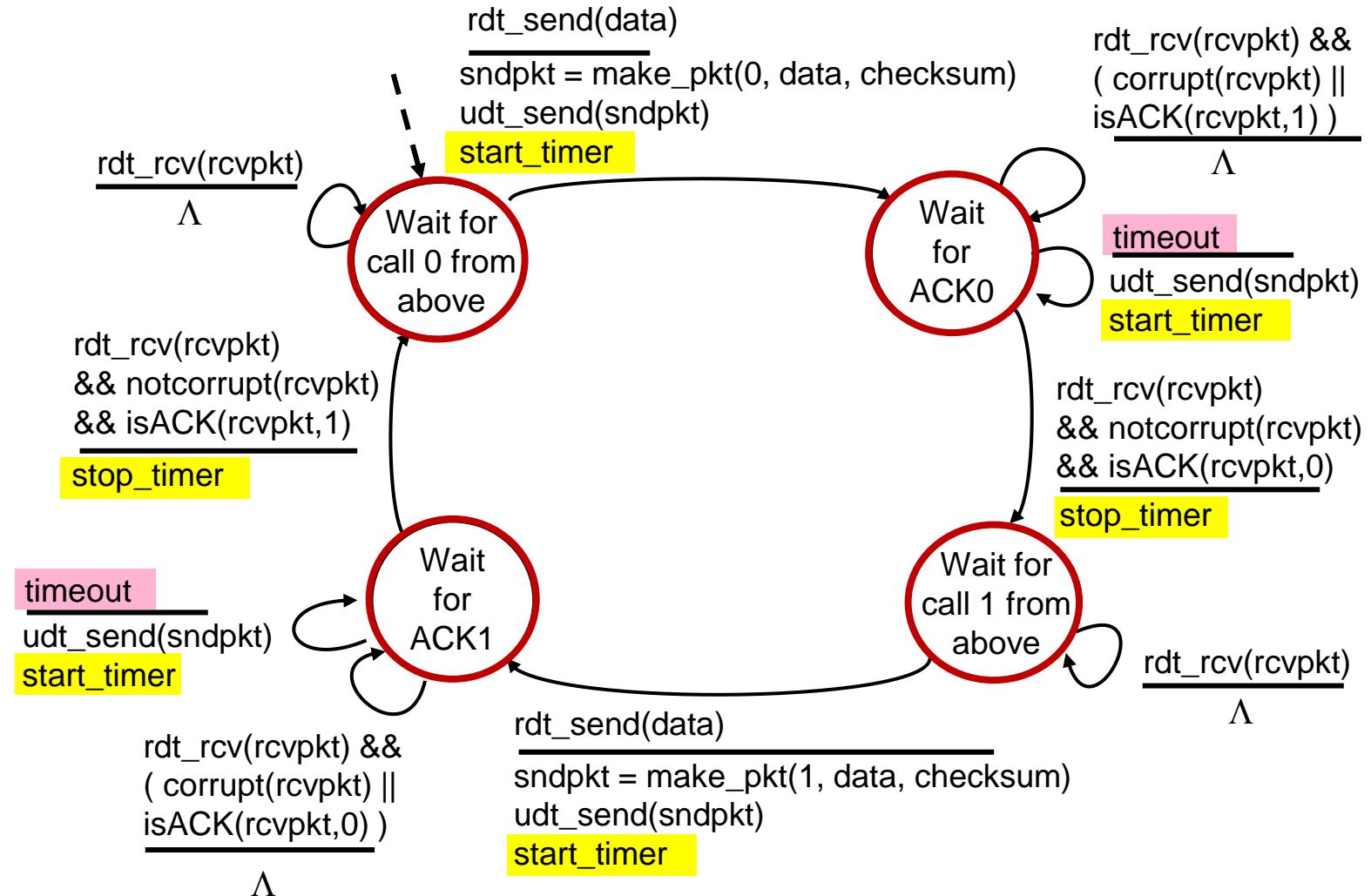
- *retransmits* if no ACK received in this time
- if pkt (or ACK) just *delayed* (not lost):
  - retransmission will be duplicate, but seq #s already handle this!
  - receiver must *specify seq #* of packet being ACKed
- use countdown *timer* to interrupt after “reasonable” amount of time



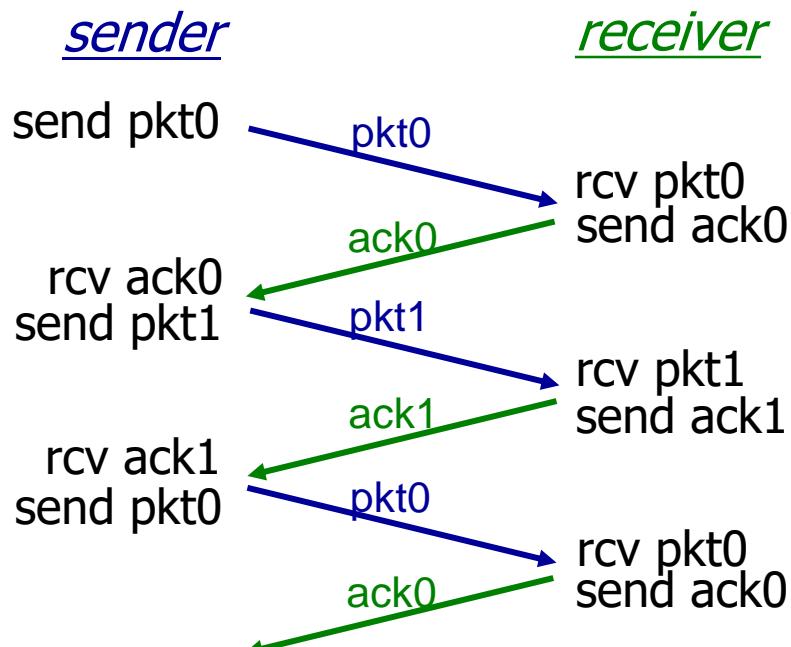
# rdt3.0 sender



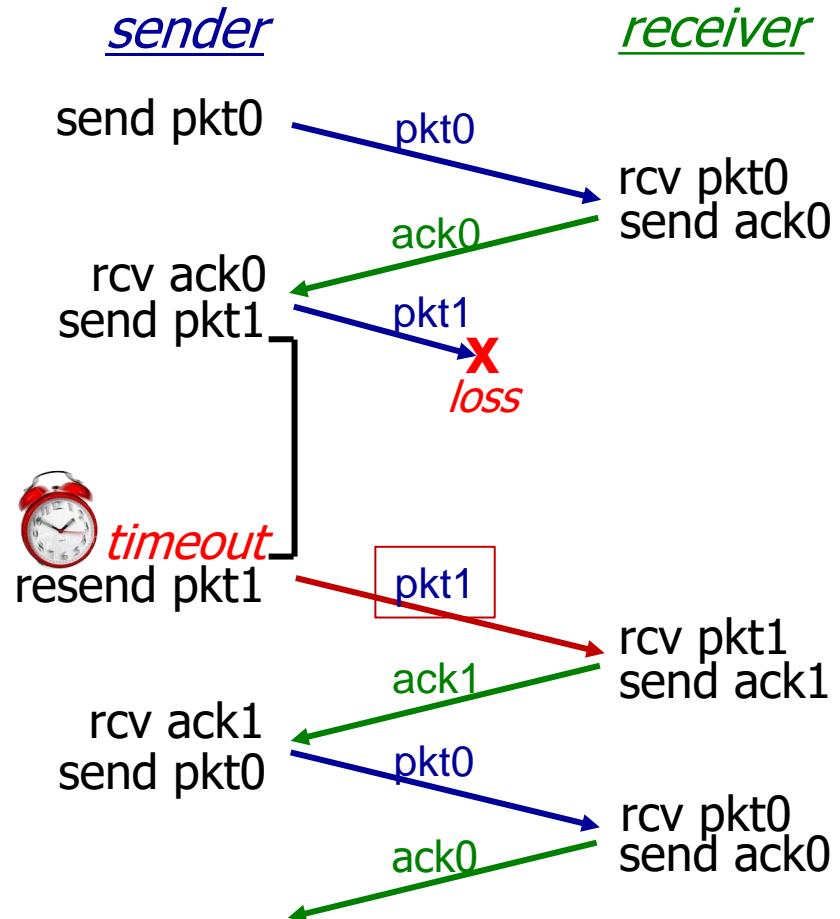
# rdt3.0 sender



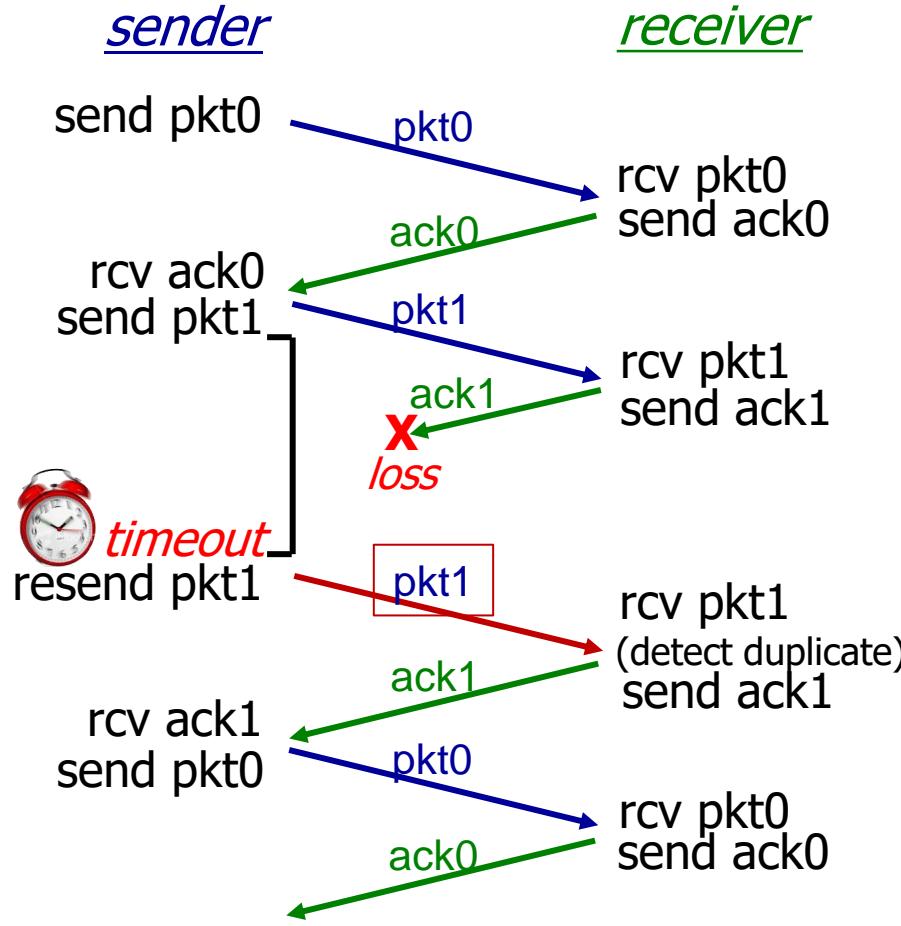
# rdt3.0 in action



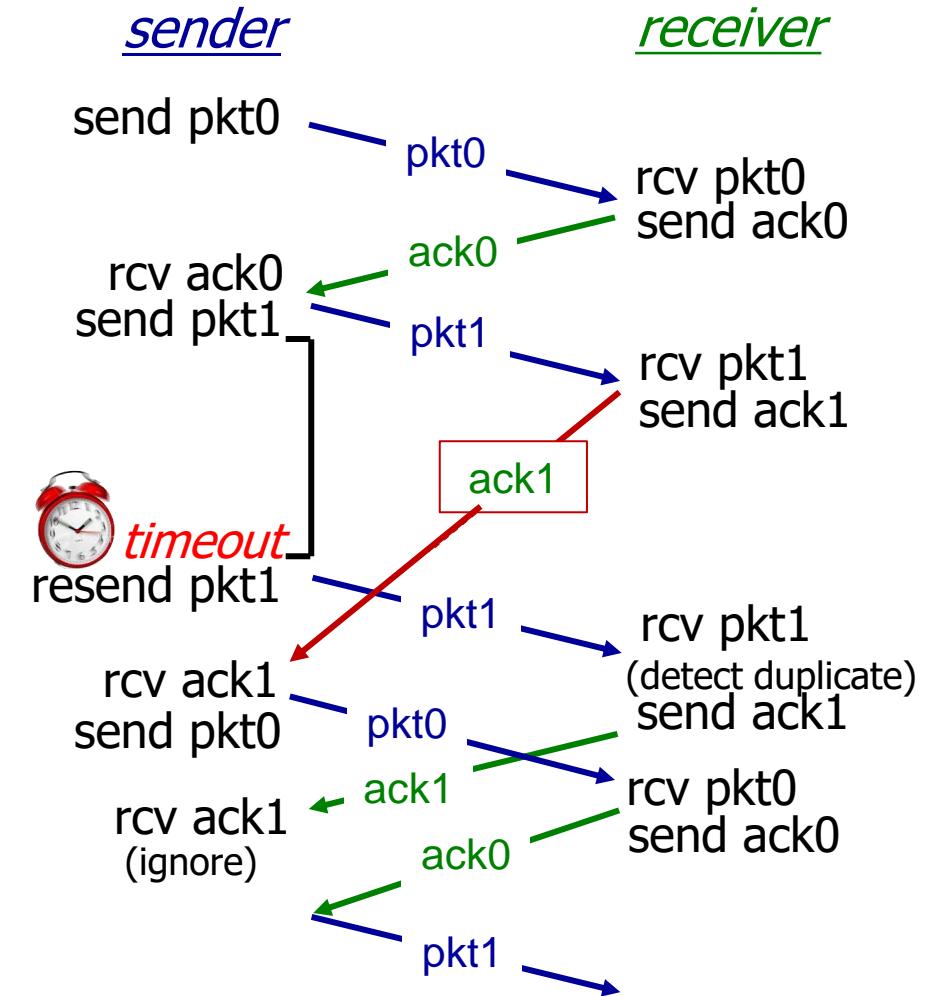
**stop and wait**  
sender sends one packet, then  
waits for receiver response



# rdt3.0 in action



(c) ACK loss



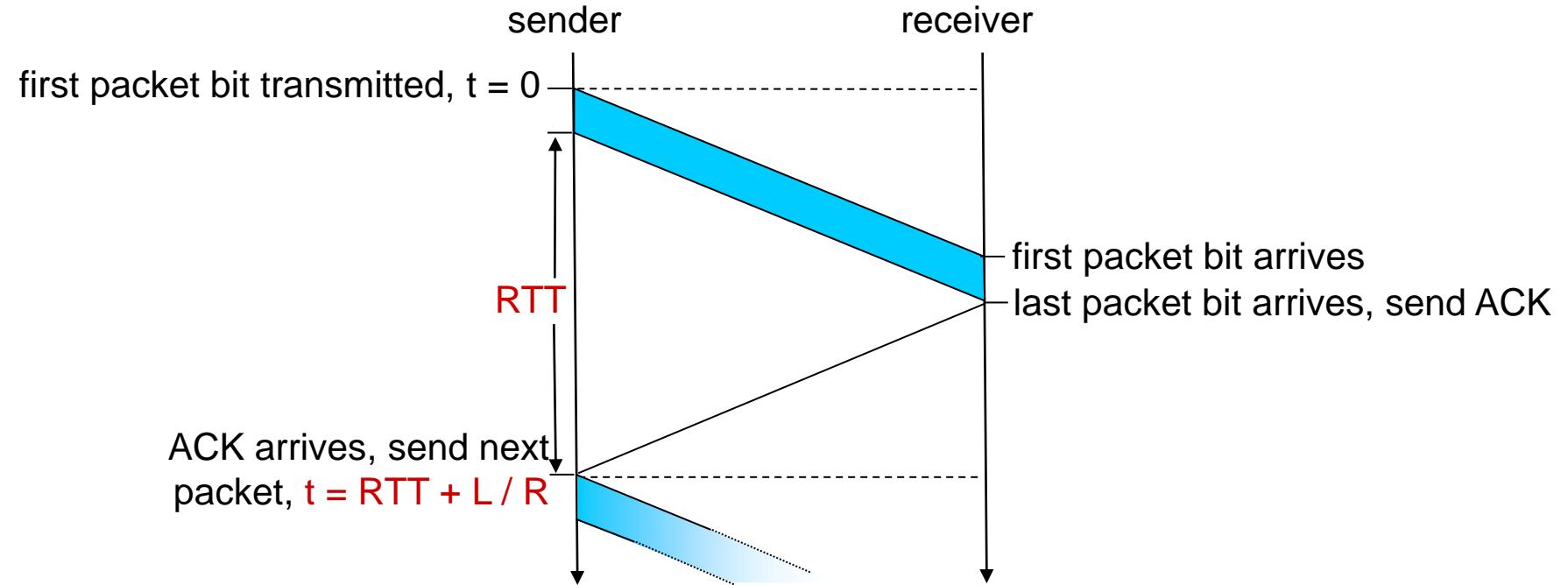
(d) premature timeout/ delayed ACK

# Performance of rdt3.0 (stop-and-wait)

- $U_{\text{sender}}$ : *utilization* – fraction of time *sender busy* sending
- example: *1 Gbps* link, *15 ms* prop. delay, *8000-bit* packet
  - time to transmit packet into channel:

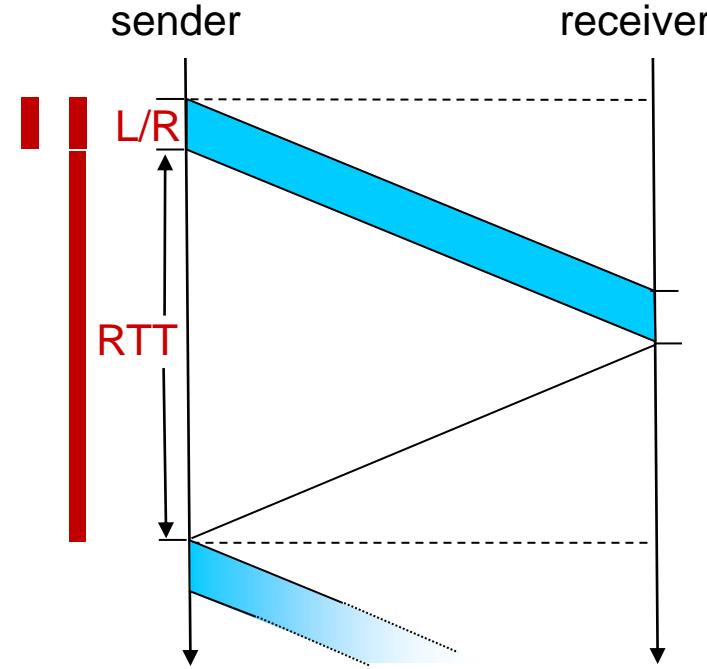
$$D_{\text{trans}} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

# rdt3.0: stop-and-wait operation



# rdt3.0: stop-and-wait operation

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

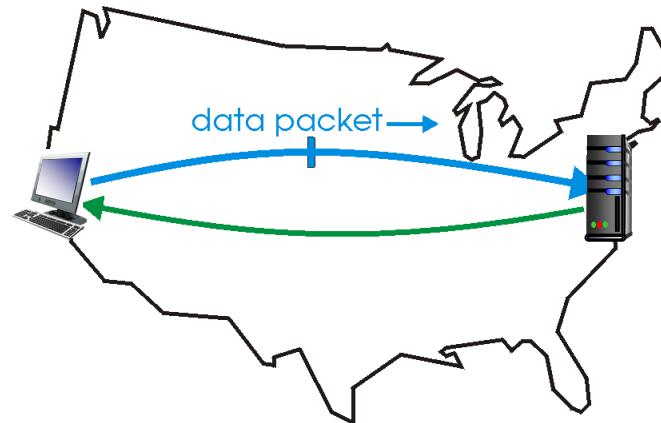


- rdt 3.0 protocol performance stinks!
- *Protocol limits performance of underlying infrastructure (channel)*

# rdt3.0: pipelined protocols operation

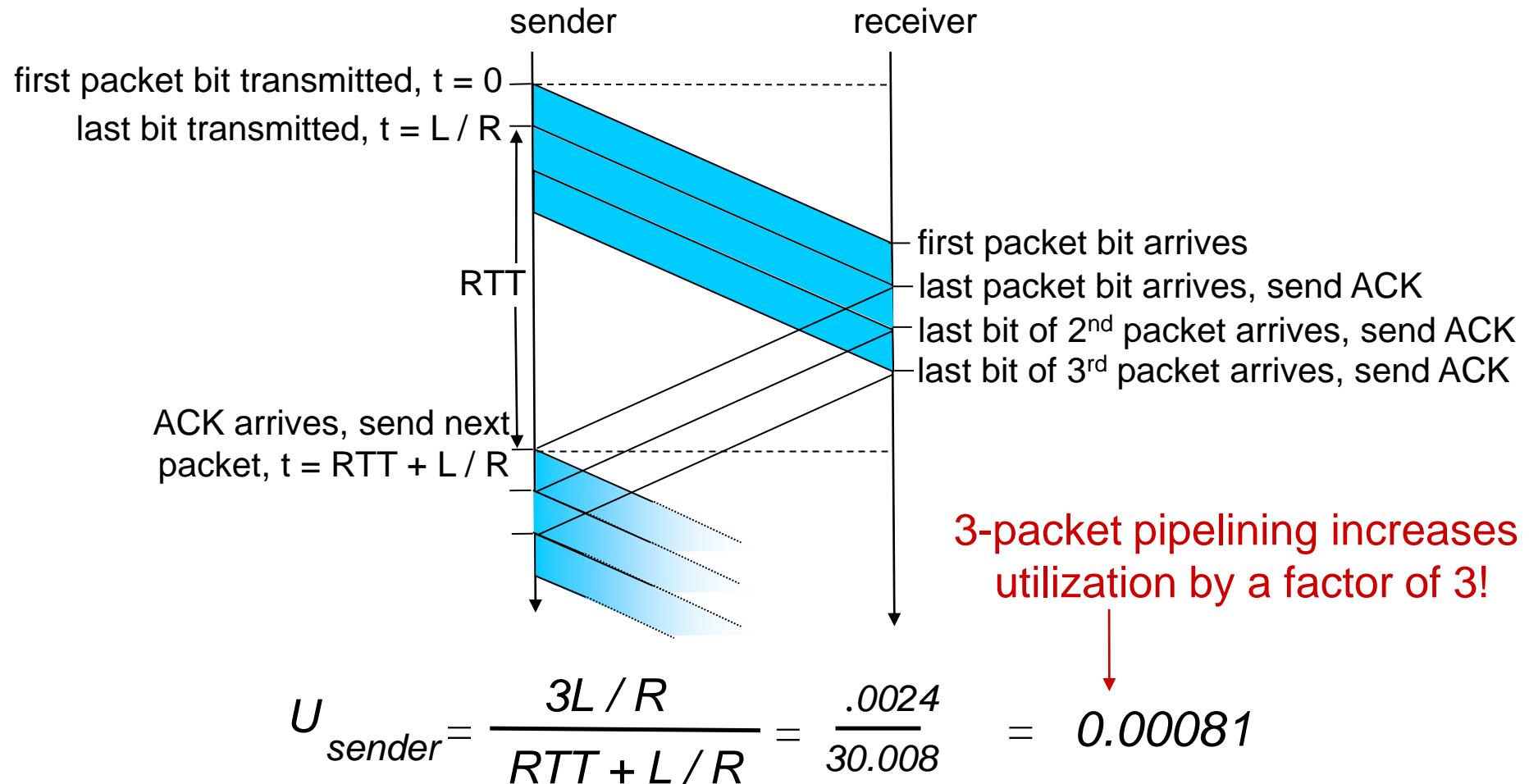
**pipelining:** sender allows multiple, “*in-flight*”, yet-to-be-acknowledged packets

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



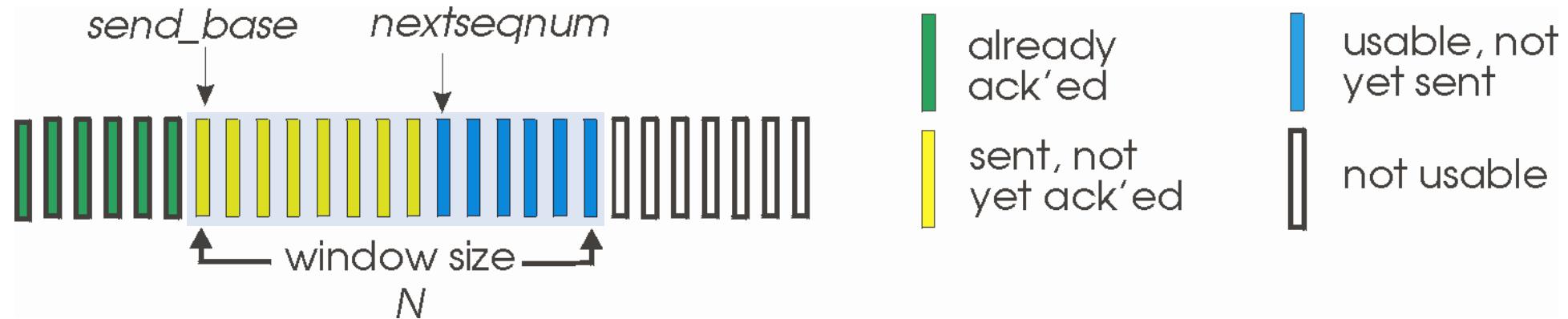
(a) a stop-and-wait protocol in operation

# Pipelining: increased utilization



# Go-Back-N: sender

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

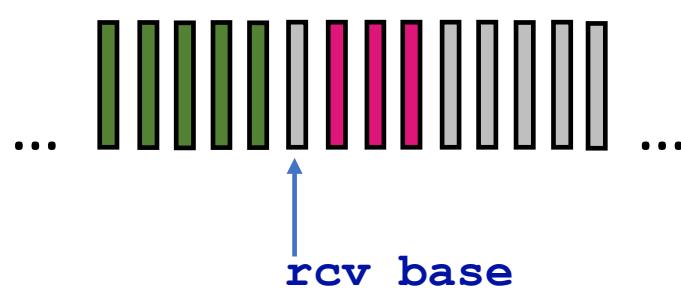


- *cumulative ACK: ACK(n)*: ACKs *all packets* up to, including seq # **n**
  - on receiving ACK( $n$ ): move window forward to begin at  $n+1$
- *timer* for *oldest* in-flight packet
- *timeout(n)*: retransmit packet **n** and *all higher seq #* packets in window

# 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:



received and ACKed

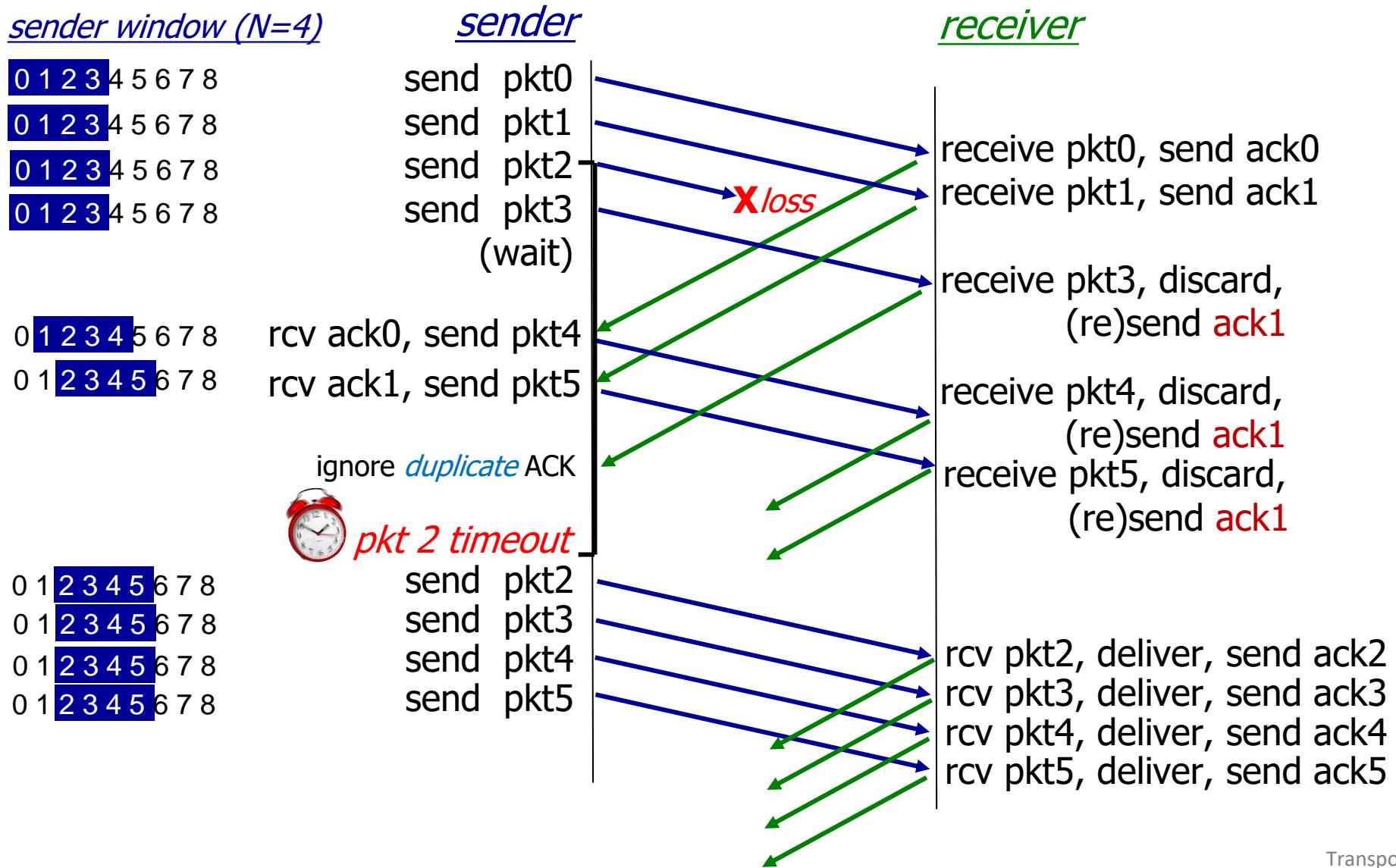


Out-of-order: received but not ACKed



Not received

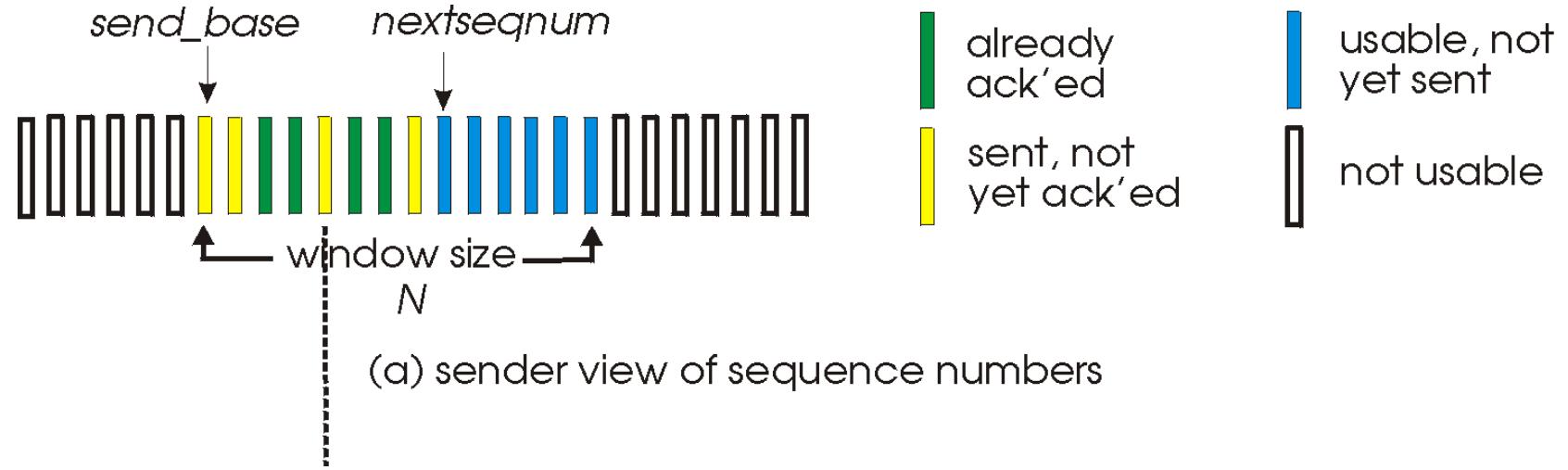
# Go-Back-N in action



# 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, receiver windows



# 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

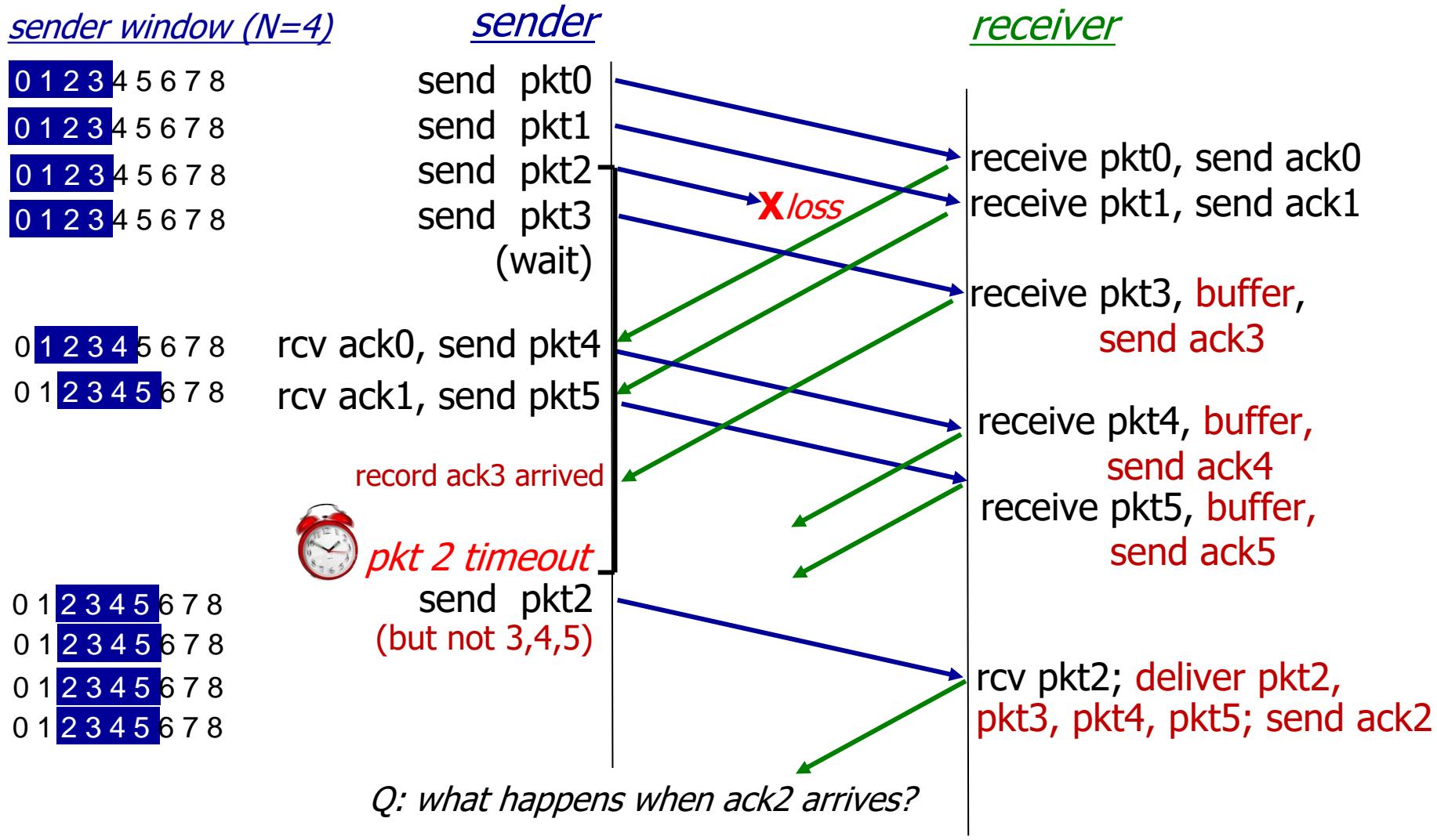
packet  $n$  in  $[rcvbase-N,rcvbase-1]$

- ACK( $n$ )

**otherwise:**

- ignore

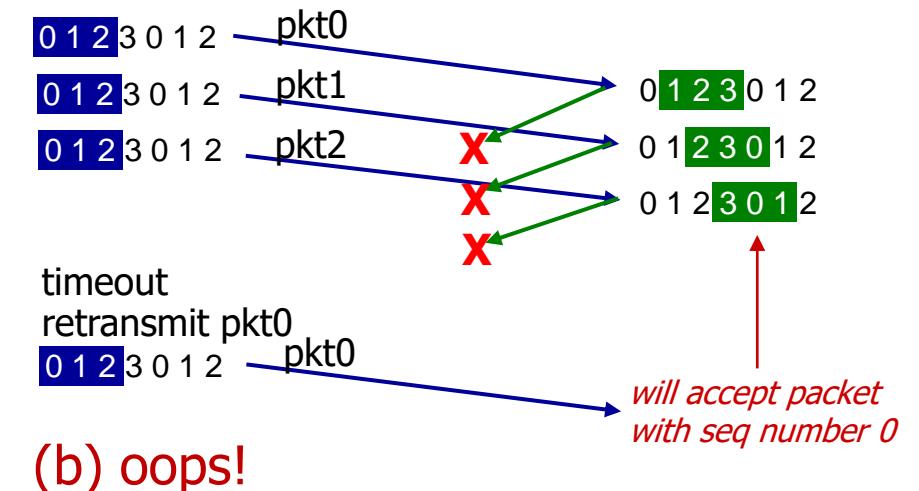
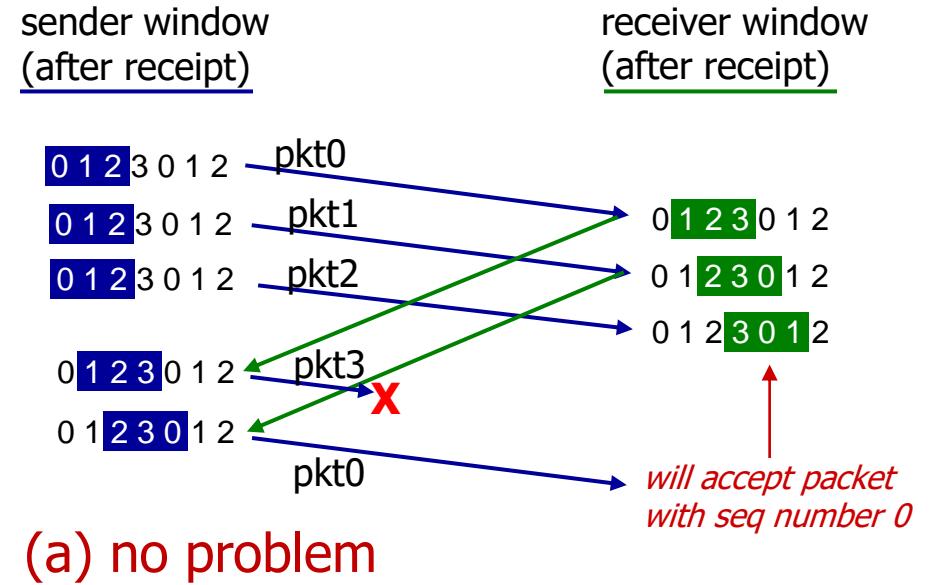
# Selective Repeat in action



# Selective repeat: a dilemma!

example:

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



# Selective repeat: a dilemma!

## example:

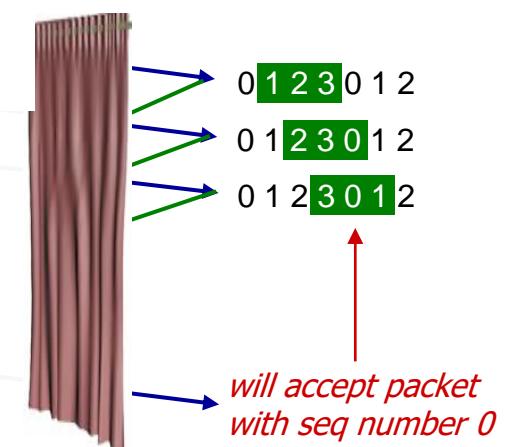
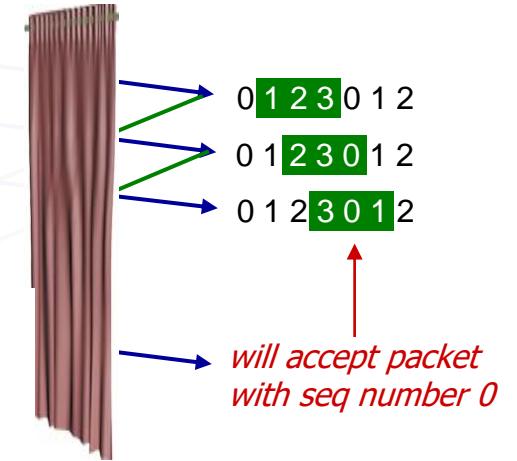
- seq #s: 0, 1, 2, 3 (base 4 counting)
  - window size=3

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

## sender window (after receipt)

- receiver can't see sender side
  - receiver behavior identical in both cases!
  - something's (very) wrong!

receiver window  
(after receipt)



# Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- **Connection-oriented transport: TCP**
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control



# TCP: overview

RFCs: 793, 1122, 2018, 5681, 7323

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream*:
  - 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

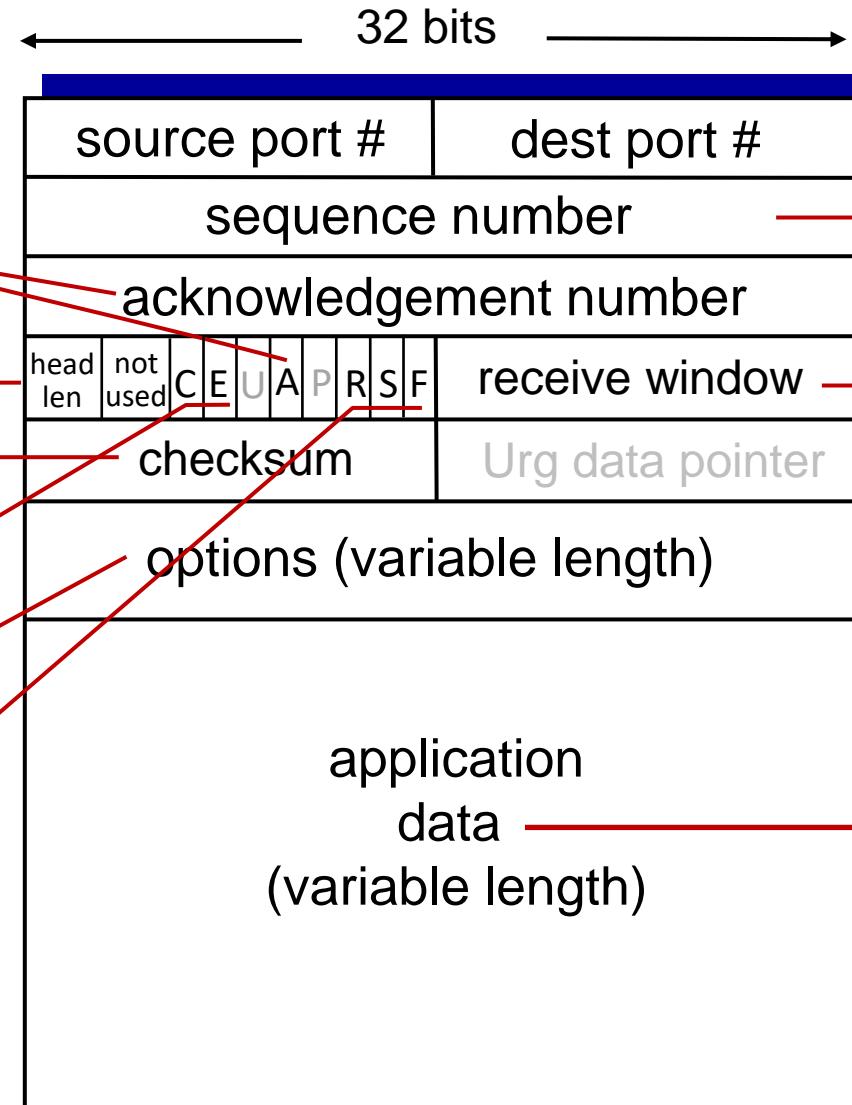
length (of TCP header)

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 numbers, ACKs

## Sequence numbers:

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

## Acknowledgements:

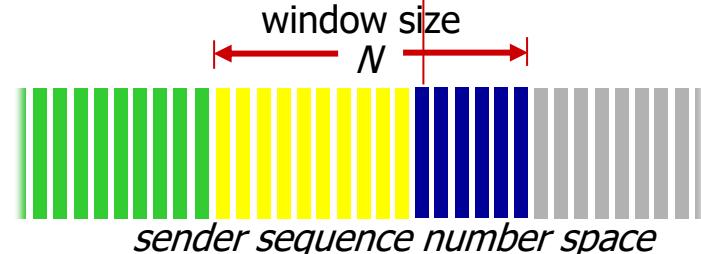
- seq # of *next byte* expected from other side
- *cumulative ACK*

*Q: how receiver handles out-of-order segments*

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

outgoing segment from sender

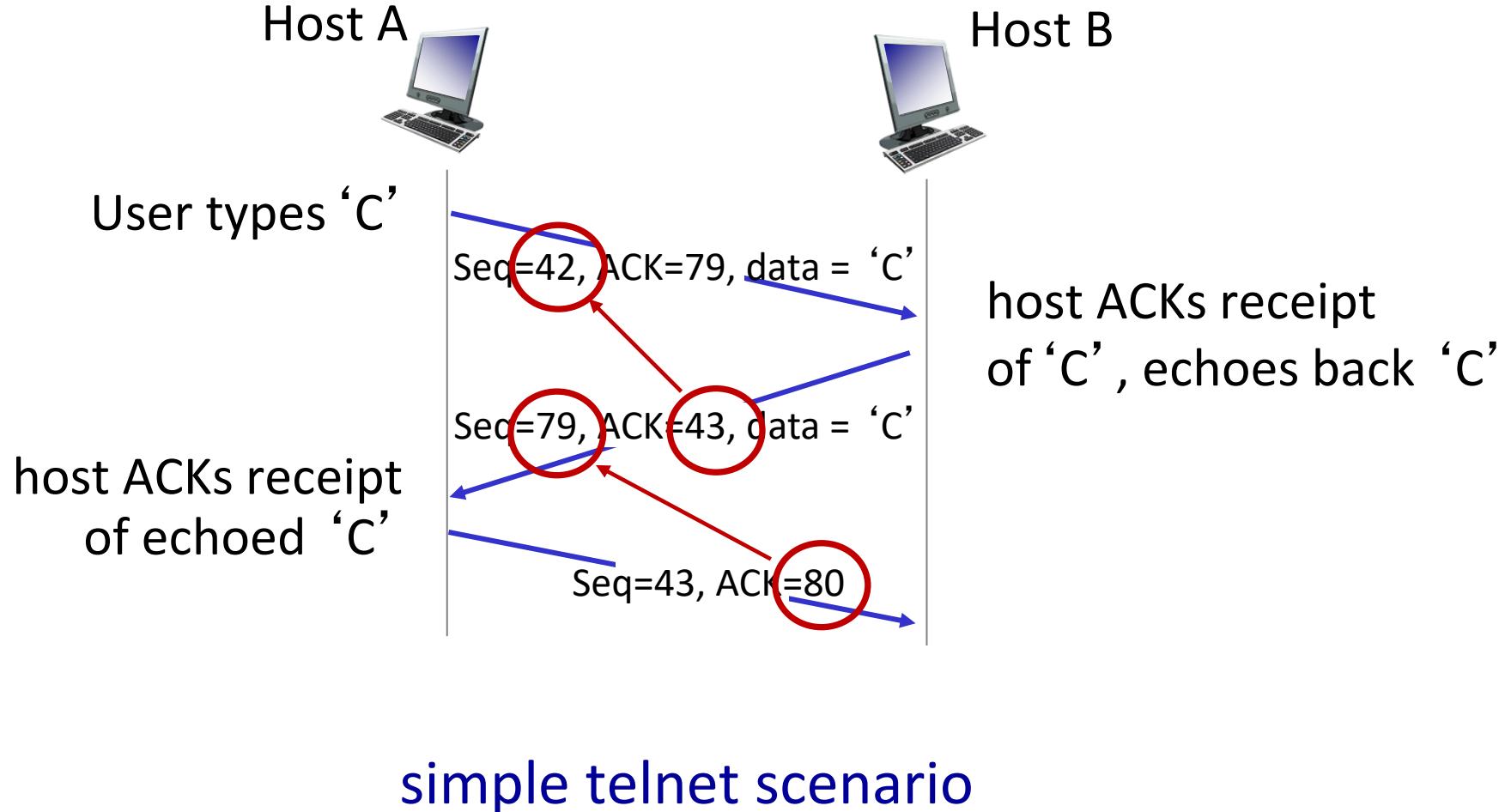
source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer



outgoing segment from receiver

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

# TCP sequence 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

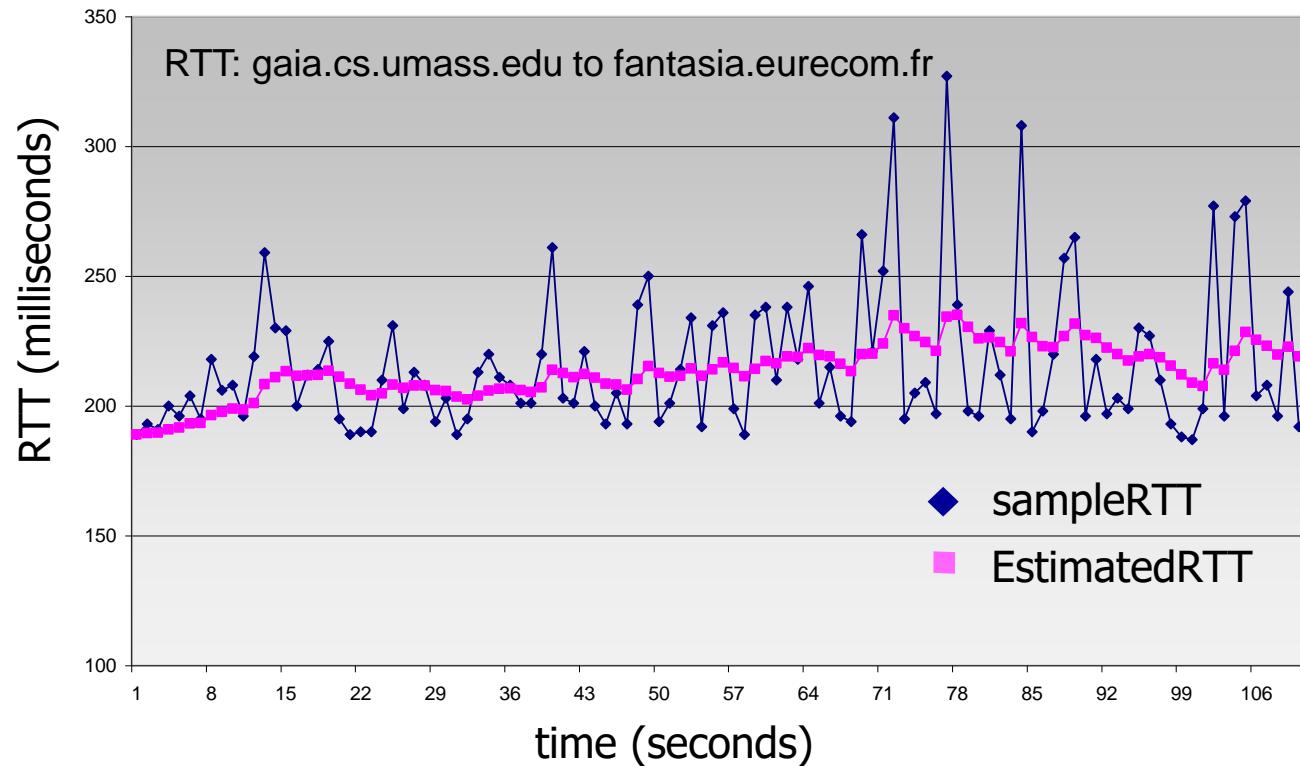
***Q:*** how to *estimate RTT*?

- *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

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

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



# TCP round trip time, 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,  $\beta = 0.25$ )

\* Check out the online interactive exercises for more examples: [http://gaia.cs.umass.edu/kurose\\_ross/interactive/](http://gaia.cs.umass.edu/kurose_ross/interactive/)

# TCP Sender (simplified)

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**

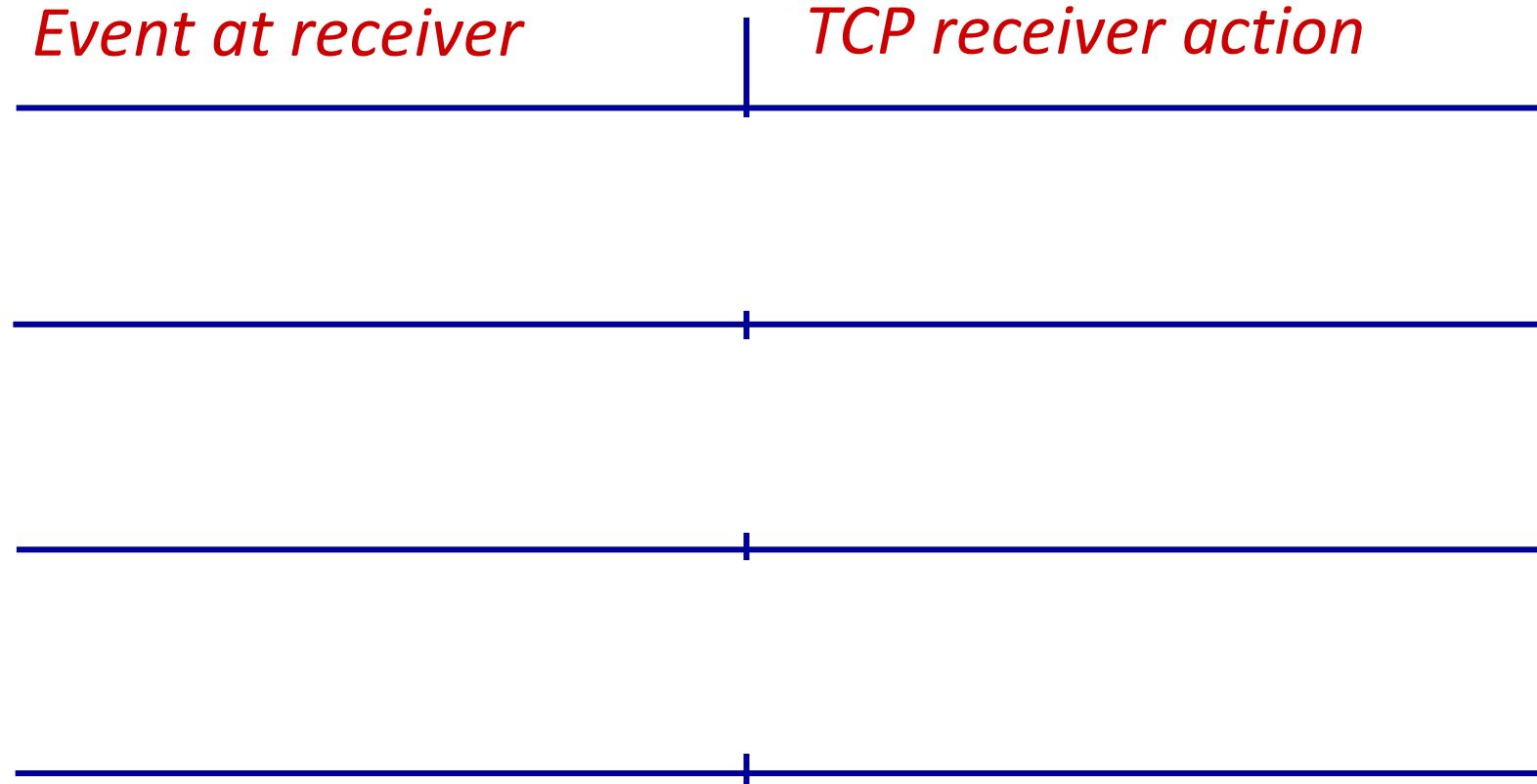
*event: timeout*

- *retransmit segment* that caused timeout
- *restart timer*

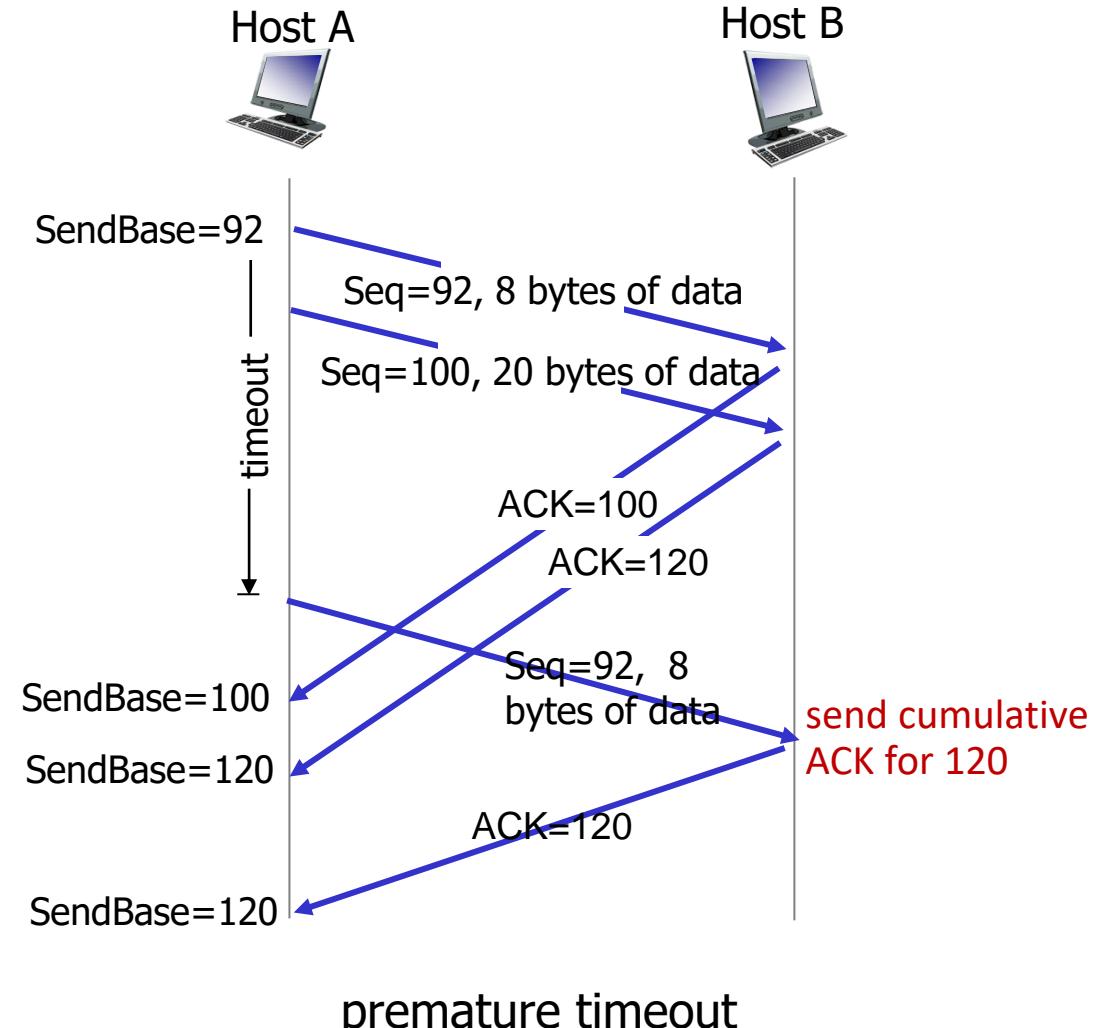
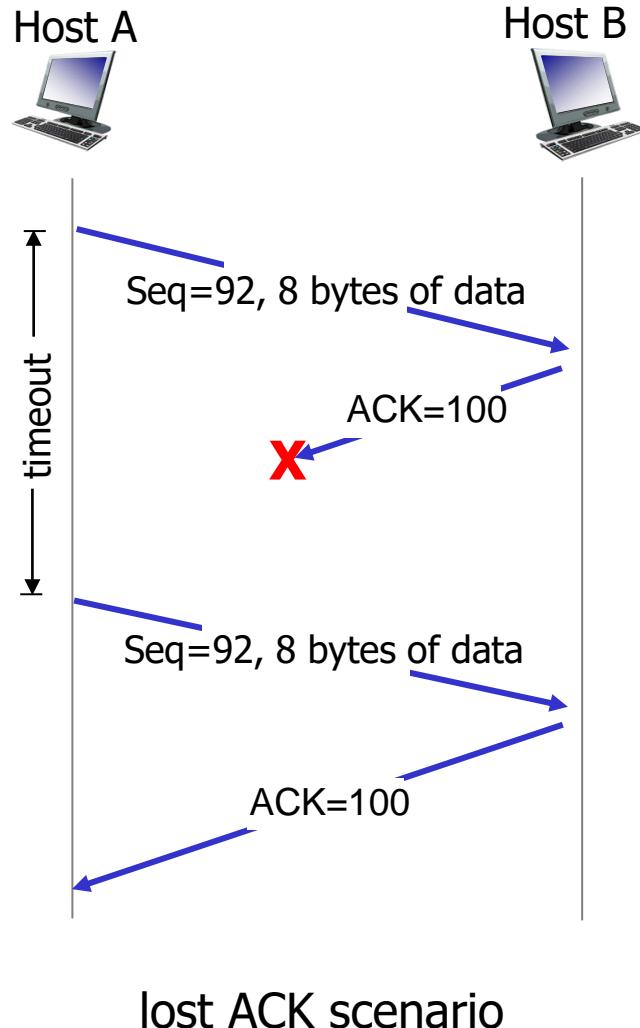
*event: ACK received*

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

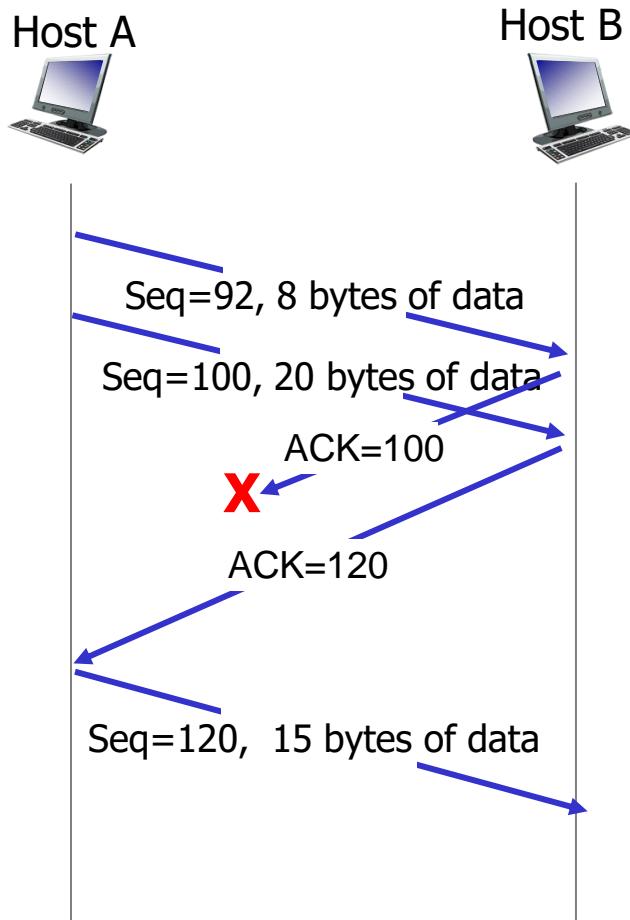
# TCP Receiver: ACK generation [RFC 5681]



# TCP: retransmission scenarios



# TCP: retransmission scenarios



cumulative ACK covers  
for earlier lost ACK

# TCP fast retransmit

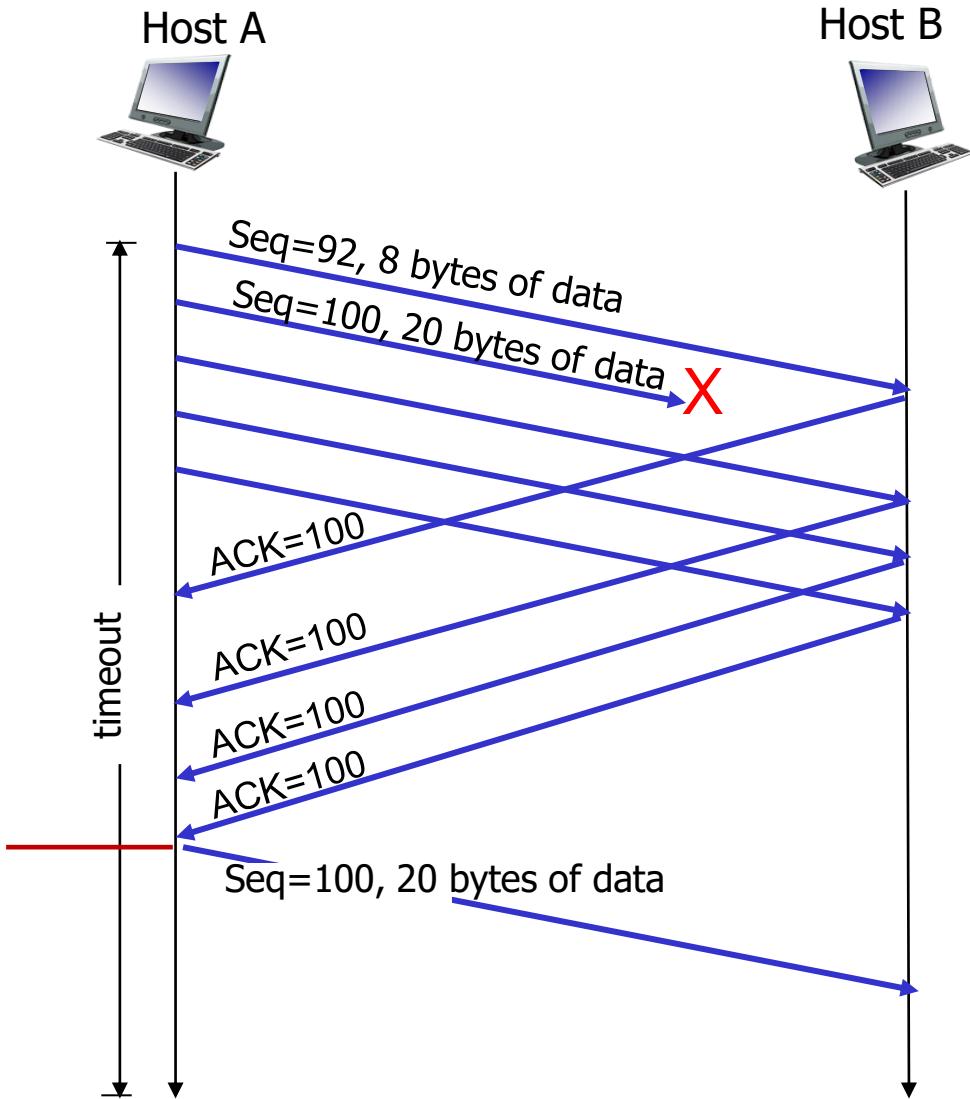
## *TCP fast retransmit*

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!



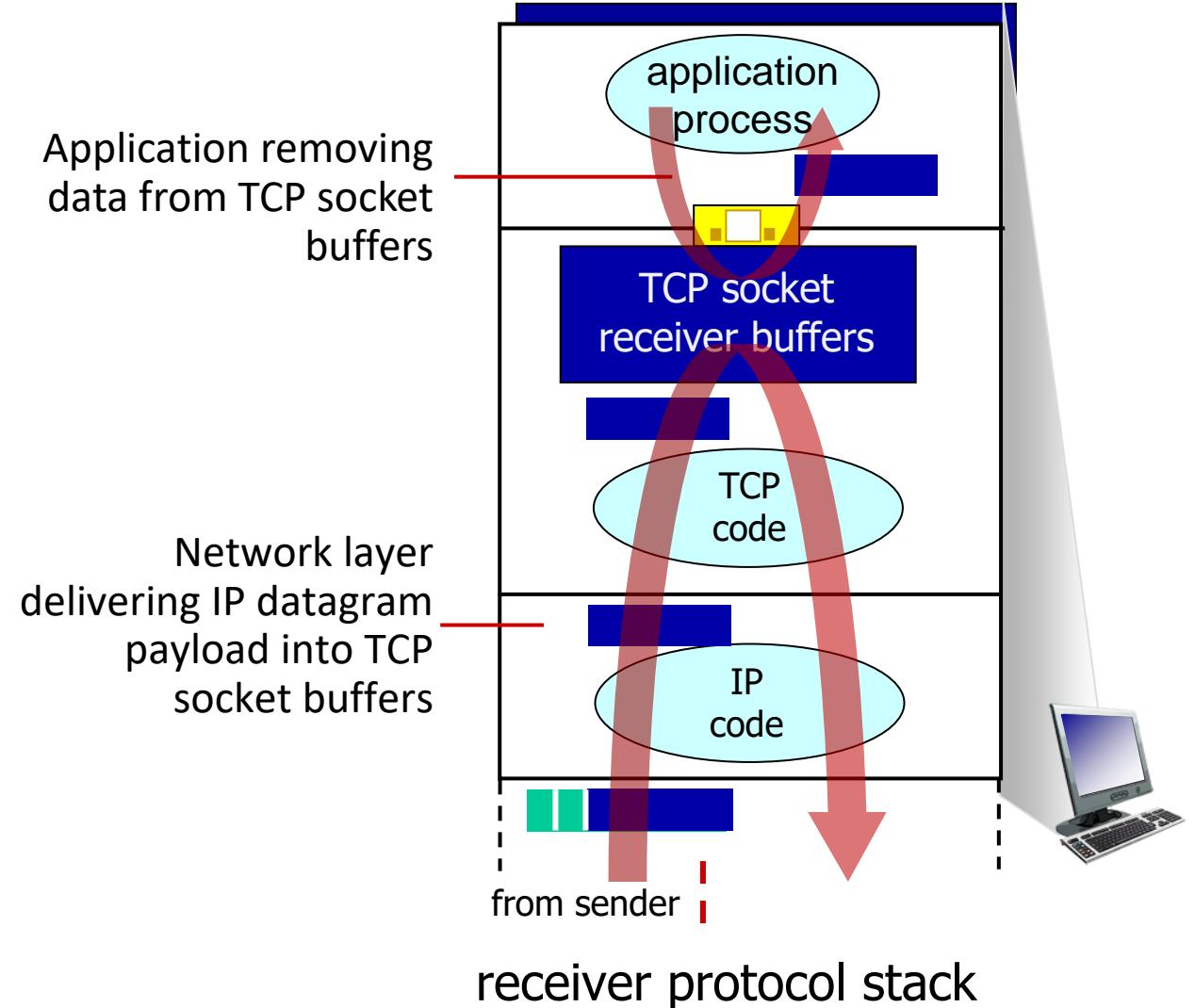
# Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- **Connection-oriented transport: TCP**
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control



# TCP flow control

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



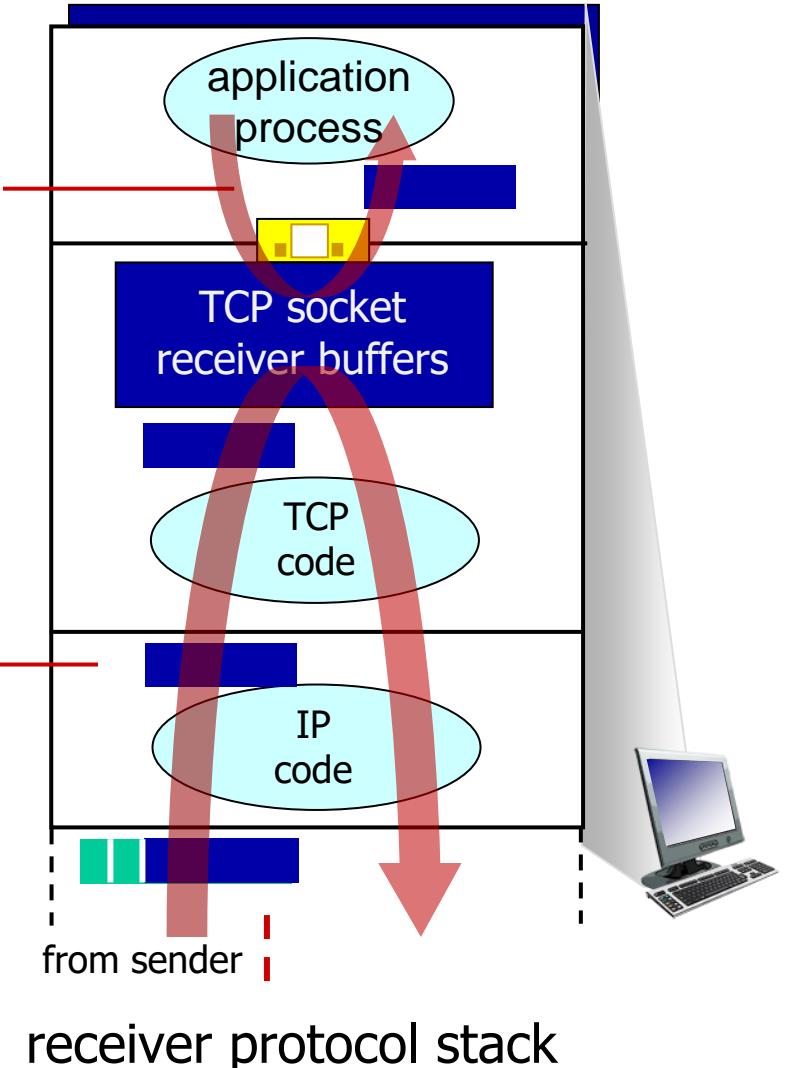
# TCP flow control

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



Application removing data from TCP socket buffers

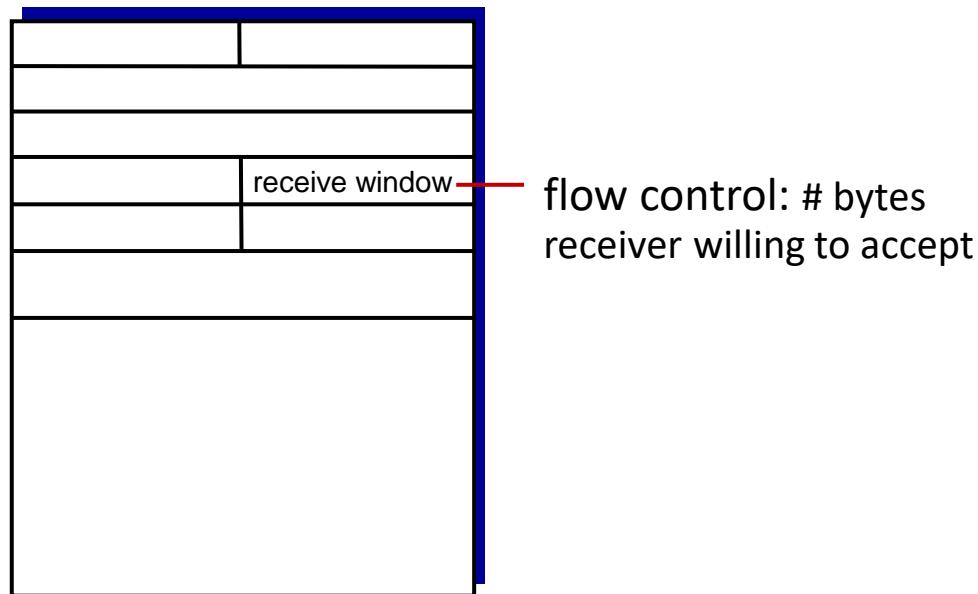
Network layer delivering IP datagram payload into TCP socket buffers



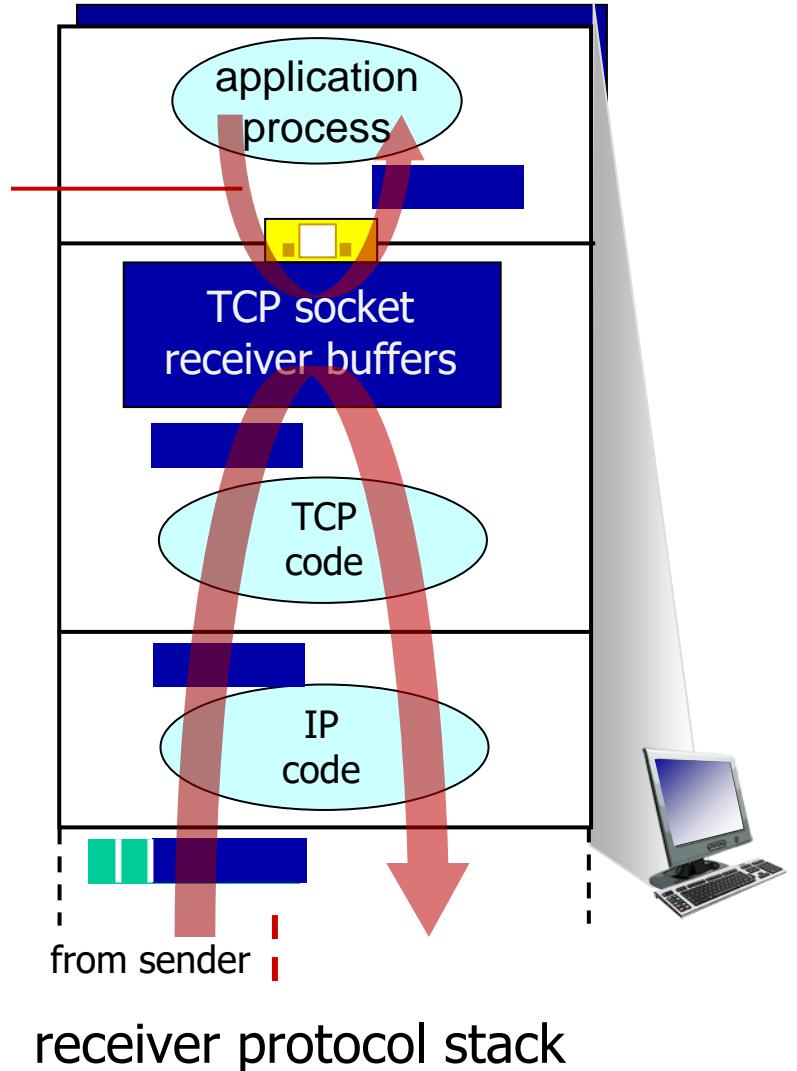
receiver protocol stack

# 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



receiver protocol stack

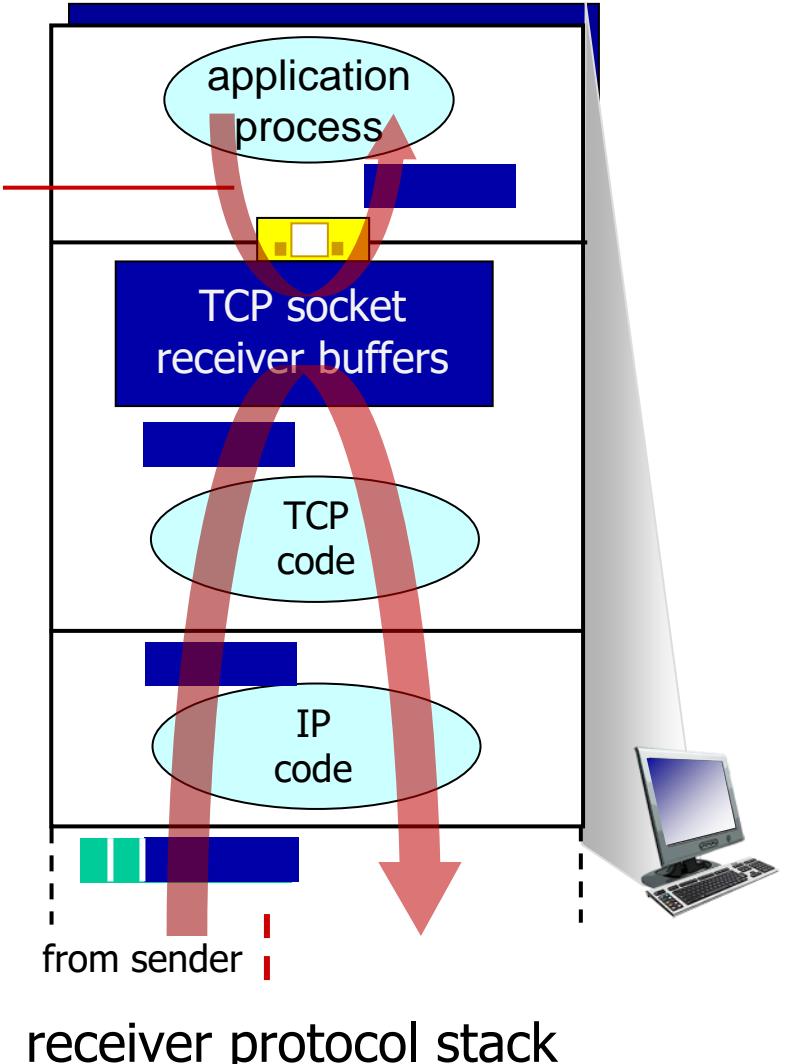
# TCP flow control

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

## flow control

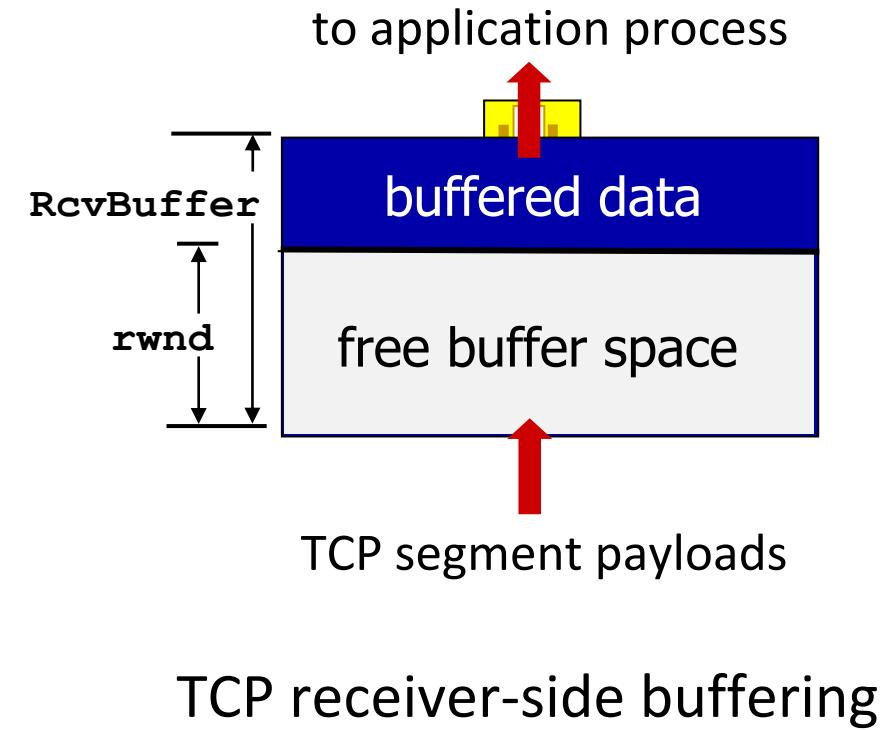
receiver controls sender, so  
*sender won't overflow receiver's buffer* by transmitting too much, too fast

Application removing data from TCP socket buffers



# TCP flow control

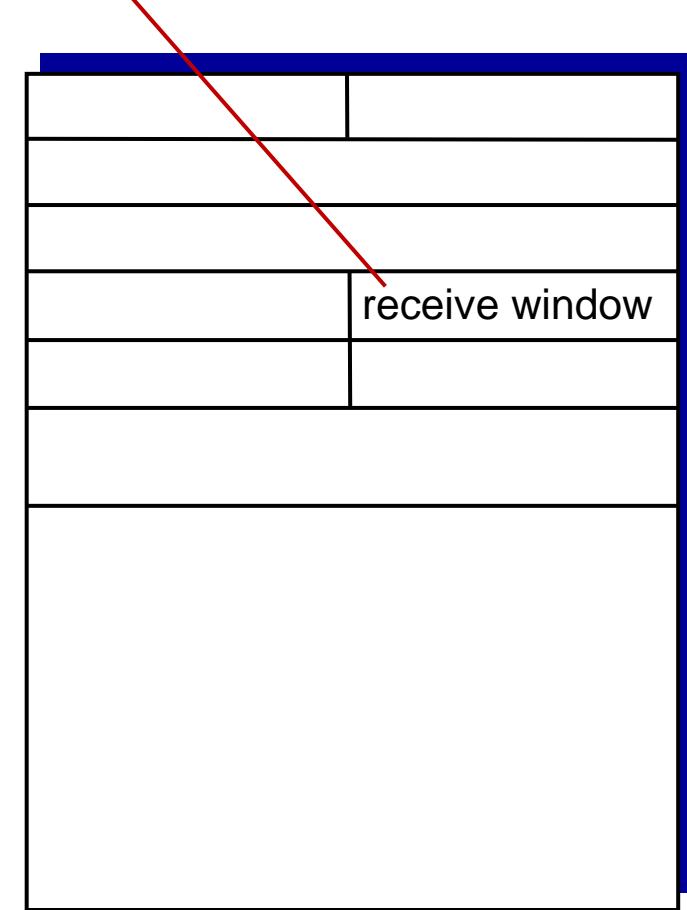
- TCP receiver “*advertises*” free buffer space in **rwnd** field in TCP header
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems auto-adjust **RcvBuffer**
- sender *limits* amount of unACKed (“in-flight”) data to received **rwnd**
  - guarantees 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

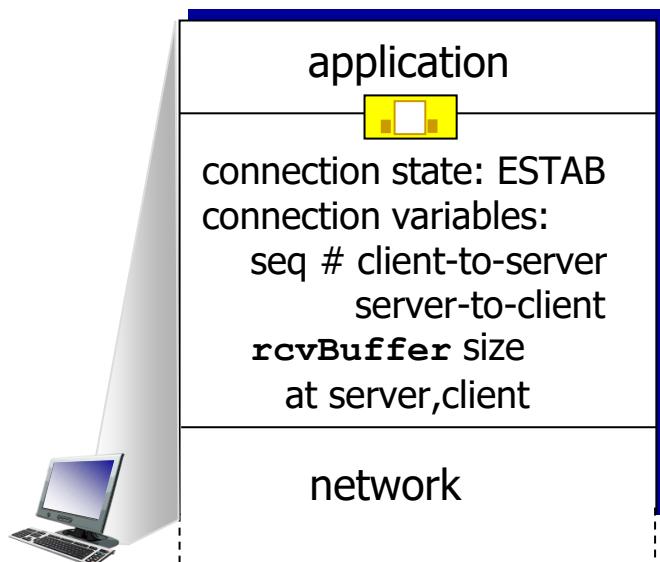


TCP segment format

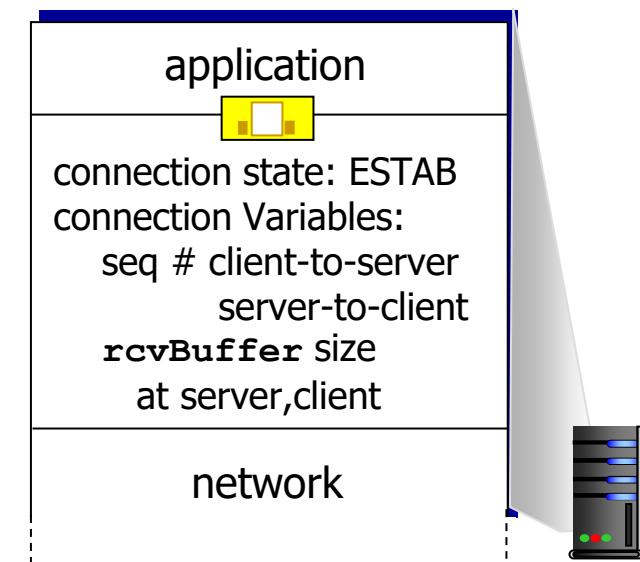
# TCP connection management

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)



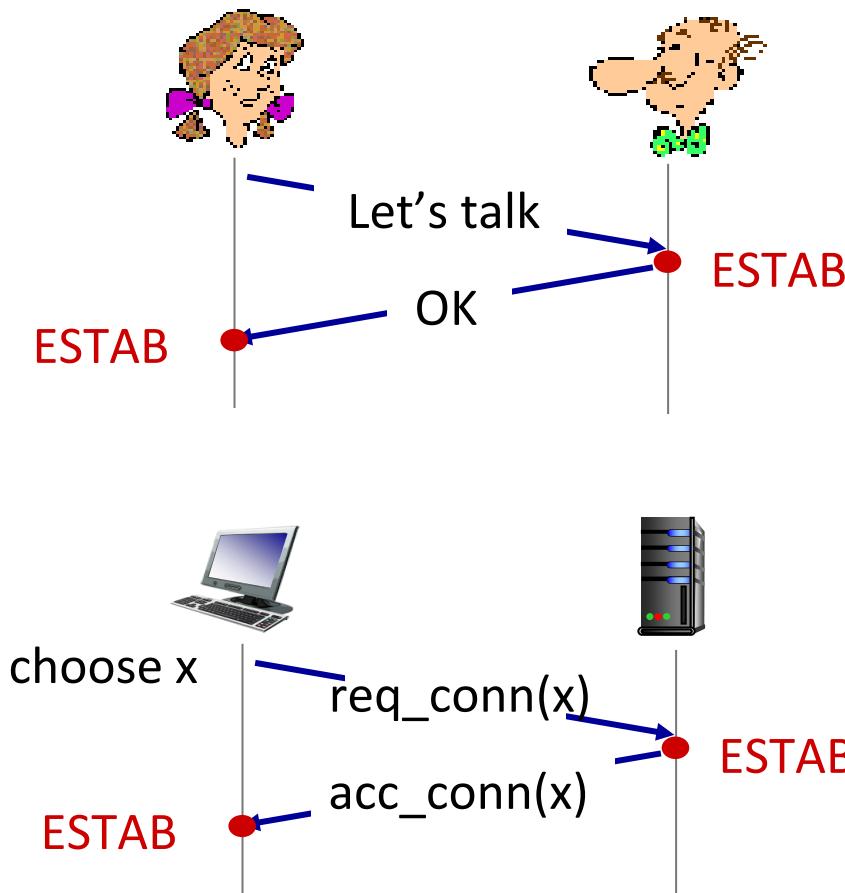
```
Socket clientSocket =  
    newSocket("hostname", "port number");
```



```
Socket connectionSocket =  
    welcomeSocket.accept();
```

# Agreeing to establish a connection

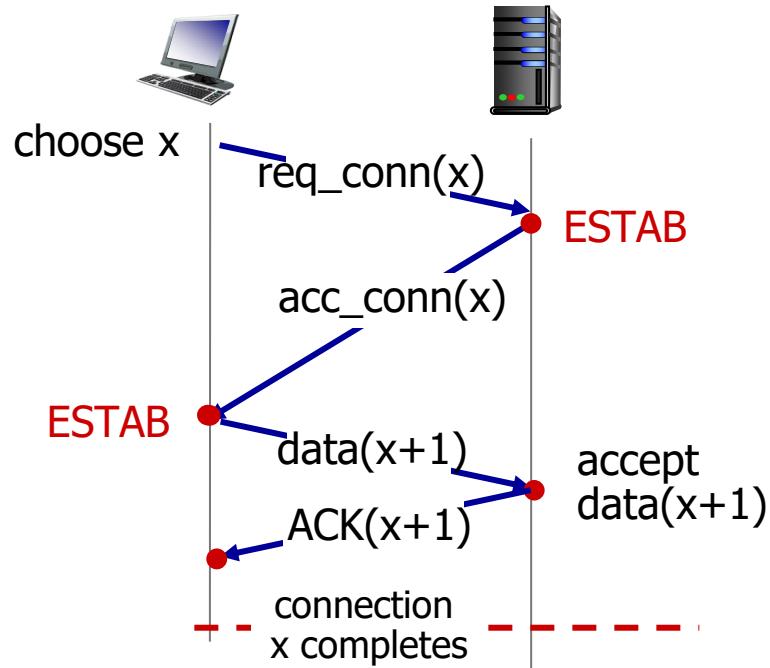
2-way handshake:



*Q: will 2-way handshake always work in network?*

- variable delays
- retransmitted messages (e.g., `req_conn(x)`) due to message loss
- message reordering
- *can't “see” other side*

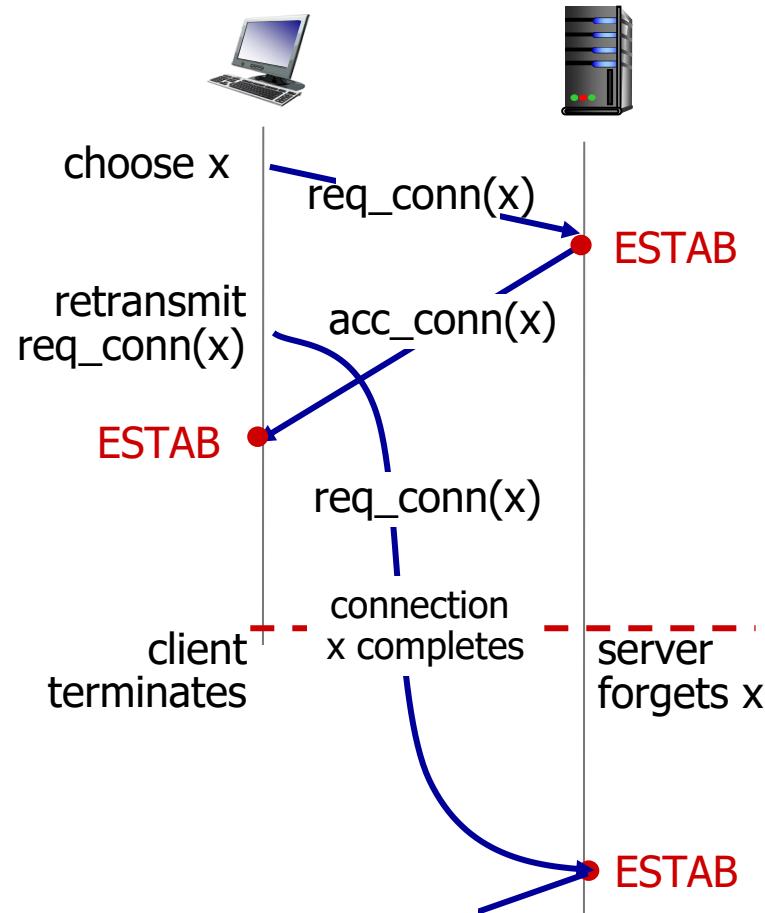
# 2-way handshake scenarios



No problem!

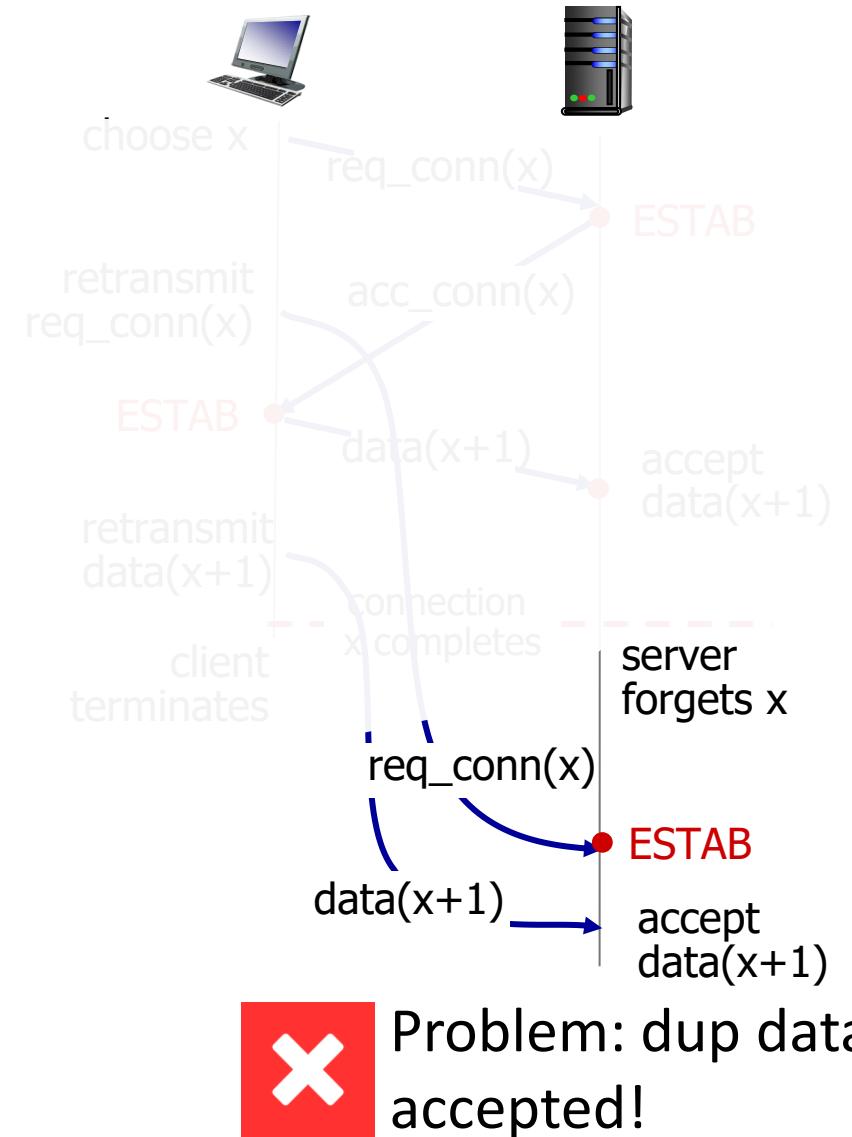


# 2-way handshake scenarios



Problem: half open  
connection! (no client)

# 2-way handshake scenarios



# TCP 3-way handshake

## Client state

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

LISTEN

```
clientSocket.connect((serverName, serverPort))
```

SYNSENT

choose init seq num,  $x$   
send TCP SYN msg



SYNbit=1, Seq= $x$

ESTAB

received SYNACK( $x$ )  
indicates server is live;  
send ACK for SYNACK;  
this segment may contain  
client-to-server data

SYNbit=1, Seq= $y$   
ACKbit=1; ACKnum= $x+1$

ACKbit=1, ACKnum= $y+1$

## Server state

```
serverSocket = socket(AF_INET, SOCK_STREAM)  
serverSocket.bind(('', serverPort))  
serverSocket.listen(1)  
connectionSocket, addr = serverSocket.accept()
```

LISTEN

SYN RCVD

choose init seq num,  $y$   
send TCP SYNACK  
msg, acking SYN

received ACK( $y$ )  
indicates client is live

ESTAB

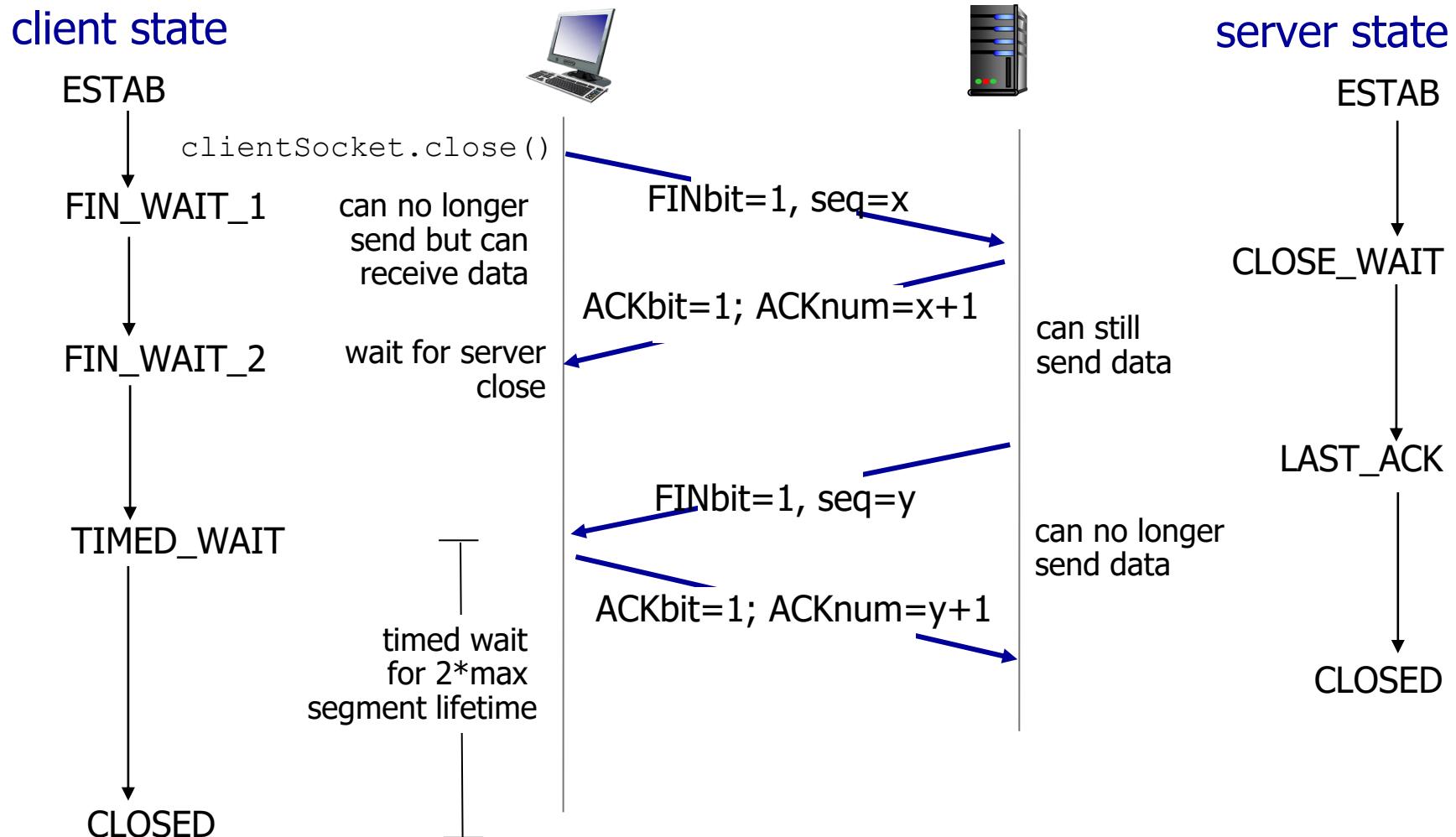
# A human 3-way handshake protocol



# Closing a TCP connection

- client, server *each closes 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*

# Closing a TCP connection



# Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- **Principles of congestion control**
- TCP congestion control
- Evolution of transport-layer functionality



# 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

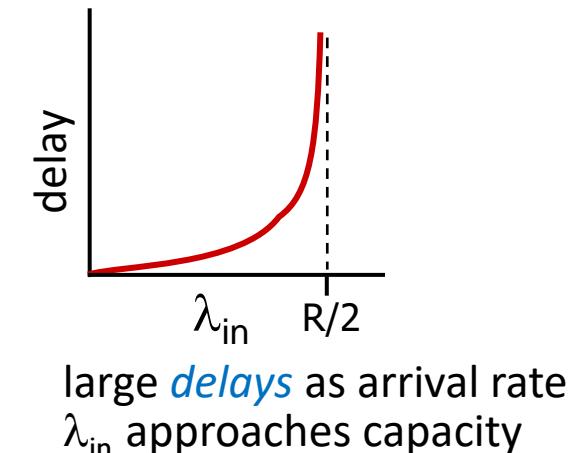
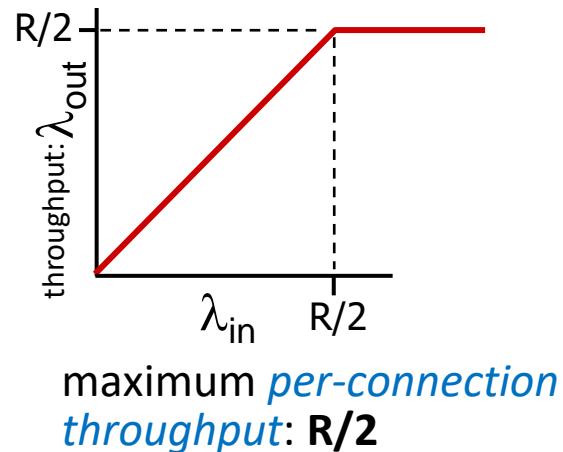
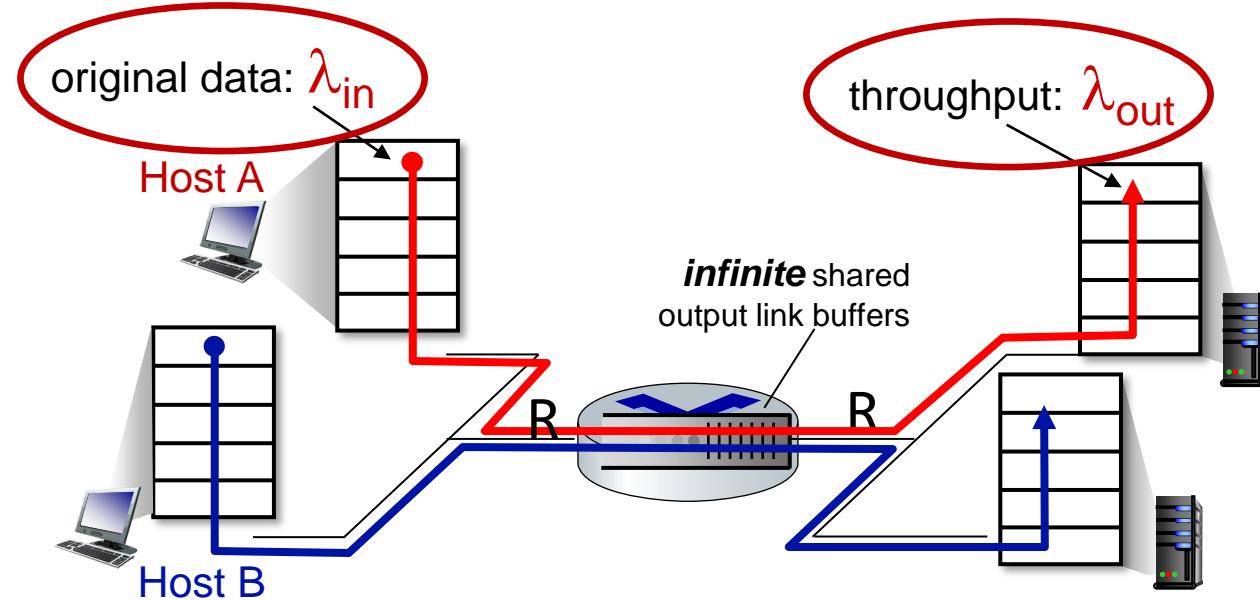
**flow control:** one sender  
too fast for one receiver

# Causes/costs of congestion: scenario 1

Simplest scenario:

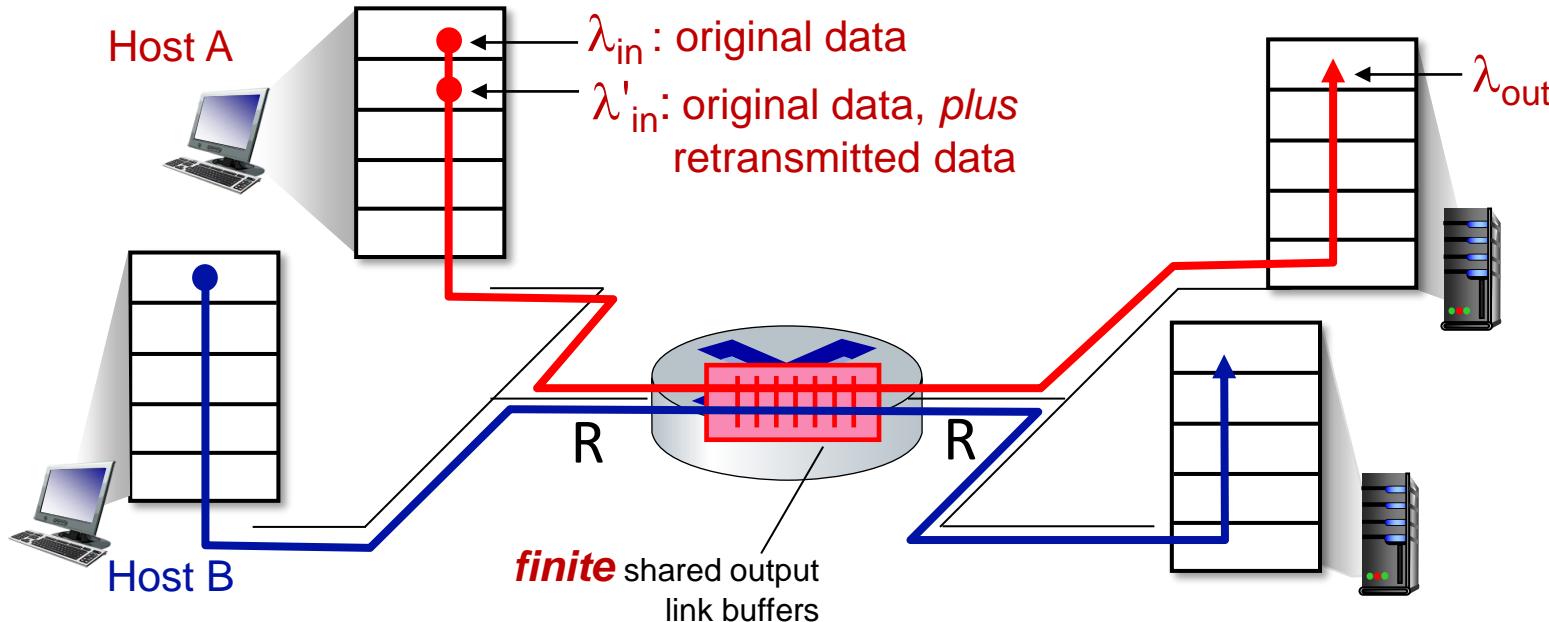
- one router, *infinite* buffers
  - no retransmissions needed
- input, output link capacity:  $R$ 
  - two flows: fairly,  $R/2$

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



# Causes/costs of congestion: scenario 2

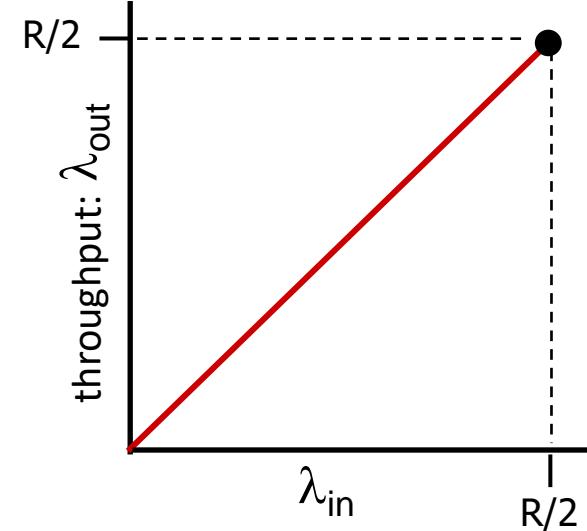
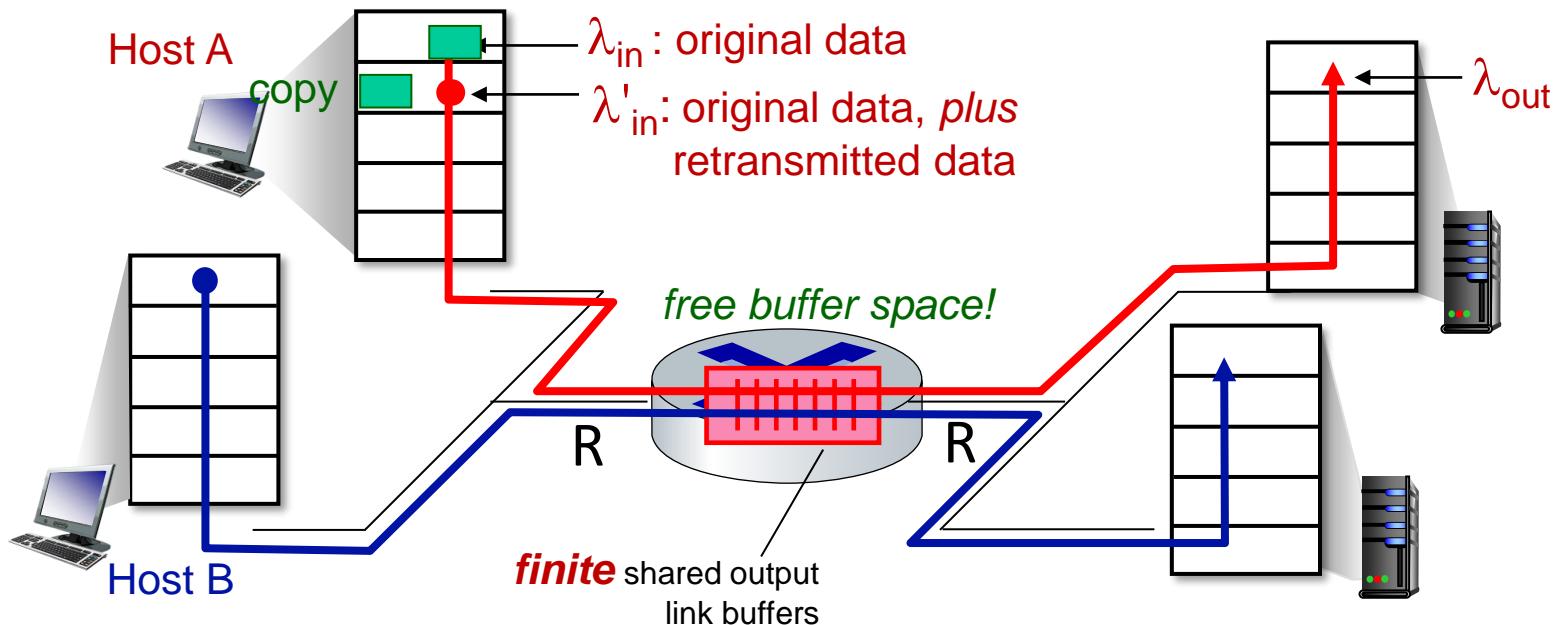
- one router, *finite* buffers
- sender *retransmits lost, timed-out packet*
  - application-layer input = application-layer output:  $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* :  $\lambda'_{in} \geq \lambda_{in}$



# Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

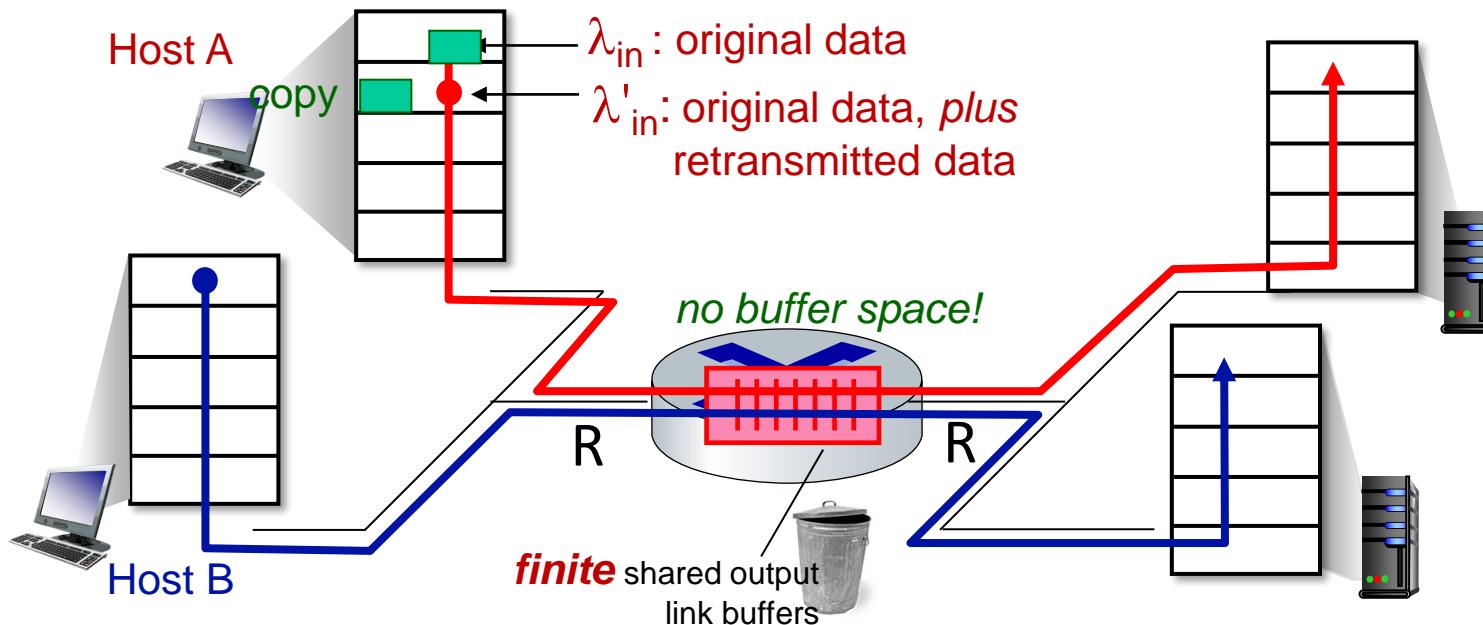
- sender sends *only* when router *buffers available*



# Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

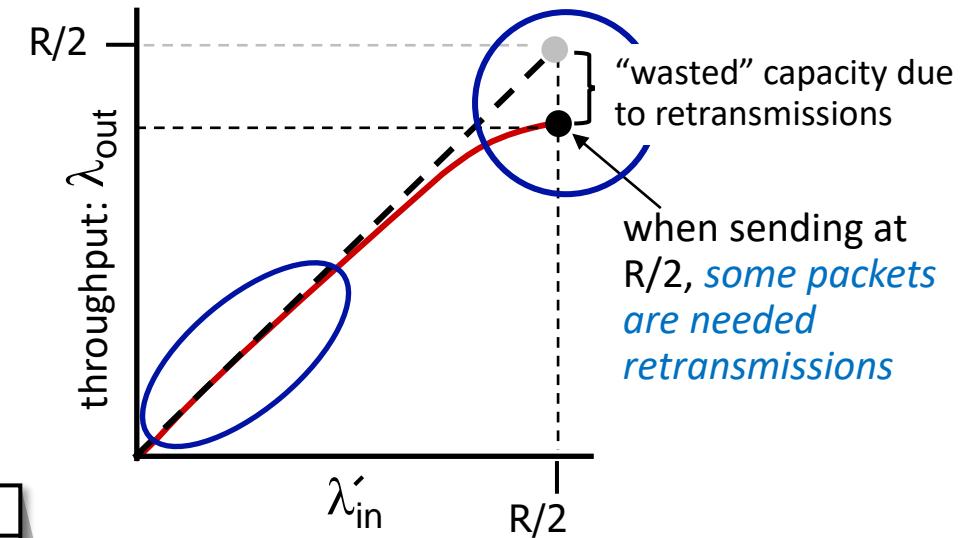
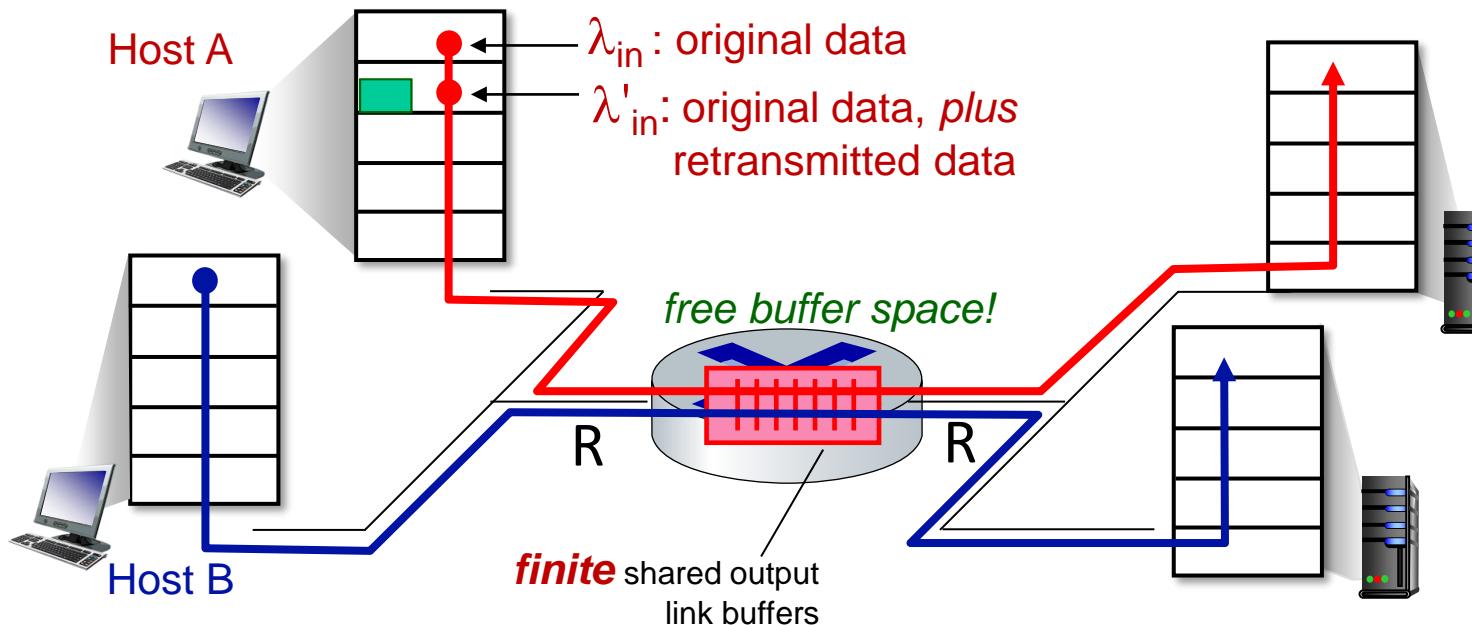
- packets can be *lost* (dropped at router) due to full buffers
- sender knows when packet has been dropped*: only resends if packet known to be lost



# Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

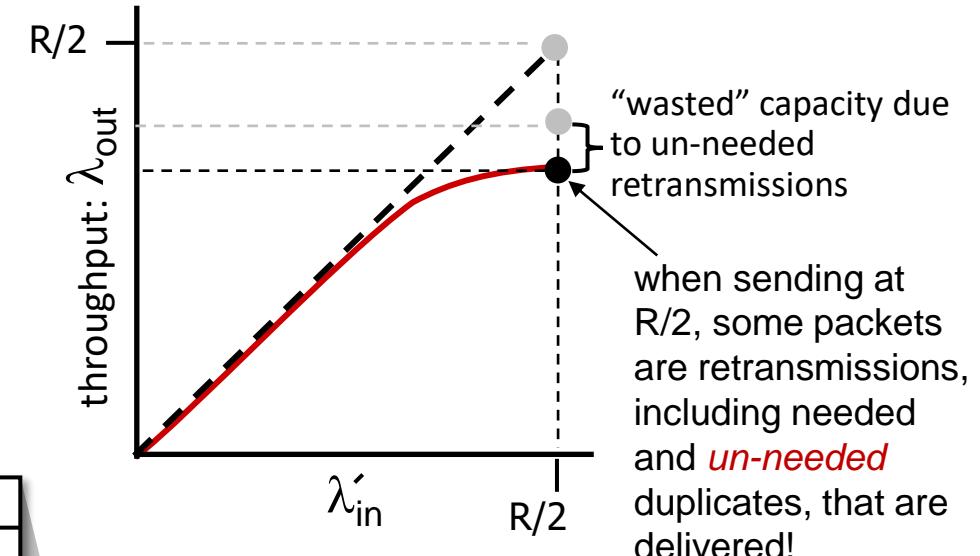
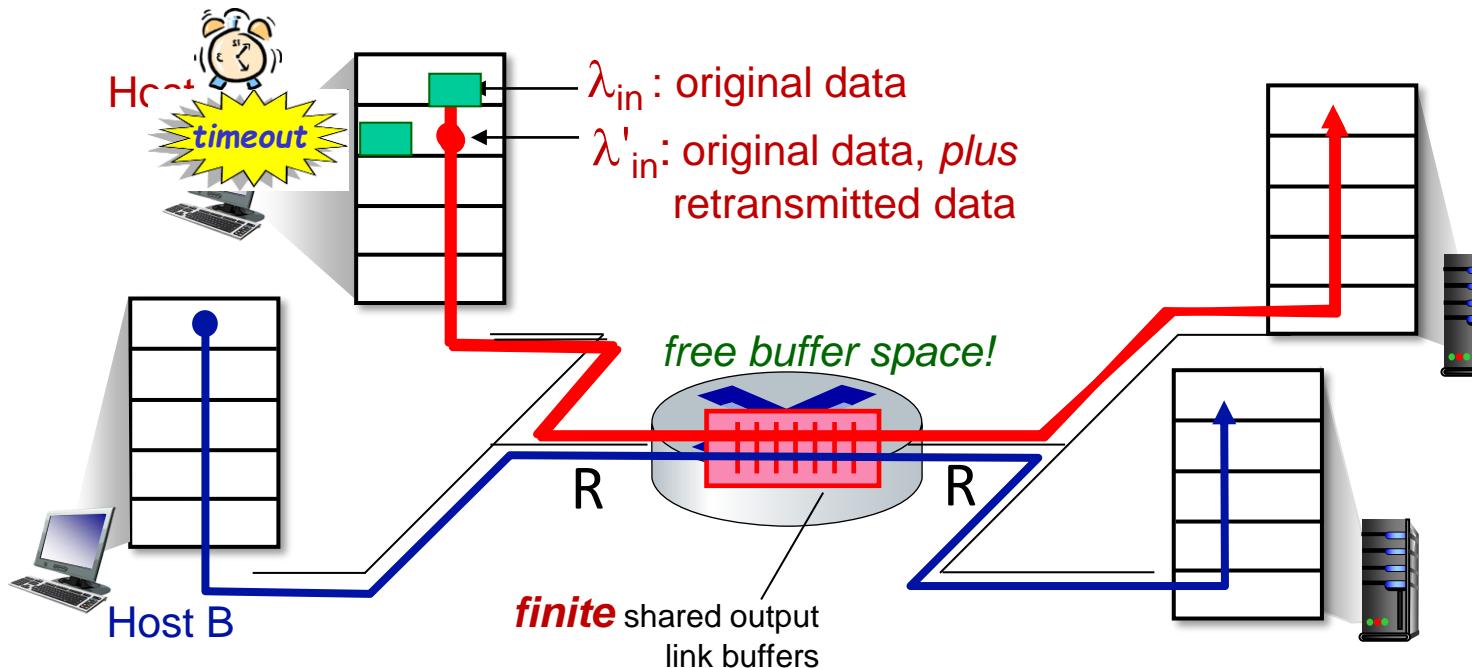
- packets can be *lost* (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost



# Causes/costs of congestion: scenario 2

## Realistic scenario: *un-needed duplicates*

- packets can be *lost*, dropped at router due to full buffers – requiring *retransmissions*
- but sender timer can *time out prematurely*, sending *two* copies, *both* of which are *delivered*



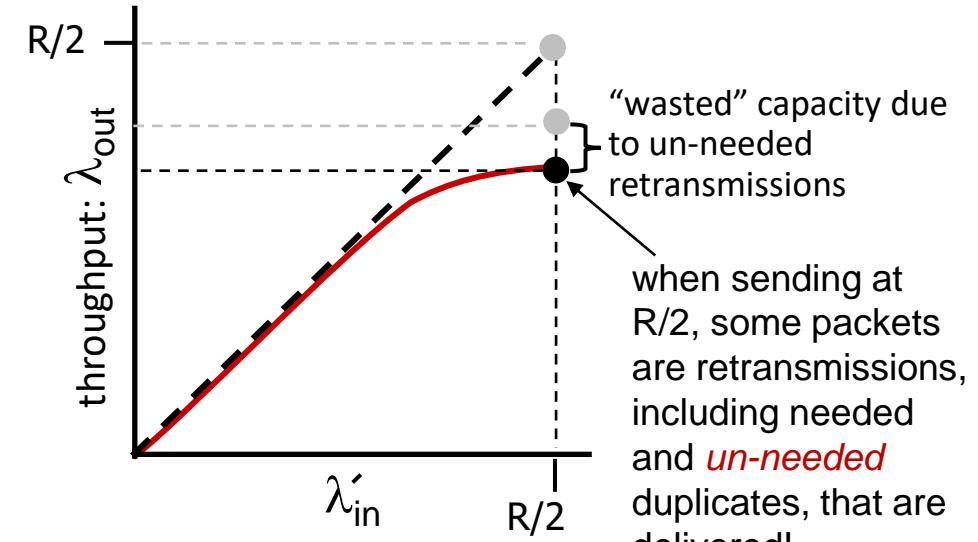
# Causes/costs of congestion: scenario 2

## Realistic scenario: *un-needed duplicates*

- packets can be *lost*, dropped at router due to full buffers – requiring retransmissions
- but sender timer can *time out prematurely*, sending *two* copies, *both* of which are *delivered*

## “costs” of congestion:

- more work (*retransmission*) for given receiver throughput
- *unneeded retransmissions*: link carries multiple copies of a packet
  - *decreasing maximum achievable throughput*

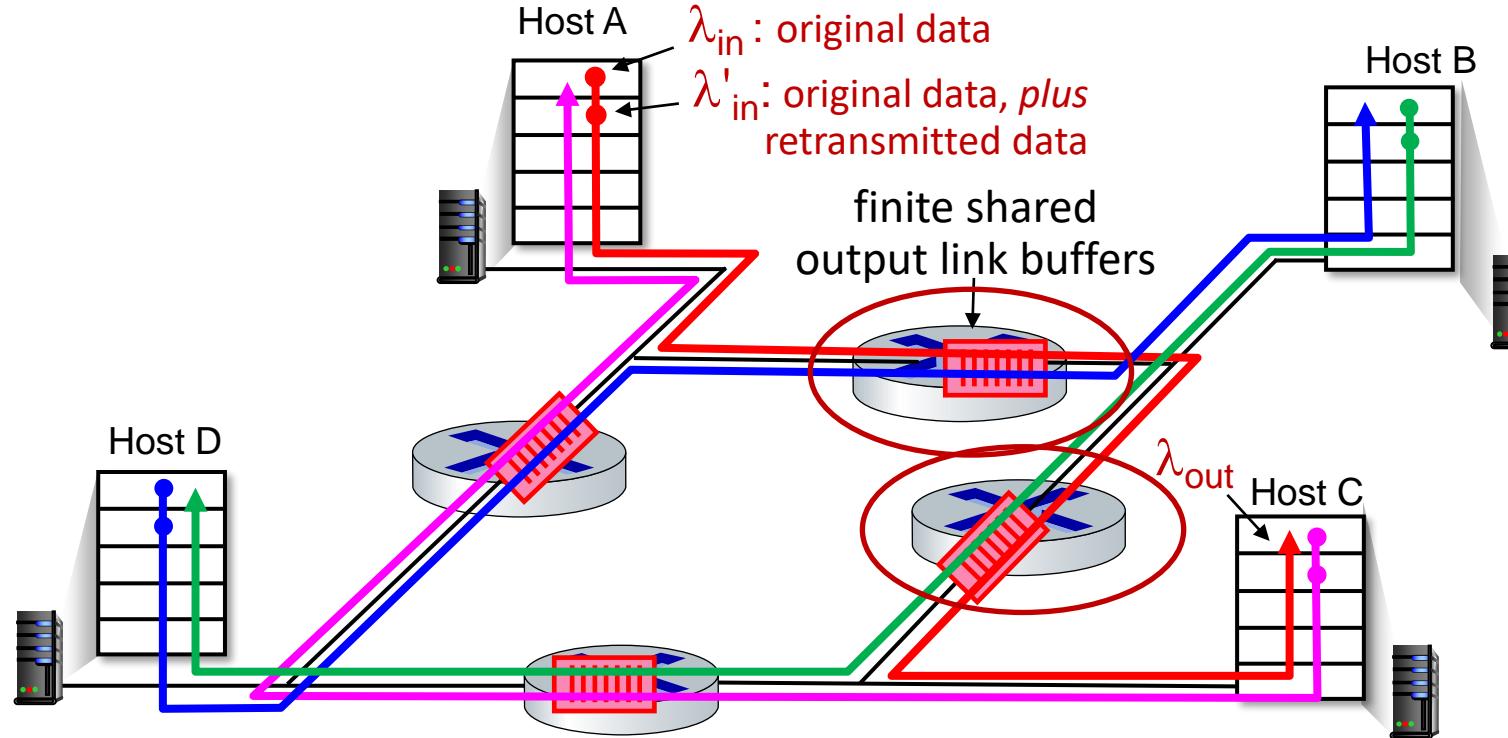


# Causes/costs of congestion: scenario 3

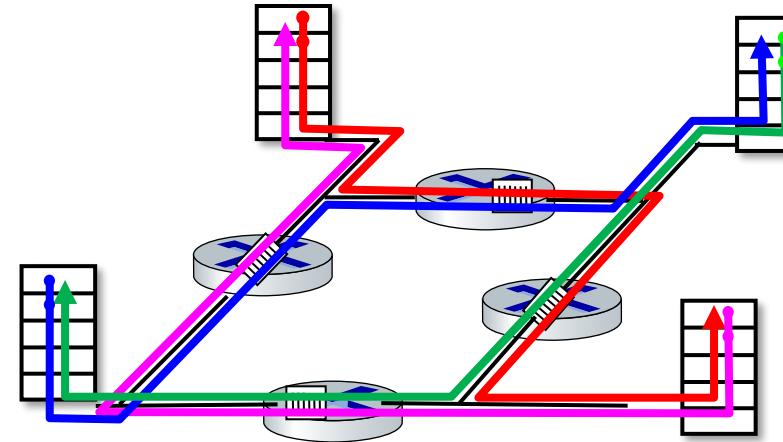
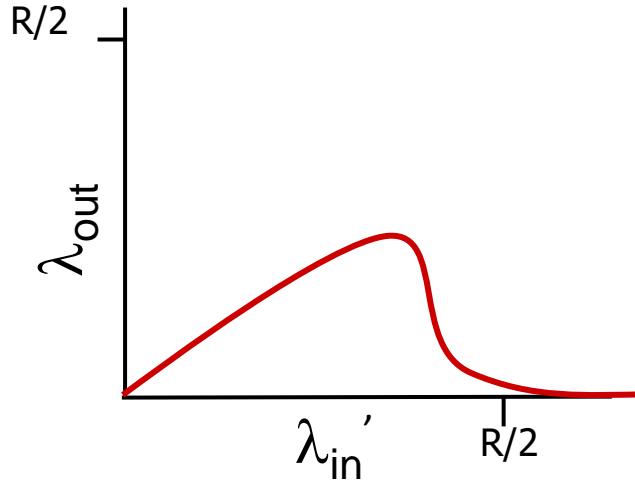
- *four senders*
- *multi-hop paths*
- *timeout/retransmit*

**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: scenario 3

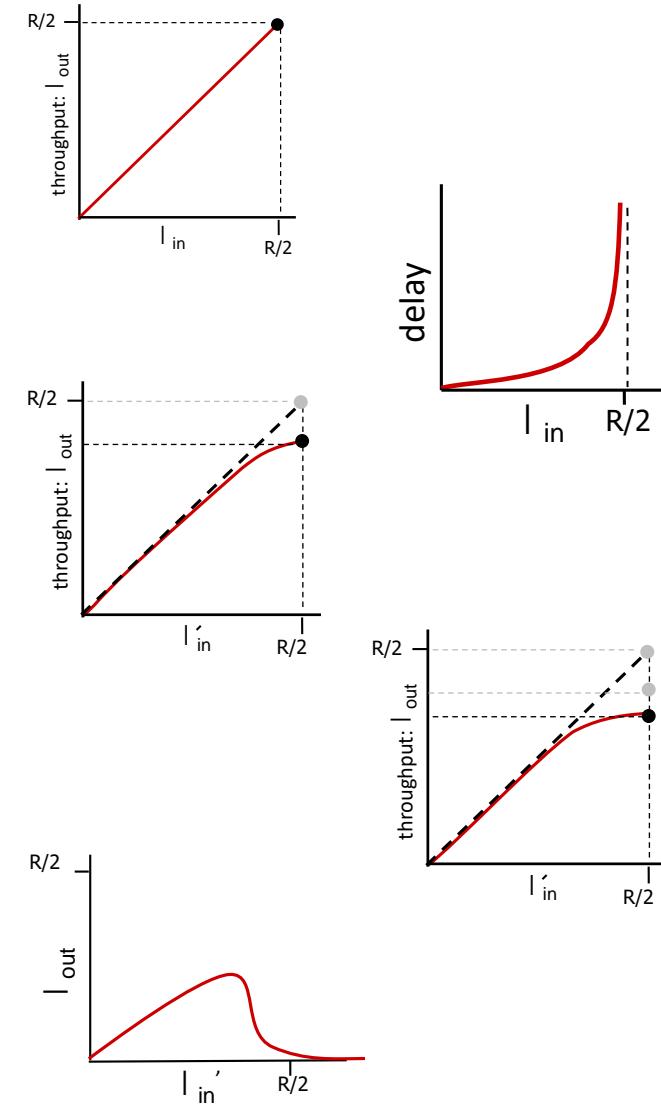


another “cost” of congestion:

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

# Causes/costs of congestion: insights

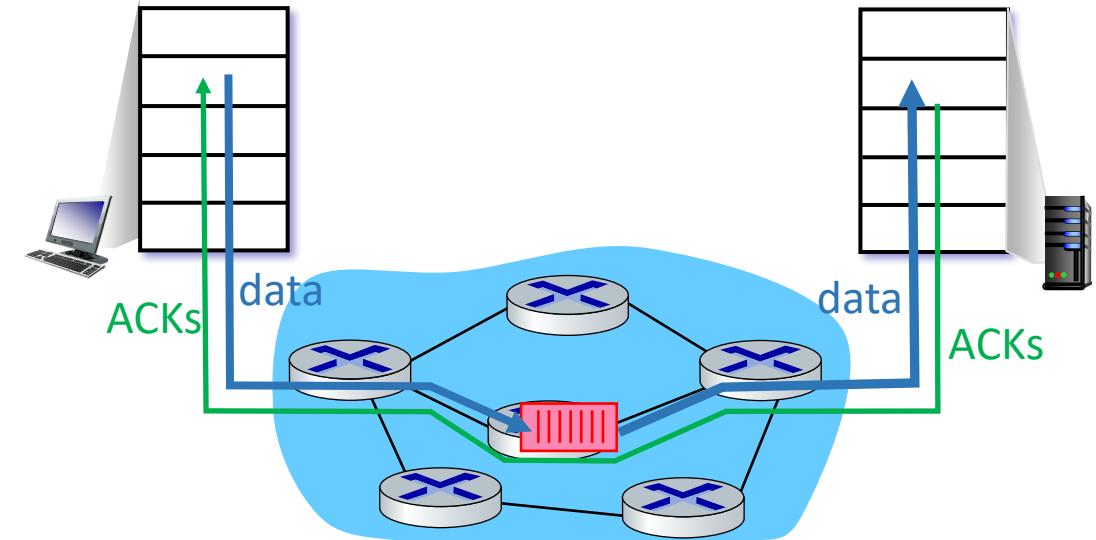
- *throughput* can never exceed *capacity*
- *delay* increases as capacity approached
- *loss/retransmission* decreases *effective throughput*
- *un-needed duplicates* further decreases *effective throughput*
- upstream transmission capacity/buffering *wasted* for packets lost downstream



# Approaches towards congestion control

## End-end congestion control:

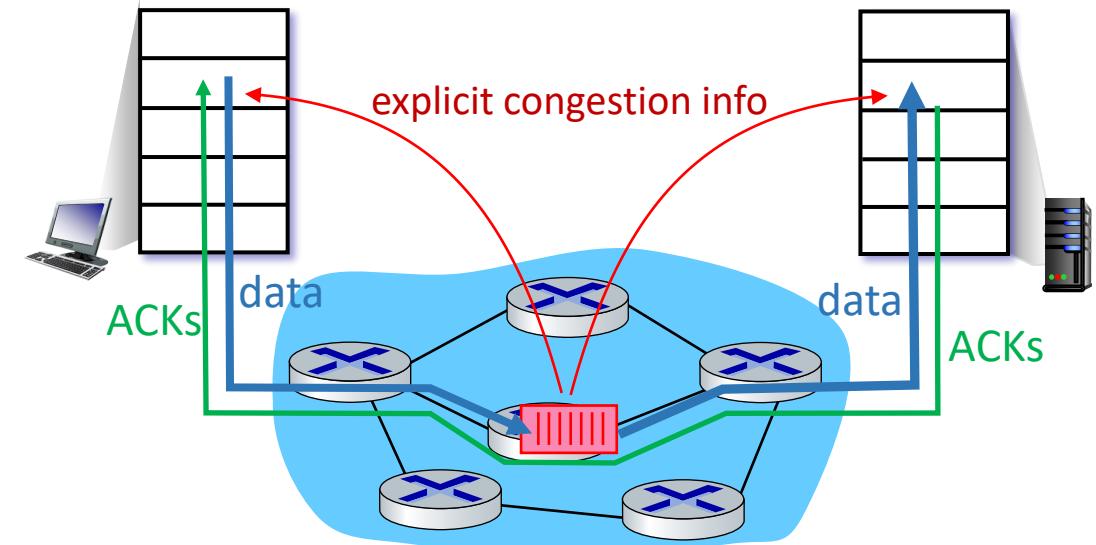
- *no explicit feedback* from network
- congestion *inferred* from *observed loss, delay*
  - approach taken by TCP



# Approaches towards congestion control

## 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*
  - e.g., TCP ECN, ATM, DECbit protocols



# Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- **TCP congestion control**
- Evolution of transport-layer functionality



# TCP congestion control: AIMD

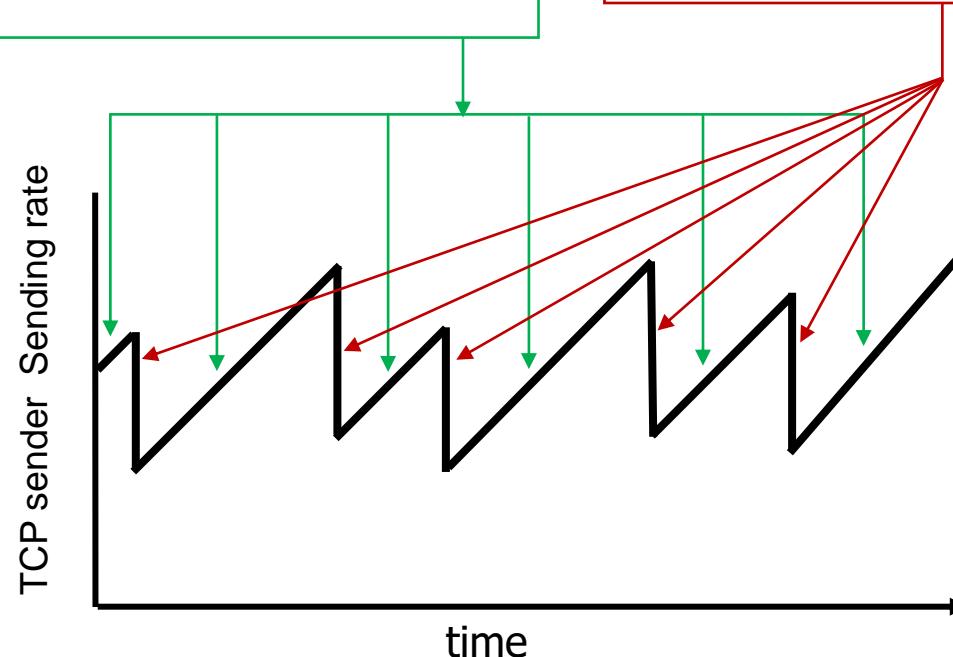
- *approach:* senders can *increase sending rate* until packet loss (congestion) occurs, then *decrease sending rate* on loss event

## Additive Increase

increase sending rate by 1  
*maximum segment size (MSS)*  
every RTT until loss detected

## Multiplicative Decrease

cut sending rate in half at  
each loss event



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

# TCP AIMD: more

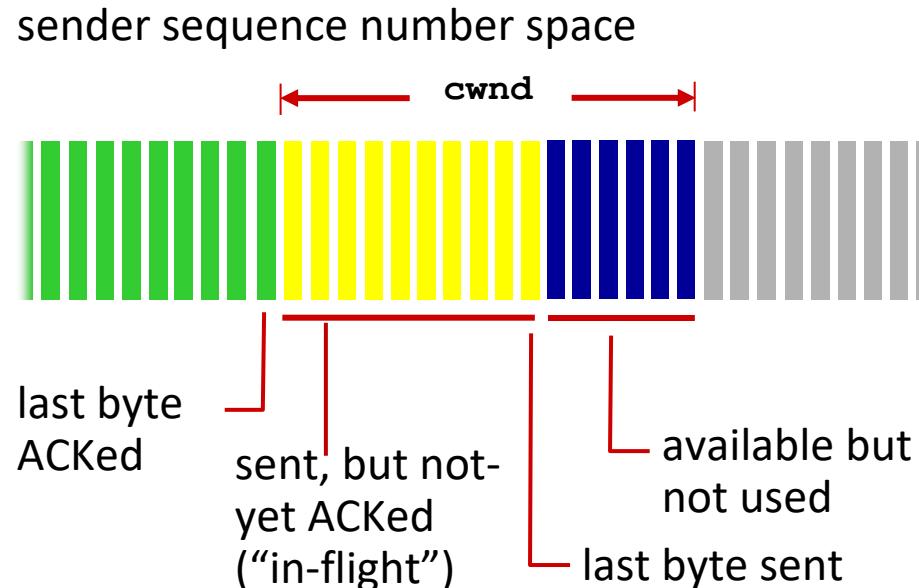
*Multiplicative decrease* detail: 2 events, sending rate is

- 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:

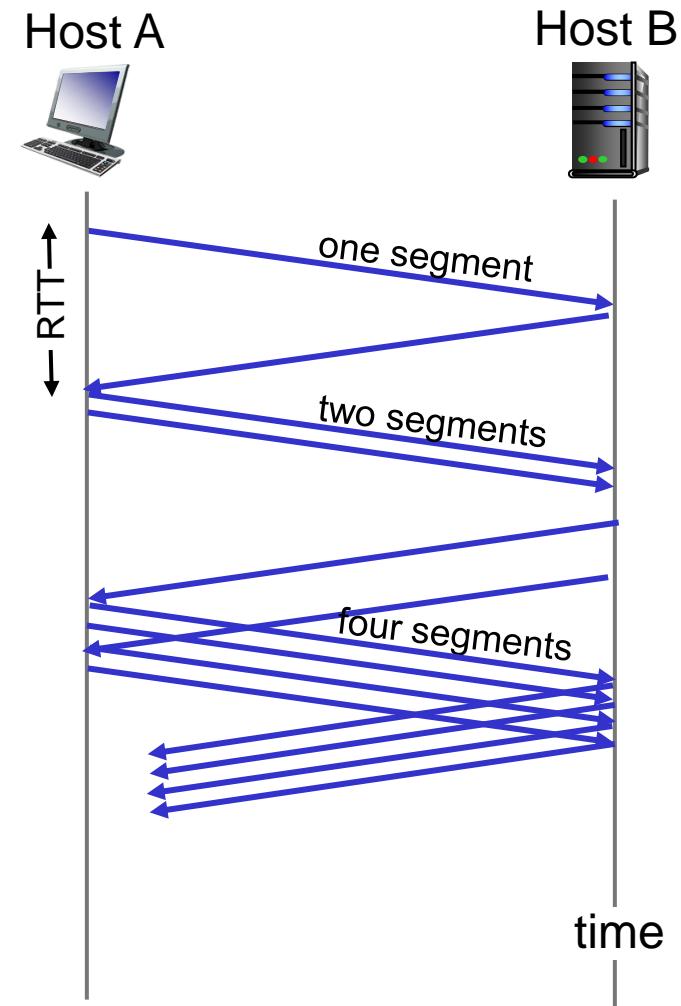
- *roughly*: send **cwnd** bytes, wait RTT for ACKs, then send more bytes

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

- 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



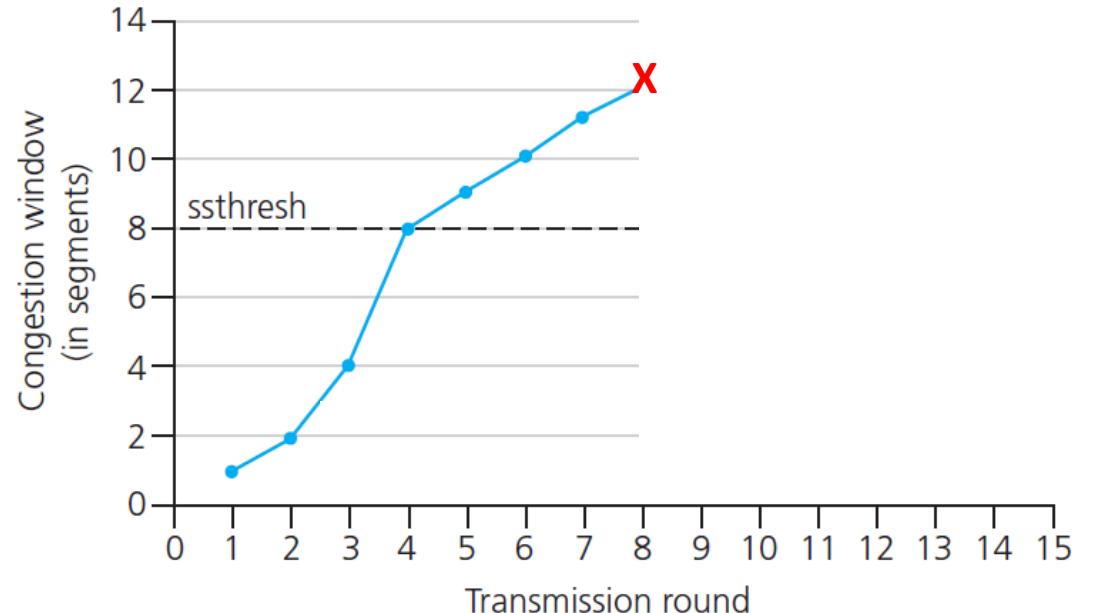
# TCP: from slow start to congestion avoidance

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

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

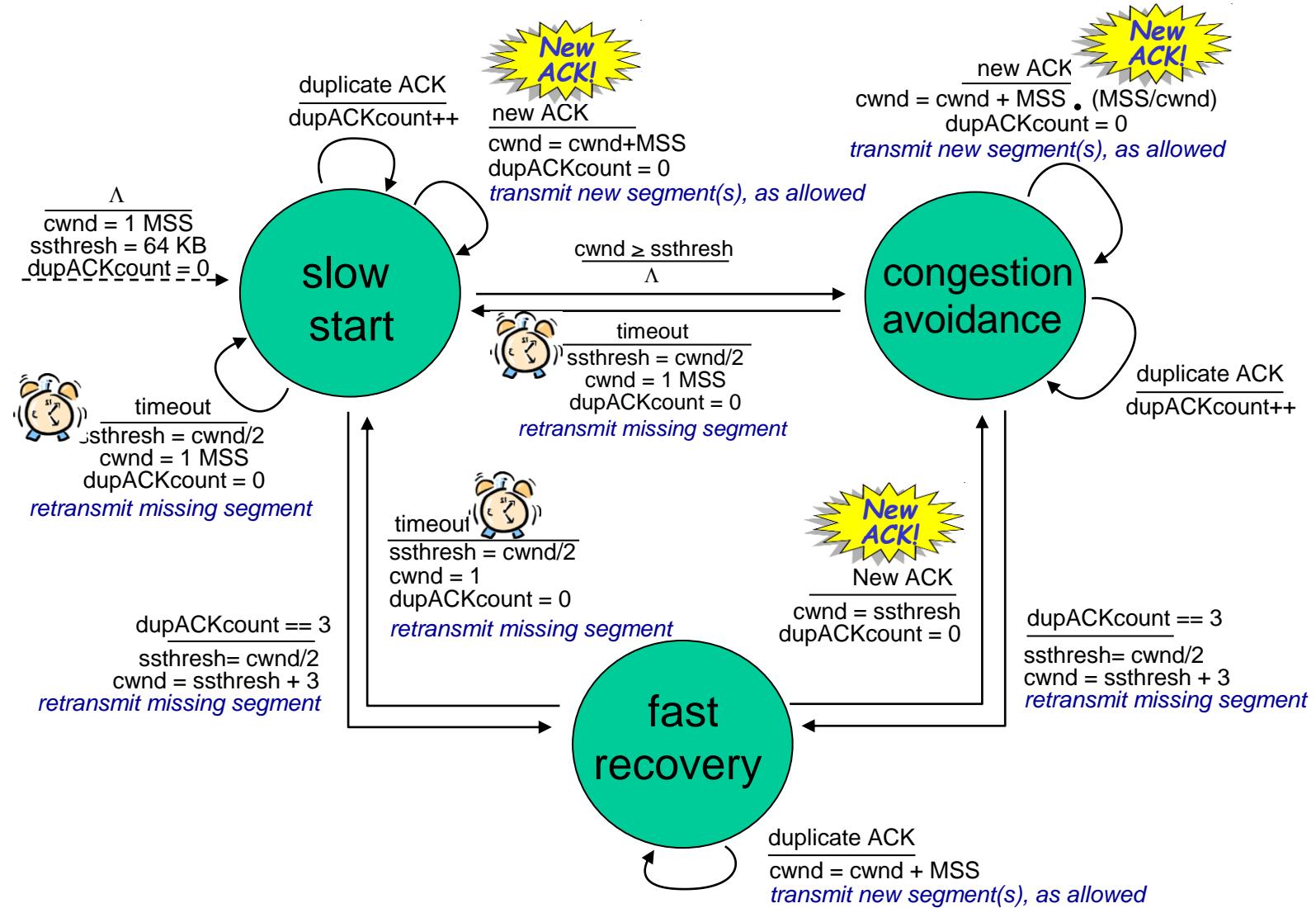
## Implementation:

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



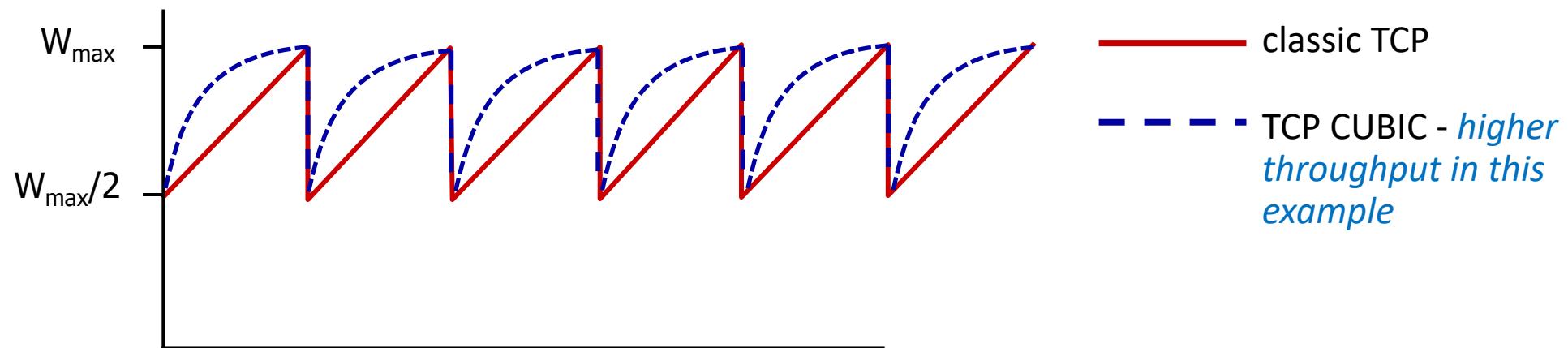
\* Check out the online interactive exercises for more examples: [http://gaia.cs.umass.edu/kurose\\_ross/interactive/](http://gaia.cs.umass.edu/kurose_ross/interactive/)

# Summary: TCP congestion control



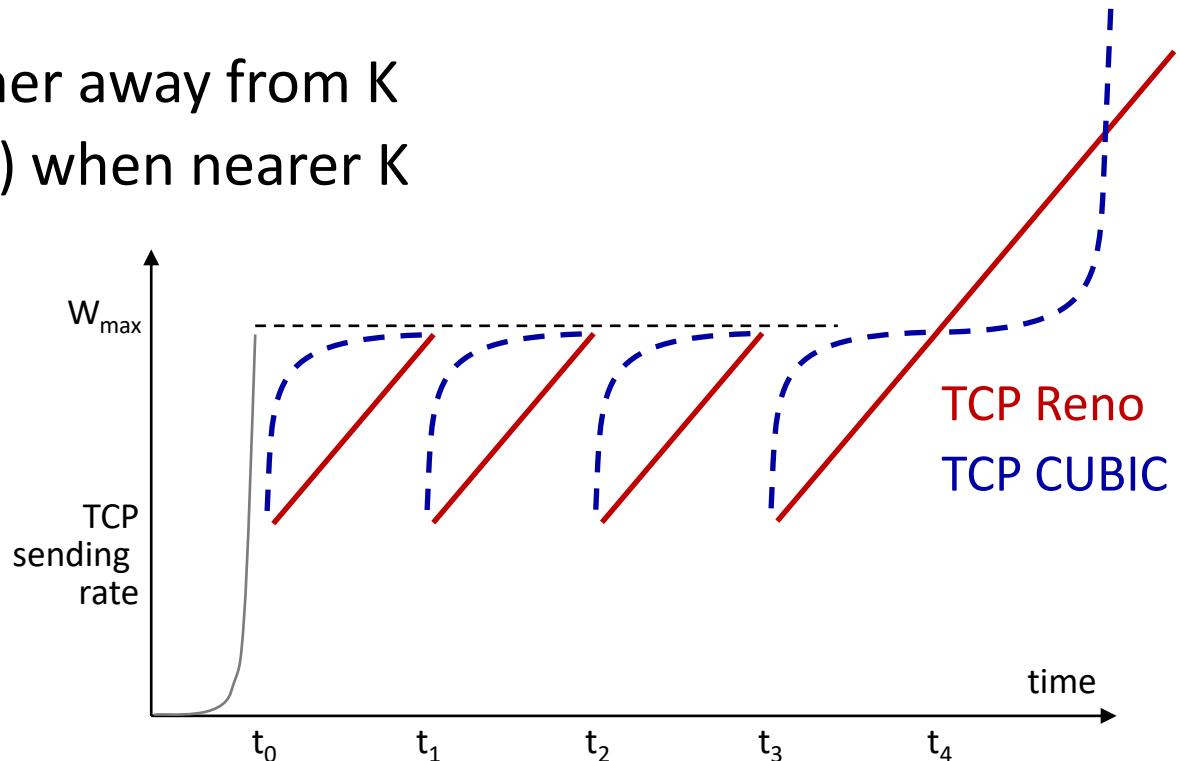
# TCP CUBIC

- 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 to  $W_{\max}$  *faster*, but then approach  $W_{\max}$  more *slowly*



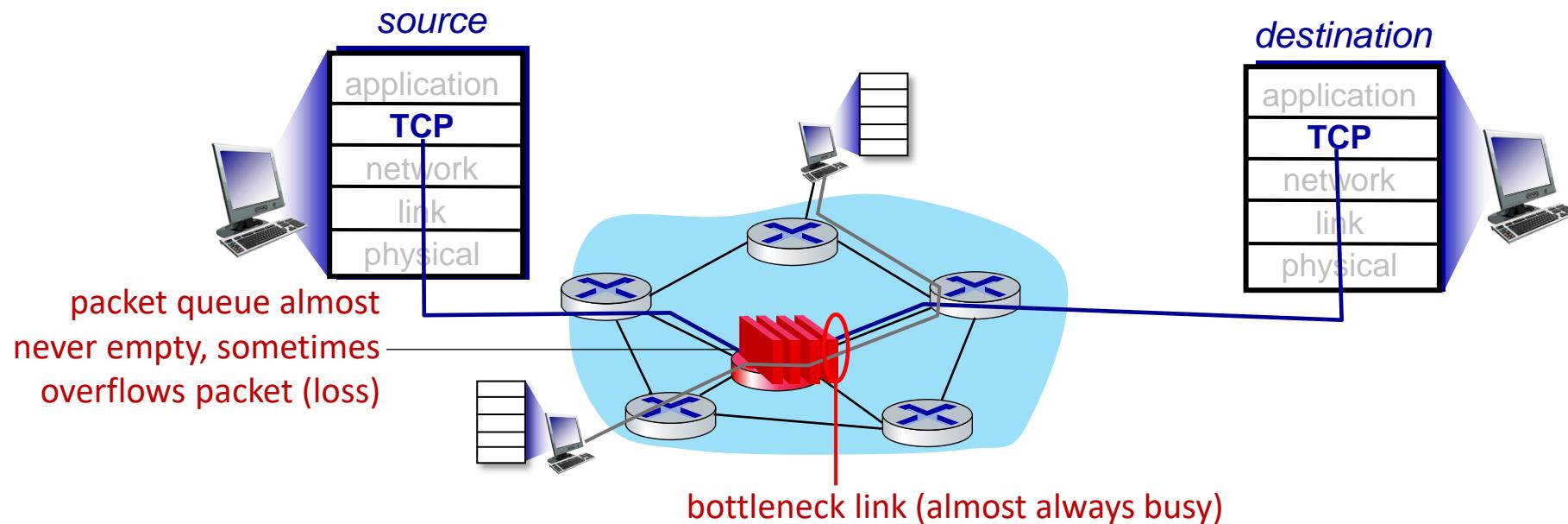
# TCP CUBIC

- $K$ : point in time when *TCP window size* will reach  $W_{\max}$ 
  - $K$  itself is tunable
- *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



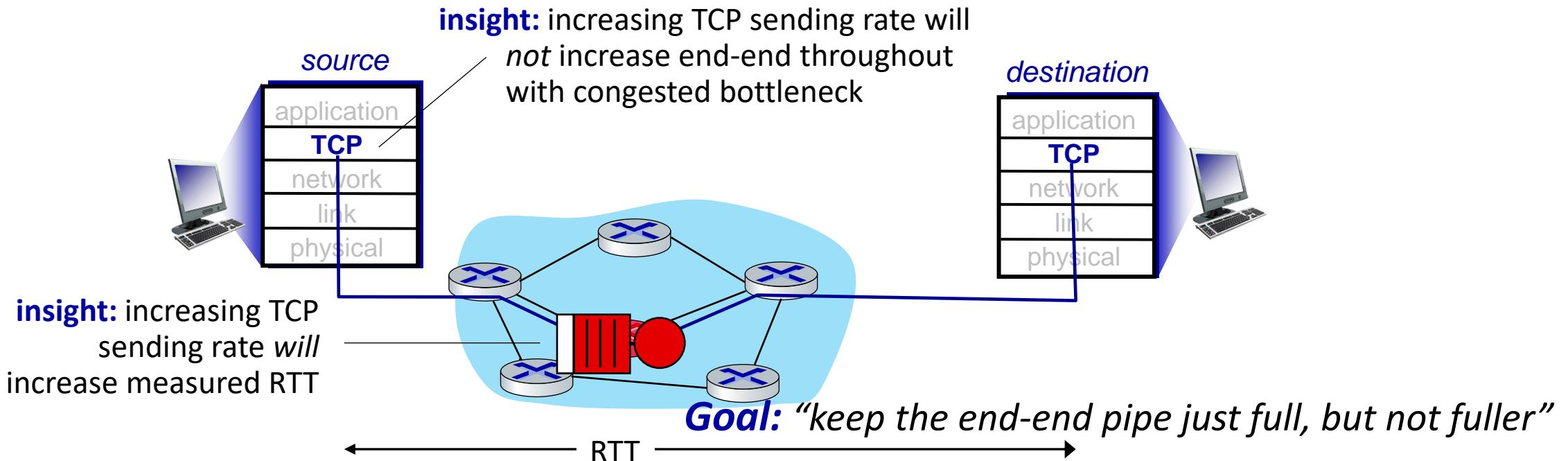
# TCP and the congested “bottleneck link”

- TCP (classic, CUBIC) *increase TCP's sending rate* until packet loss occurs at some router's output: the *bottleneck link*



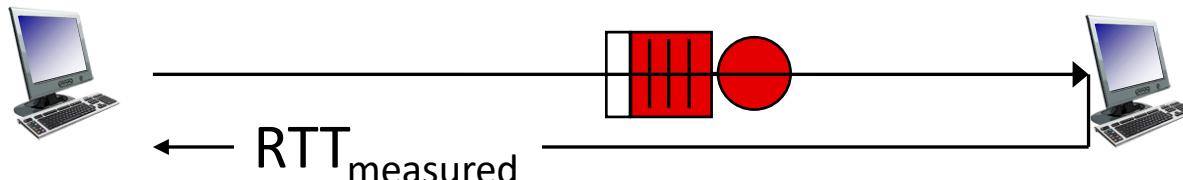
# TCP and the congested “bottleneck link”

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*
- understanding congestion: useful to focus on congested bottleneck link



# Delay-based TCP congestion control

Keeping sender-to-receiver pipe “just full enough, but no fuller”: keep bottleneck link *busy transmitting, but avoid high delays/buffering*



$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{\text{RTT}_{\text{measured}}}$$

## Delay-based approach:

- $\text{RTT}_{\min}$  - minimum observed RTT (uncongested path)
- uncongested throughput with congestion window **cwnd** is  $\text{cwnd}/\text{RTT}_{\min}$

if measured throughput “very close” to uncongested throughput

*increase cwnd* linearly /\* since path not congested \*/

else if measured throughput “far below” uncongested throughput

*decrease cwnd* linearly /\* since path is congested \*/

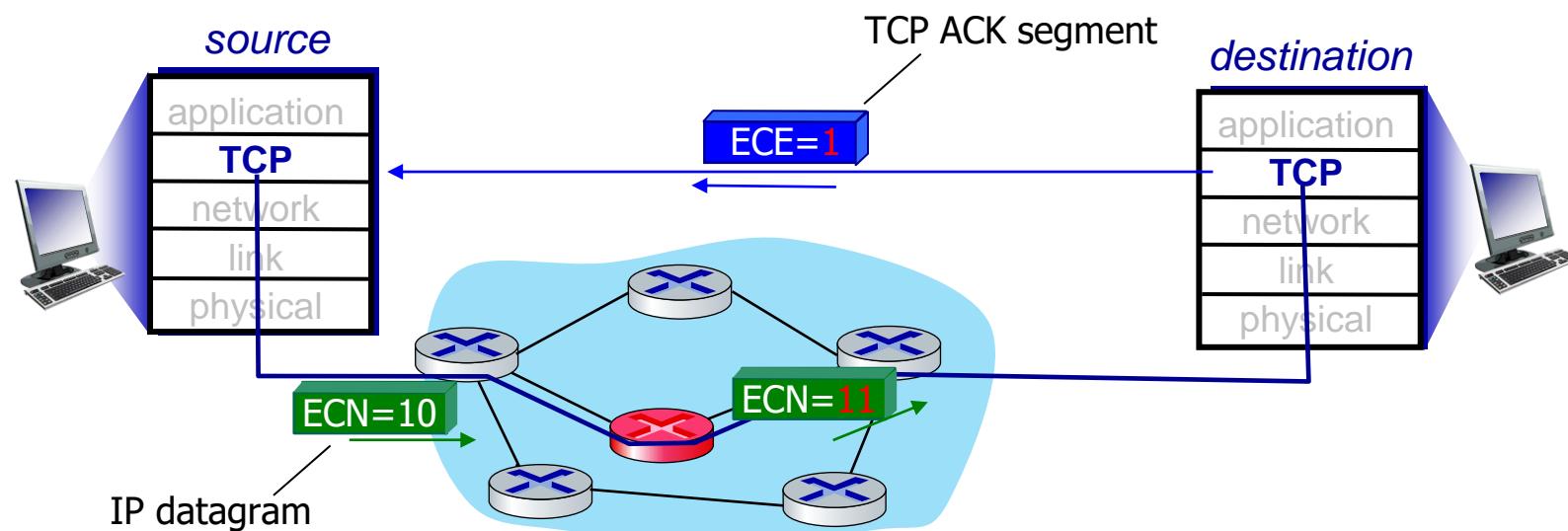
# Delay-based TCP congestion control

- congestion control without inducing/forcing loss
- *maximizing throughput* (“keeping the just pipe full...”) while keeping *delay low* (“...but not fuller”)
- a number of deployed TCPs take a delay-based approach
  - e.g., *Bottleneck Bandwidth and Round-trip propagation time (BBR)* deployed on Google’s (internal) backbone network

# Explicit congestion notification (ECN)

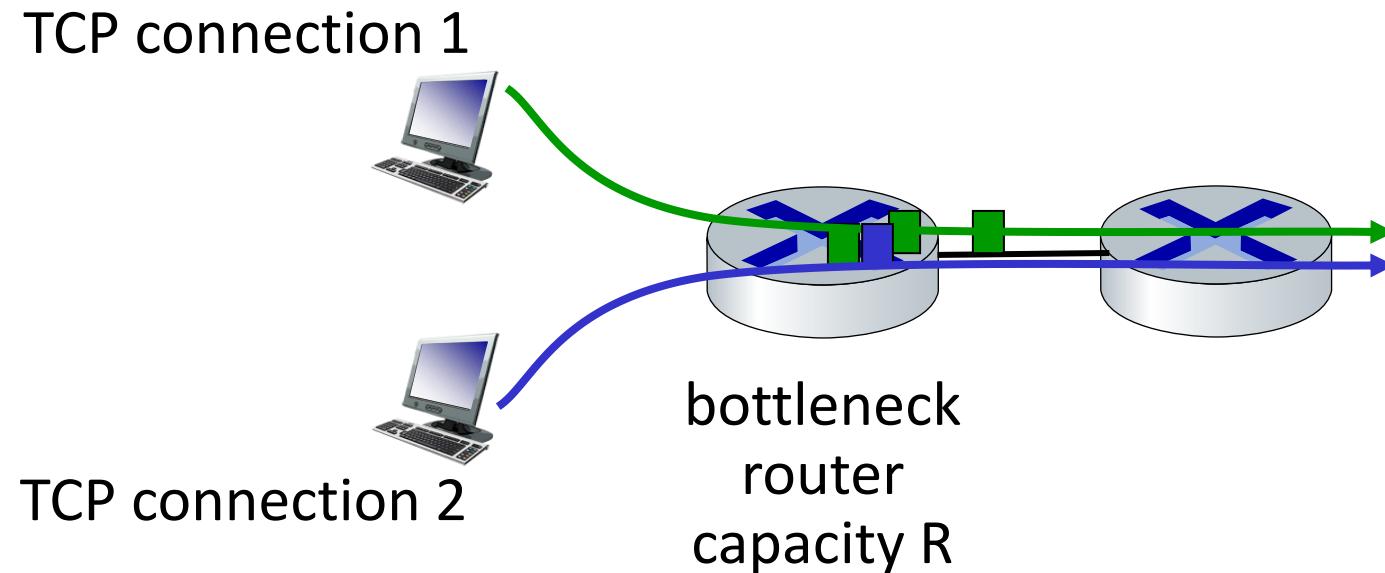
TCP deployments often implement *network-assisted* congestion control:

- 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)



# TCP fairness

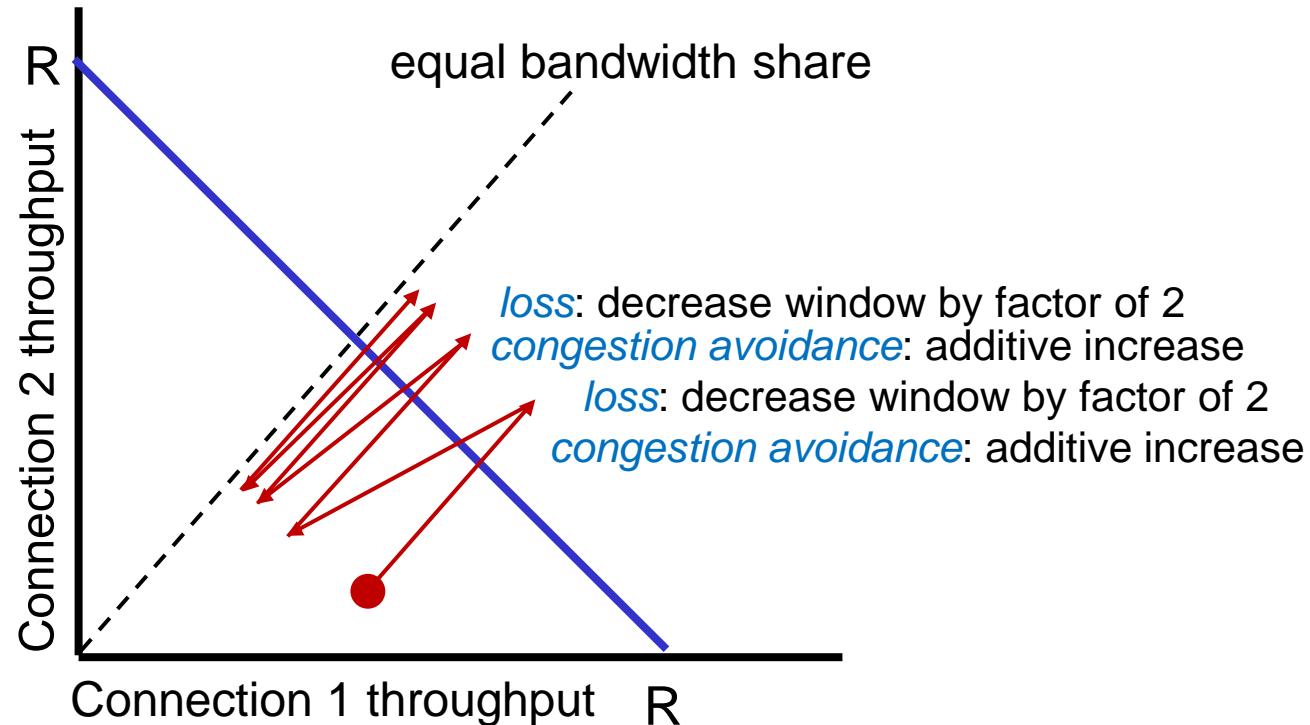
Fairness goal: if  $K$  TCP sessions share same bottleneck link of bandwidth  $R$ , each should have average rate of  $R/K$



# Q: is TCP Fair?

Example: two competing TCP sessions:

- *additive increase* gives slope of 1, as throughput increases
- *multiplicative decrease* decreases throughput proportionally



Is TCP fair?

A: Yes, under idealized assumptions:

- *same RTT*
- *fixed number of sessions*  
only in congestion avoidance

# Fairness: must all network apps be “fair”?

## Fairness and UDP

- *multimedia apps* often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss
- there is no “Internet police” policing use of congestion control

## Fairness, parallel TCP connections

- application can open *multiple parallel connections* between two hosts
- web browsers do this, e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/20

# Transport layer: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality



# Evolving transport-layer functionality

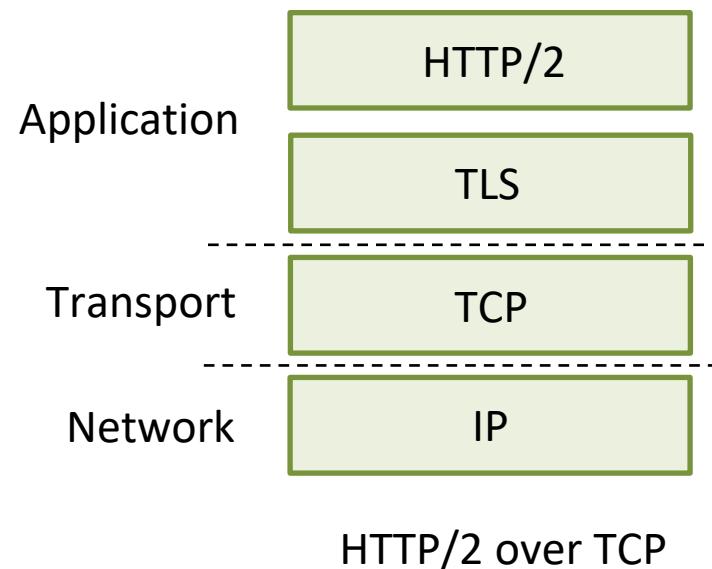
- **TCP, UDP**: principal transport protocols for 40 years
- different “*flavors*” of TCP developed, for specific scenarios:

Scenario	Challenges
Long, fat pipes (large data transfers)	Many packets “in flight”; loss shuts down pipeline
Wireless networks	Loss due to noisy wireless links, mobility; TCP treat this as congestion loss
Long-delay links	Extremely long RTTs
Data center networks	Latency sensitive
Background traffic flows	Low priority, “background” TCP flows

- *moving transport–layer functions to application layer, on top of UDP*
  - HTTP/3, QUIC

# QUIC: Quick UDP Internet Connections

- *application-layer protocol*, on top of **UDP**
  - increase performance of HTTP
  - deployed on many Google servers, apps (**Chrome**, **mobile YouTube** app)

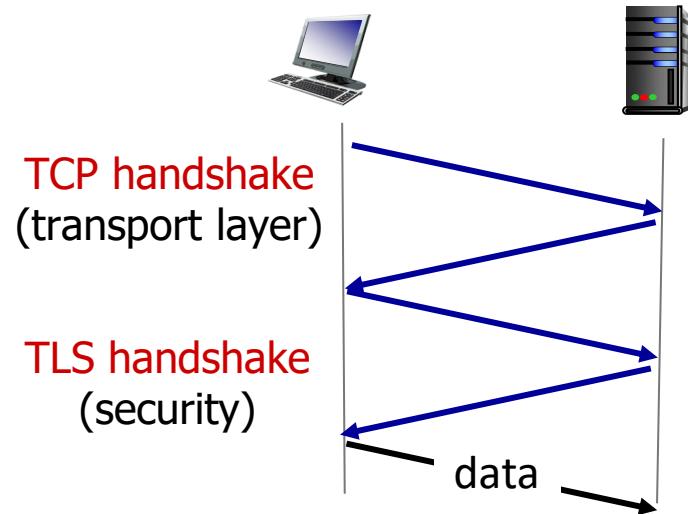


# QUIC: Quick UDP Internet Connections

adopts approaches we've studied in this chapter for  
*connection establishment, error control, congestion control*

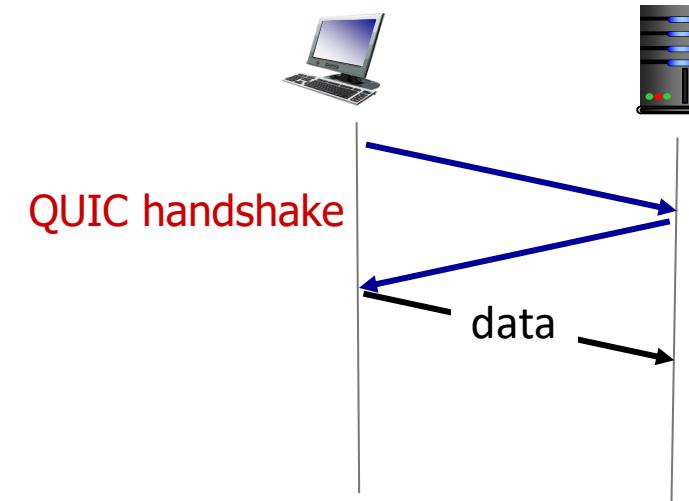
- **error and congestion control:** “Readers familiar with TCP’s loss detection and congestion control will find algorithms here that *parallel well-known TCP ones.*” [from QUIC specification]
- **connection establishment:** reliability, congestion control, authentication, encryption, state established *in one RTT*
- *multiple application-level “streams” multiplexed* over single QUIC connection
  - separate reliable data transfer, security
  - common congestion control

# QUIC: Connection establishment



TCP (reliability, congestion control state)  
+ TLS (authentication, crypto state)

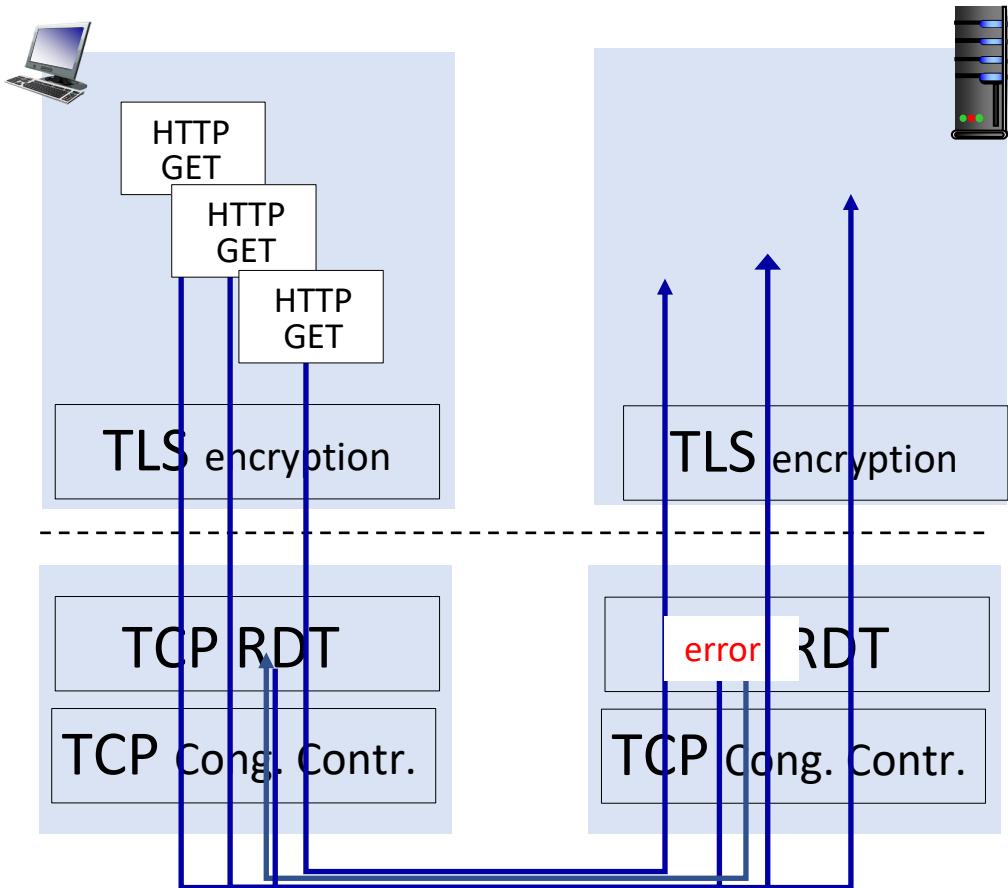
- *2 serial handshakes*



QUIC: reliability, congestion control,  
authentication, crypto state

- *1 handshake*

# QUIC: streams: parallelism, no HOL blocking



(a) HTTP 1.1

# Chapter 3: summary

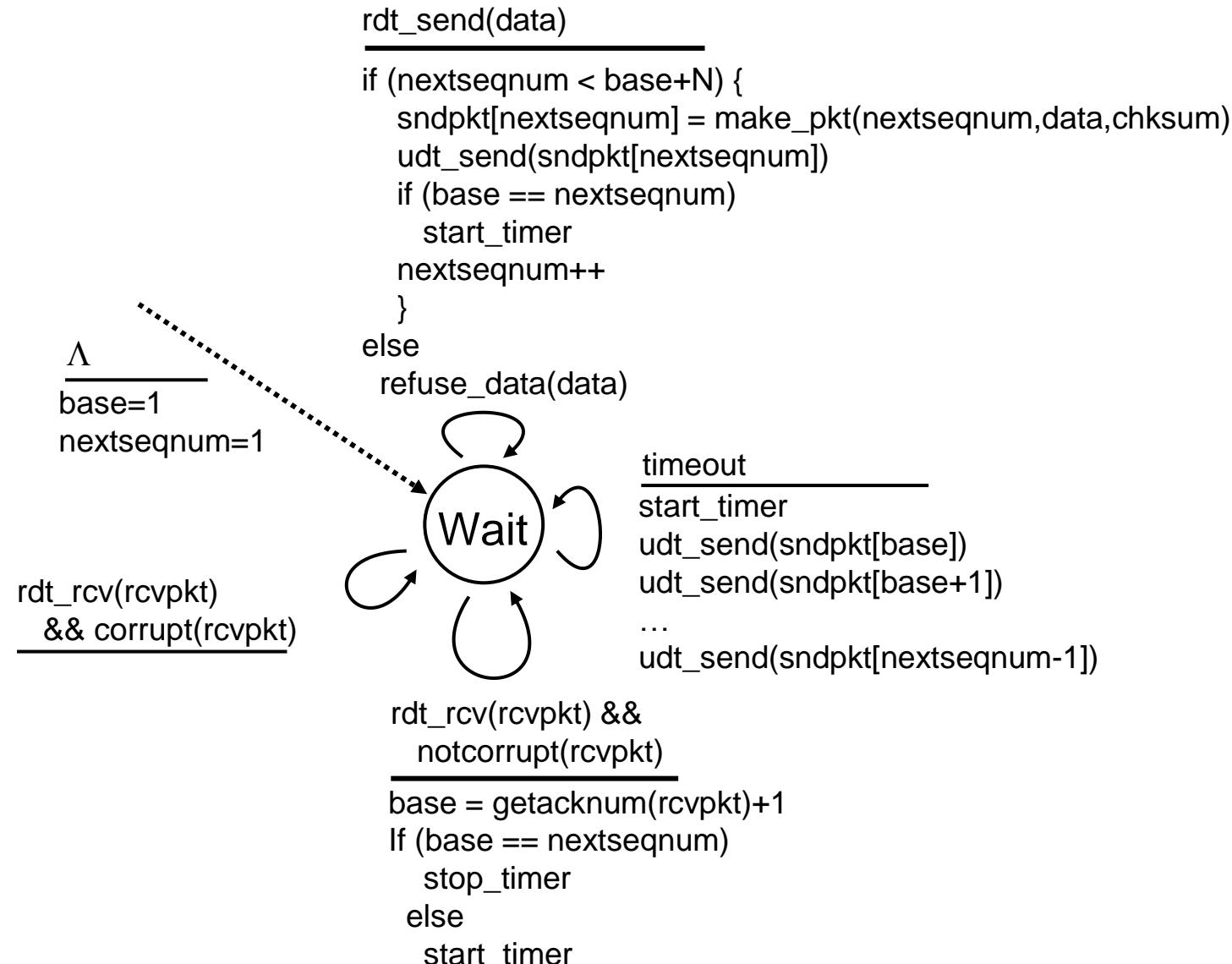
- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation on the Internet
  - UDP
  - TCP

Up next:

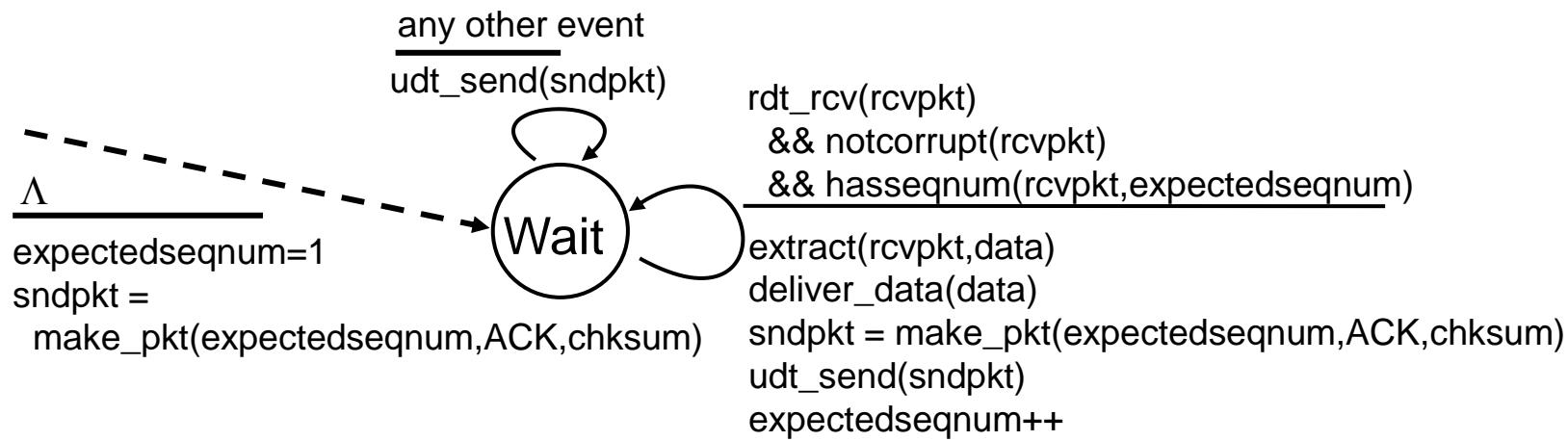
- *leaving the network*  
“*edge*” (application, transport layers)
- into the network “*core*”
- two network-layer chapters:
  - *data plane*
  - *control plane*

# Additional Chapter 3 slides

# Go-Back-N: sender extended FSM



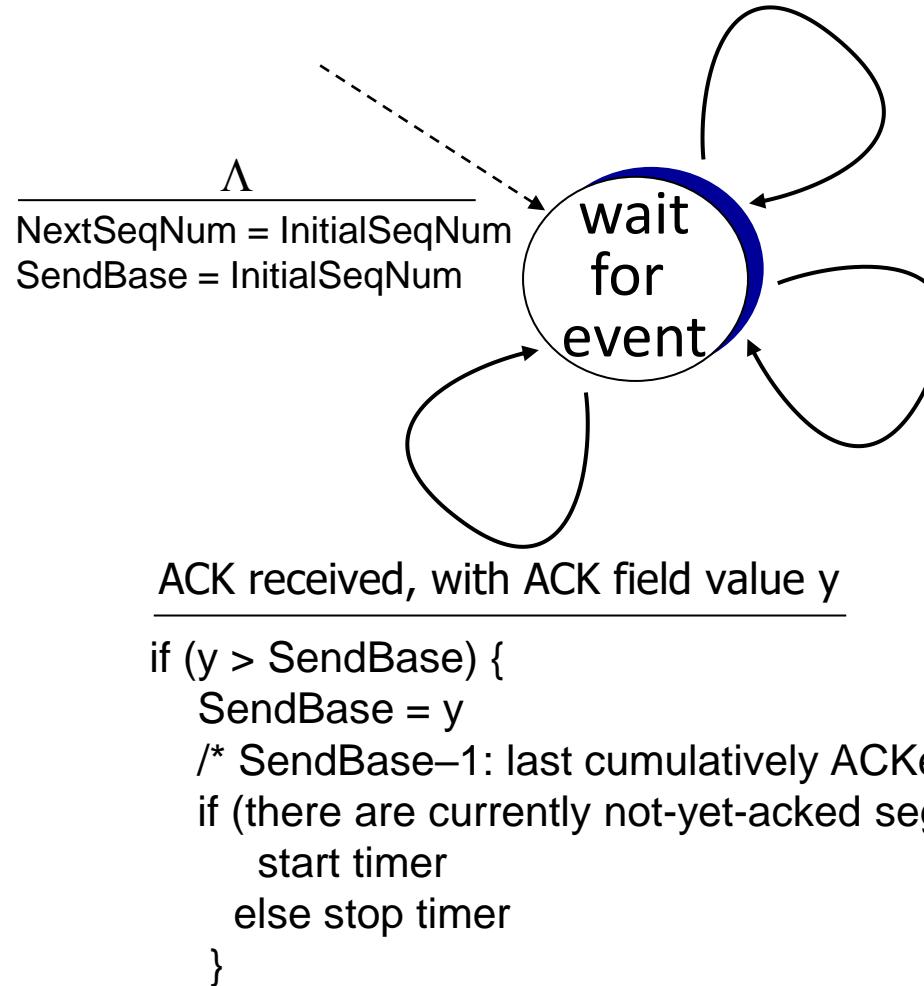
# Go-Back-N: receiver extended FSM



ACK-only: always send ACK for correctly-received packet with highest *in-order* seq #

- may generate duplicate ACKs
  - need only remember **expectedseqnum**
- out-of-order packet:
- discard (don't buffer): *no receiver buffering!*
  - re-ACK pkt with highest in-order seq #

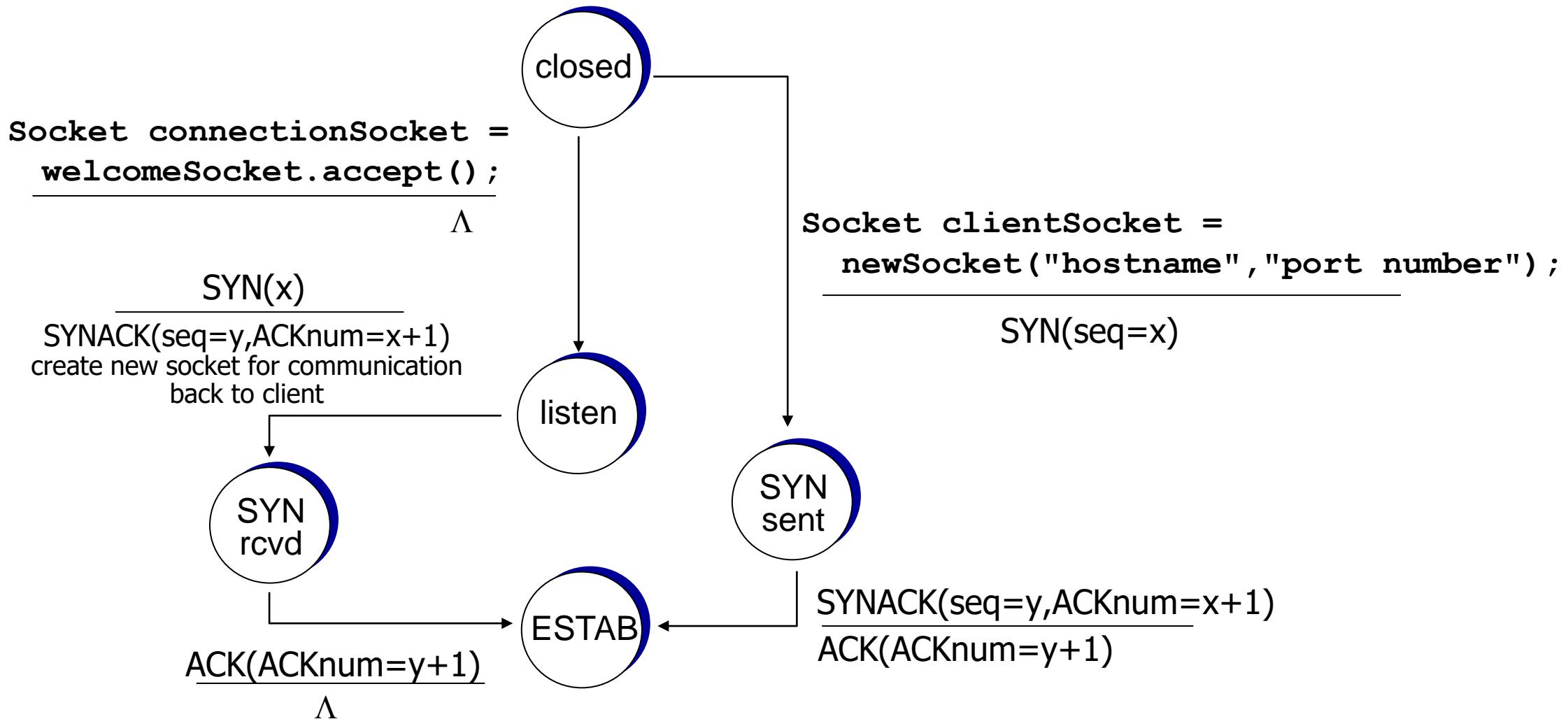
# TCP sender (simplified)



data received from application above  
create segment, seq. #: NextSeqNum  
pass segment to IP (i.e., “send”)  
 $\text{NextSeqNum} = \text{NextSeqNum} + \text{length(data)}$   
if (timer currently not running)  
start timer

timeout  
retransmit not-yet-acked segment  
with smallest seq. #  
start timer

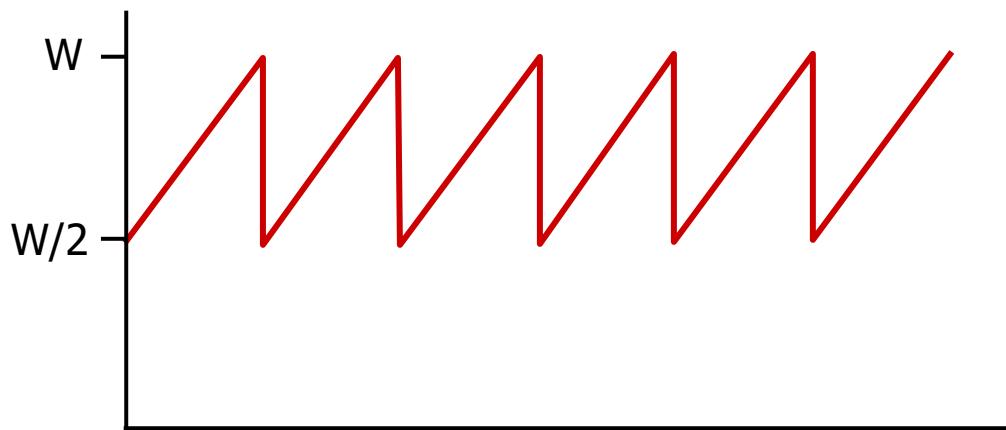
# TCP 3-way handshake FSM



# TCP throughput

- avg. TCP throughput as function of window size, RTT?
  - ignore slow start, assume there is always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is  $\frac{3}{4} W$
  - avg. throughput is  $\frac{3}{4}W$  per RTT

$$\text{avg TCP throughput} = \frac{3}{4} \frac{W}{\text{RTT}} \text{ bytes/sec}$$



# TCP over “long, fat pipes”

- example: 1500-byte segments, 100ms RTT, want 10 Gbps throughput
- requires  $W = 83,333$  in-flight segments
- throughput in terms of segment loss probability,  $L$  [Mathis 1997]:

$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT} \sqrt{L}}$$

→ to achieve 10 Gbps throughput, need a loss rate of  $L = 2 \cdot 10^{-10}$  – *a very small loss rate!*

- versions of TCP for long, high-speed scenarios