Chapter 3
Transport Layer

A note on the use of these PowerPoint slides:
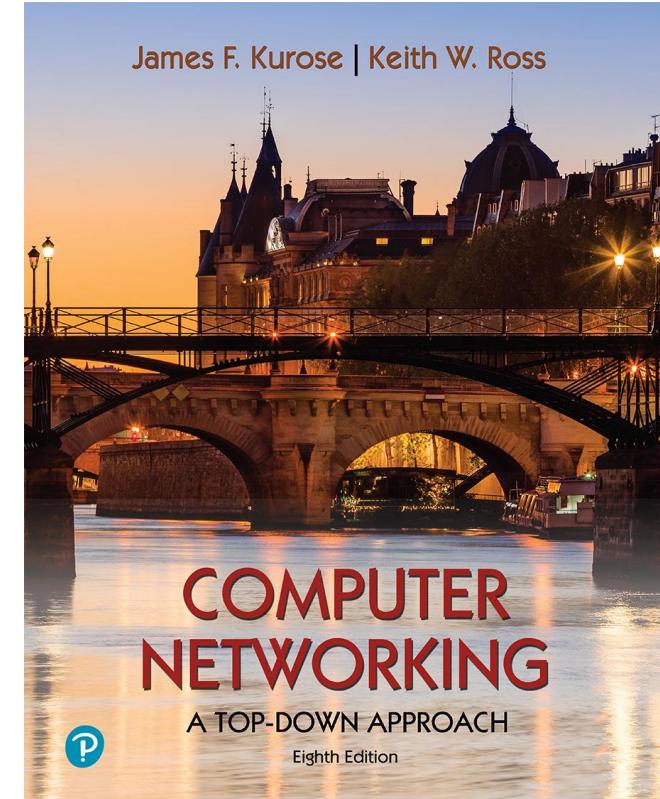
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

For a revision history, see the slide note for this page.

Thanks and enjoy! JFK/KWR

James F. Kurose | Keith W. Ross

COMPUTER NETWORKING
A TOP-DOWN APPROACH
Eighth Edition

P

Computer Networking: A Top-Down Approach

8th 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
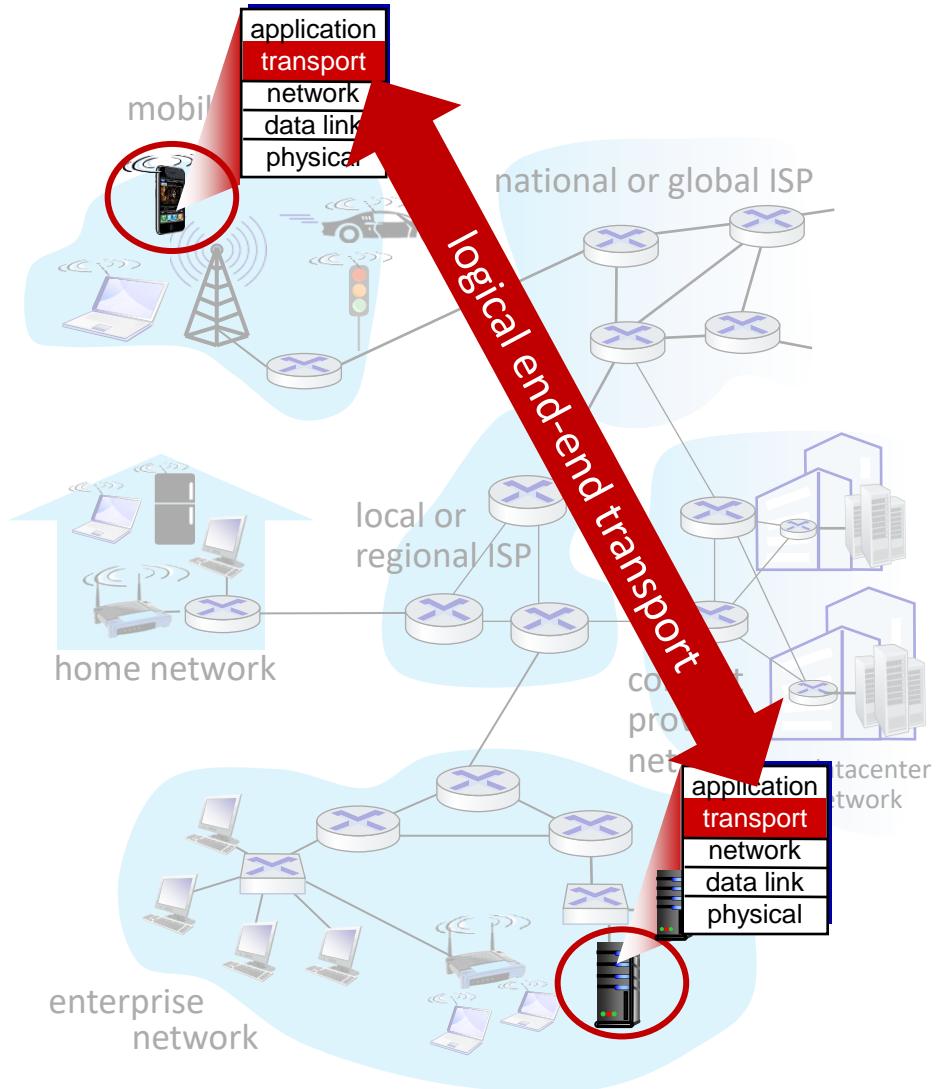
Transport layer: roadmap

- 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

© 1822. 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 vs. network layer services and protocols

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

- **network layer:**
 communication between
  hosts

Transport Layer Actions

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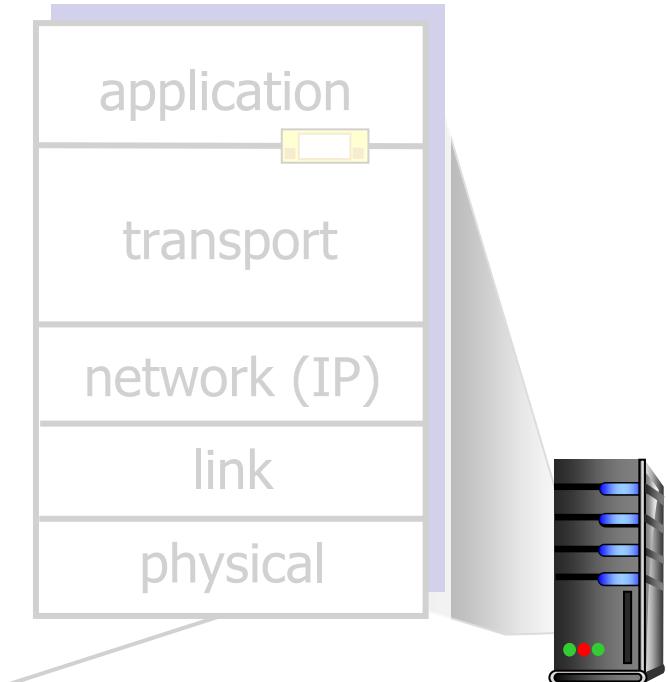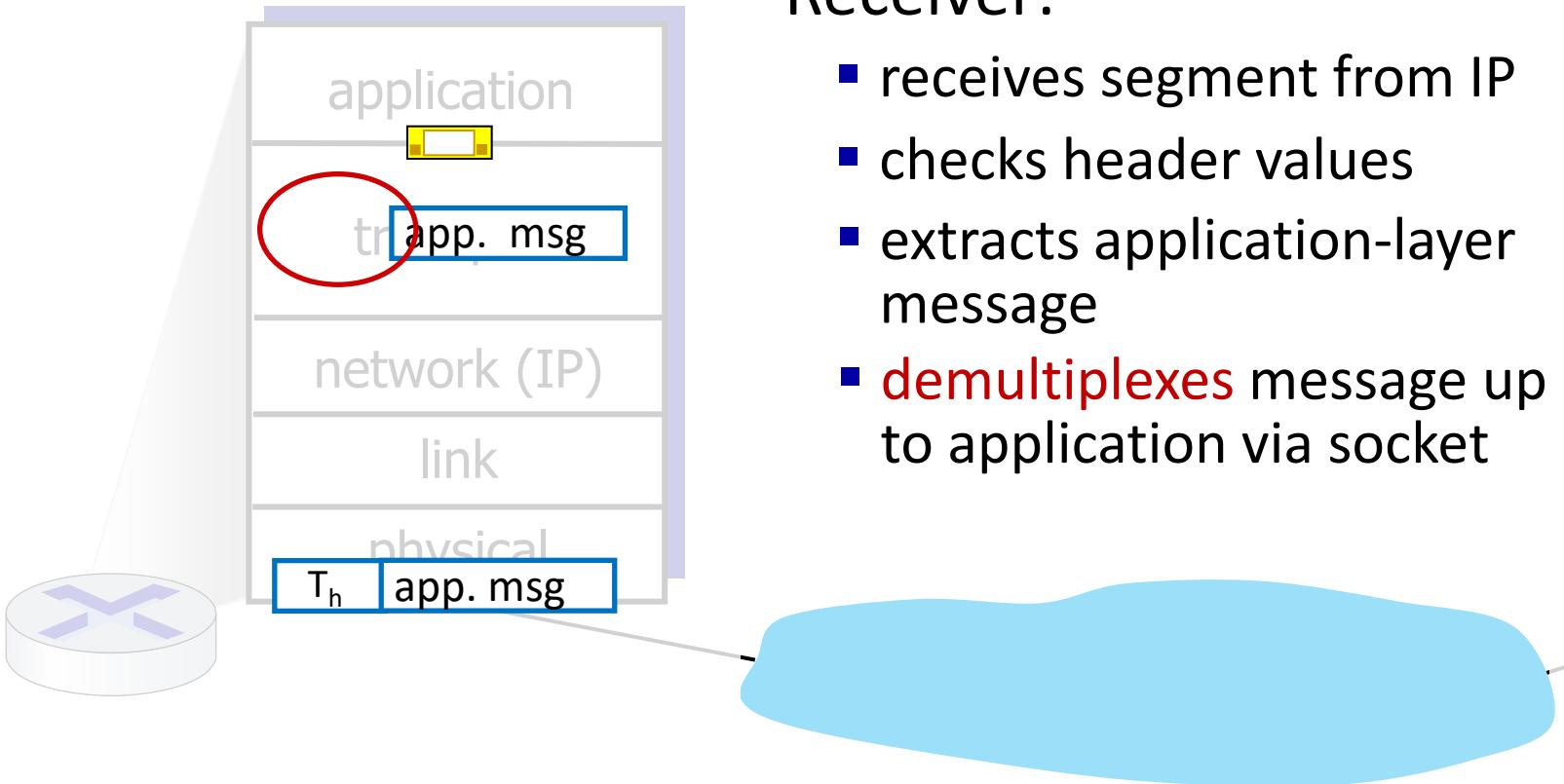
- is passed an application-layer message
- determines segment header fields values
- creates segment
- passes segment to IP

application

transport

network (IP)

link

physical

app. msg

T_h app. msg

network (IP)

link

physical

Transport Layer Actions
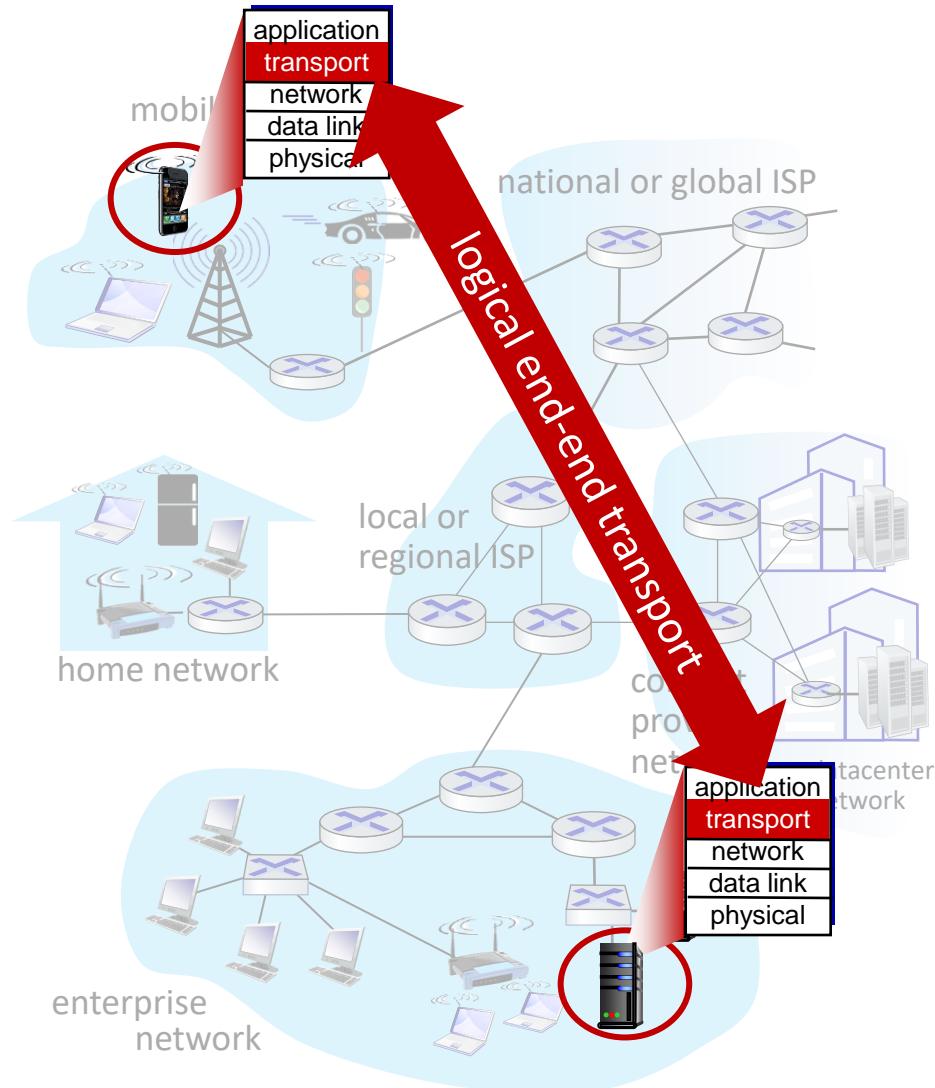
Receiver:

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

app. msg

T_h app. msg

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
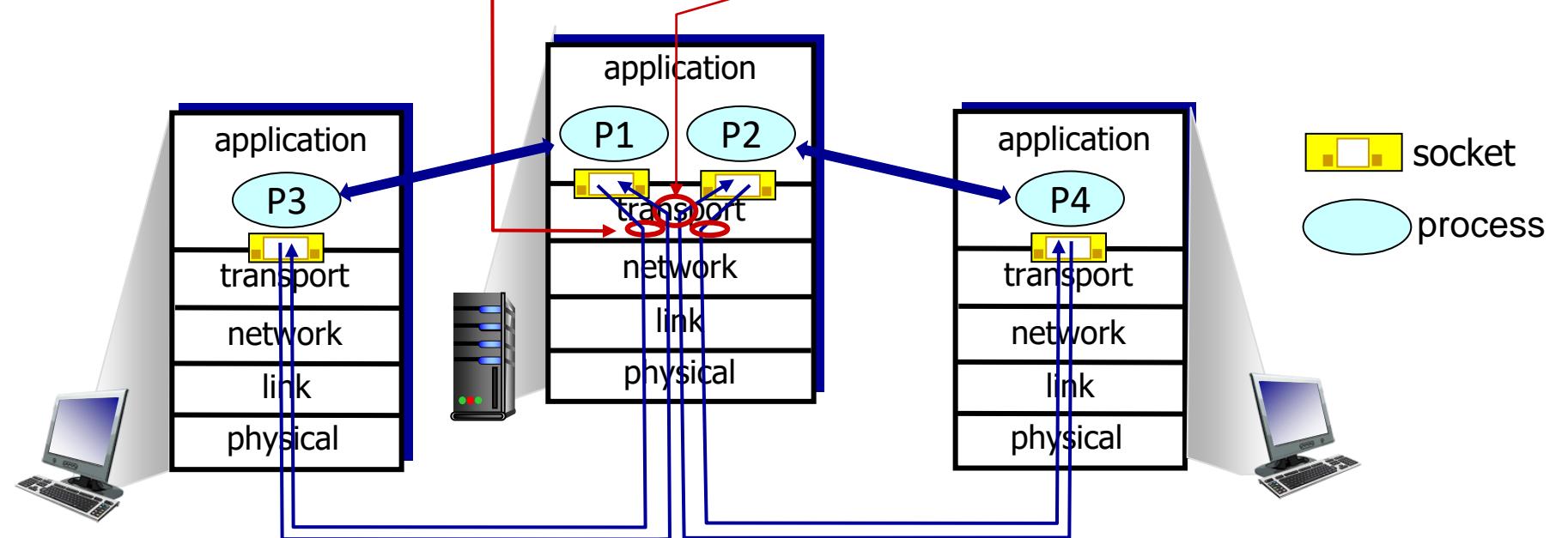
Multiplexing/demultiplexing
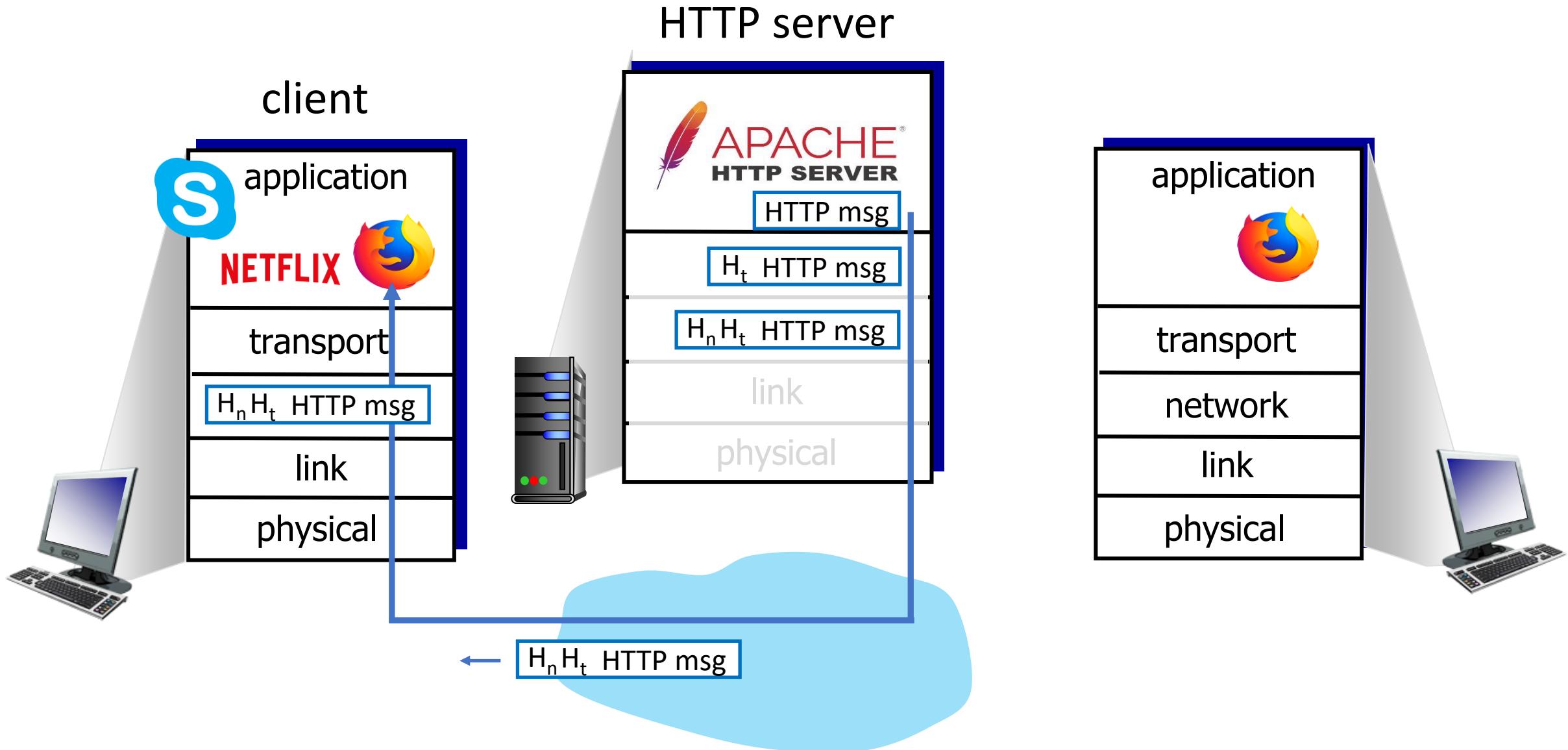
multiplexing as sender:

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

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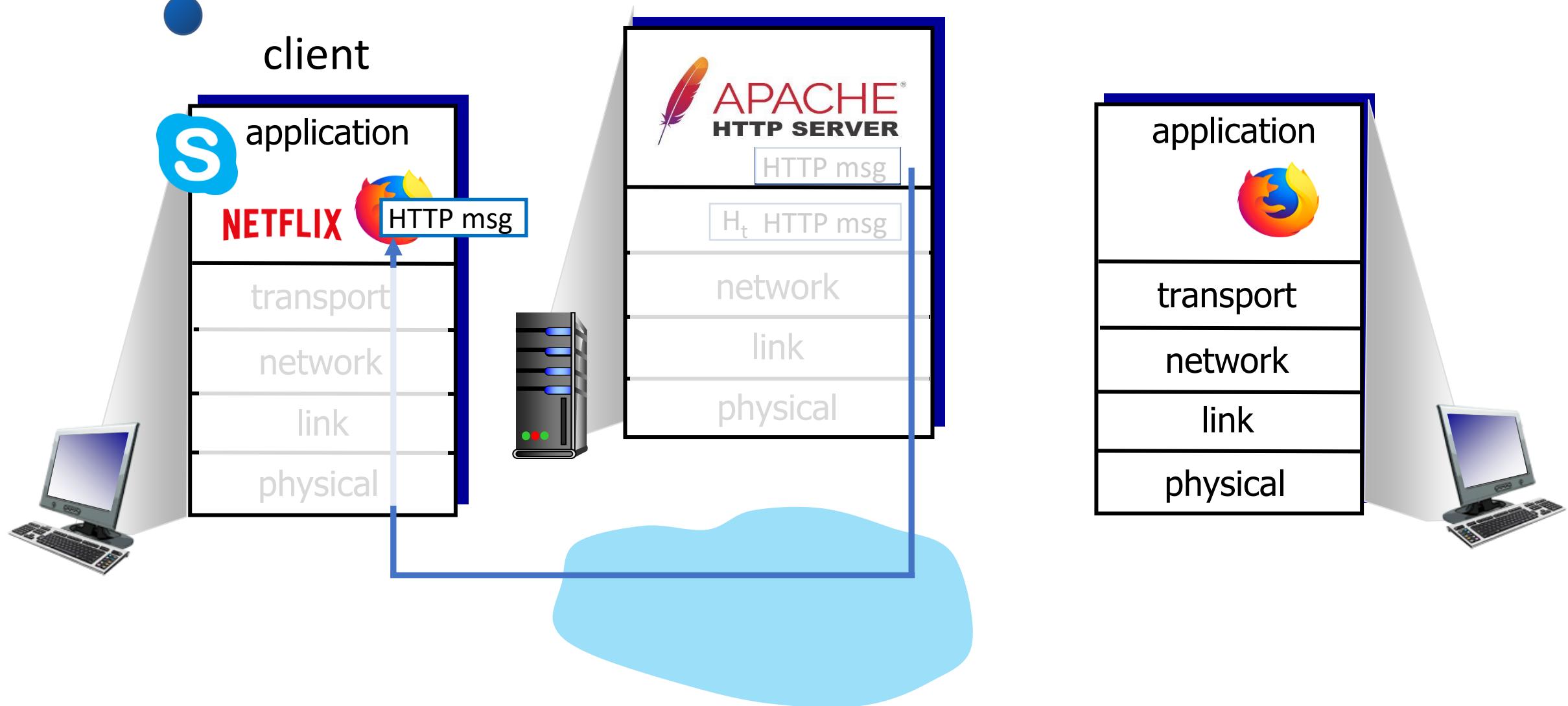
use header info to deliver received segments to correct socket

client

HTTP server

application

transport

$H_n H_t$ HTTP msg

link

physical

HTTP msg

H_t HTTP msg

$H_n H_t$ HTTP msg

link

physical

application

transport

network

link

physical

$H_n H_t$ HTTP msg
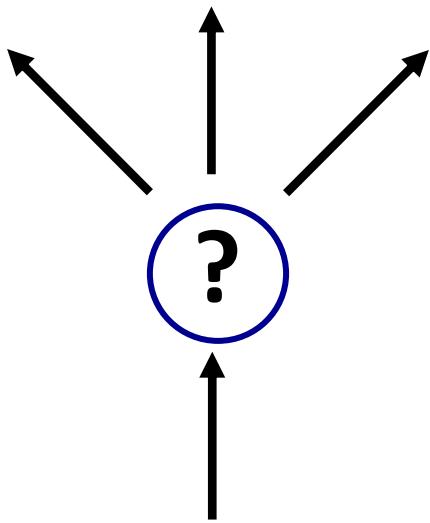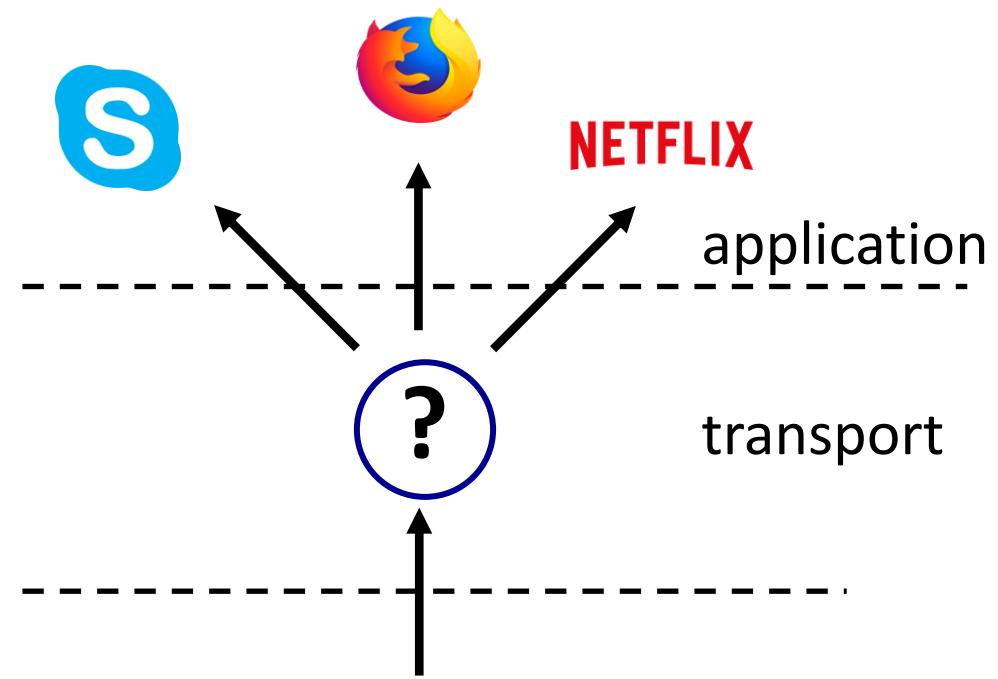
Q: how did transport layer know to deliver message to Firefox browser process rather then Netflix process or Skype process?

client

application

NETFLIX

HTTP msg

transport

network

link

physical

APACHE® HTTP SERVER

HTTP msg

H_t HTTP msg

network

link

physical

application

transport

network

link

physical

de-multiplexing

application

transport

de-multiplexing

Demultiplexing

multiplexing

application
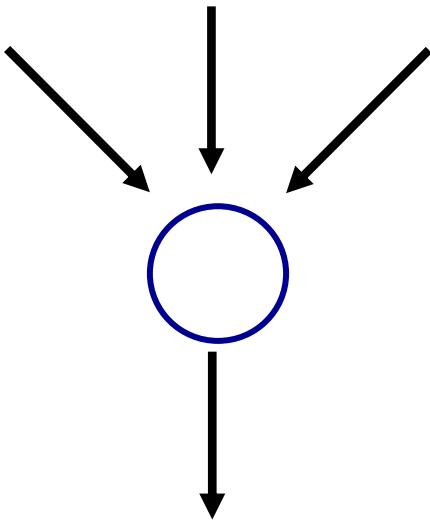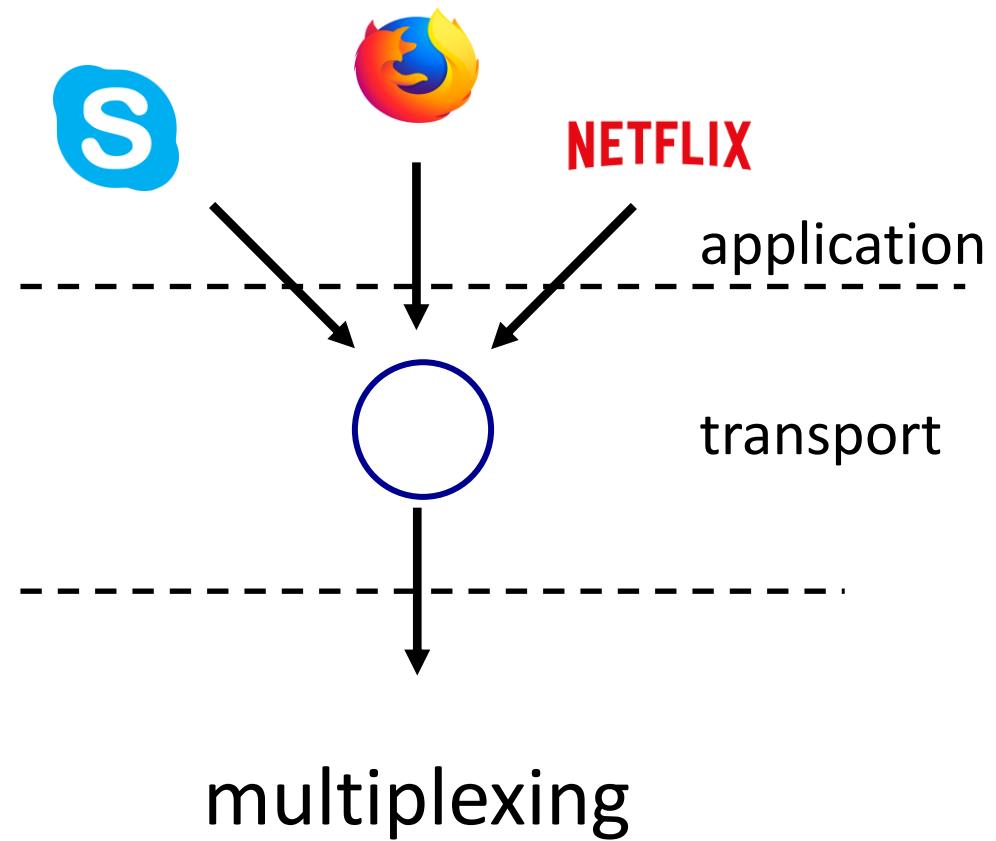
transport
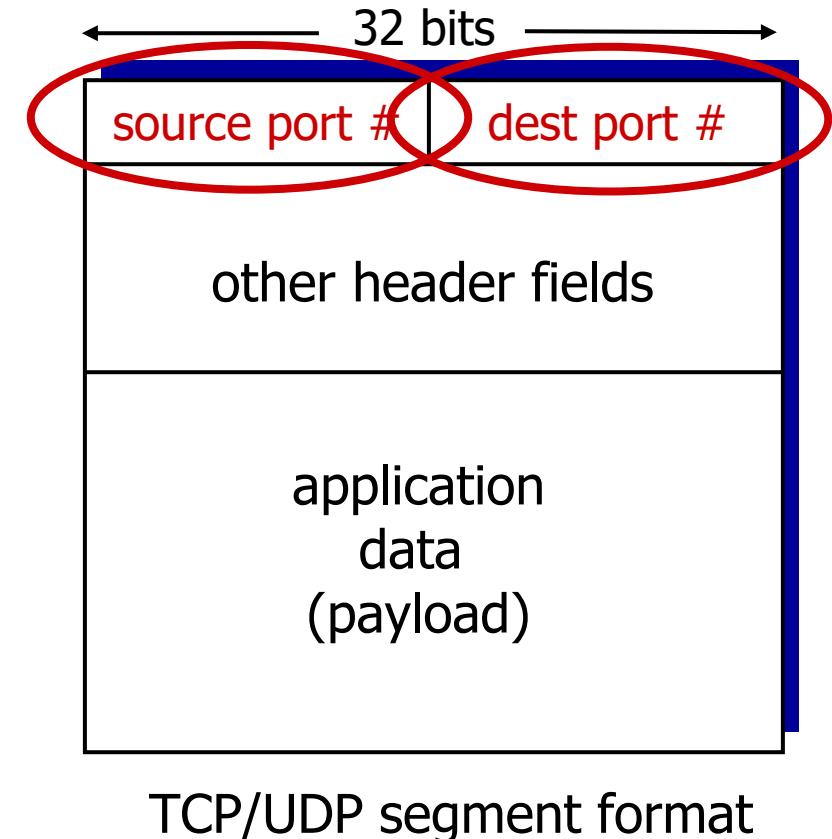
multiplexing

Multiplexing

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

32 bits

source port #	dest port #
other header fields	
application data (payload)	

TCP/UDP segment format

Connectionless demultiplexing

Recall:

- when creating socket, must specify *host-local* port #:

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

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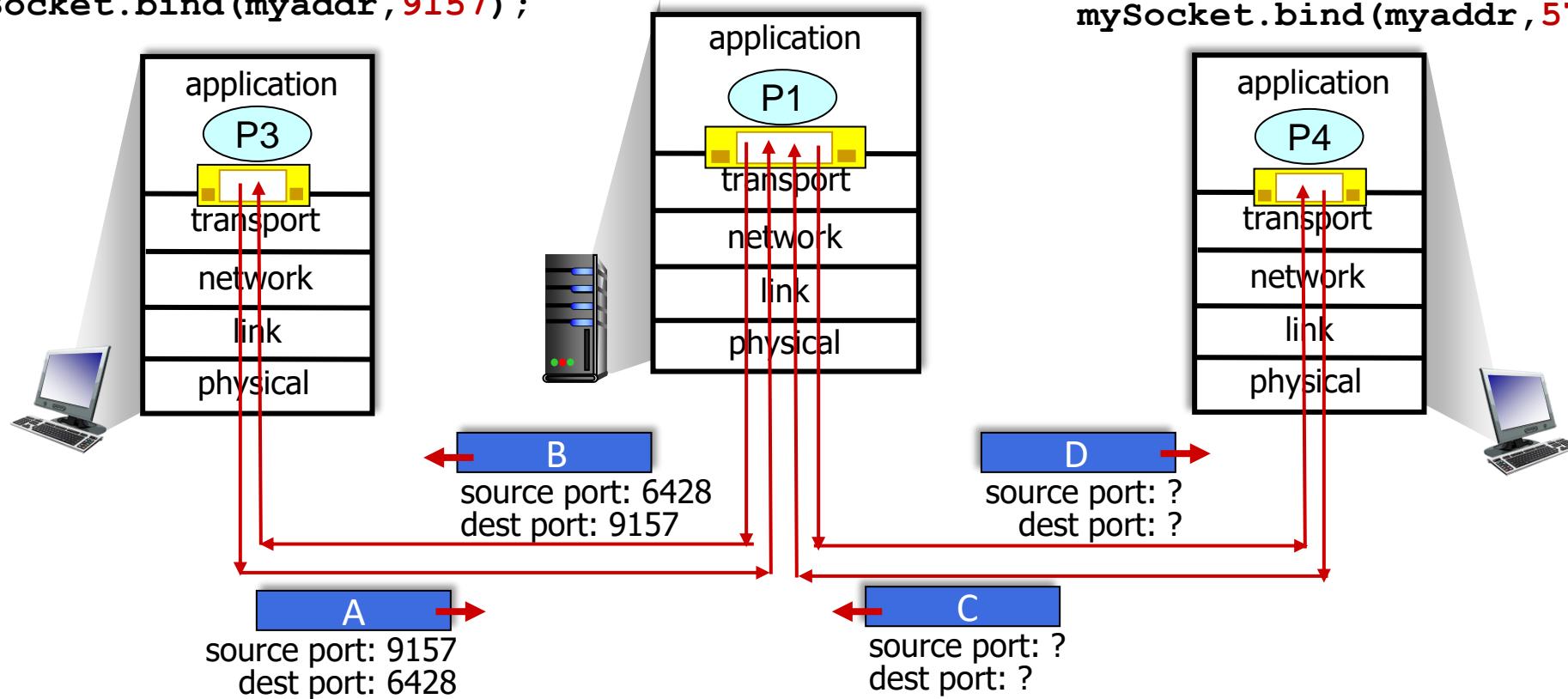
IP/UDP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

Connectionless demultiplexing: an example

```
mySocket =
  socket(AF_INET,SOCK_DGRAM)
mySocket.bind(myaddr,6428);
```

```
mySocket =
  socket(AF_INET,SOCK_STREAM)
mySocket.bind(myaddr,9157);
```
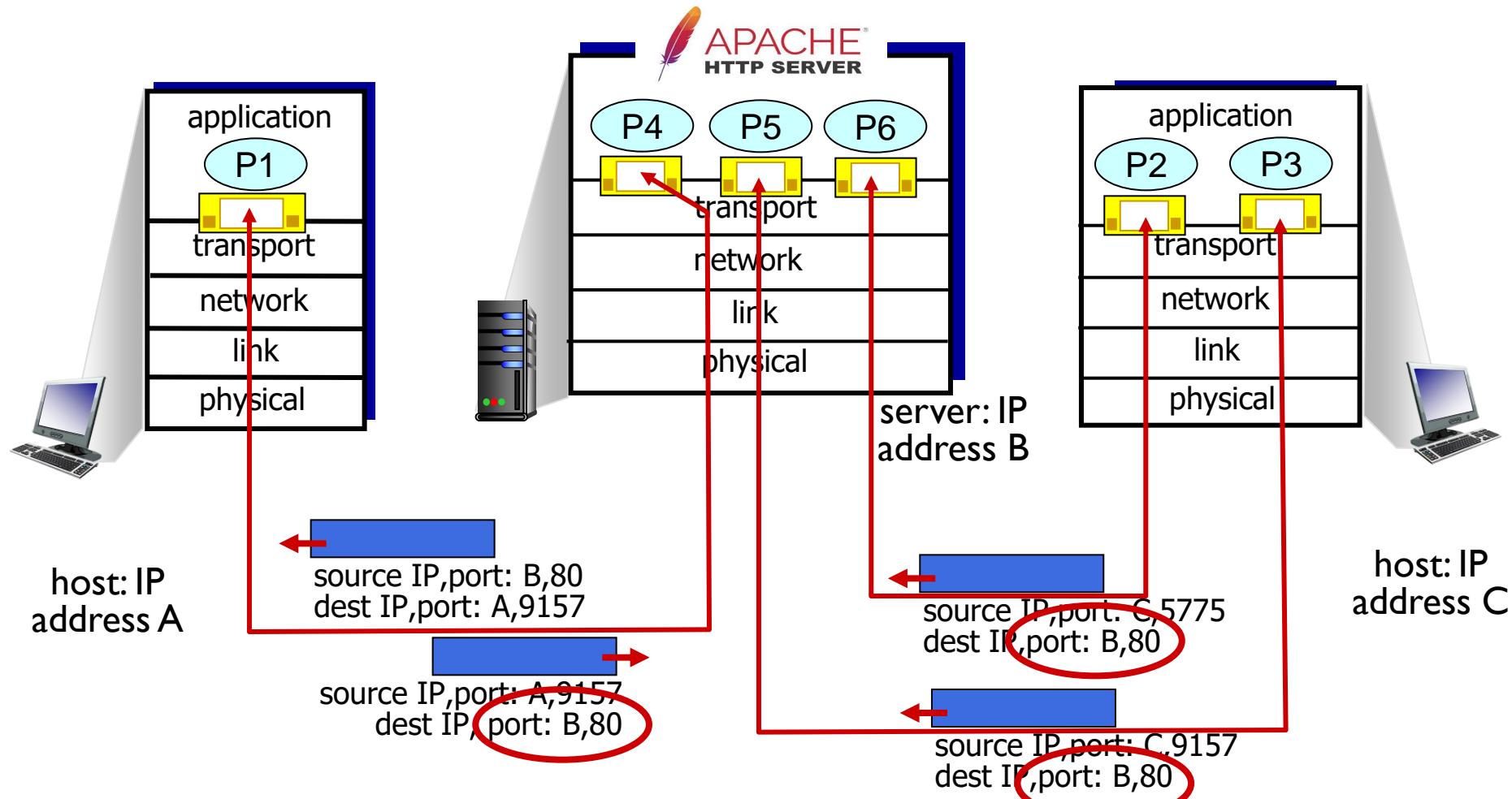
```
mySocket =
  socket(AF_INET,SOCK_STREAM)
mySocket.bind(myaddr,5775);
```


B
source port: 6428
dest port: 9157

D
source port: ?
dest port: ?

A
source port: 9157
dest port: 6428

C
source port: ?
dest port: ?

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

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

- 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
------


 0      7 8     15 16    23 24    31
+--------+--------+--------+--------+
|     Source      |   Destination   |
|      Port       |      Port       |
+--------+--------+--------+--------+
|                 |                 |
|     Length      |    Checksum     |
+--------+--------+--------+--------+
|
|          data octets ...
+--------------- ...
```
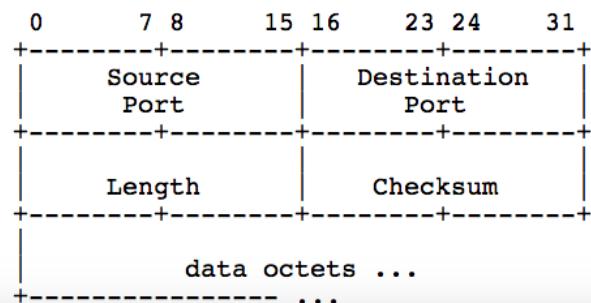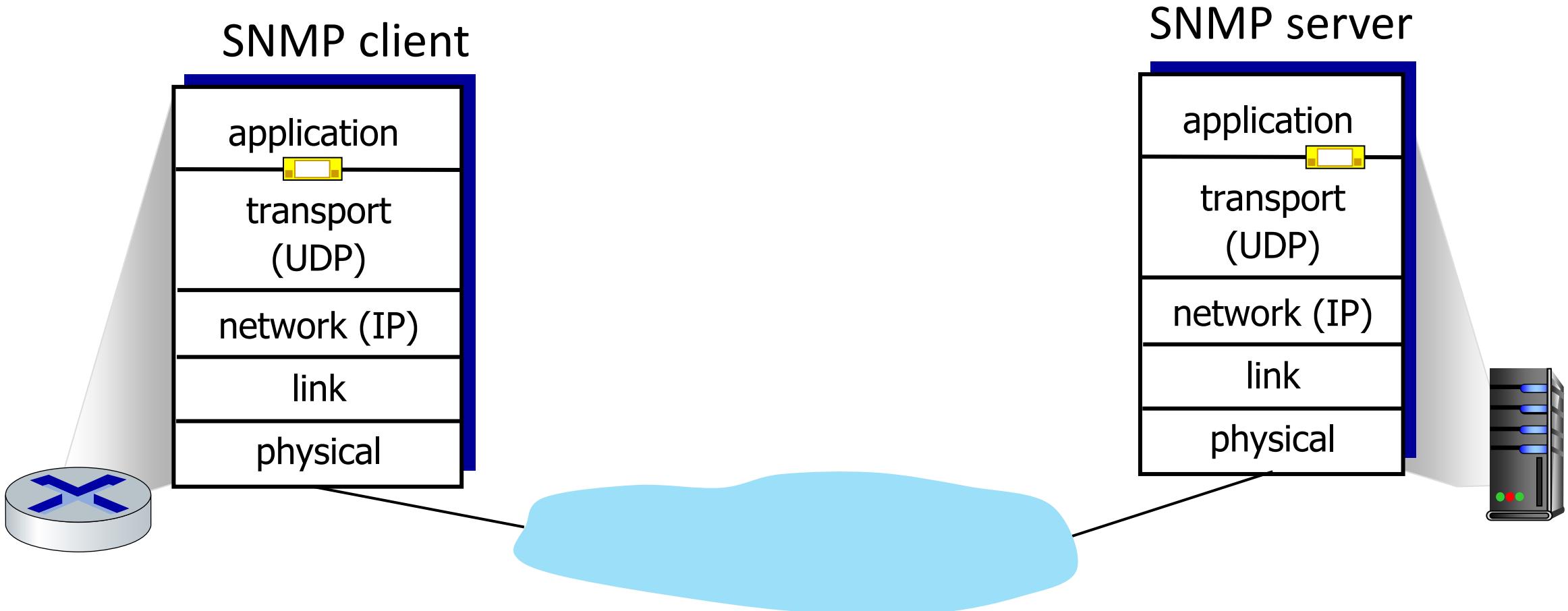
UDP: Transport Layer Actions

SNMP client

SNMP server

application

transport (UDP)

network (IP)

link

physical

application

transport (UDP)

network (IP)

link

physical

UDP: Transport Layer Actions

SNMP client

application

transport (UDP)

network (IP)

link

physical

SNMP server

application

transport (UDP)

network (IP)

link

physical
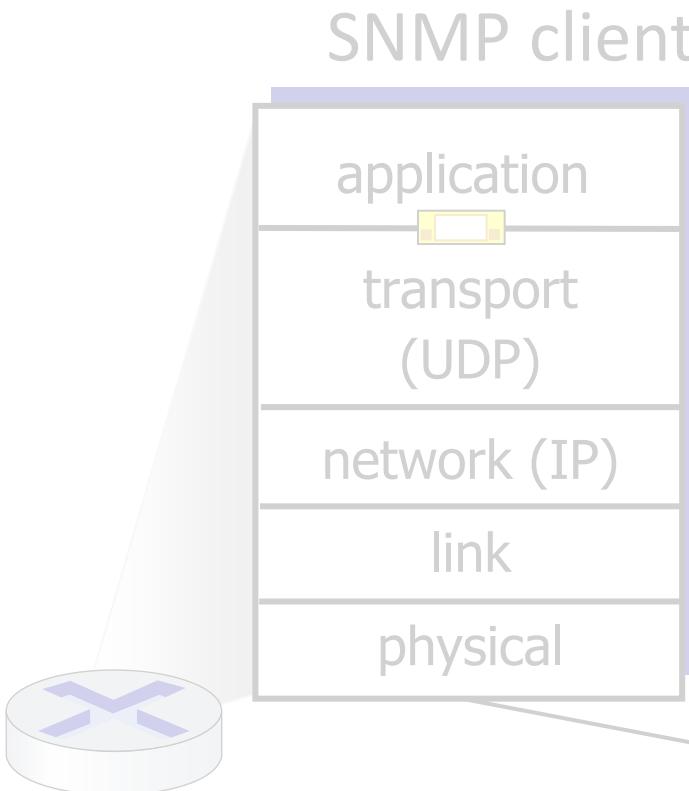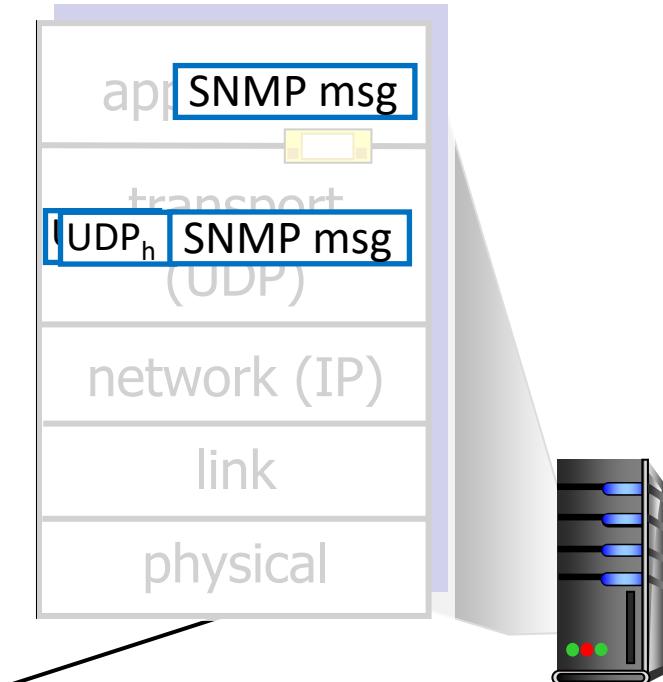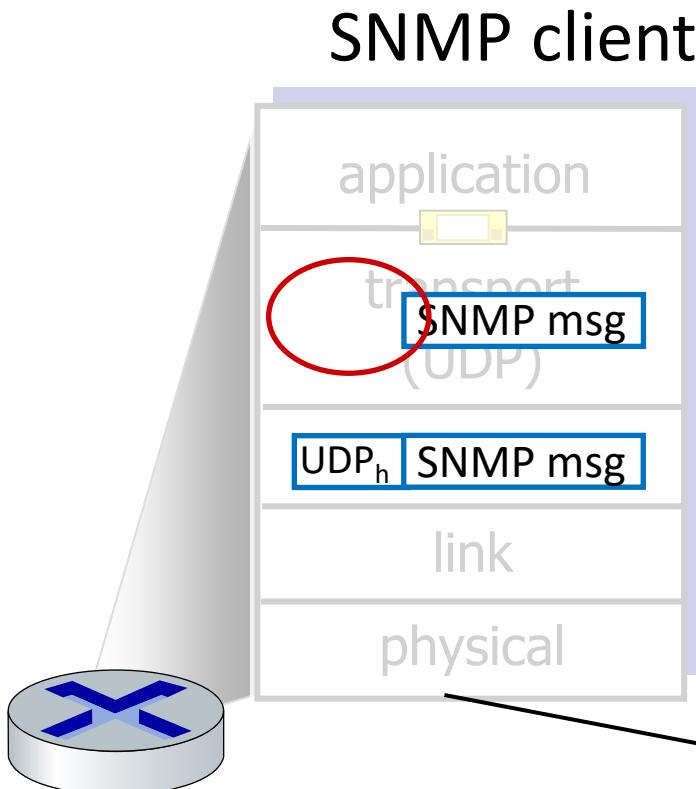
UDP sender actions:

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

SNMP msg

UDP$_h$ | SNMP msg

UDP: Transport Layer Actions

SNMP client

application

transport (UDP)

SNMP msg

UDP_h SNMP msg

link

physical

SNMP server

application

transport (UDP)

network (IP)

link

physical

UDP receiver actions:

- receives segment from IP
- checks UDP checksum header value
- extracts application-layer message
- demultiplexes message up to application via socket
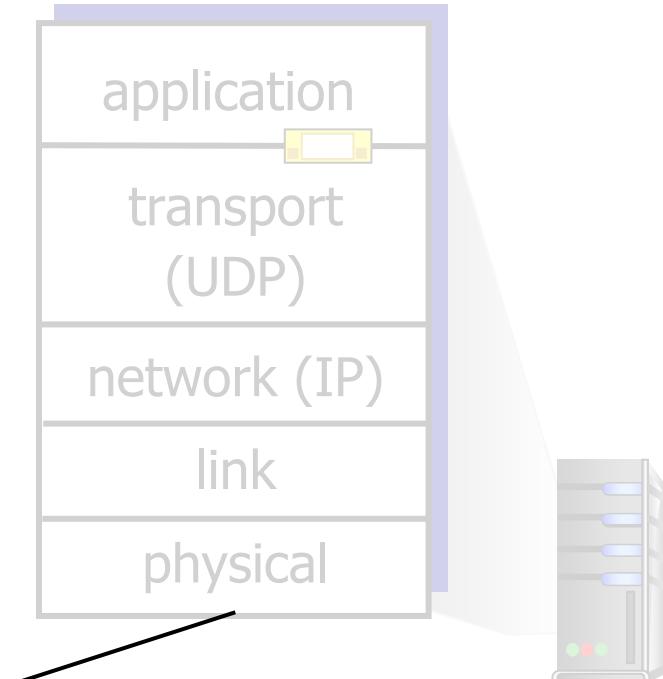
Figure 23.2 *Port numbers*
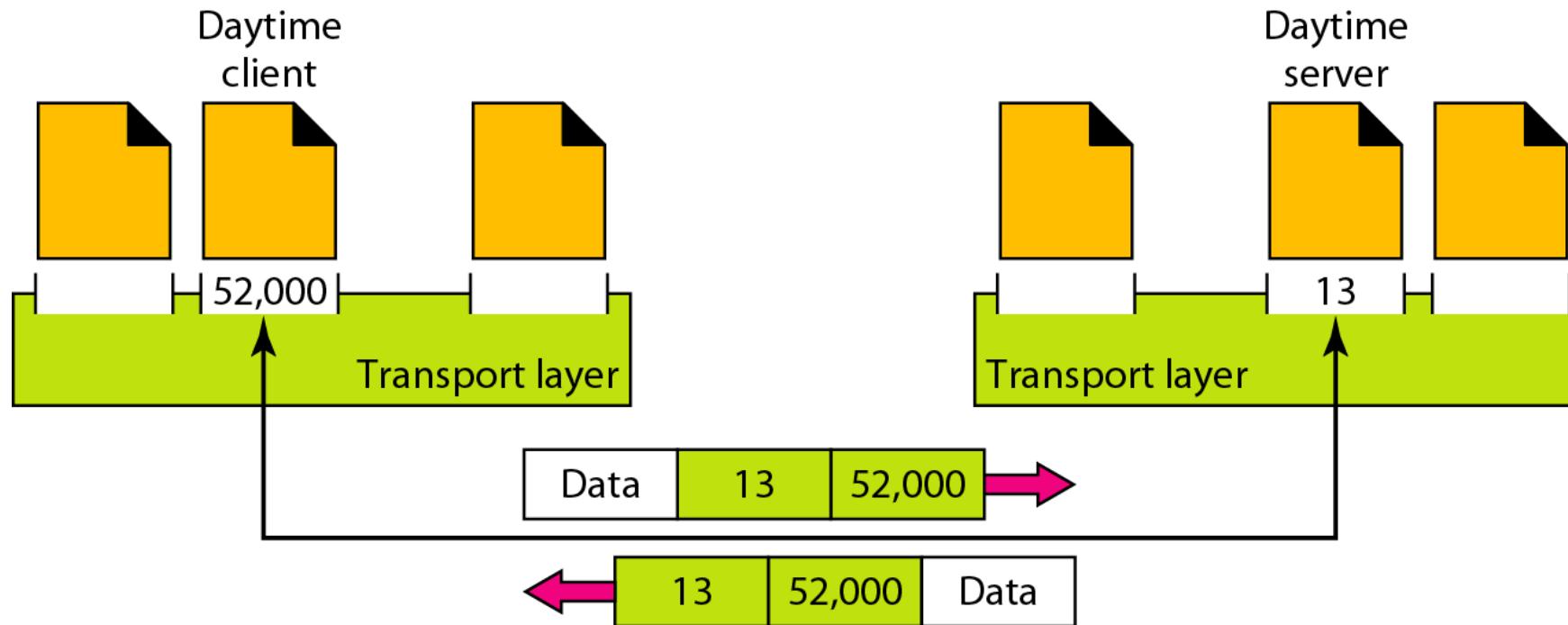
Figure 23.3 *IP addresses versus port numbers*
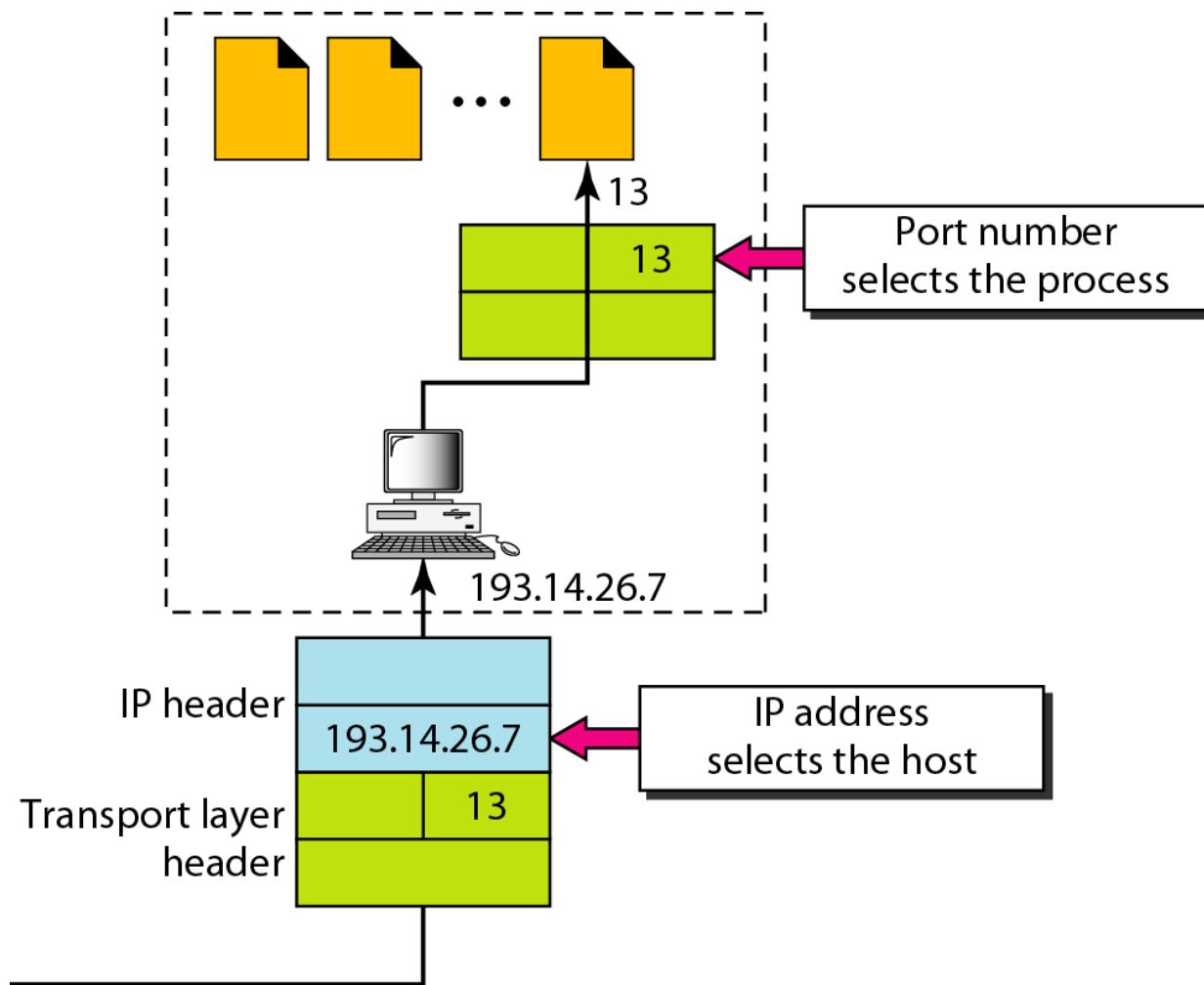
Figure 23.4 *IANA ranges*
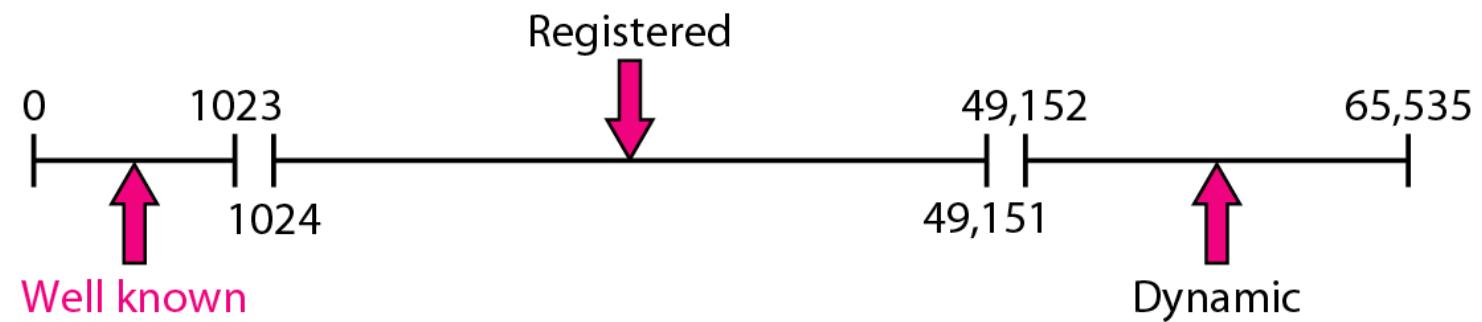
Figure 23.5 *Socket address*

UDP segment header
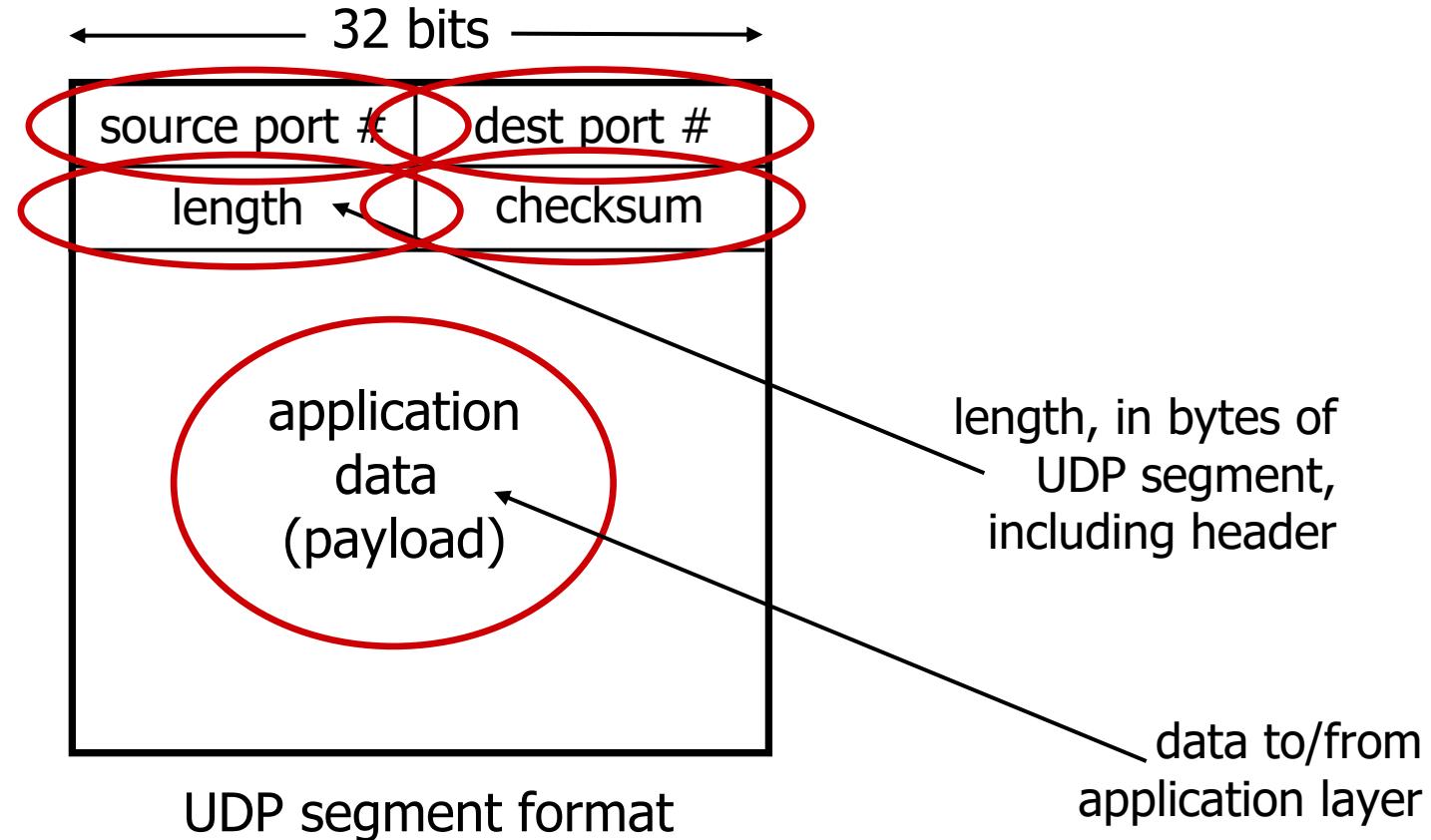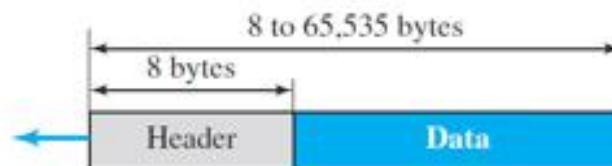
UDP segment format

Figure 24.2   *User datagram packet format*

Example 24.1

The following is the content of a UDP header in hexadecimal format.

CB84000D001C001C

a. What is the source port number?

b. What is the destination port number?

c. What is the total length of the user datagram?

d. What is the length of the data?

e. Is the packet directed from a client to a server or vice versa?

f. What is the client process?

8 to 65,535 bytes

8 bytes

Header   Data

a. UDP user datagram

0	16	31
Source port number		Destination port number
Total length		Checksum

b. Header format

Figure 24.2   *User datagram packet format*
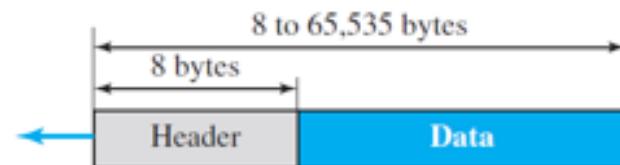
Example 24.1

The following is the content of a UDP header in hexadecimal format.

CB84000D001C001C

a. What is the source port number?

b. What is the destination port number?

c. What is the total length of the user datagram?

d. What is the length of the data?

e. Is the packet directed from a client to a server or vice versa?

f. What is the client process?

8 to 65,535 bytes

8 bytes

Header	Data

a. UDP user datagram

0	16	31
Source port number		Destination port number
Total length		Checksum

b. Header format
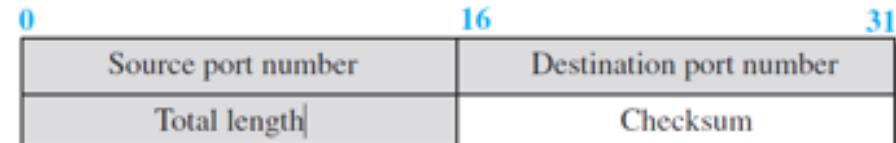
Solution

a. The source port number is the first four hexadecimal digits $(CB84)_{16}$, which means that the source port number is 52100.

b. The destination port number is the second four hexadecimal digits $(000D)_{16}$, which means that the destination port number is 13.

c. The third four hexadecimal digits $(001C)_{16}$ define the length of the whole UDP packet as 28 bytes.

d. The length of the data is the length of the whole packet minus the length of the header, or $28 - 8 = 20$ bytes.

e. Since the destination port number is 13 (well-known port), the packet is from the client to the server.

f. The client process is the Daytime (see Table 24.1).
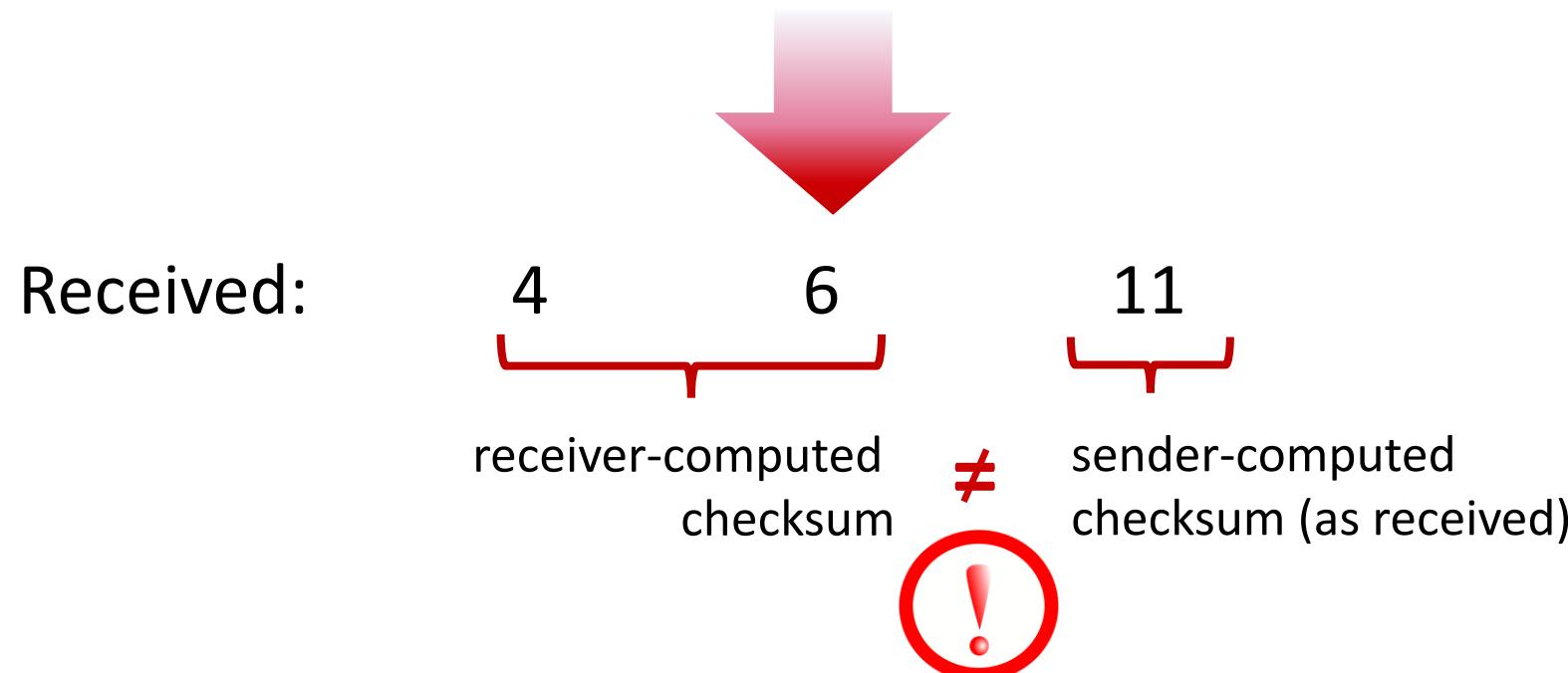
Service	Port
FTP	21
HTTP	80
SMTP	25
DNS	53
TFTP	69
Daytime	13
HTTPS	443

$$
\begin{array}{ccc}
2 & A & 5 \\
16^2 & 16^1 & 16^0 \\
256 \times 2 = 512 & 16 \times 10 = 160 & 1 \times 5 = 5 \\
512 & + \quad 160 & + \quad 5 \\
\end{array}
$$

677

$$(2A5)_{16} = (677)_{10}$$

UDP checksum

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

	1st number	2nd number	sum
Transmitted:	5	6	11

	1st number	2nd number	sum
Received:	4	6	11

receiver-computed checksum ≠ sender-computed checksum (as received)

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
          ─────────────────────────────────
wraparound ⓵ 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1

sum          1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
checksum     0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
```

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

* Check out the online interactive exercises for more examples: 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 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1          1 0
```

wraparound ①1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1

sum   1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0

checksum   0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1

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
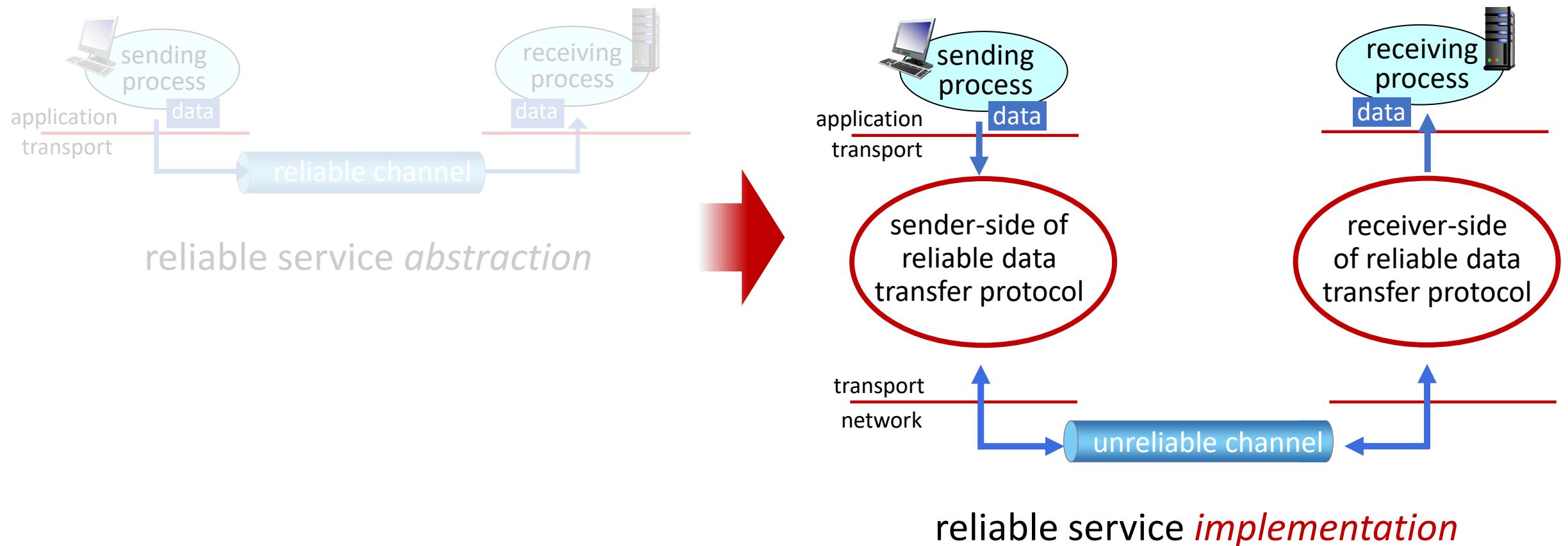
Principles of reliable data transfer
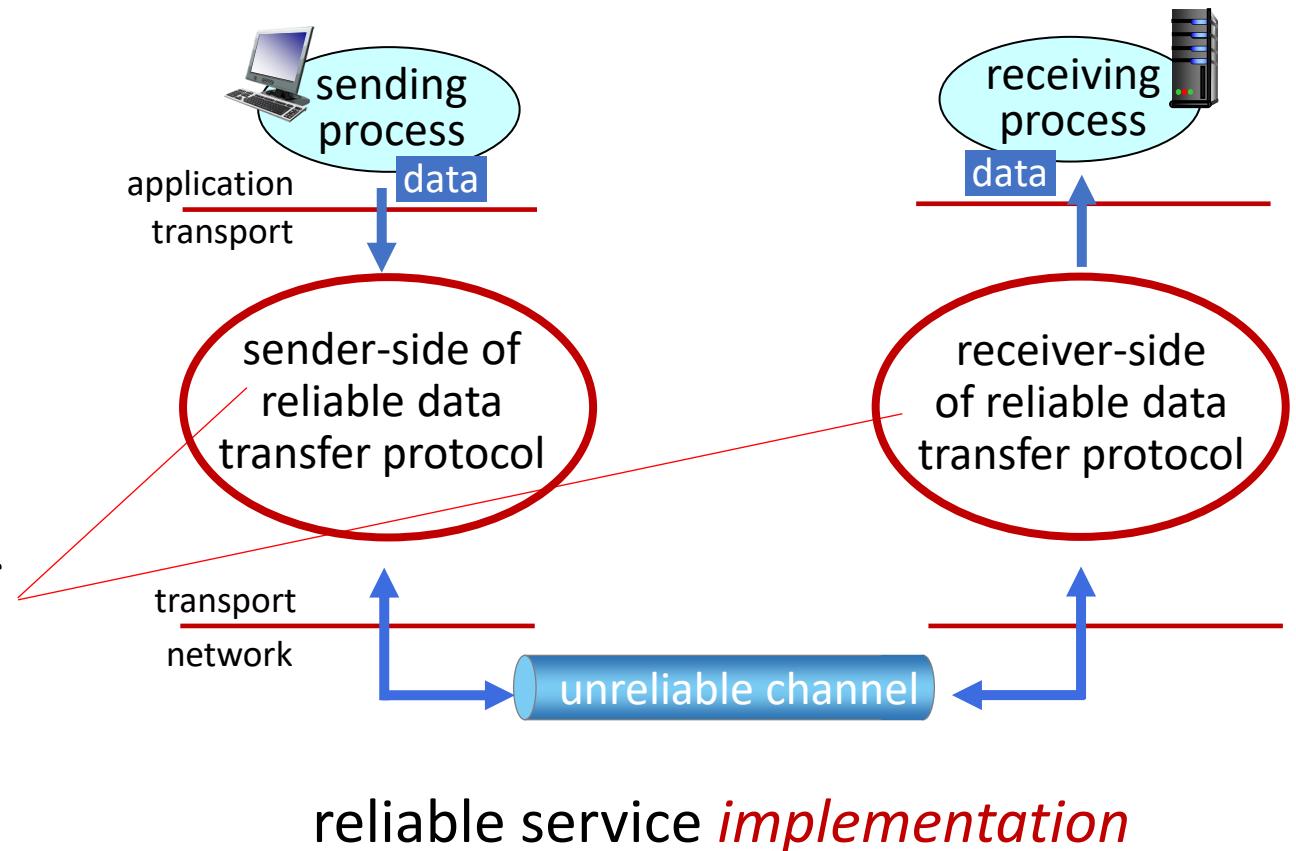
reliable service *abstraction*

Principles of reliable data transfer

reliable service *abstraction*

reliable service *implementation*

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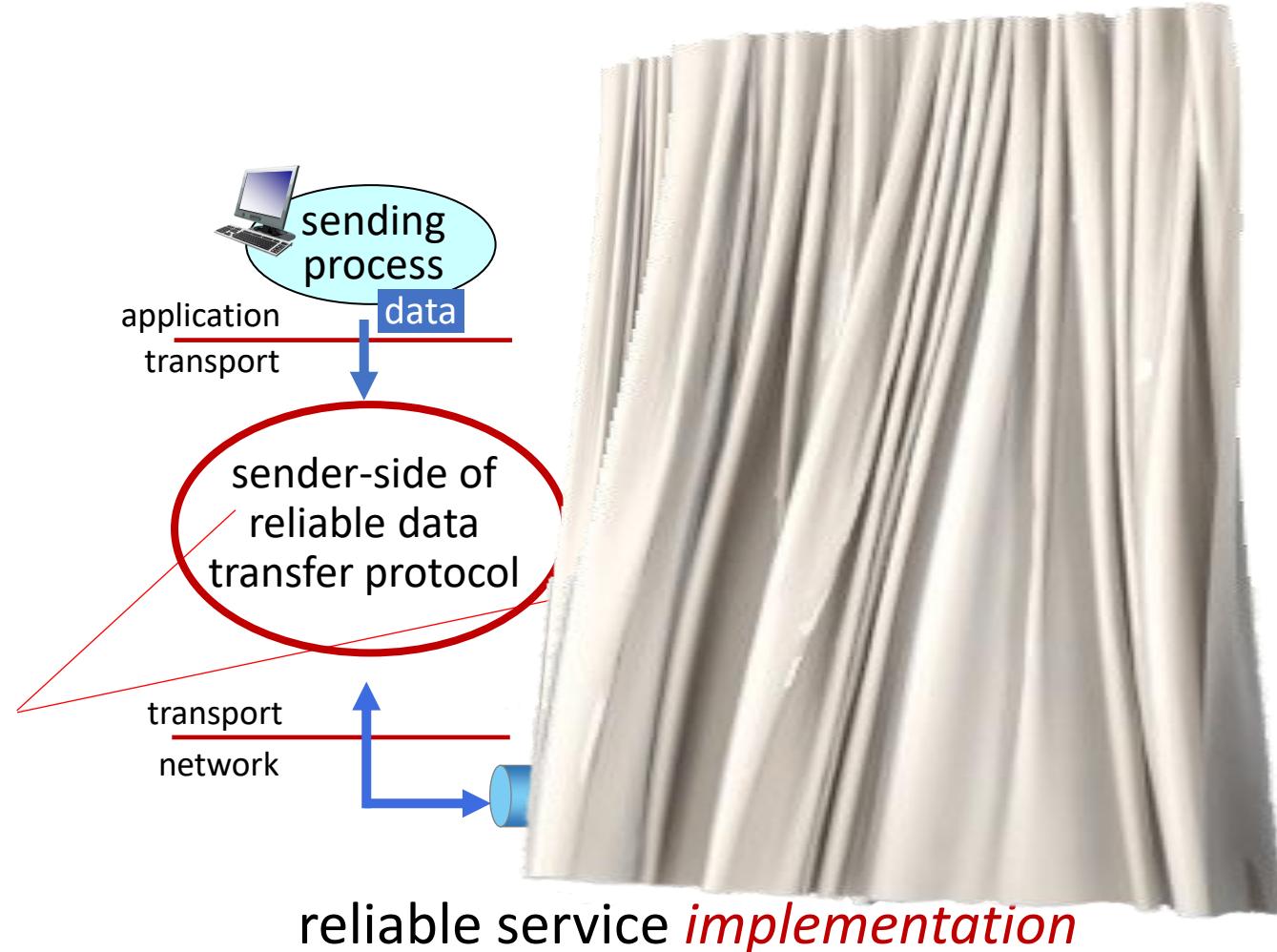
Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)

reliable service *implementation*

Principles of reliable data transfer
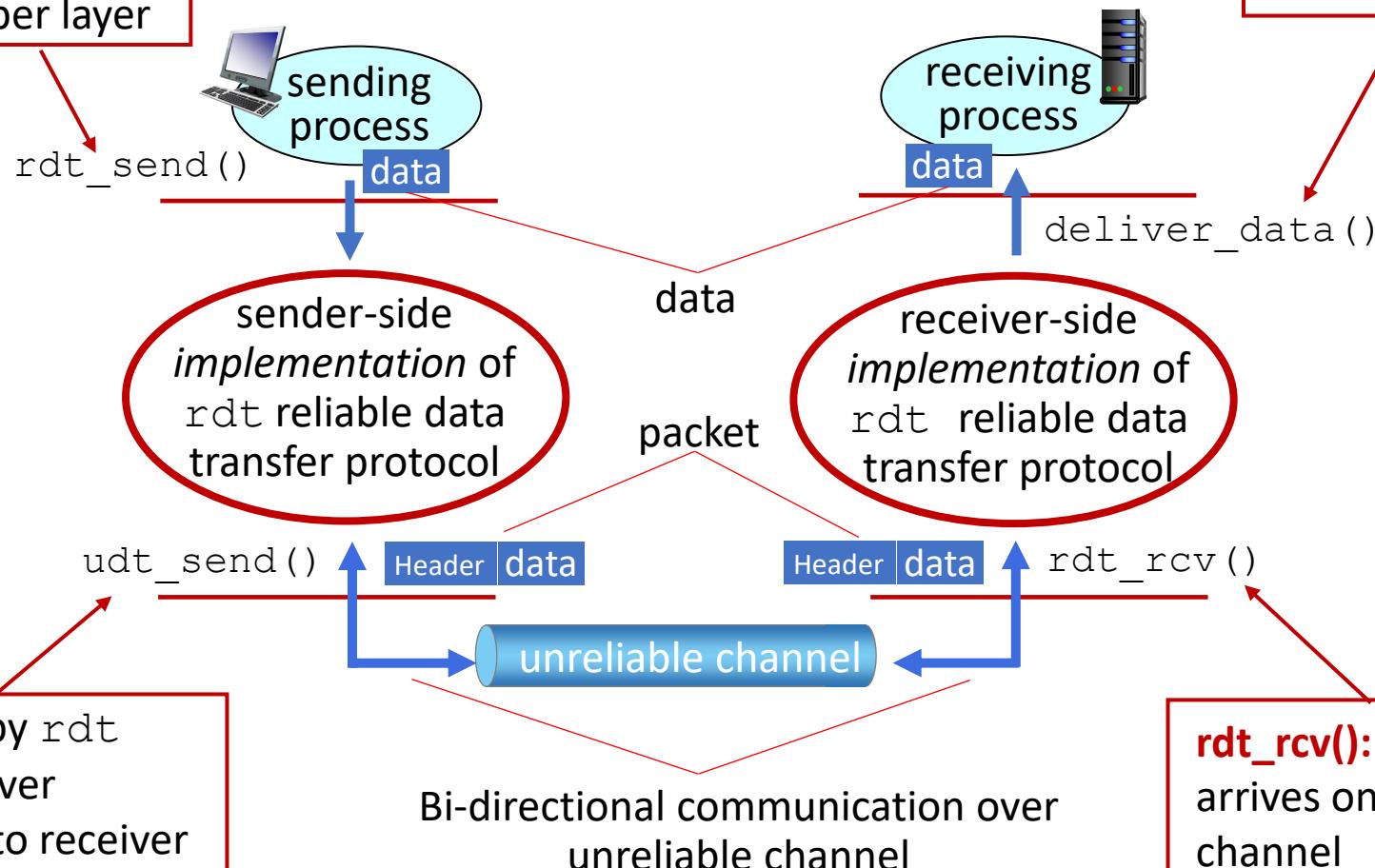
Sender, receiver do *not* know the "state" of each other, e.g., was a message received?

- unless communicated via a message

sending process

application
transport
data

sender-side of reliable data transfer protocol

transport
network

reliable service *implementation*

Reliable data transfer protocol (rdt): interfaces
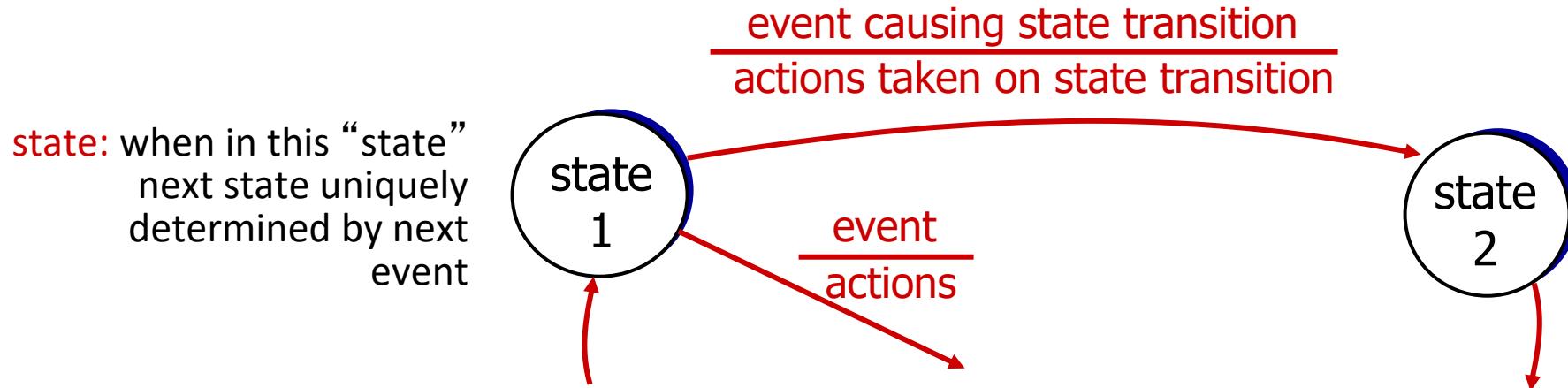
rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by `rdt` to deliver data to upper layer

sending process

receiving process

`rdt_send()`

data

data

`deliver_data()`

sender-side *implementation* of `rdt` reliable data transfer protocol

data

receiver-side *implementation* of `rdt` reliable data transfer protocol

packet

`udt_send()`

Header data

Header data

`rdt_rcv()`

unreliable channel

udt_send(): called by `rdt` to transfer packet over unreliable channel to receiver

Bi-directional communication over unreliable channel

rdt_rcv(): called when packet arrives on receiver side of channel

Reliable data transfer: getting started
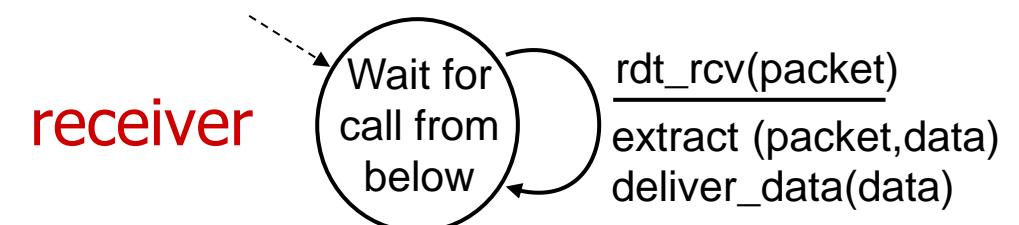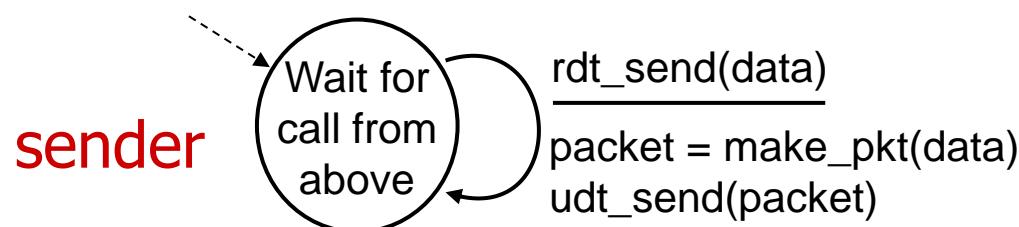
We will:

- 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

event causing state transition
_____
actions taken on state transition

state: when in this "state"
next state uniquely
determined by next
event

state 1

event
_____
actions

state 2

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

sender

Wait for call from above

rdt_send(data)
─────────────────
packet = make_pkt(data)
udt_send(packet)

receiver

Wait for call from below

rdt_rcv(packet)
─────────────────
extract (packet,data)
deliver_data(data)

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?
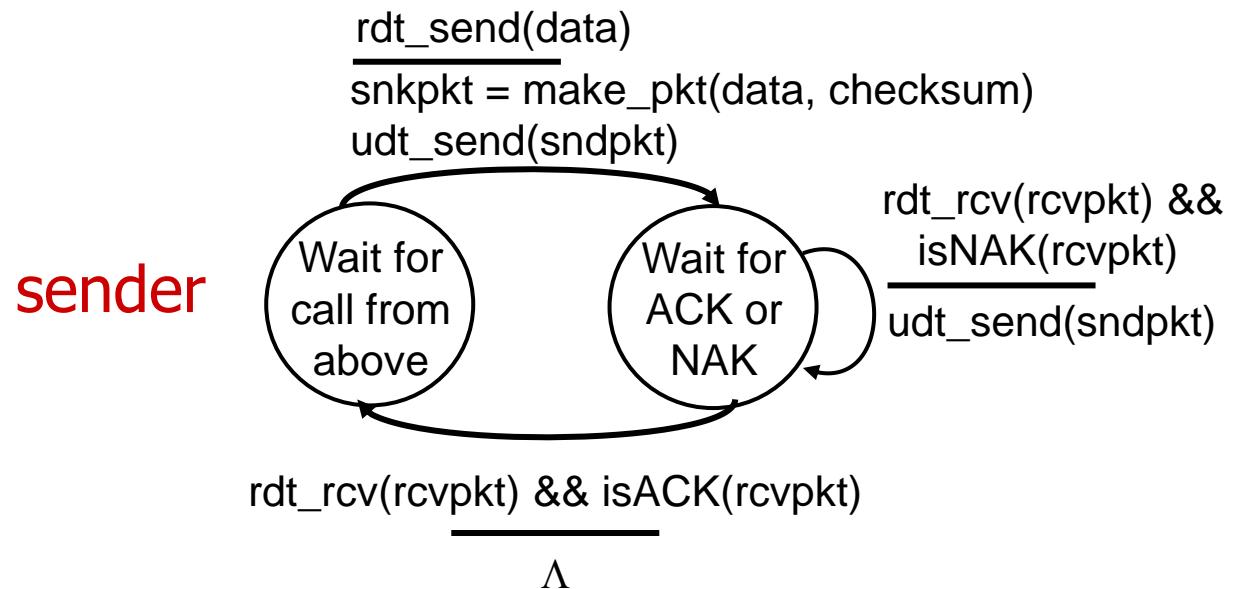
rdt2.0: channel with bit errors

- 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
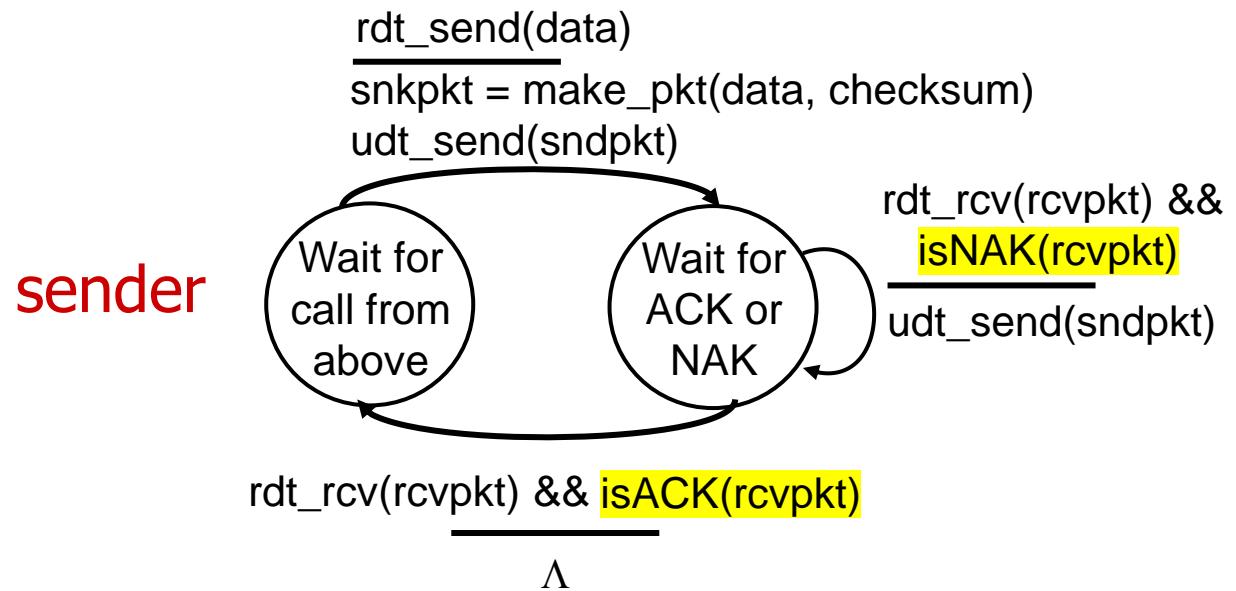
stop and wait

sender sends one packet, then waits for receiver response

rdt2.0: FSM specifications

sender

rdt_send(data)
_____
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
_____
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
_____
Λ

rdt2.0: FSM specification

sender

rdt_send(data)
_____
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

Wait for call from above → **Wait for ACK or NAK**

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
_____
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
_____
Λ
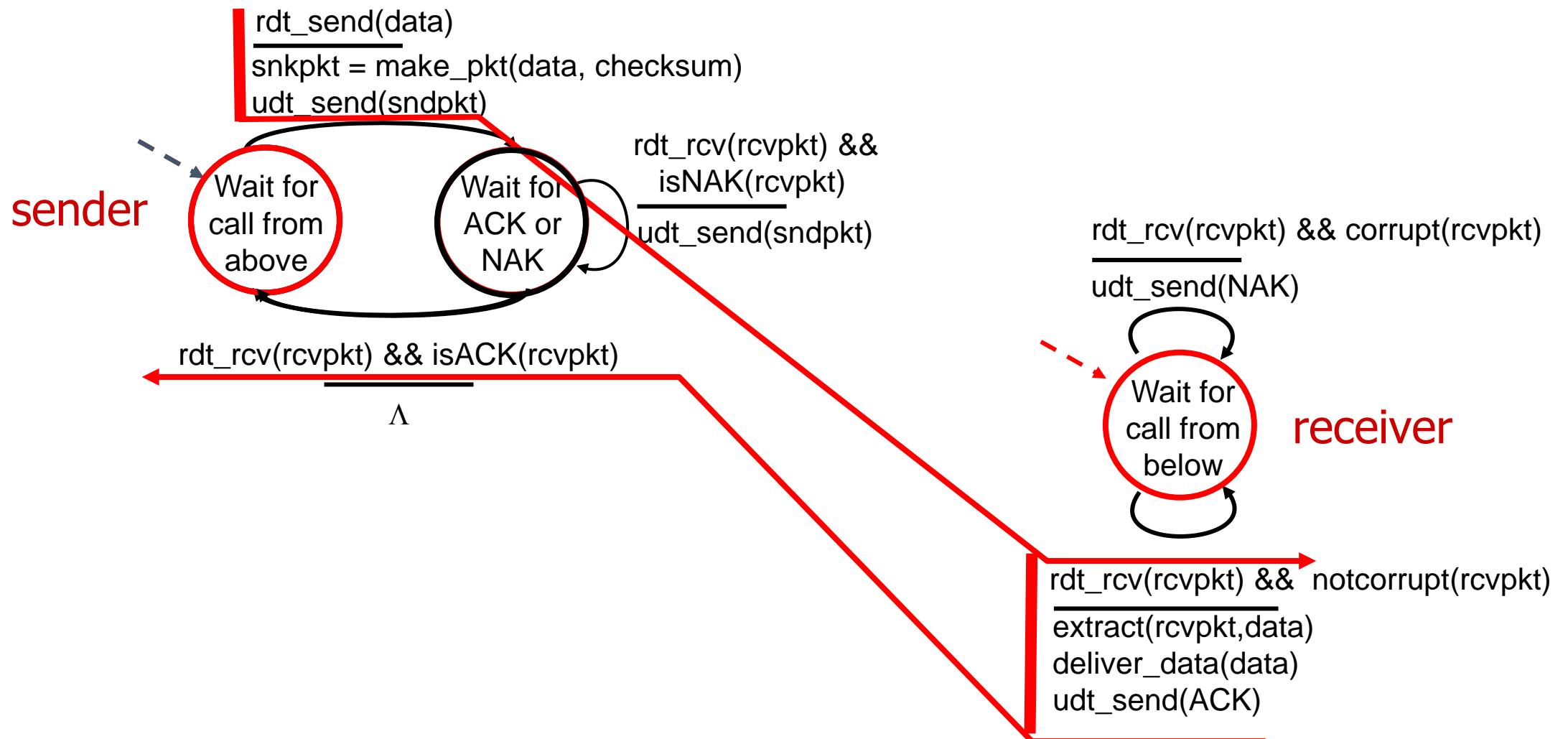
Note: "state" of receiver (did the receiver get my message correctly?) isn't known to sender unless somehow communicated from receiver to sender
  ▪ that's why we need a protocol!
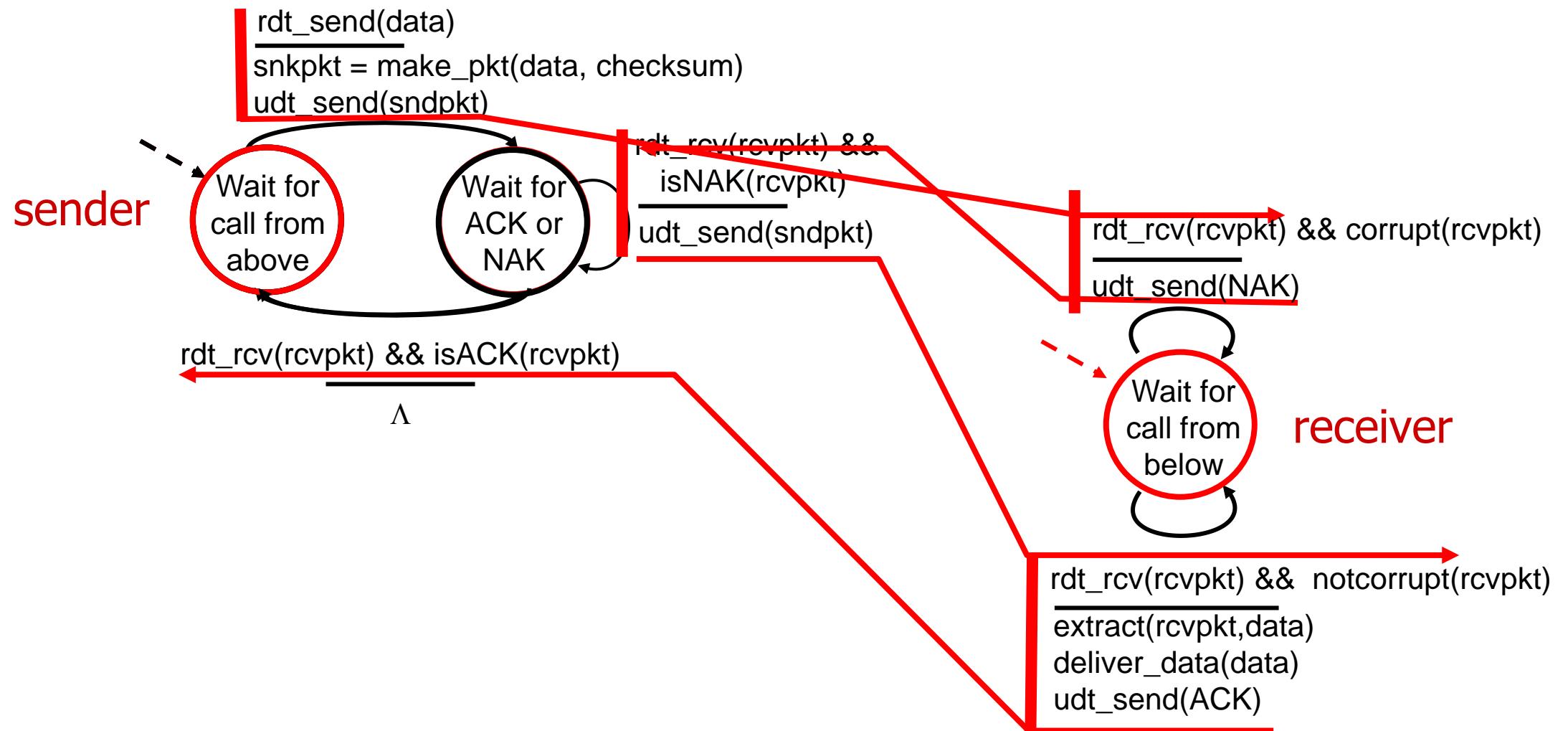
rdt2.0: operation with no errors

rdt_send(data)
———————————
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

sender

Wait for
call from
above

Wait for
ACK or
NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
———————————
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
———————————
Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
———————————
udt_send(NAK)

Wait for
call from
below

receiver

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
———————————
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: corrupted packet scenario

sender

rdt_send(data)

snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) && isNAK(rcvpkt)

udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)

udt_send(NAK)

receiver

Wait for call from below

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

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
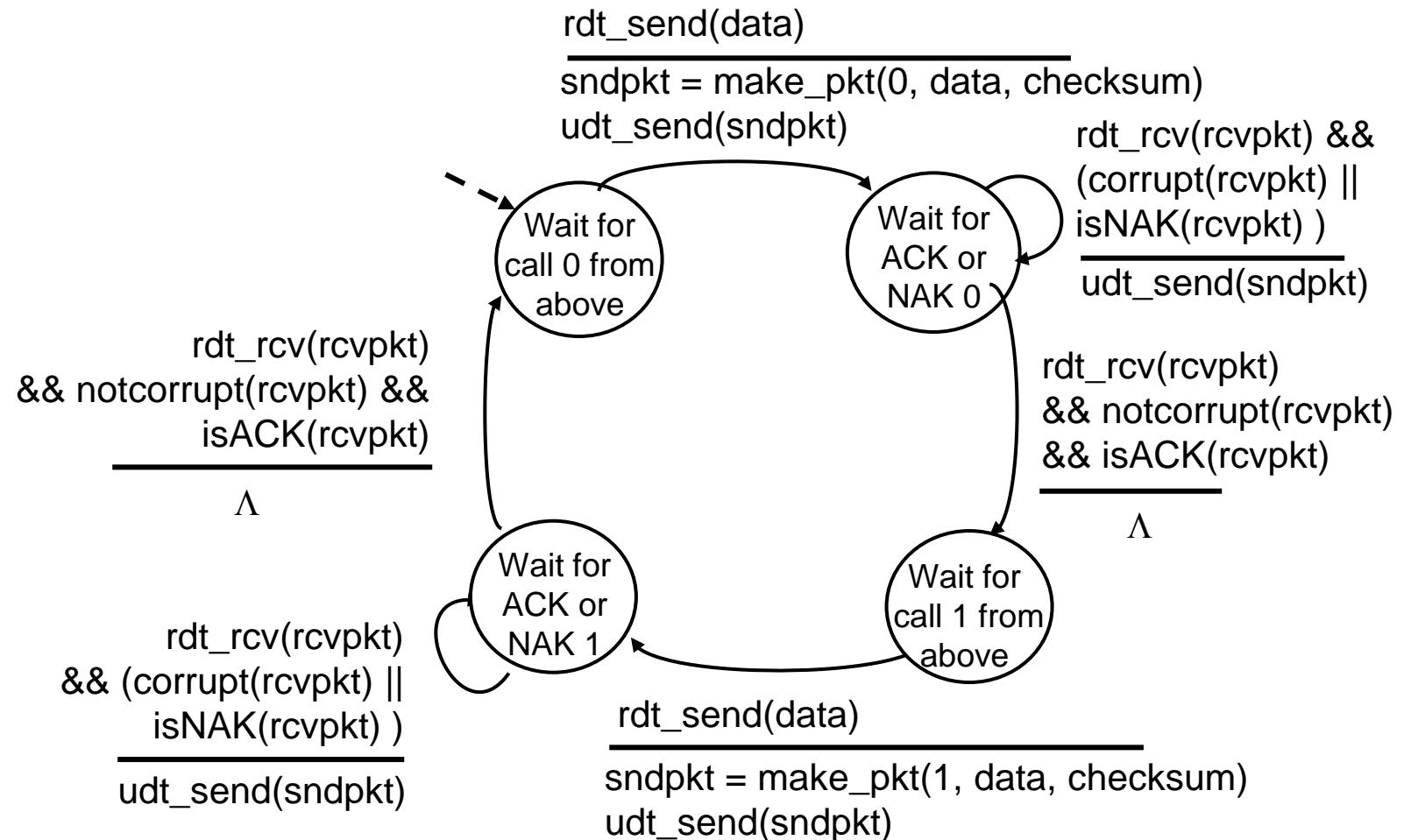
handling duplicates:

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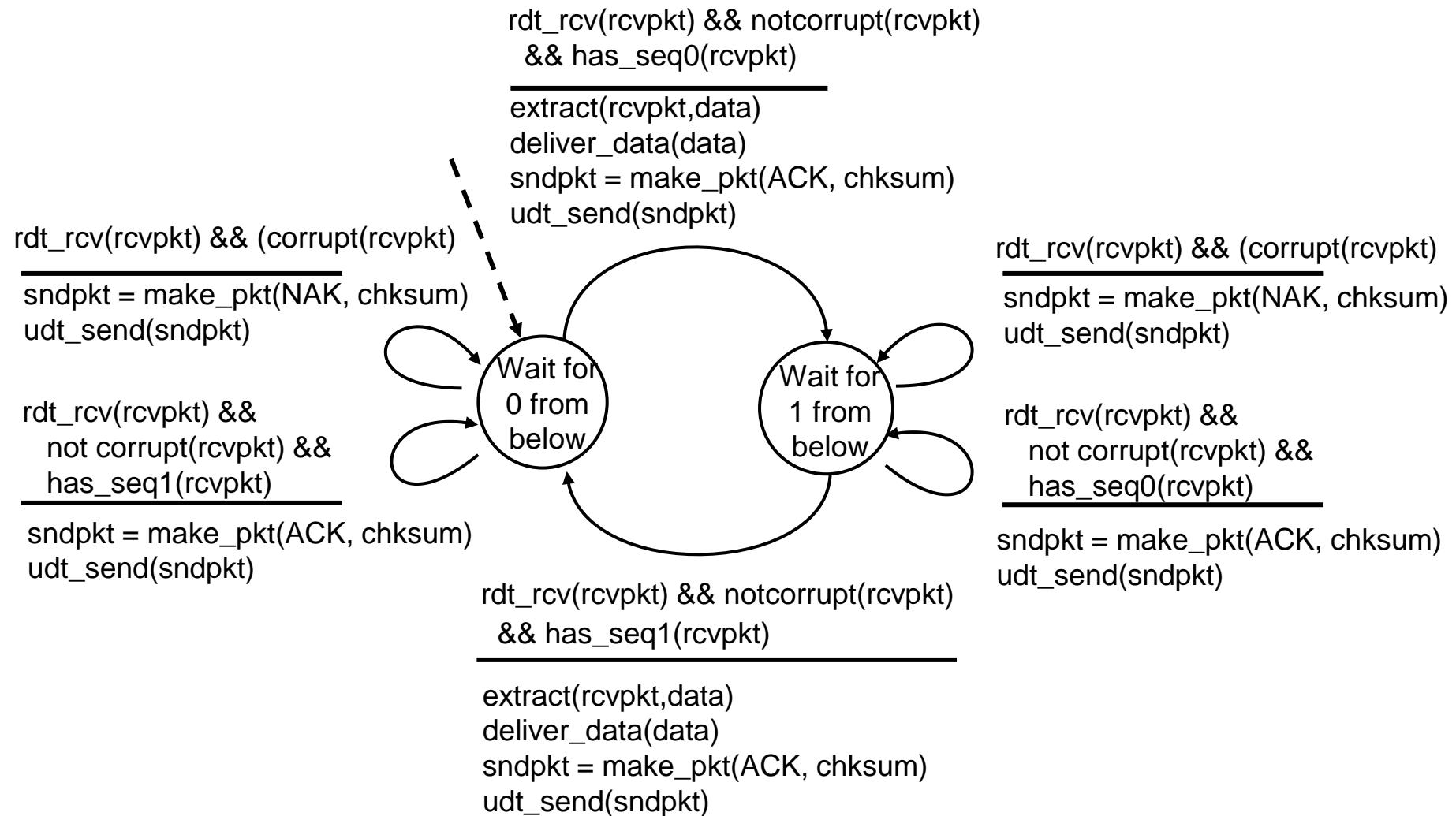
stop and wait

sender sends one packet, then waits for receiver response

rdt2.1: sender, handling garbled ACK/NAKs

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

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
 && has_seq0(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
_____
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
  not corrupt(rcvpkt) &&
  has_seq1(rcvpkt)
_____
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
_____
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
  not corrupt(rcvpkt) &&
  has_seq0(rcvpkt)
_____
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

(**Wait for 0 from below**)   (**Wait for 1 from below**)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
 && has_seq1(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt2.1: discussion

- 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
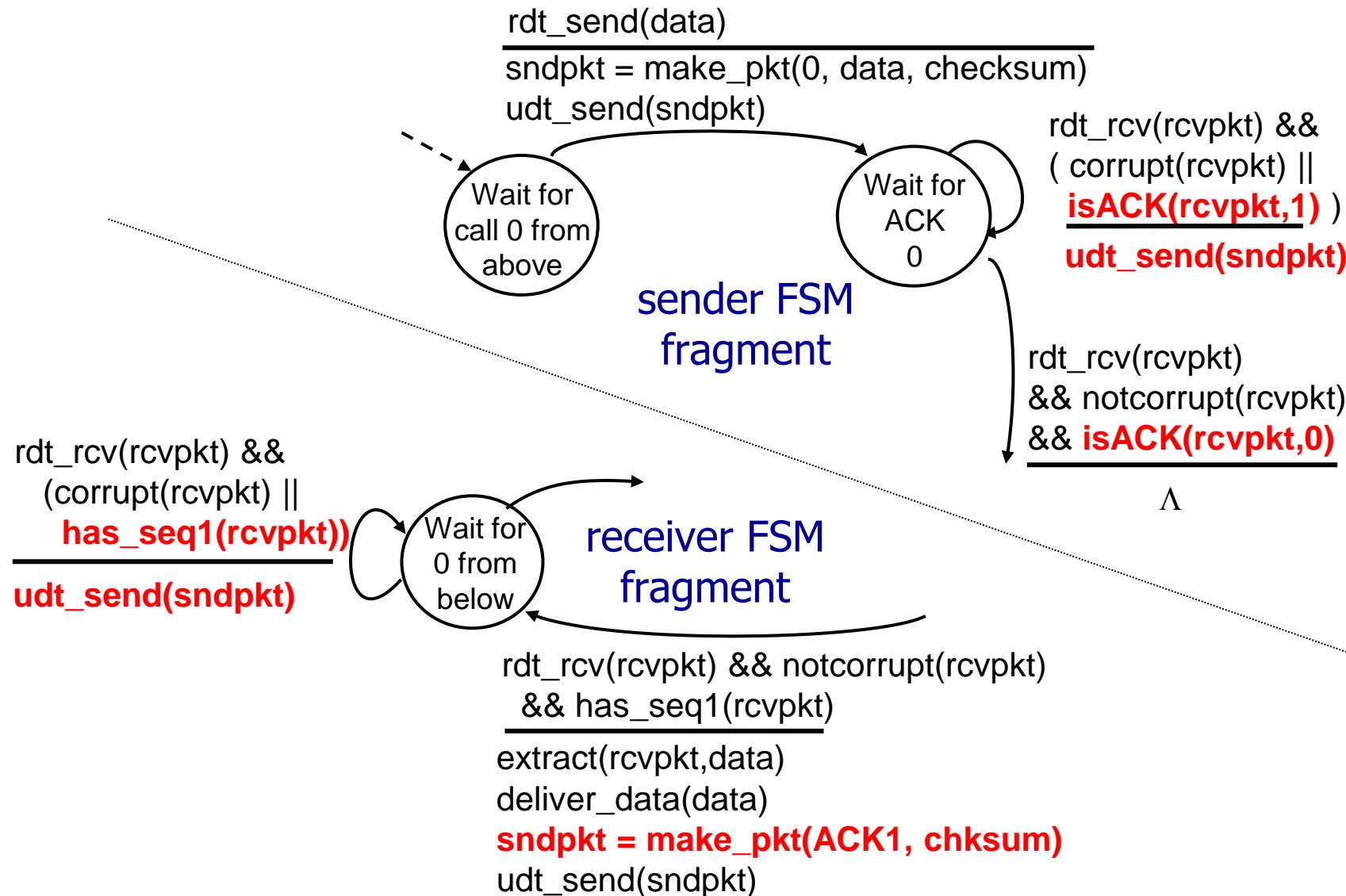
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #

- 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

rdt_send(data)
_____
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)

(Wait for call 0 from above) (Wait for ACK 0)

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
  isACK(rcvpkt,1))
_____
udt_send(sndpkt)

sender FSM fragment

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& **isACK(rcvpkt,0)**
_____
Λ

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
  has_seq1(rcvpkt))
_____
udt_send(sndpkt)

(Wait for 0 from below) **receiver FSM fragment**

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK1, chksum)
udt_send(sndpkt)

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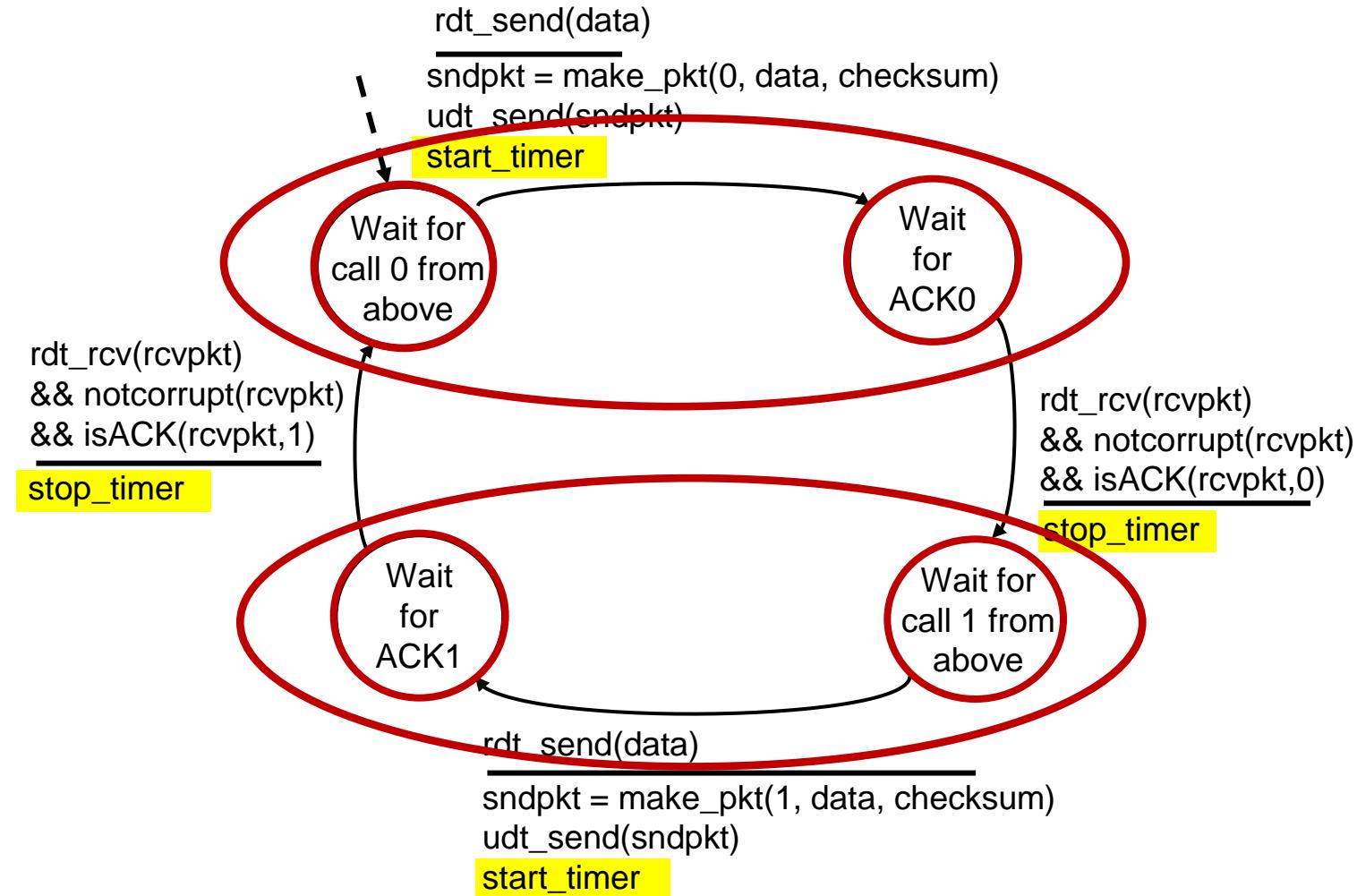
rdt3.0: channels with errors *and* loss

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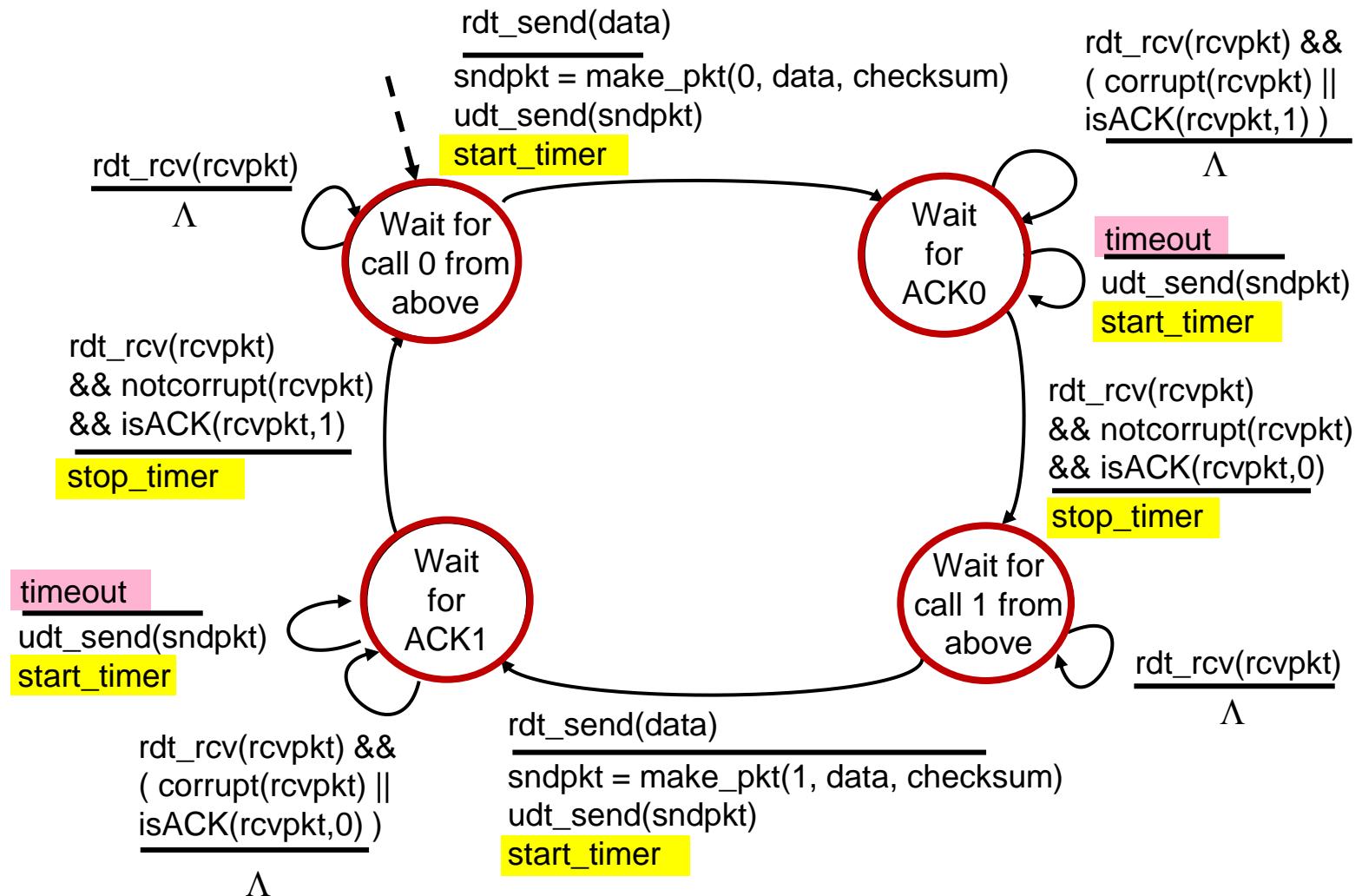
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq #s already handles this!
  - receiver must specify seq # of packet being ACKed
- use countdown timer to interrupt after "reasonable" amount of time

timeout

rdt3.0 sender

rdt_send(data)

sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer

Wait for call 0 from above

Wait for ACK0

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
stop_timer

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
stop_timer

Wait for ACK1

Wait for call 1 from above

rdt_send(data)

sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer

rdt3.0 sender

rdt_send(data)
_____
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt)
_____
Λ

Wait for call 0 from above

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))
_____
Λ

Wait for ACK0

timeout
_____
udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
_____
stop_timer

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
_____
stop_timer

timeout
_____
udt_send(sndpkt)
start_timer

Wait for ACK1

Wait for call 1 from above

rdt_rcv(rcvpkt)
_____
Λ

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,0))
_____
Λ

rdt_send(data)
_____
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer

rdt3.0 in action

(a) no loss
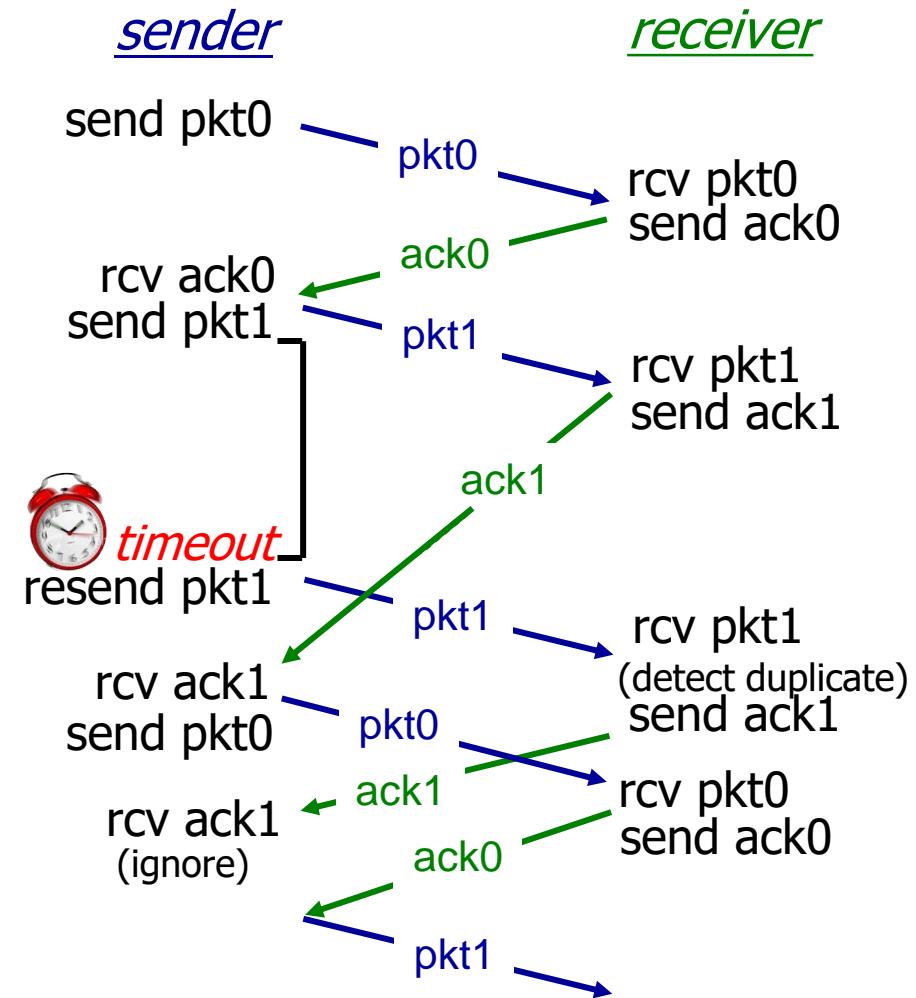
(b) packet loss

rdt3.0 in action

(c) ACK loss
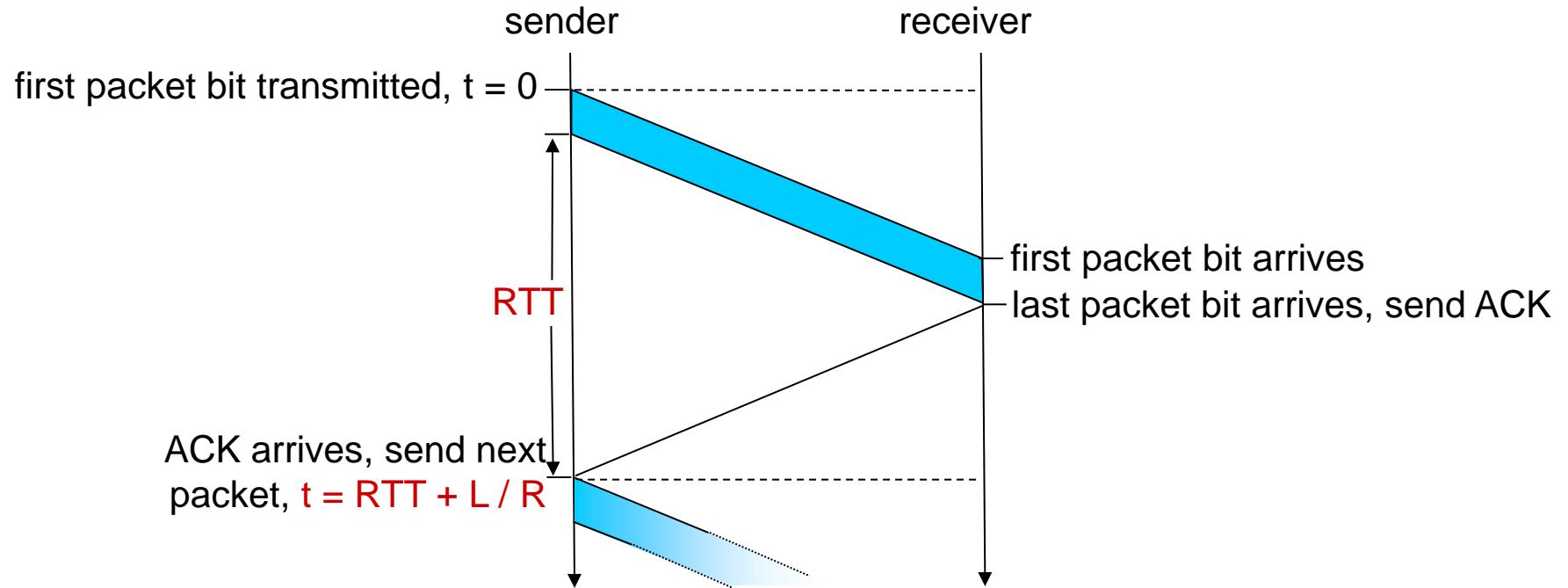
(d) premature timeout/ delayed ACK

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

- U_{sender}: *utilization* – fraction of time sender busy sending

- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet

 - time to transmit packet into channel:
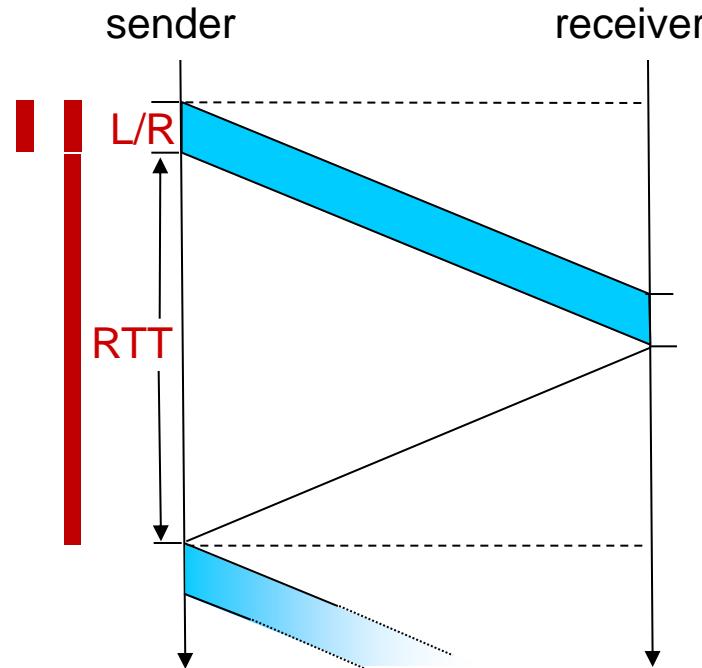
  $$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

rdt3.0: stop-and-wait operation

sender   receiver

first packet bit transmitted, t = 0

first packet bit arrives

RTT   last packet bit arrives, send ACK

ACK arrives, send next
packet, t = RTT + L / R

rdt3.0: stop-and-wait operation

$$U_{sender} = \frac{L / R}{RTT + L / R}$$

$$= \frac{.008}{30.008}$$

$$= 0.00027$$

sender   receiver

L/R

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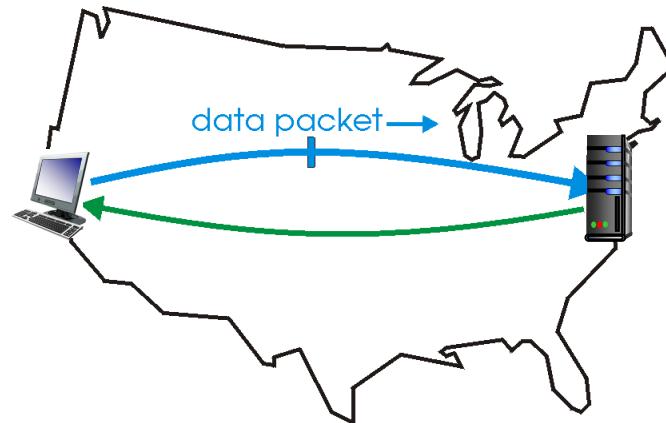
- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

rdt3.0: pipelined protocols operation
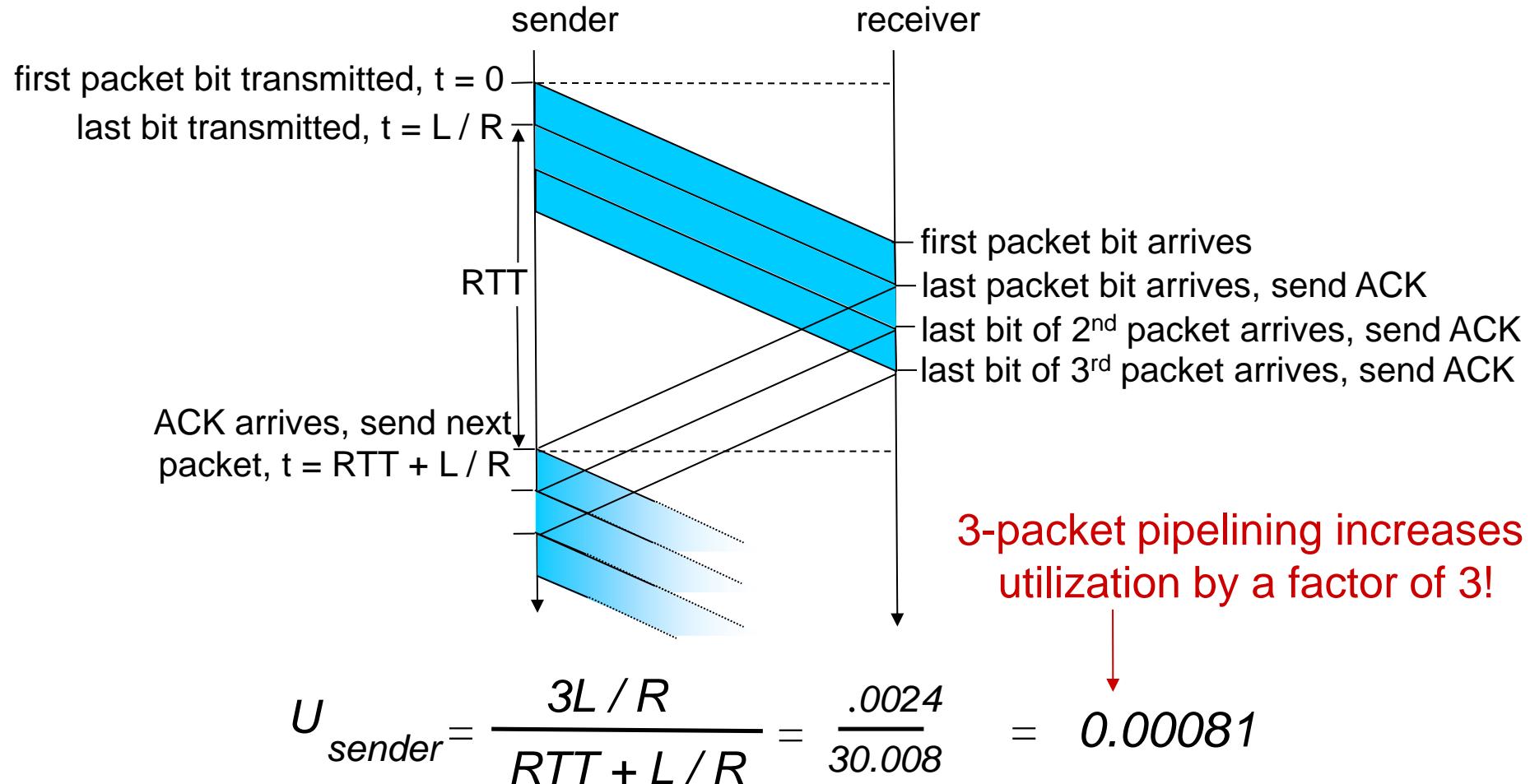
pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged packets

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

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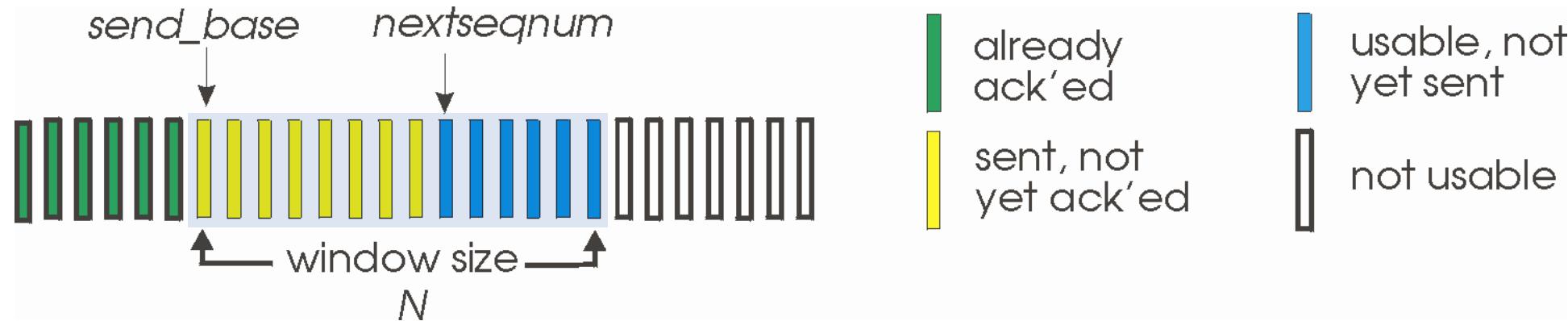
(a) a stop-and-wait protocol in operation

Pipelining: increased utilization

sender   receiver

first packet bit transmitted, t = 0

last bit transmitted, t = L / R

RTT

first packet bit arrives

last packet bit arrives, send ACK

last bit of 2nd packet arrives, send ACK

last bit of 3rd packet arrives, send ACK

ACK arrives, send next
packet, t = RTT + L / R

3-packet pipelining increases
utilization by a factor of 3!

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$
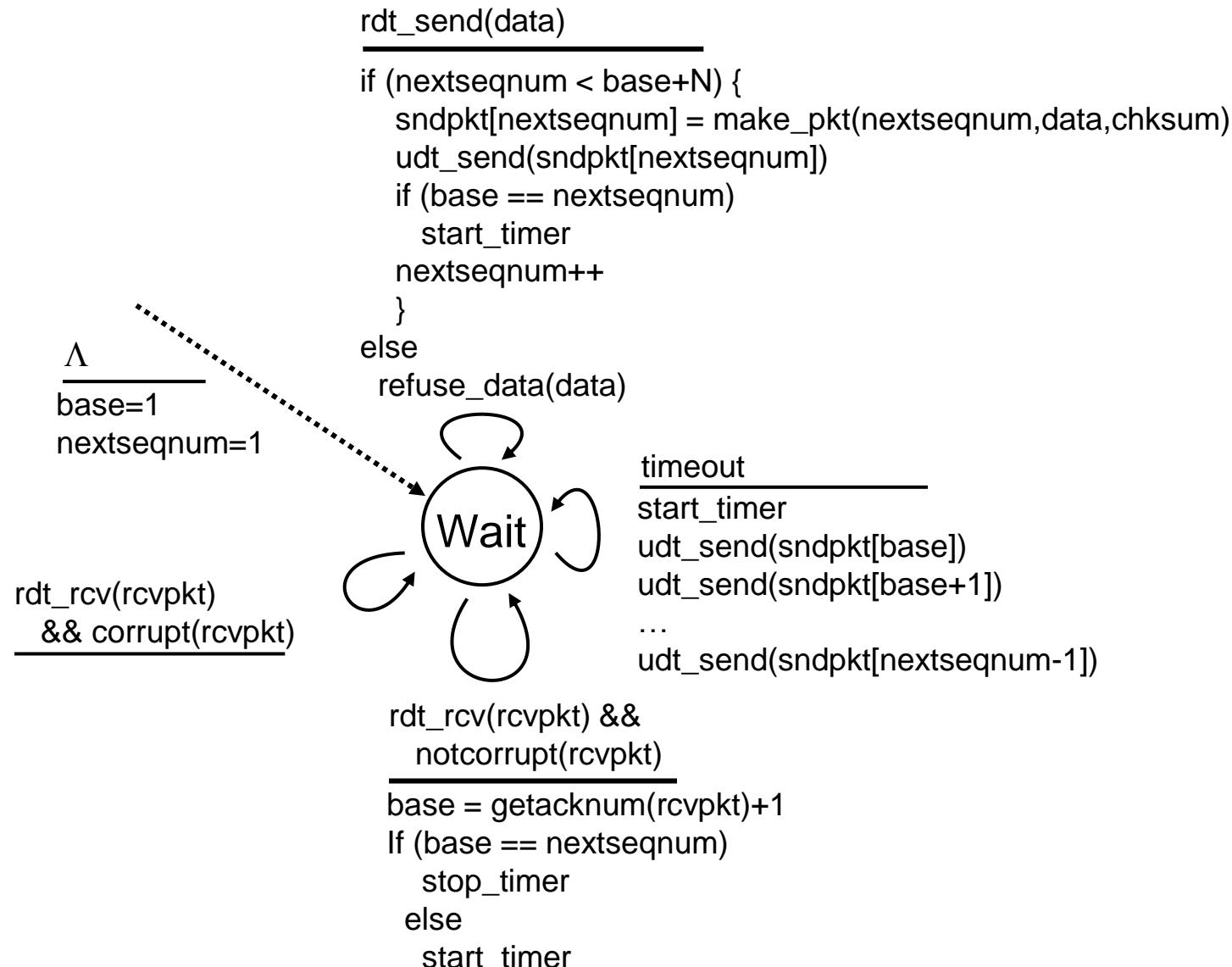
Go-Back-N: sender

- 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
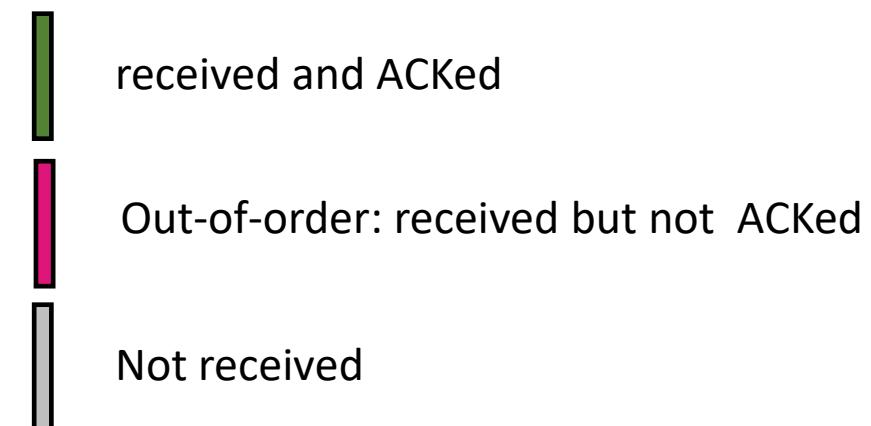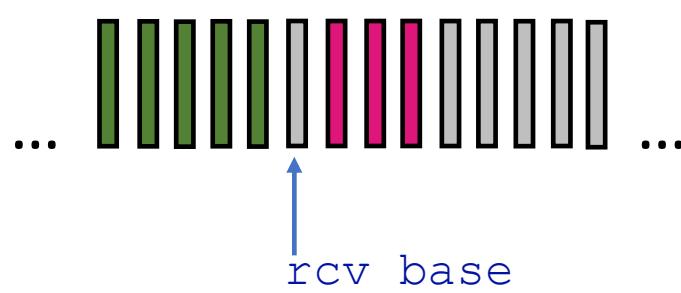
Go-Back-N: sender extended FSM

rdt_send(data)
‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾

if (nextseqnum < base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum)
        start_timer
    nextseqnum++
    }
else
  refuse_data(data)

Λ
‾‾‾‾
base=1
nextseqnum=1

(Wait)

rdt_rcv(rcvpkt)
   && corrupt(rcvpkt)
‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾

timeout
‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])

rdt_rcv(rcvpkt) &&
   notcorrupt(rcvpkt)
‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
 else
    start_timer

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:

... rcv_base ...

received and ACKed

Out-of-order: received but not ACKed
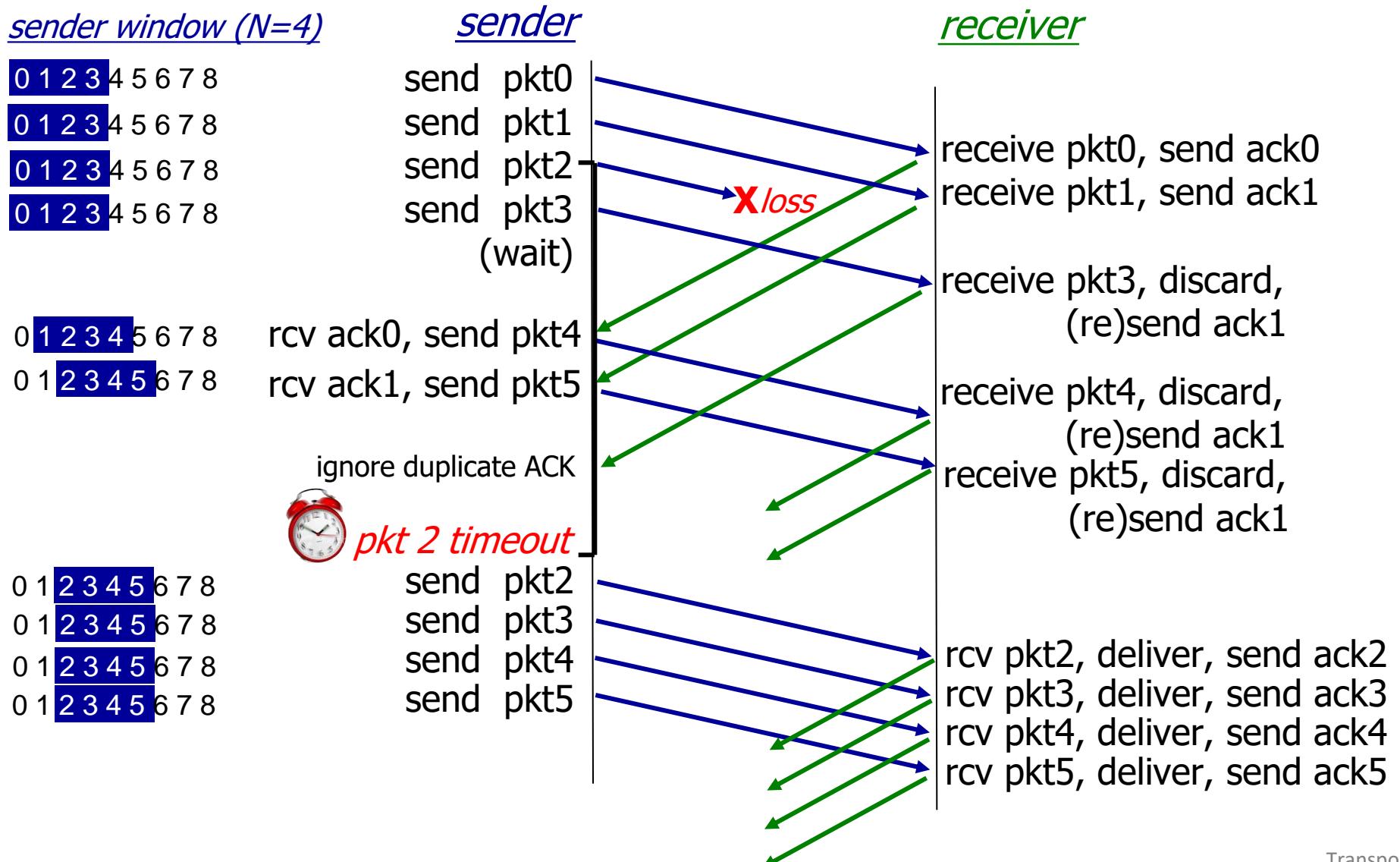
Not received

Go-Back-N: receiver extended FSM

any other event
_____
udt_send(sndpkt)

rdt_rcv(rcvpkt)
  && notcorrupt(rcvpkt)
  && hasseqnum(rcvpkt,expectedseqnum)
_____

Λ
_____
expectedseqnum=1
sndpkt =
  make_pkt(expectedseqnum,ACK,chksum)

Wait

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(expectedseqnum,ACK,chksum)
udt_send(sndpkt)
expectedseqnum++

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 #

Go-Back-N in action

sender window (N=4)　　sender　　receiver

0 1 2 3 4 5 6 7 8　　send pkt0
0 1 2 3 4 5 6 7 8　　send pkt1
0 1 2 3 4 5 6 7 8　　send pkt2
0 1 2 3 4 5 6 7 8　　send pkt3
　　　　　　　　　　　(wait)

X *loss*

receive pkt0, send ack0
receive pkt1, send ack1

receive pkt3, discard,
　　　(re)send ack1

0 1 2 3 4 5 6 7 8　　rcv ack0, send pkt4
0 1 2 3 4 5 6 7 8　　rcv ack1, send pkt5

receive pkt4, discard,
　　　(re)send ack1
receive pkt5, discard,
　　　(re)send ack1

ignore duplicate ACK

pkt 2 timeout

0 1 2 3 4 5 6 7 8　　send pkt2
0 1 2 3 4 5 6 7 8　　send pkt3
0 1 2 3 4 5 6 7 8　　send pkt4
0 1 2 3 4 5 6 7 8　　send pkt5

rcv pkt2, deliver, send ack2
rcv pkt3, deliver, send ack3
rcv pkt4, deliver, send ack4
rcv pkt5, deliver, send ack5

Selective repeat: the approach

- *pipelining*: *multiple* packets in flight

- *receiver individually ACKs* all correctly received packets
  - buffers packets, as needed, for in-order delivery to upper layer

- sender:
  - maintains (conceptually) a timer for each unACKed pkt
    - timeout: retransmits single unACKed packet associated with timeout
  - maintains (conceptually) "window" over *N* consecutive seq #s
    - limits pipelined, "in flight" packets to be within this window

Selective repeat: sender, receiver windows

(a) sender view of sequence numbers

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-1]:

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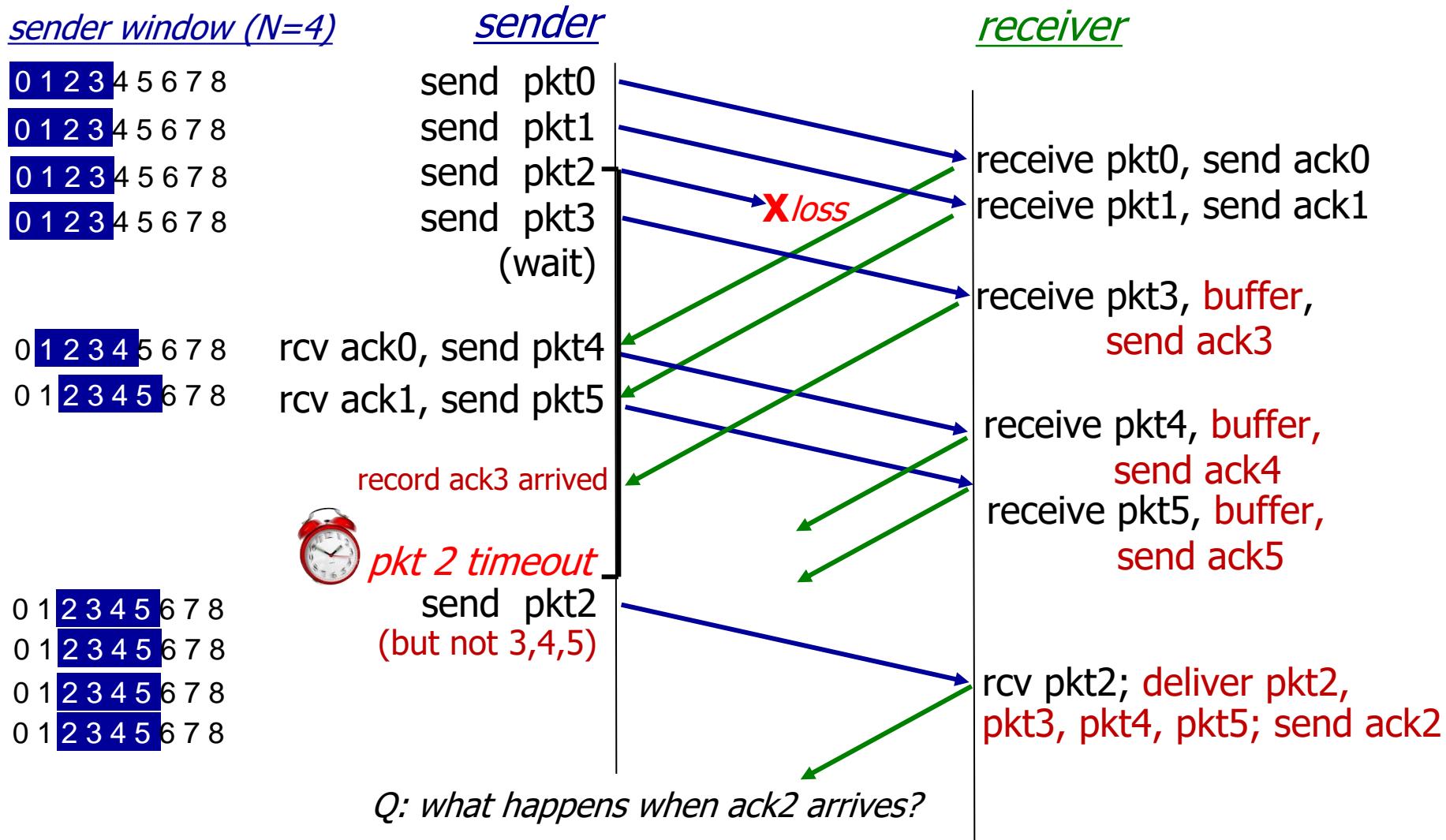
### packet n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore
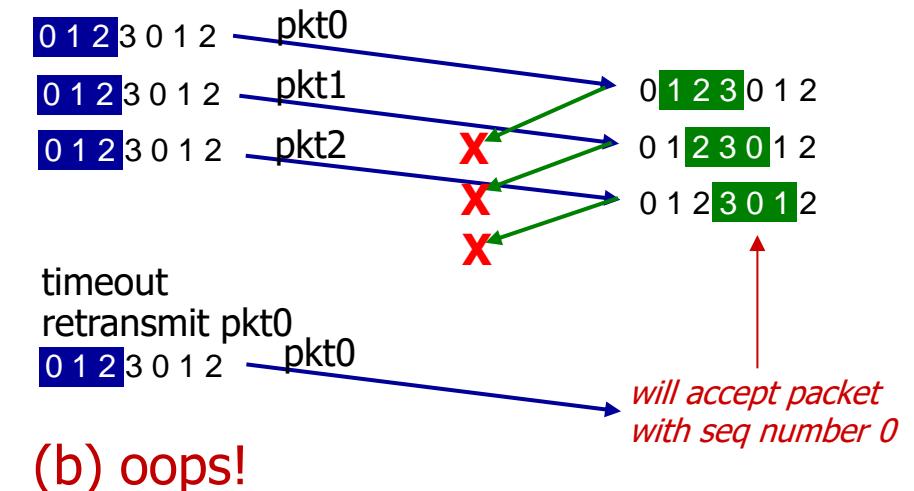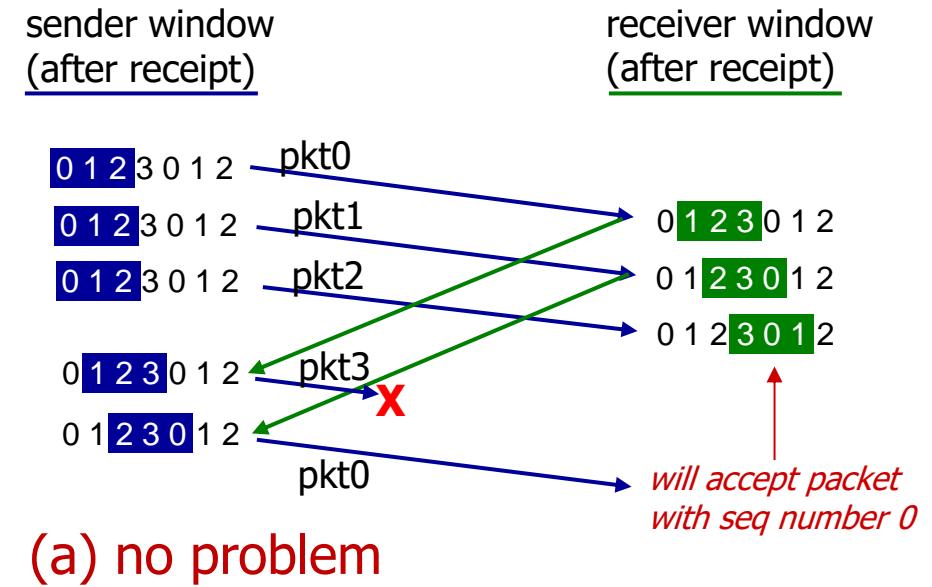
Selective Repeat in action

sender window (N=4)

sender

receiver

0 1 2 3 4 5 6 7 8   send pkt0

0 1 2 3 4 5 6 7 8   send pkt1

0 1 2 3 4 5 6 7 8   send pkt2   receive pkt0, send ack0

0 1 2 3 4 5 6 7 8   send pkt3   **X** *loss*   receive pkt1, send ack1

                 (wait)   receive pkt3, buffer,   send ack3

0 1 2 3 4 5 6 7 8   rcv ack0, send pkt4

0 1 2 3 4 5 6 7 8   rcv ack1, send pkt5   receive pkt4, buffer,   send ack4

     record ack3 arrived   receive pkt5, buffer,   send ack5

pkt 2 timeout

0 1 2 3 4 5 6 7 8   send pkt2

0 1 2 3 4 5 6 7 8   (but not 3,4,5)

0 1 2 3 4 5 6 7 8   rcv pkt2; deliver pkt2,

0 1 2 3 4 5 6 7 8   pkt3, pkt4, pkt5; send ack2

Q: what happens when ack2 arrives?

Selective repeat: a dilemma!

example:

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

0 1 2 3 0 1 2   pkt0

0 1 2 3 0 1 2   pkt1   0 1 2 3 0 1 2

0 1 2 3 0 1 2   pkt2   0 1 2 3 0 1 2

                             0 1 2 3 0 1 2

0 1 2 3 0 1 2   pkt3   X

0 1 2 3 0 1 2
            pkt0

*will accept packet
with seq number 0*

(a) no problem

0 1 2 3 0 1 2   pkt0

0 1 2 3 0 1 2   pkt1   0 1 2 3 0 1 2

0 1 2 3 0 1 2   pkt2   X   0 1 2 3 0 1 2

                      X   0 1 2 3 0 1 2

                      X

timeout
retransmit pkt0

0 1 2 3 0 1 2   pkt0

*will accept packet
with seq number 0*

(b) oops!

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)?

0 1 2 3 0 1 2   pkt0
0 1 2 3 0 1 2   pkt1
0 1 2 3 0 1 2   pkt2

0 1 2 3 0 1 2   pkt3
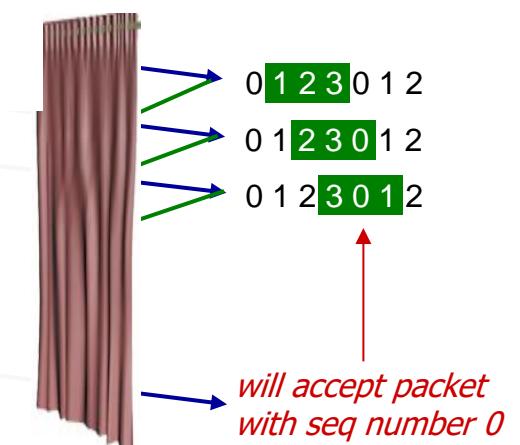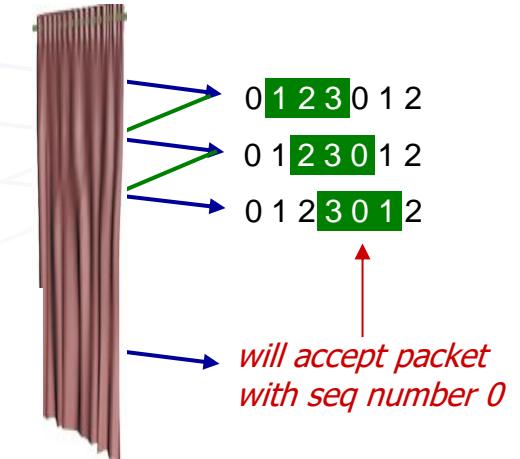
0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2

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

will accept packet with seq number 0

0 1 2 3 0 1 2   pkt2

timeout
retransmit pkt0
0 1 2 3 0 1 2   pkt0

0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2

will accept packet with seq number 0

(b) oops!

	Max. Sender Window size	Max. Receiver Window size	Equation for sequence number	Sequence number
Stop & Wait	1	1	Modulo 2	0,1,0,1,0, 1
GBN	$2^m - 1$	1	Modulo 2^m	0 to $2^m - 1$
SR	2^{m-1}	2^{m-1}	Modulo 2^m	0 to $2^m - 1$

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 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

32 bits

ACK: seq # of next expected byte; A bit: this is an ACK
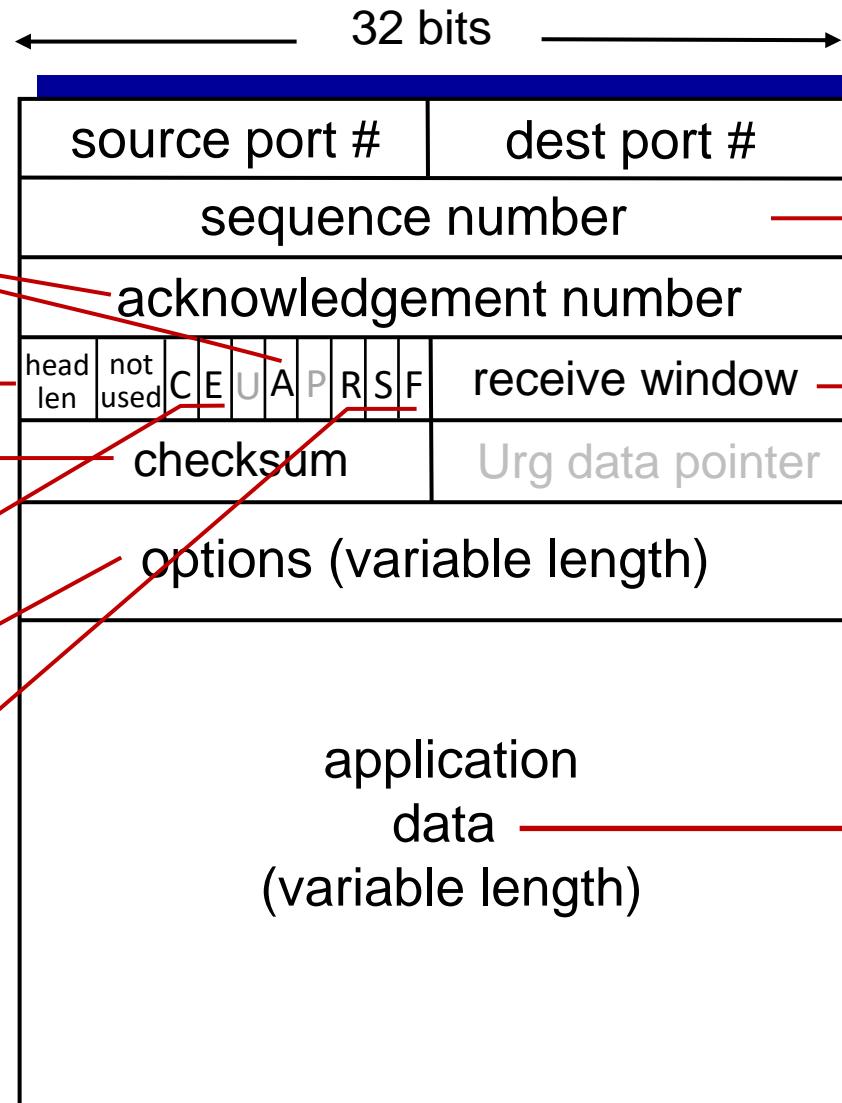
length (of TCP header)

Internet checksum

C, E: congestion notification

TCP options

RST, SYN, FIN: connection management

source port #	dest port #
sequence number	
acknowledgement number	

head len	not used	C E U A P R S F	receive window

checksum	Urg data pointer

options (variable length)

application data (variable length)

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:
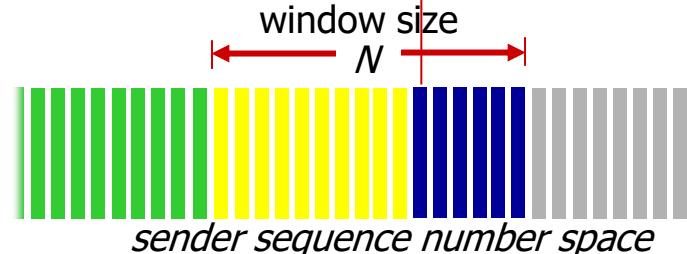
- seq # of next byte expected from other side

- cumulative ACK

Q: how receiver handles out-of-order segments

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

outgoing segment from sender
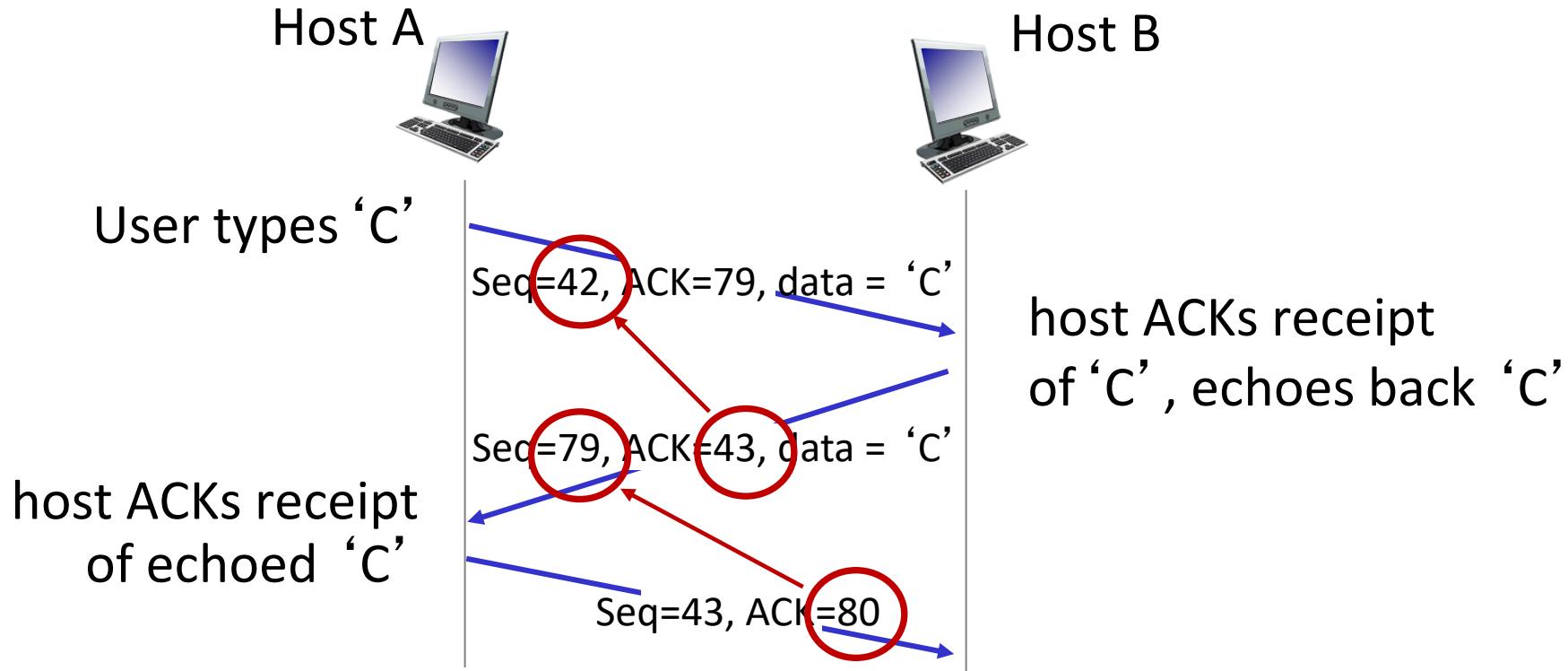
source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer

window size
N

sender sequence number space

sent ACKed

sent, not-yet ACKed ("in-flight")

usable but not yet sent

not usable

outgoing segment from receiver

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

TCP sequence numbers, ACKs

Host A

Host B

User types 'C'

Seq=42, ACK=79, data = 'C'

host ACKs receipt
of 'C', echoes back 'C'

Seq=79, ACK=43, data = 'C'

host ACKs receipt
of echoed 'C'

Seq=43, ACK=80

simple telnet scenario

TCP round trip time, timeout

Q: how to set TCP timeout value?

- longer than RTT, but RTT varies!
- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss
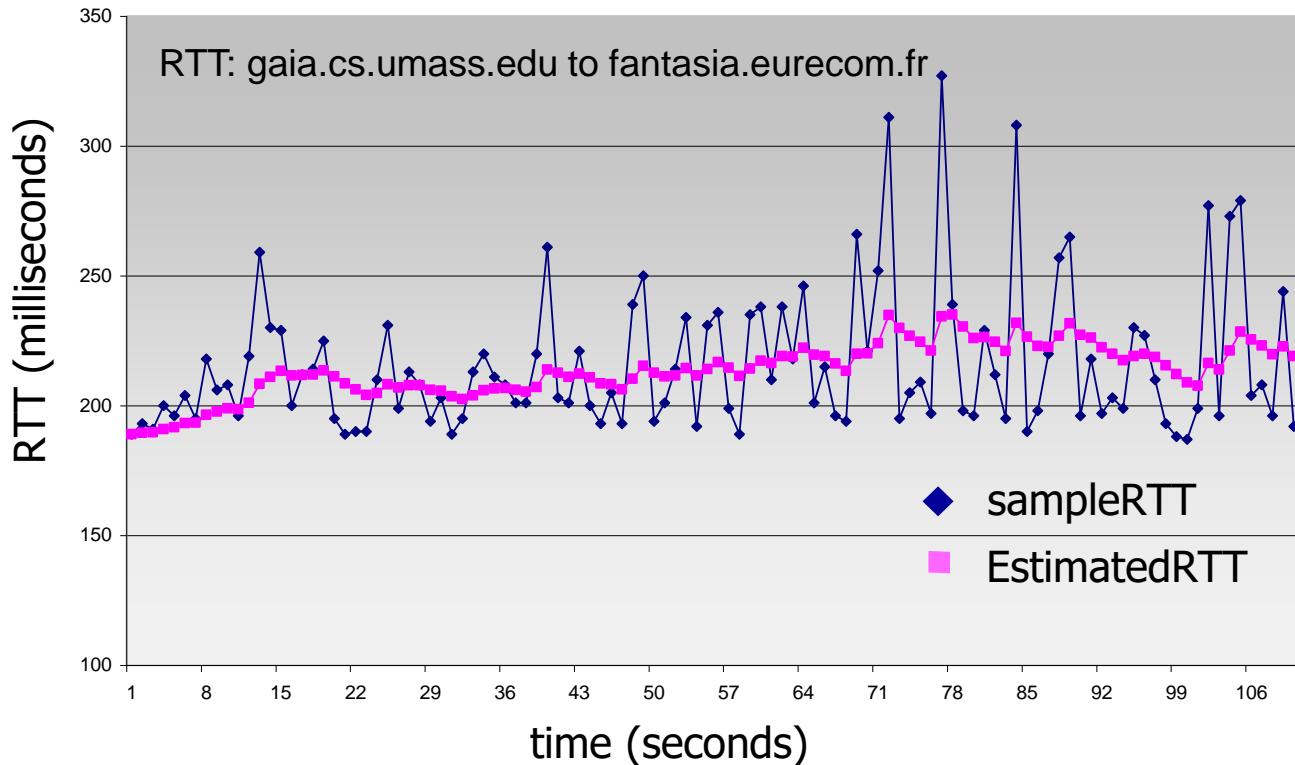
Q: how to estimate RTT?

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

TCP round trip time, timeout

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

- <u>e</u>xponential <u>w</u>eighted <u>m</u>oving <u>a</u>verage (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.125

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr

RTT (milliseconds)

time (seconds)

◆ sampleRTT

■ EstimatedRTT

TCP round trip time, timeout

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

TimeoutInterval = EstimatedRTT + 4*DevRTT

estimated RTT   "safety margin"

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

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

(typically, β = 0.25)

* Check out the online interactive exercises for more examples: 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`
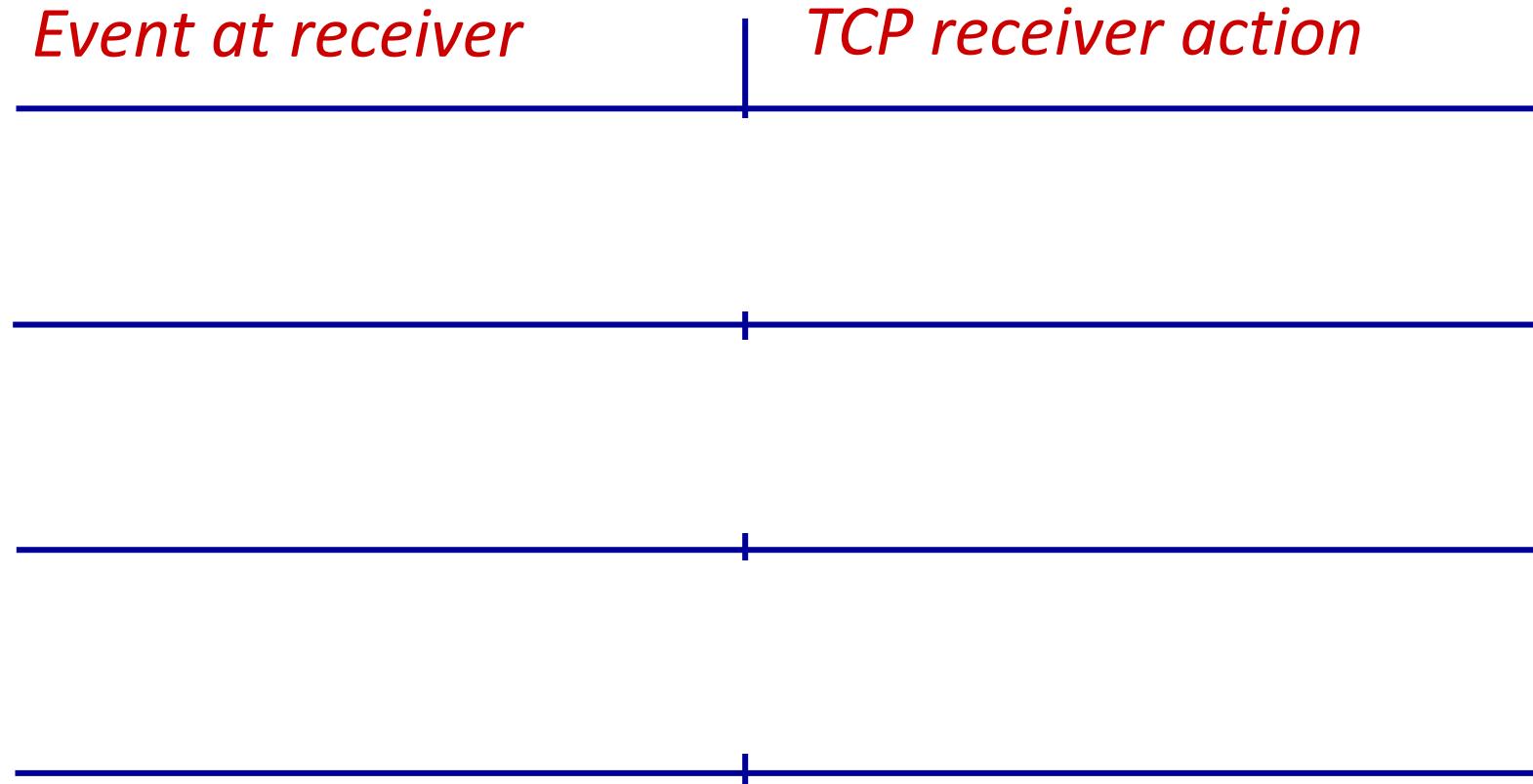
event: timeout

- retransmit segment that caused timeout
- restart timer
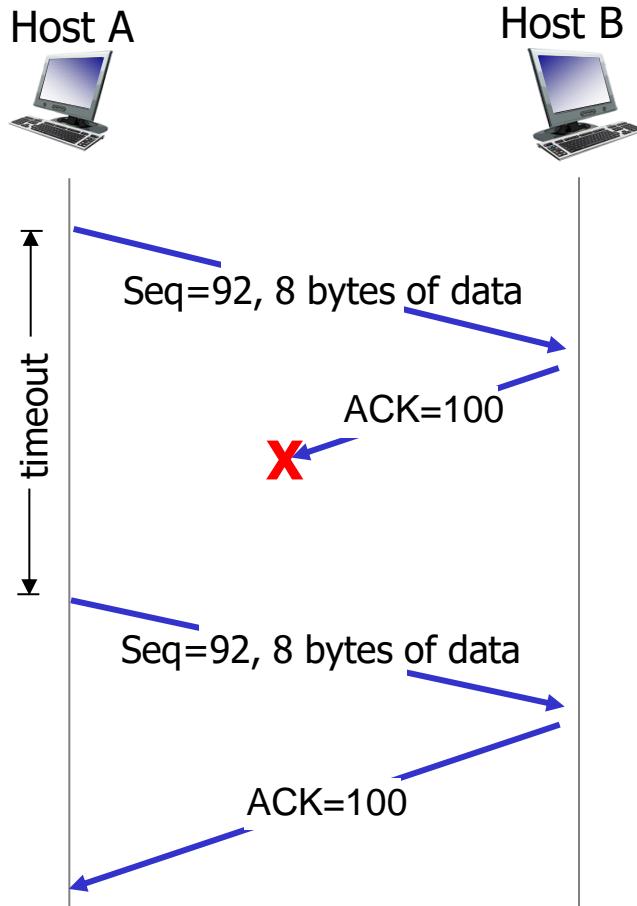
event: ACK received

- if ACK acknowledges previously unACKed segments
  - update what is known to be ACKed
  - start timer if there are still unACKed segments
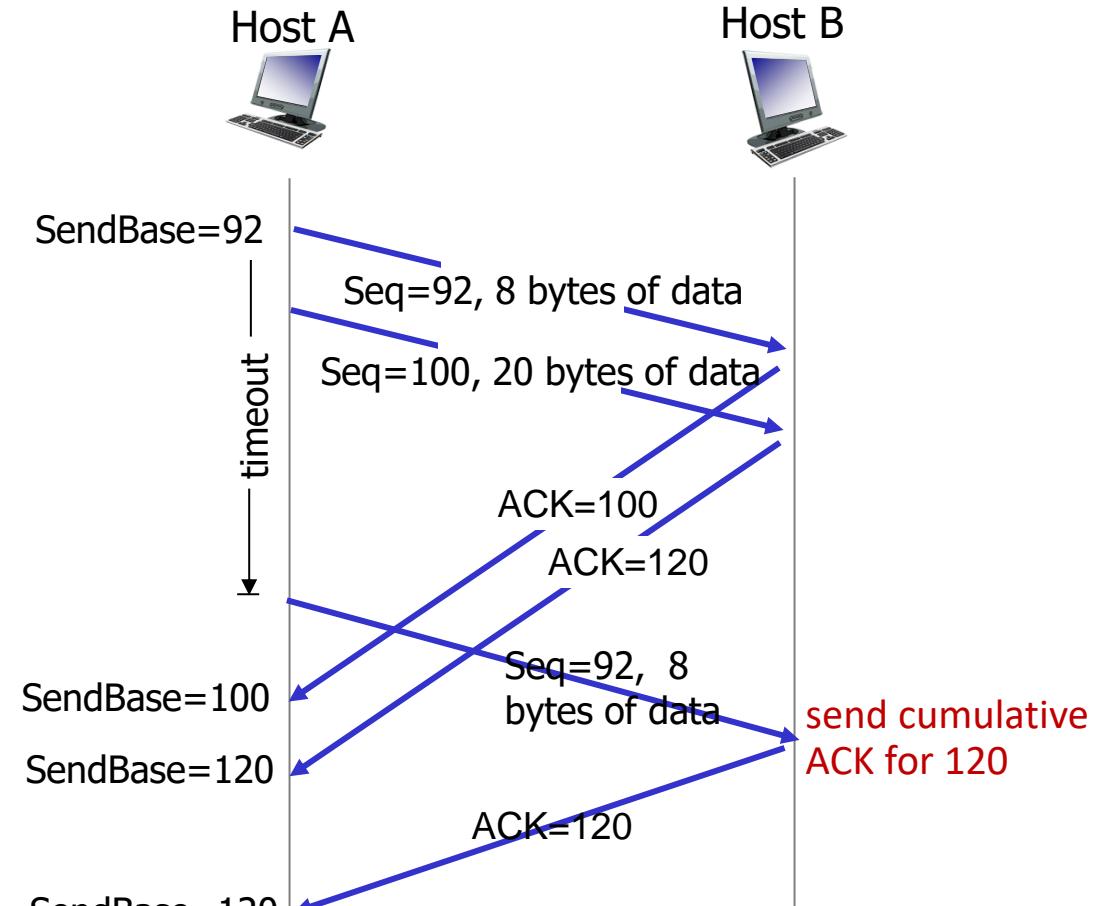
TCP Receiver: ACK generation [RFC 5681]

Event at receiver	TCP receiver action

TCP: retransmission scenarios

Host A   Host B

Seq=92, 8 bytes of data

ACK=100

X

timeout

Seq=92, 8 bytes of data

ACK=100

lost ACK scenario

Host A   Host B

SendBase=92

Seq=92, 8 bytes of data

Seq=100, 20 bytes of data

timeout

ACK=100

ACK=120

SendBase=100

Seq=92, 8 bytes of data

SendBase=120

send cumulative ACK for 120

ACK=120

SendBase=120
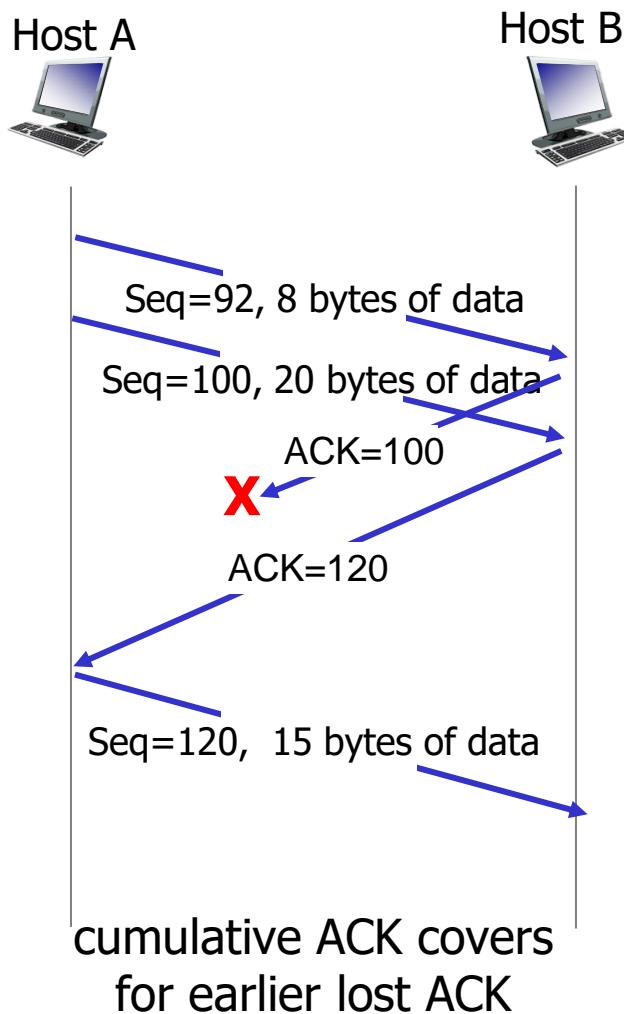
premature timeout

TCP: retransmission scenarios
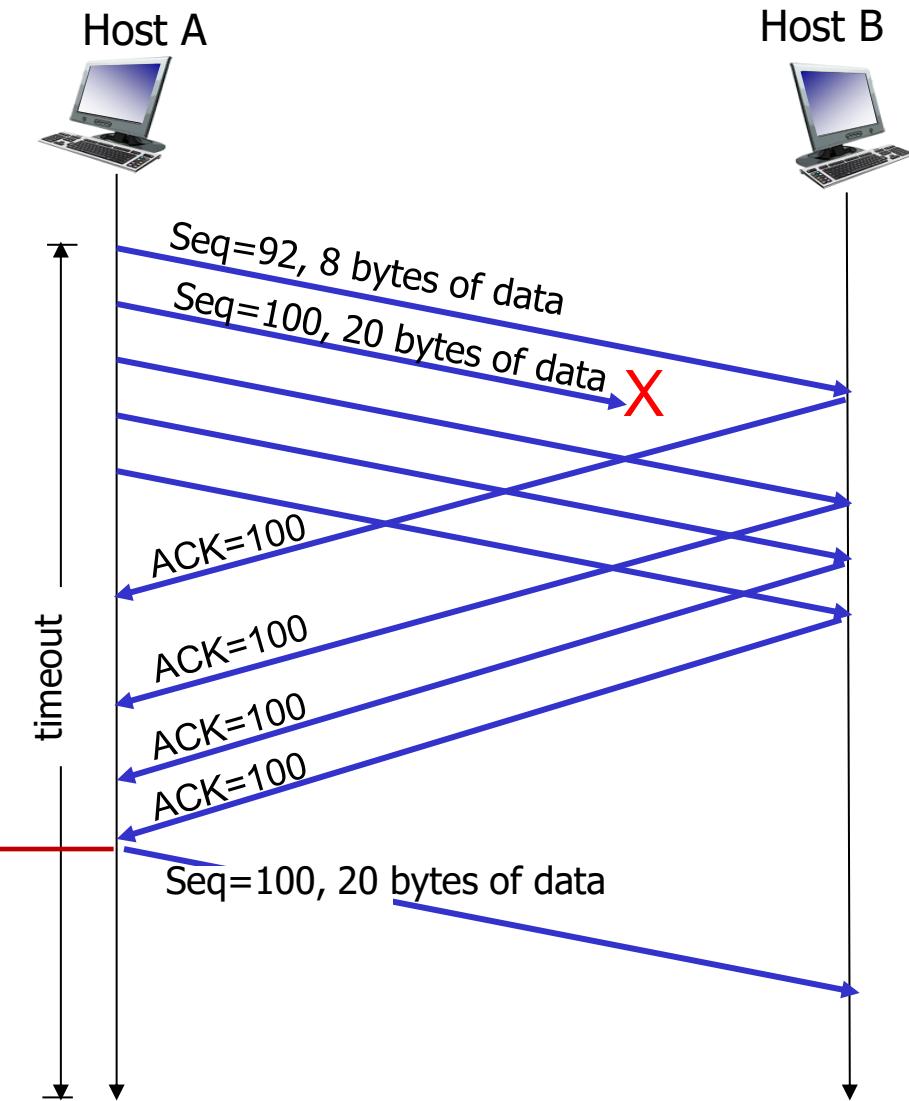
Host A   Host B

Seq=92, 8 bytes of data

Seq=100, 20 bytes of data

ACK=100

X

ACK=120

Seq=120, 15 bytes of data

cumulative ACK covers
for earlier lost ACK

TCP fast retransmit

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

Host A

Host B

Seq=92, 8 bytes of data

Seq=100, 20 bytes of data

X

ACK=100

ACK=100

ACK=100

ACK=100

Seq=100, 20 bytes of data

timeout

Chapter 3: roadmap

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

application process

TCP socket receiver buffers

TCP code

Network layer delivering IP datagram payload into TCP socket buffers

IP code

from sender

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

Network layer delivering IP datagram payload into TCP socket buffers

application process

TCP socket receiver buffers

TCP code

IP code

from sender

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

receive window
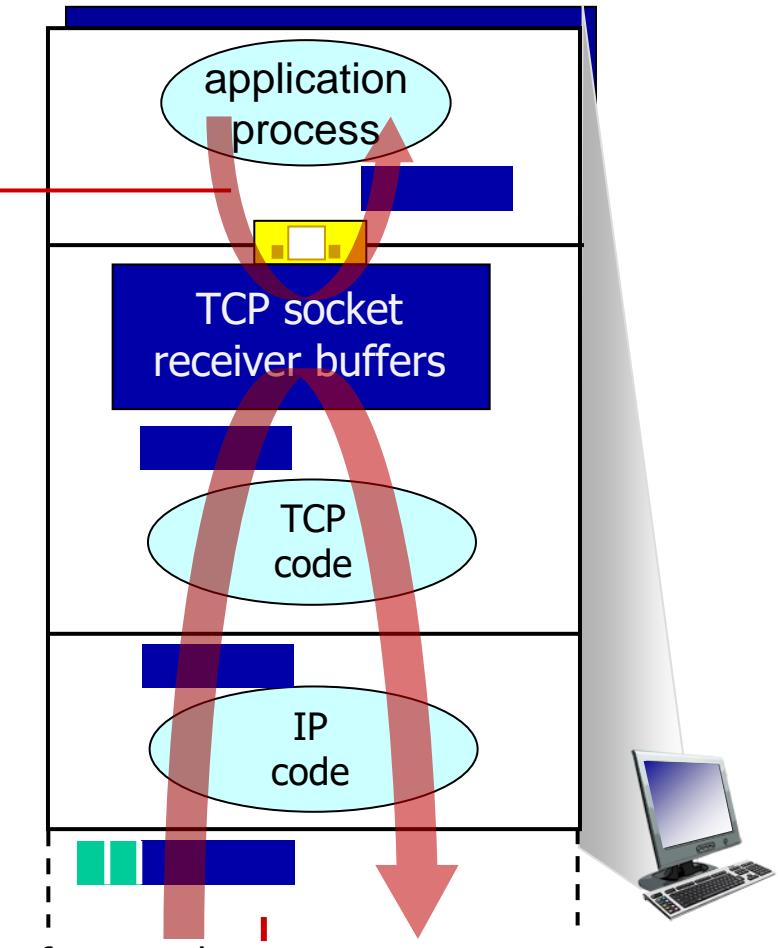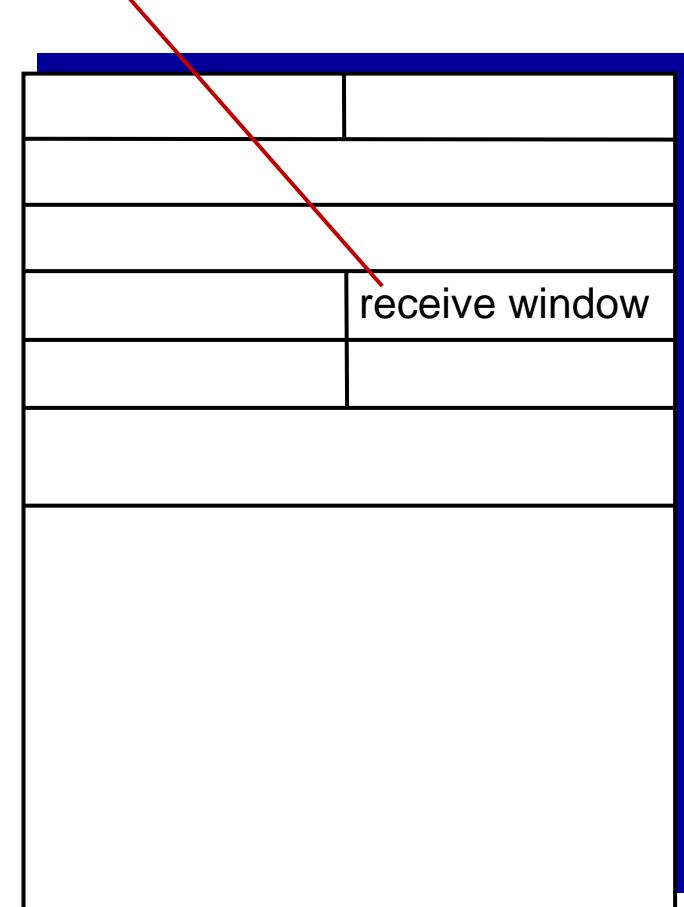
flow control: # bytes receiver willing to accept

application process

TCP socket receiver buffers

TCP code

IP code

from sender

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

application process

TCP socket receiver buffers

TCP code

IP code

from sender

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 auto-adjust **RcvBuffer**

- sender limits amount of unACKed ("in-flight") data to received **rwnd**

- guarantees receive buffer will not overflow

to application process

buffered data

RcvBuffer

rwnd

free buffer space

TCP segment payloads

TCP receiver-side buffering

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

flow control: # bytes receiver willing to accept

receive window
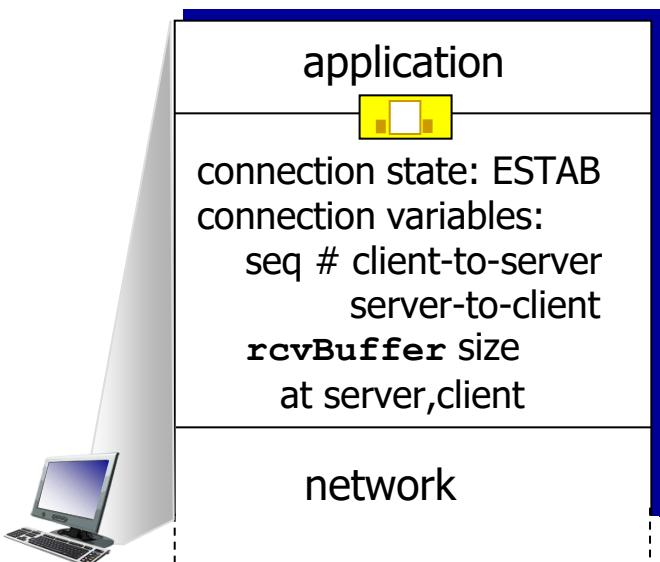
TCP segment format
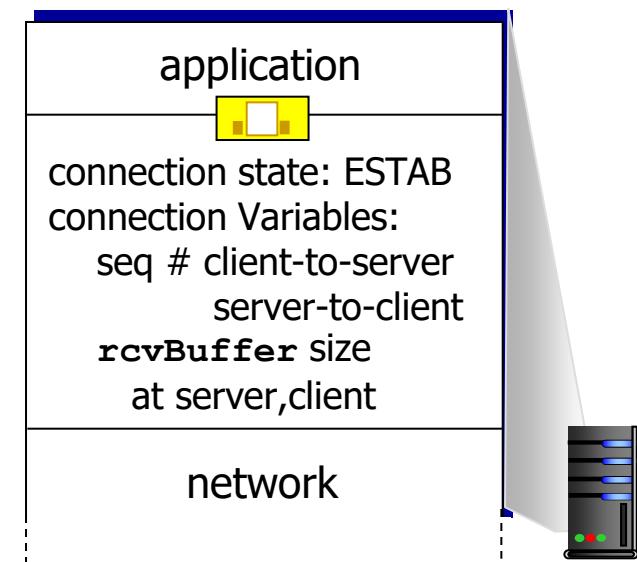
TCP connection management

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)

application

connection state: ESTAB
connection variables:
   seq # client-to-server
      server-to-client
rcvBuffer size
   at server,client

network

application

connection state: ESTAB
connection Variables:
   seq # client-to-server
      server-to-client
rcvBuffer size
   at server,client

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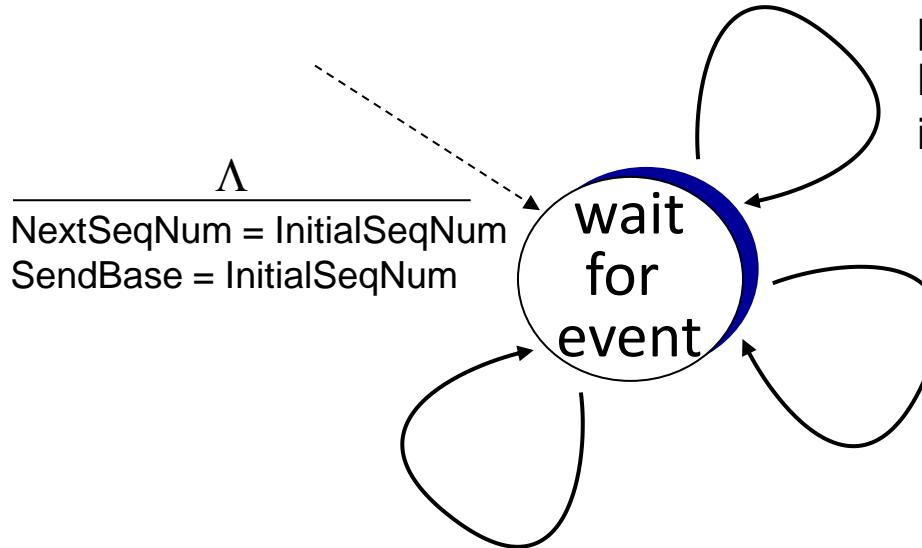
```
Socket clientSocket =
   newSocket("hostname","port number");
```

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

TCP sender (simplified)

Λ
—————————
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

wait for event

data received from application above
———————————————————————
create segment, seq. #: NextSeqNum
pass segment to IP (i.e., "send")
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
   start timer

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

ACK received, with ACK field value y
———————————————————————
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
     start timer
    else stop timer
   }
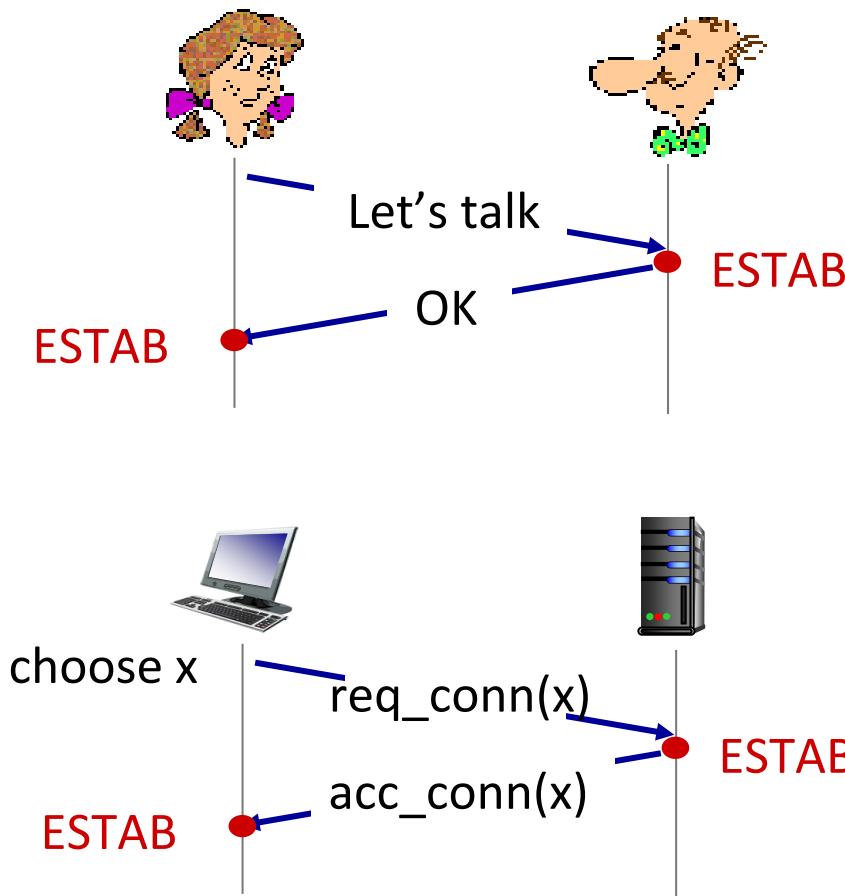
Agreeing to establish a connection
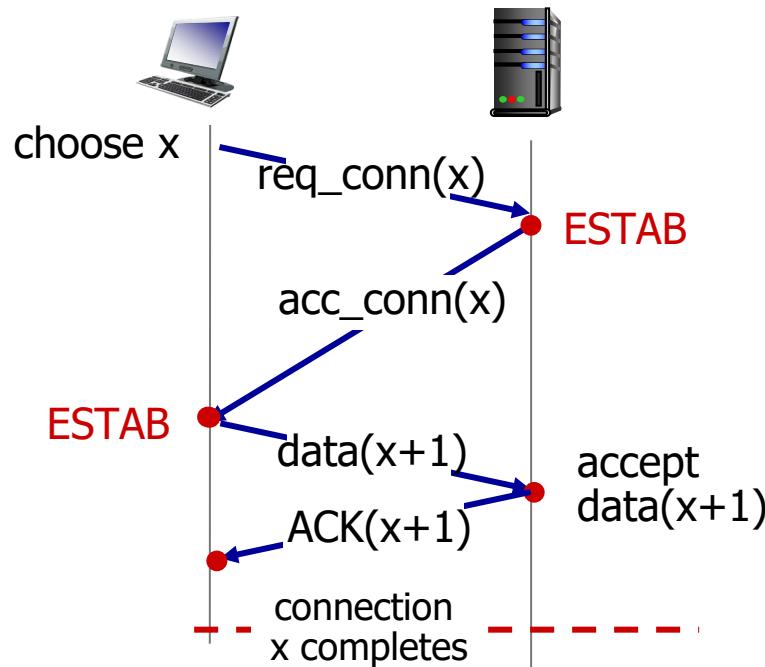
2-way handshake:

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

choose x

req_conn(x)

ESTAB

acc_conn(x)

ESTAB

data(x+1)

accept
data(x+1)

ACK(x+1)
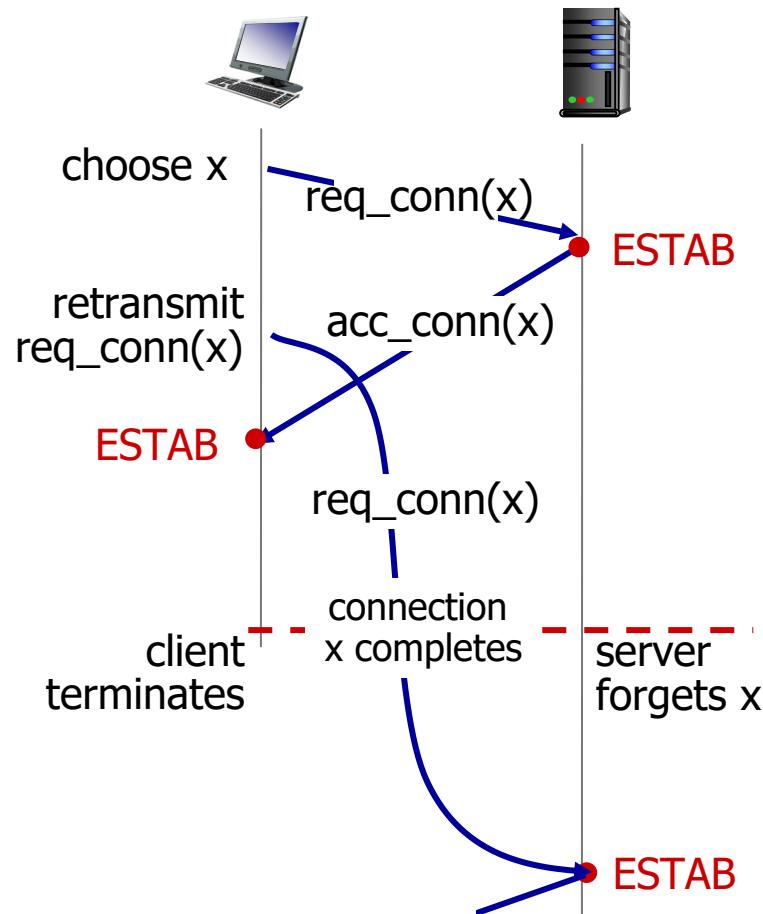
connection
x completes

No problem!

2-way handshake scenarios

choose x

req_conn(x) → ESTAB

retransmit
req_conn(x)

acc_conn(x)

ESTAB

req_conn(x)

connection
x completes

client
terminates

server
forgets x

ESTAB
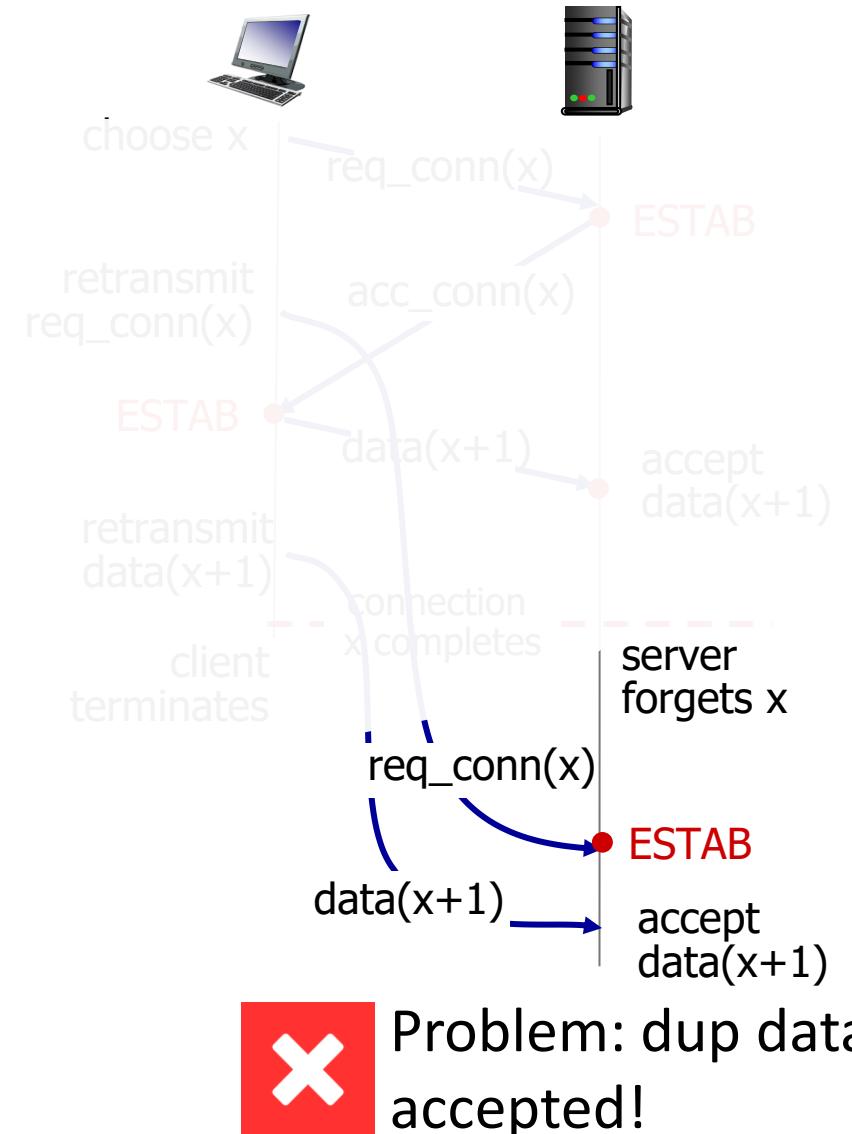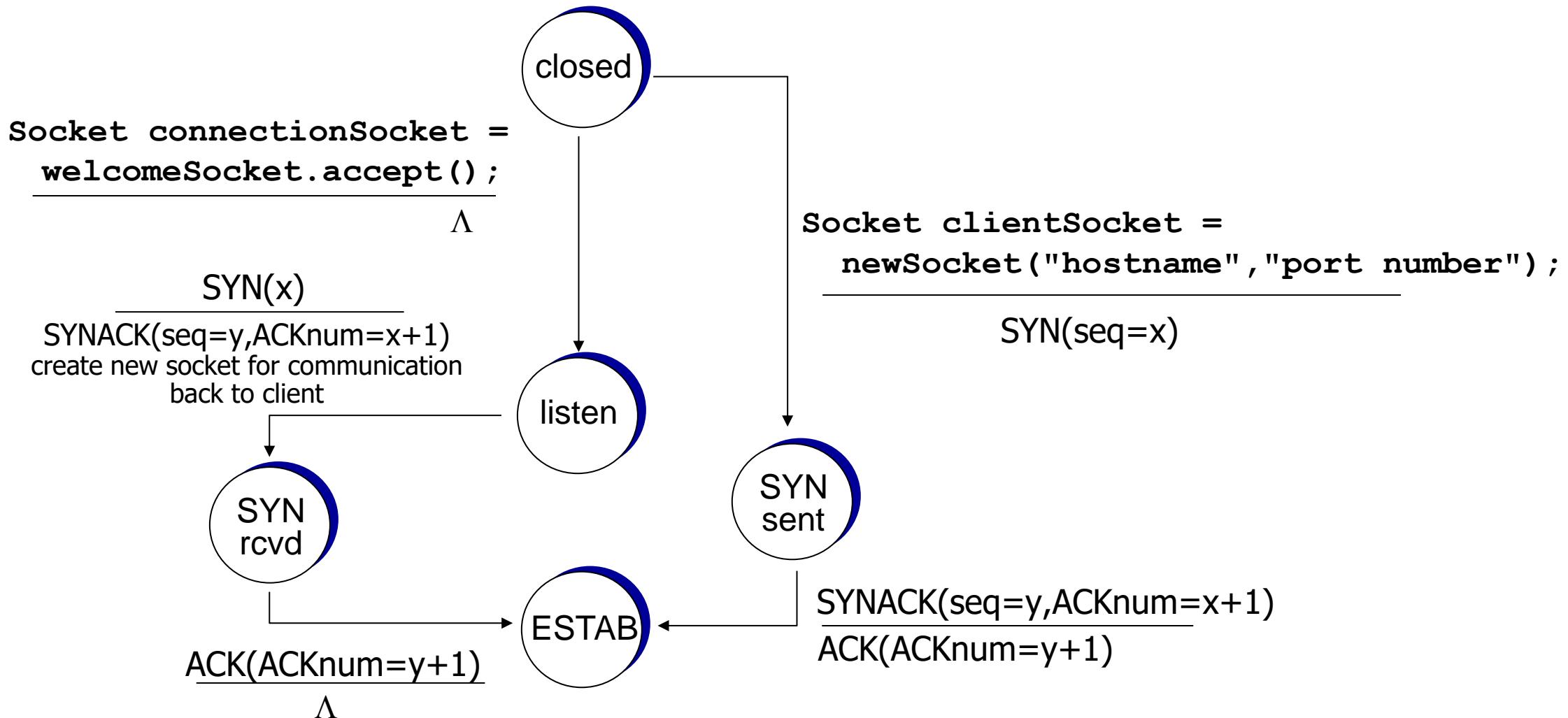
❌ Problem: half open
connection! (no client)

2-way handshake scenarios

choose x

req_conn(x)

ESTAB

retransmit
req_conn(x)

acc_conn(x)

ESTAB

data(x+1)

accept
data(x+1)

retransmit
data(x+1)

connection
x completes

client
terminates

server
forgets x

req_conn(x)

ESTAB

data(x+1)

accept
data(x+1)

❌ Problem: dup data accepted!

TCP 3-way handshake FSM

closed

Socket connectionSocket =
welcomeSocket.accept();
$\overline{}$
Λ

Socket clientSocket =
newSocket("hostname","port number");
$\overline{}$
SYN(seq=x)

$\dfrac{\text{SYN(x)}}{\begin{array}{c}\text{SYNACK(seq=y,ACKnum=x+1)}\\\text{create new socket for communication}\\\text{back to client}\end{array}}$

listen

SYN
rcvd

SYN
sent

$\dfrac{\text{ACK(ACKnum=y+1)}}{\Lambda}$

ESTAB

$\dfrac{\text{SYNACK(seq=y,ACKnum=x+1)}}{\text{ACK(ACKnum=y+1)}}$

TCP 3-way handshake

Server state

Client state

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

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

LISTEN

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

LISTEN

choose init seq num, x
send TCP SYN msg

SYNSENT

SYNbit=1, Seq=x

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

SYN RCVD

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

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

ESTAB

ACKbit=1, ACKnum=y+1

received ACK(y)
indicates client is live

ESTAB
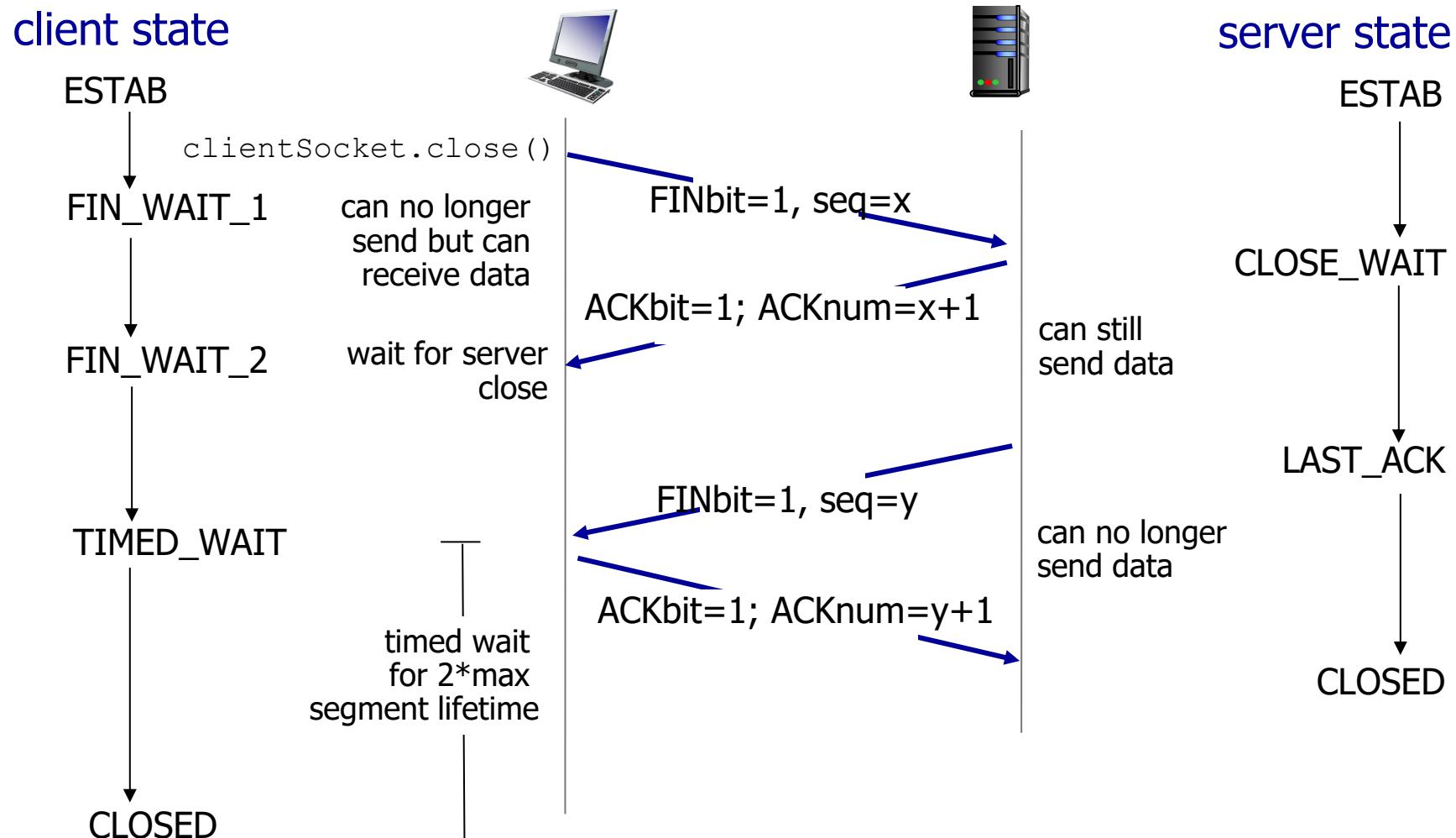
A human 3-way handshake protocol

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

Closing a TCP connection

client state

ESTAB

`clientSocket.close()`

FIN_WAIT_1 — can no longer send but can receive data

FIN_WAIT_2 — wait for server close

TIMED_WAIT

timed wait for 2*max segment lifetime

CLOSED

FINbit=1, seq=x

ACKbit=1; ACKnum=x+1

FINbit=1, seq=y

ACKbit=1; ACKnum=y+1

server state

ESTAB

CLOSE_WAIT — can still send data

LAST_ACK — can no longer send data

CLOSED

Chapter 3: roadmap

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!
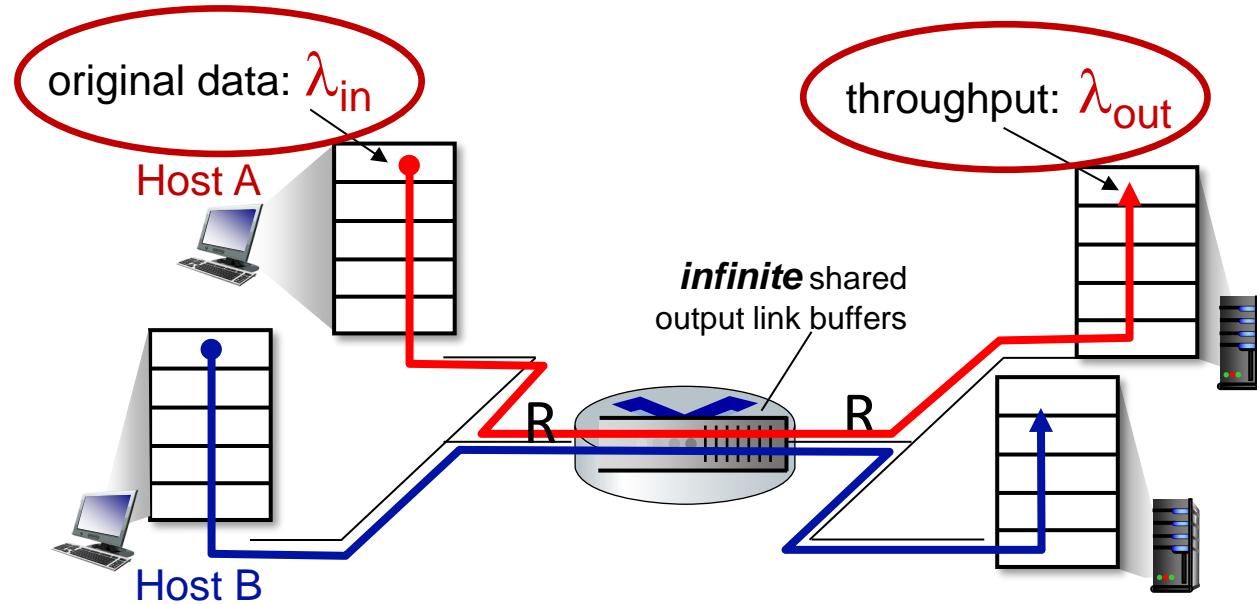
congestion control:
too many senders,
sending too fast

flow control: one sender
too fast for one receiver
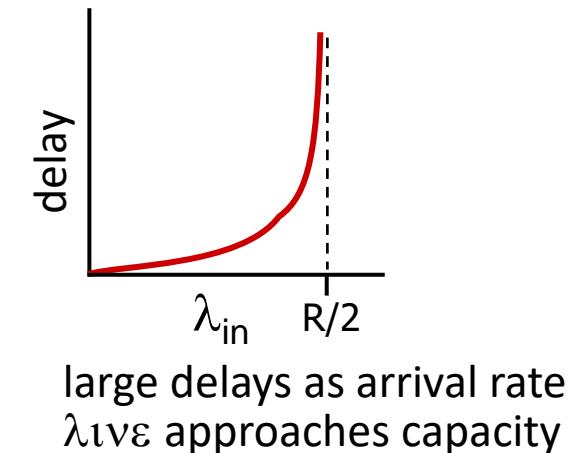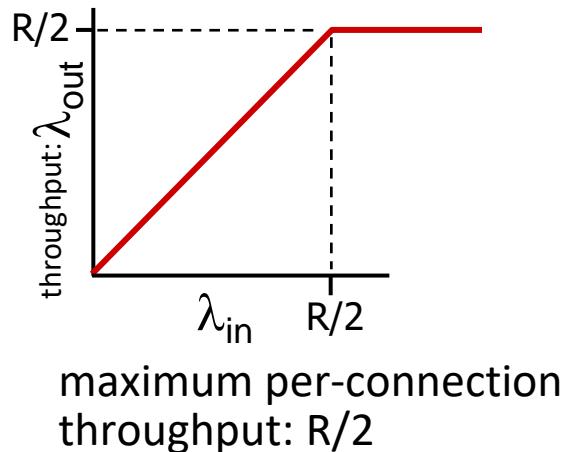
Causes/costs of congestion: scenario 1

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

original data: λ_{in}

Host A

throughput: λ_{out}

infinite shared output link buffers

R   R

Host B
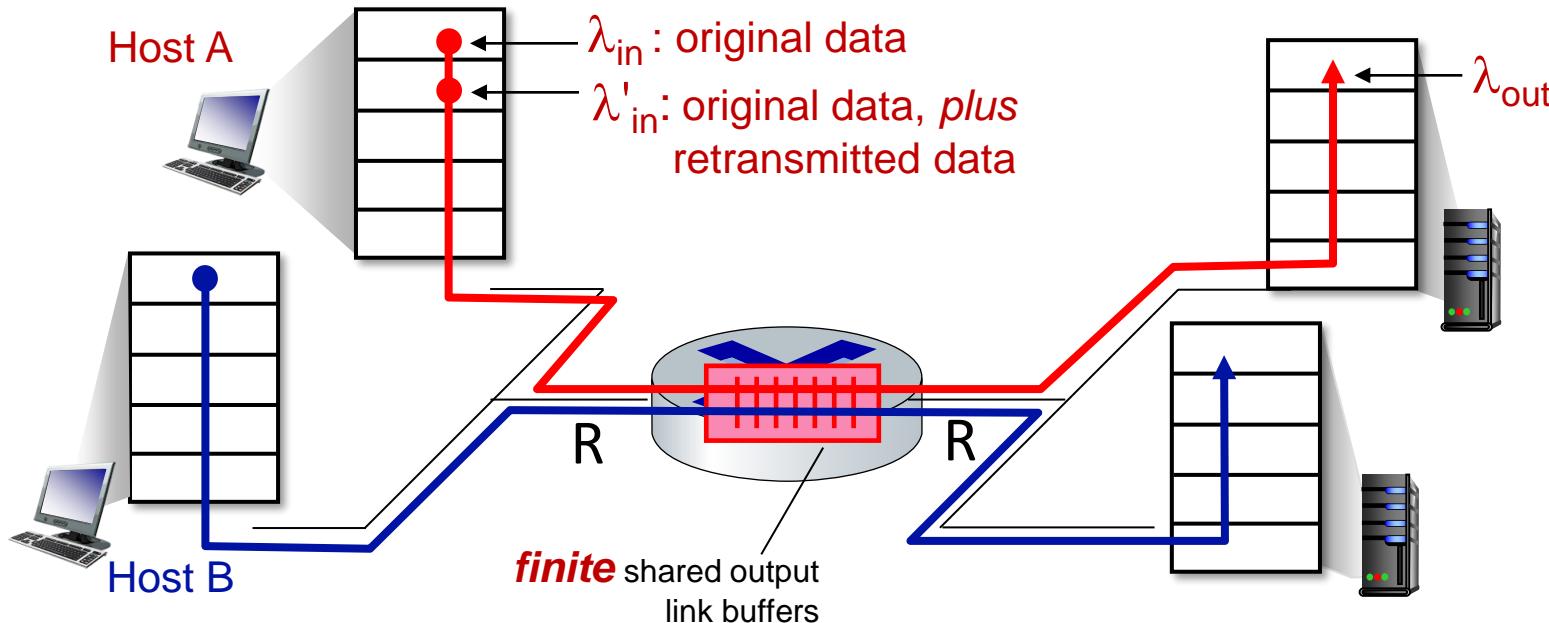
Q: What happens as arrival rate λ_{in} approaches R/2?

throughput: λ_{out} — R/2

λ_{in} R/2

maximum per-connection throughput: R/2

delay

λ_{in} R/2

large delays as arrival rate $\lambda\iota\nu\varepsilon$ approaches capacity

Causes/costs of congestion: scenario 2
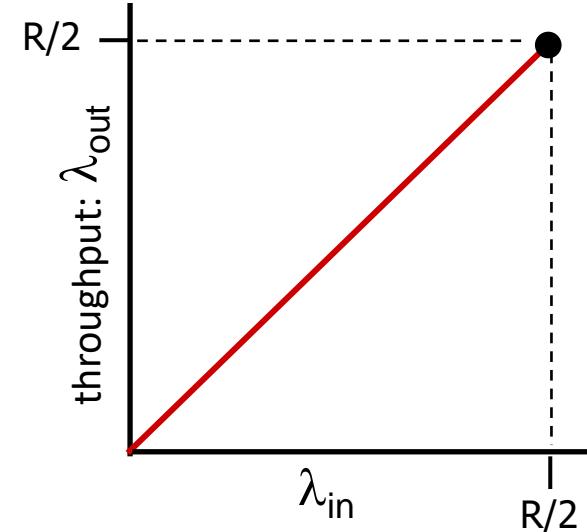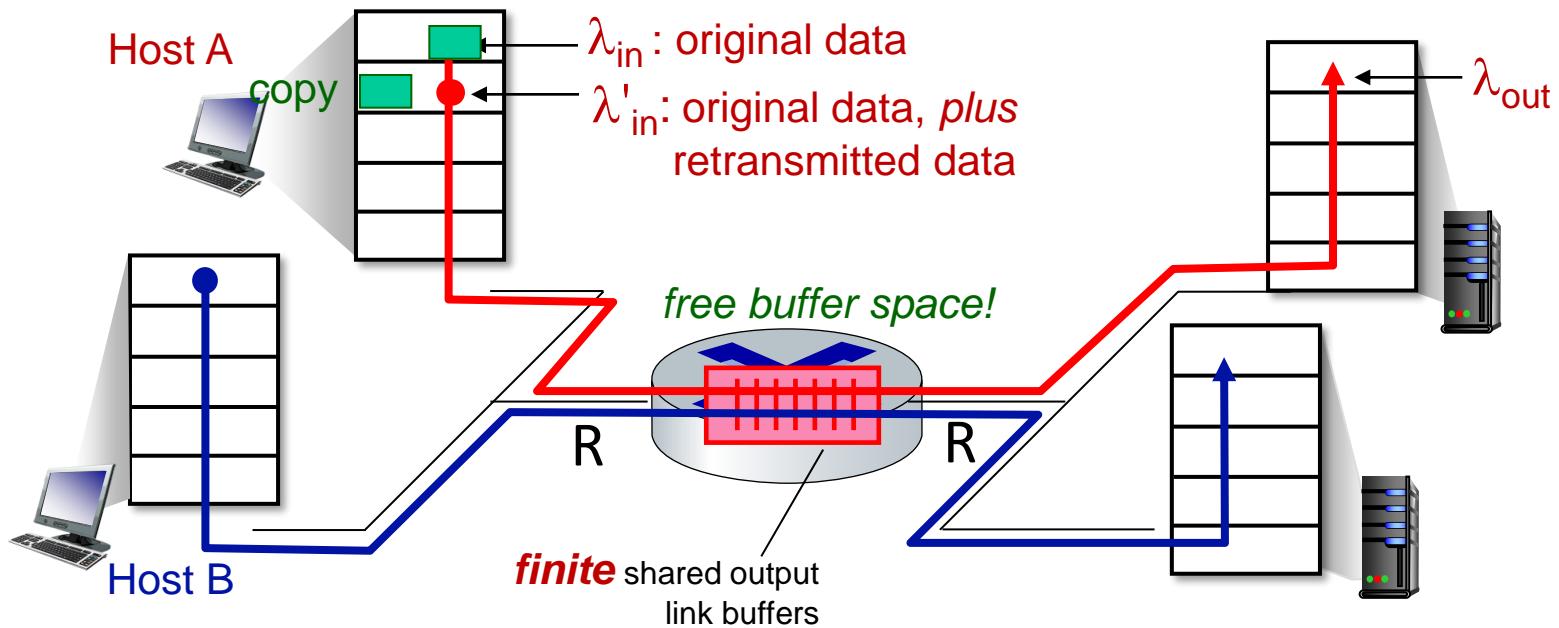
- one router, *finite* buffers

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

Host A

λ_{in} : original data

λ'_{in}: original data, *plus* retransmitted data

λ_{out}

Host B

finite shared output link buffers

Causes/costs of congestion: scenario 2

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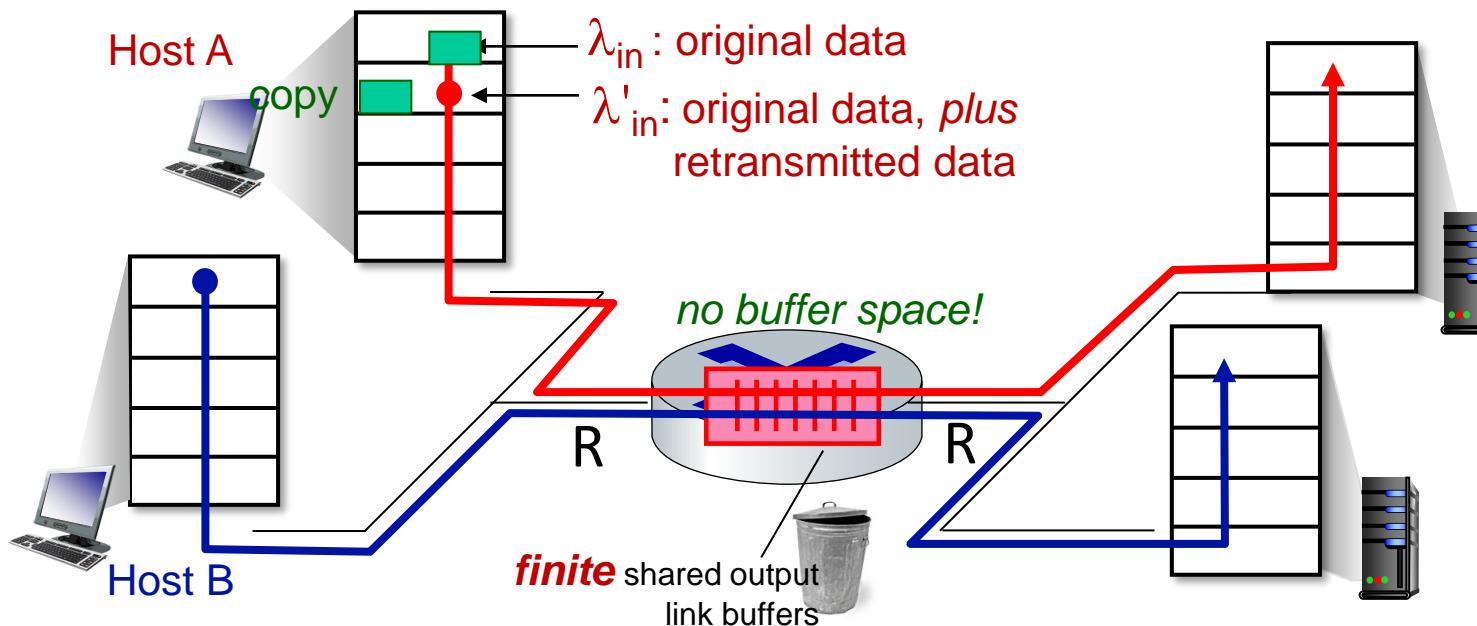
- sender sends only when router buffers available

λ_{in} : original data

λ'_{in}: original data, *plus* retransmitted data

Host A

copy

free buffer space!

λ_{out}

R

R

finite shared output link buffers

Host B

R/2

throughput: λ_{out}

λ_{in}

R/2

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

Host A

copy

λ_{in} : original data

λ'_{in}: original data, *plus* retransmitted data

Host B

no buffer space!

R   R

finite shared output link buffers

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

throughput: λ_{out}

R/2

"wasted" capacity due to retransmissions

when sending at R/2, some packets are needed retransmissions

λ'_{in} R/2

Host A

λ_{in} : original data

λ'_{in}: original data, *plus* retransmitted data

free buffer space!

R   R

Host B

finite shared output link buffers

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 times can time out prematurely, sending *two* copies, *both* of which are delivered

λ_{in} : original data

λ'_{in}: original data, *plus* retransmitted data

timeout

Host

Host B

free buffer space!

R   R

finite shared output link buffers

throughput: λ_{out}

R/2

λ'_{in} R/2

"wasted" capacity due to un-needed retransmissions

when sending at R/2, some packets are retransmissions, including needed and *un-needed* duplicates, that 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 times can time out prematurely, sending *two* copies, *both* of which are delivered

R/2

throughput: λ_{out}

"wasted" capacity due to un-needed retransmissions

λ'_{in} R/2

when sending at R/2, some packets are retransmissions, including needed and *un-needed* duplicates, that 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
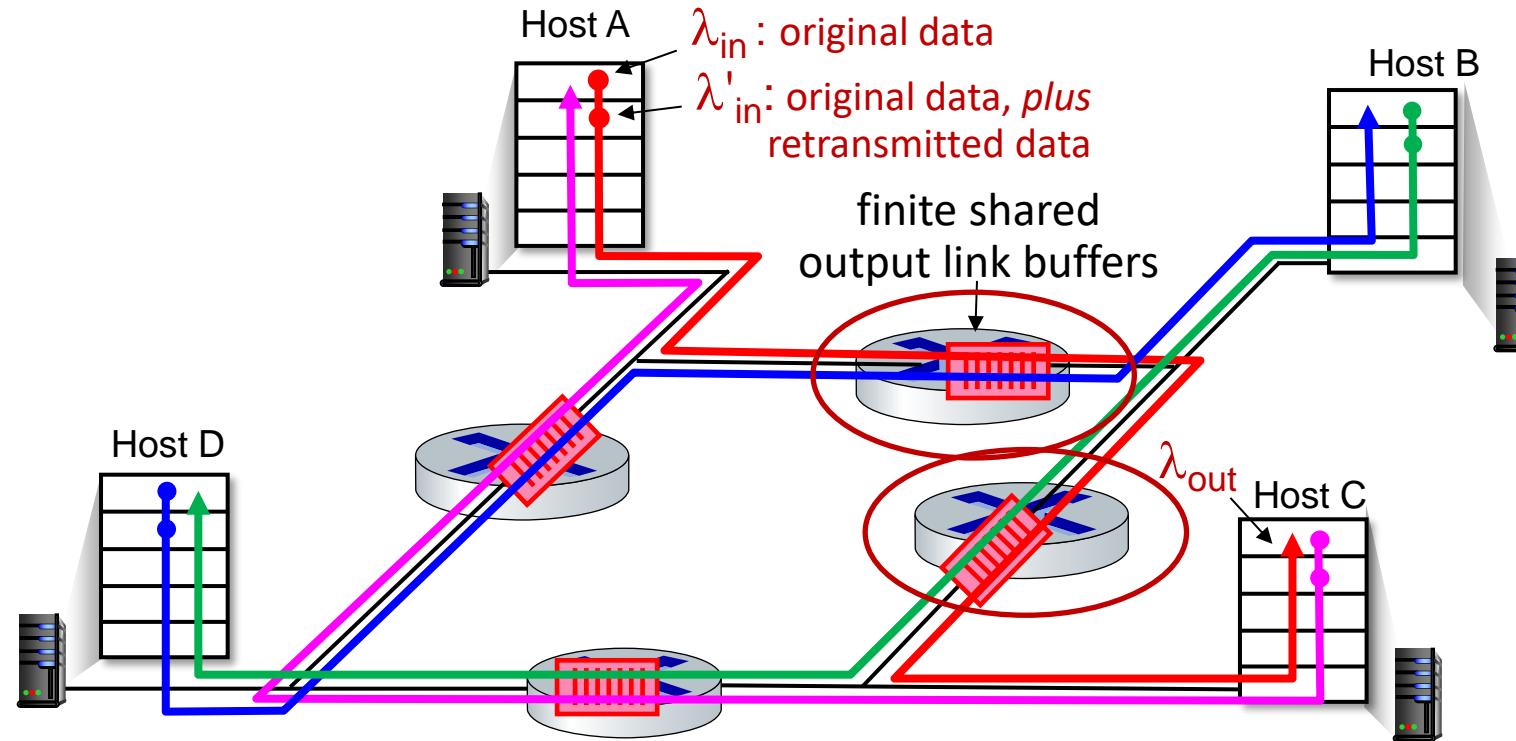
Causes/costs of congestion: scenario 3
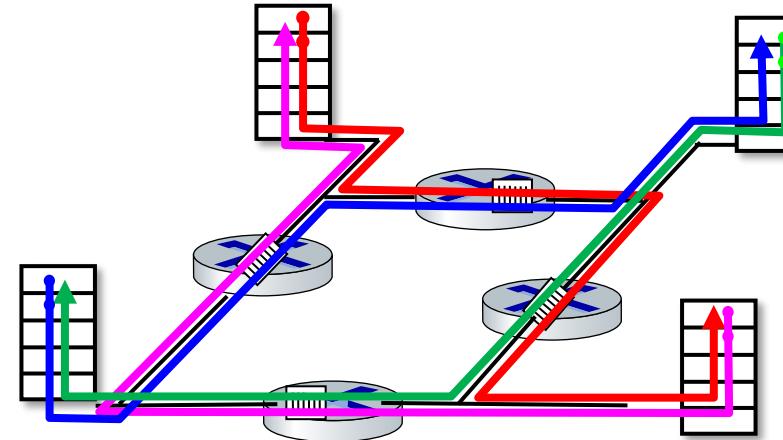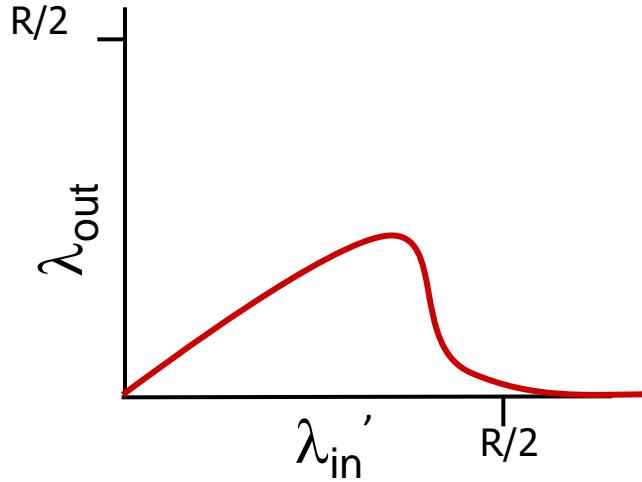
- *four* senders
- *multi-hop* paths
- timeout/retransmit

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

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

Host A

λ_{in} : original data

λ'_{in} : original data, *plus* retransmitted data

finite shared output link buffers

Host B

Host D

λ_{out}

Host C

Causes/costs of congestion: scenario 3
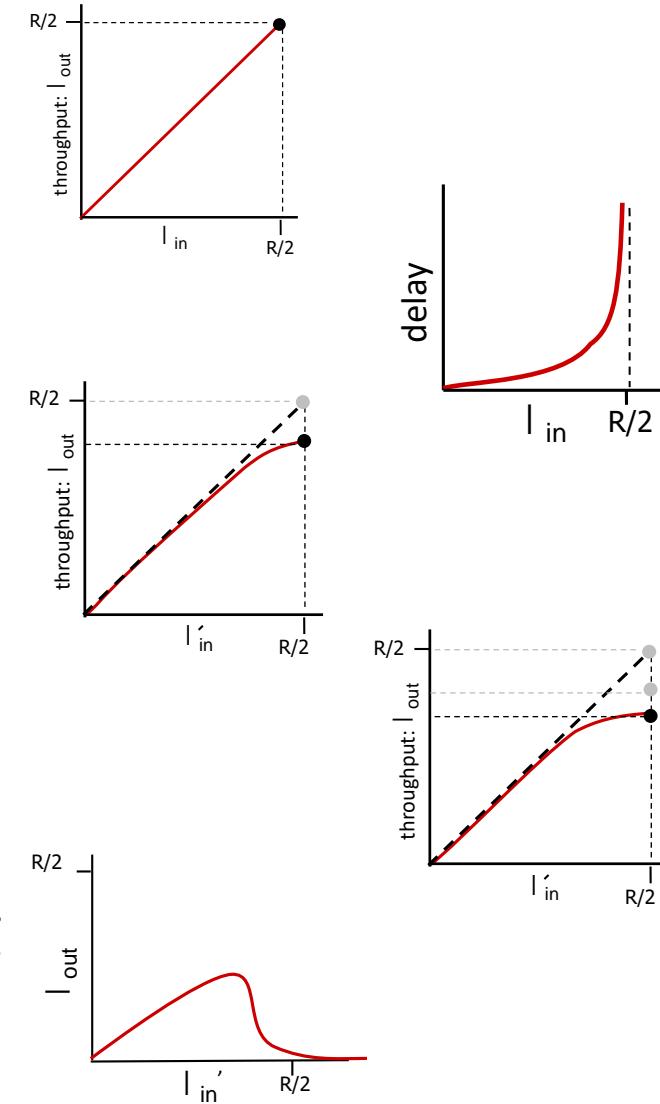
another "cost" of congestion:

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

Causes/costs of congestion: insights
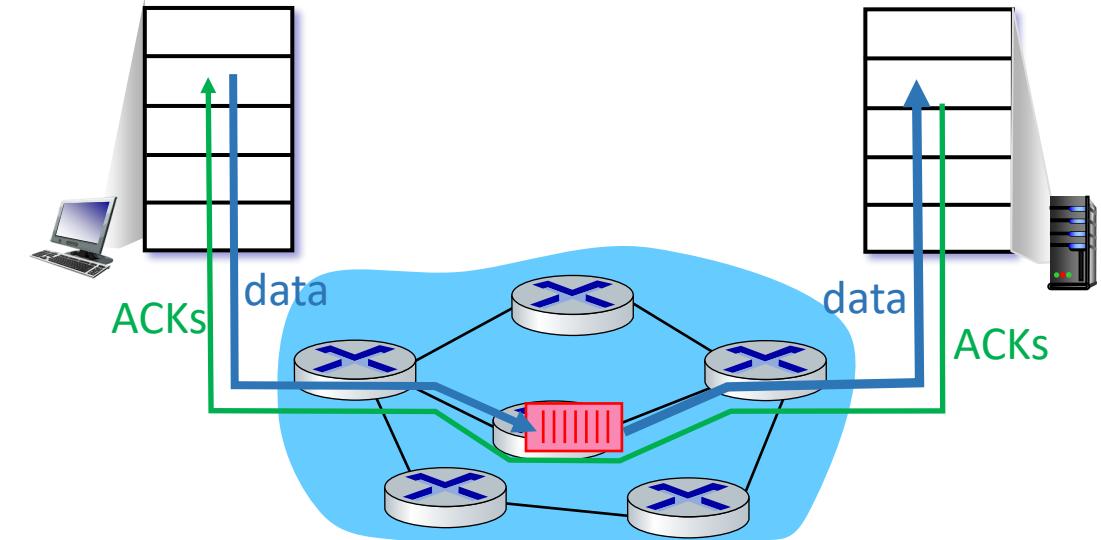
- throughput can never exceed capacity

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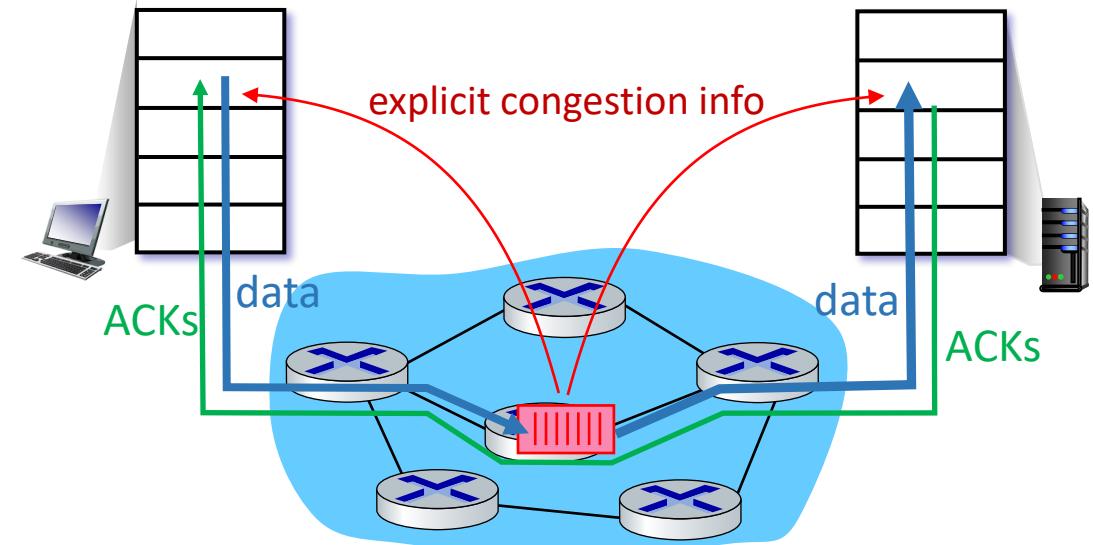
- no explicit feedback from network
- congestion *inferred* from observed loss, delay
- approach taken by TCP

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

- TCP ECN, ATM, DECbit protocols

explicit congestion info

data

ACKs

data

ACKs

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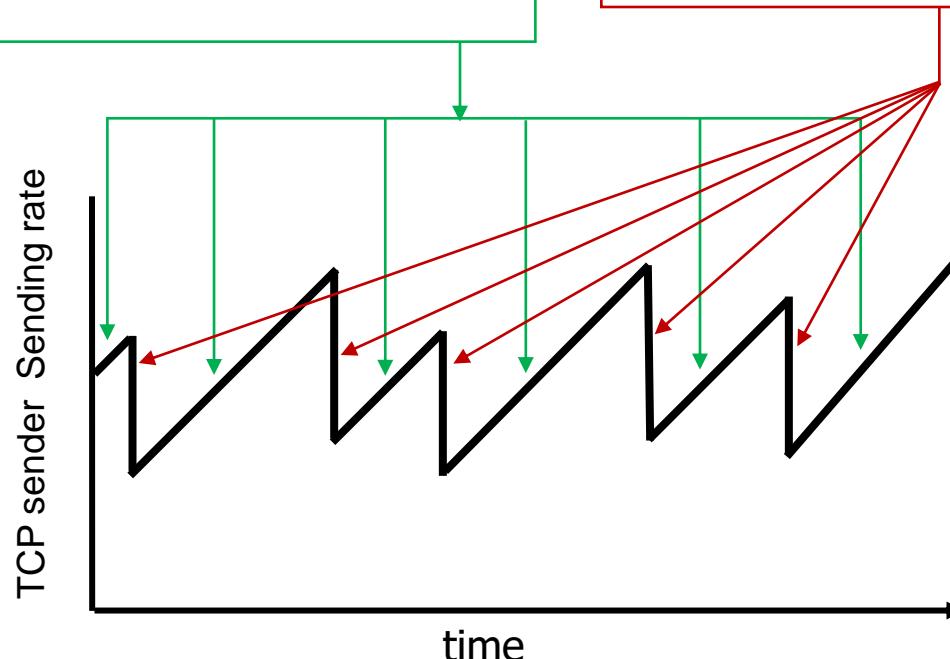
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

Additive Increase

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

Multiplicative Decrease

cut sending rate in half at each loss event

TCP sender Sending rate

time

AIMD sawtooth behavior: *probing* for bandwidth

TCP AIMD: more
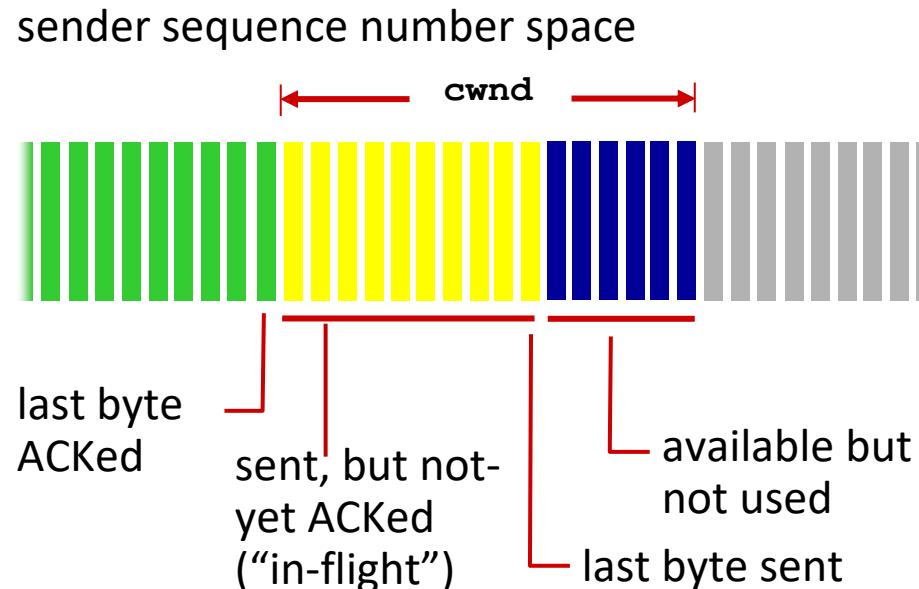
Multiplicative decrease detail: sending rate is
- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

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

sender sequence number space

last byte
ACKed

sent, but not-
yet ACKed
("in-flight")

last byte sent

available but
not used

TCP sending behavior:
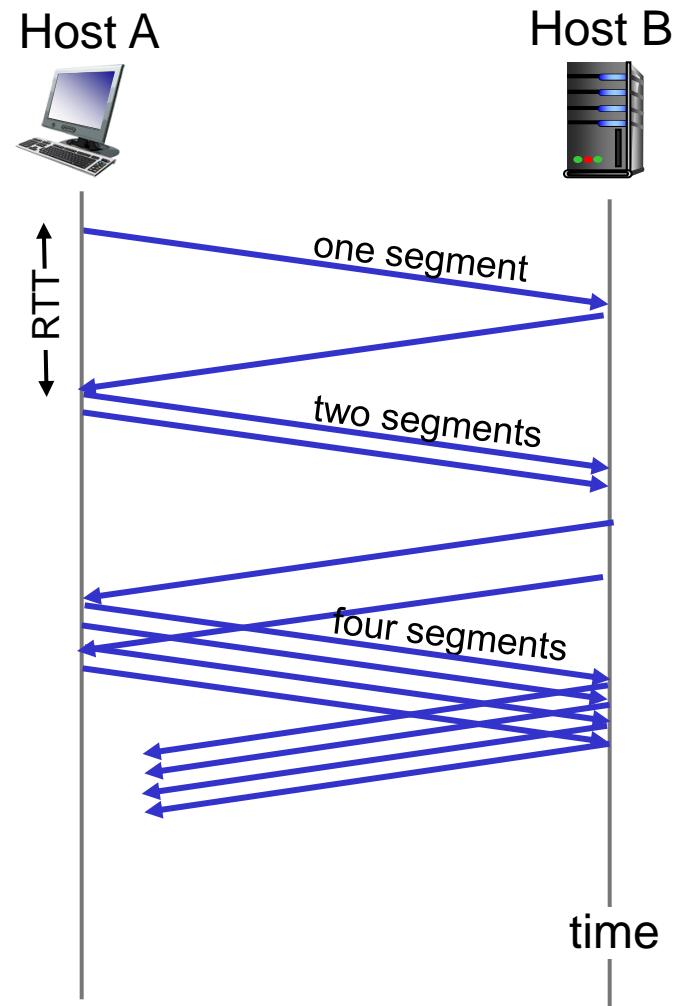
- *roughly:* send `cwnd` bytes, wait RTT for ACKS, then send more bytes

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

- TCP sender limits transmission: `LastByteSent- LastByteAcked <  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
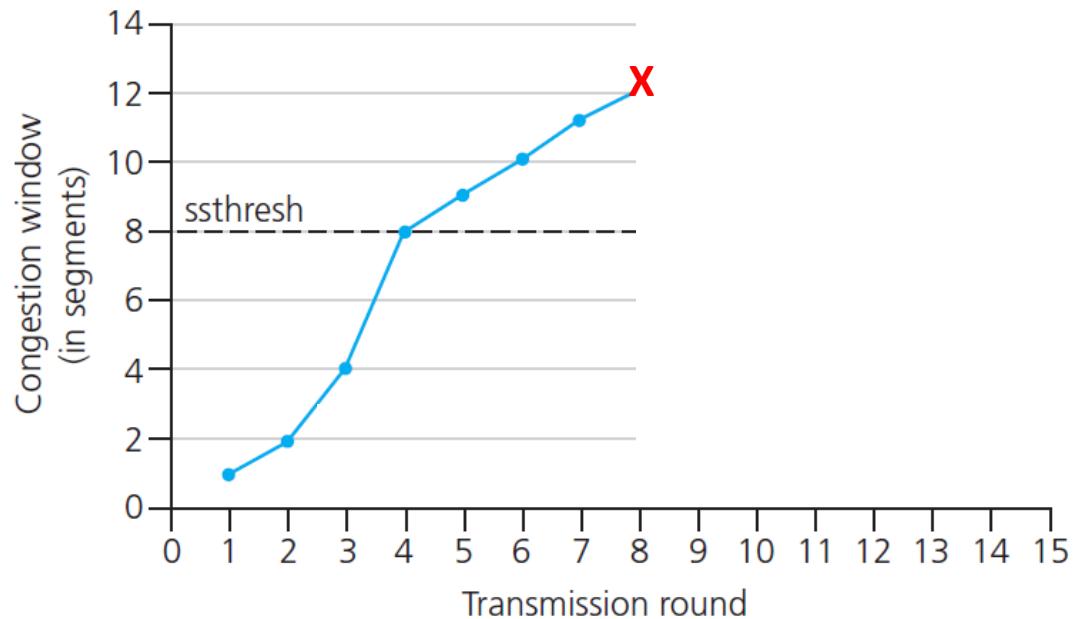
TCP: from slow start to congestion avoidance

Q: when should the exponential increase switch to linear?

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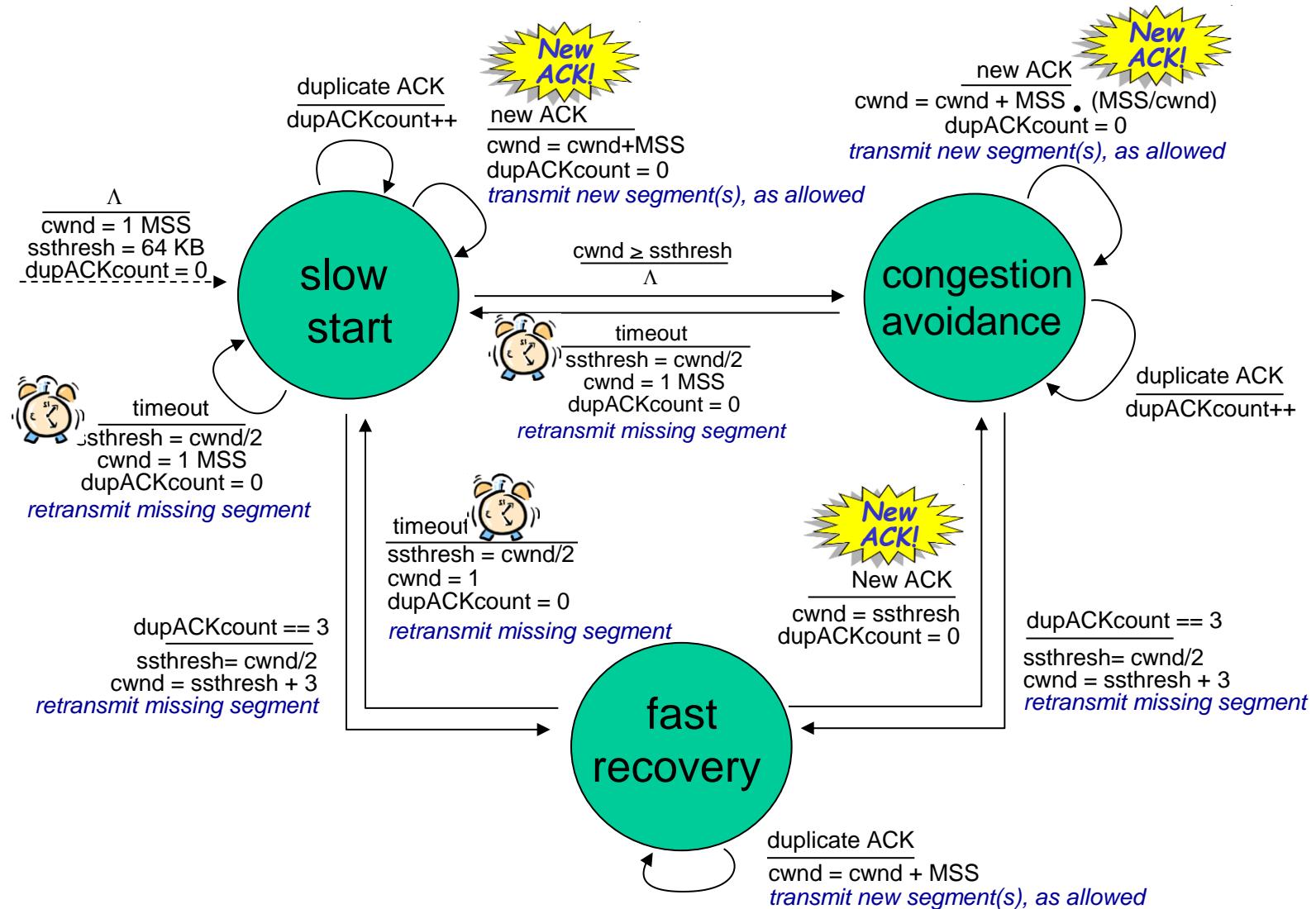
Implementation:

- variable `ssthresh`

- on loss event, `ssthresh` is set to 1/2 of `cwnd` just before loss event
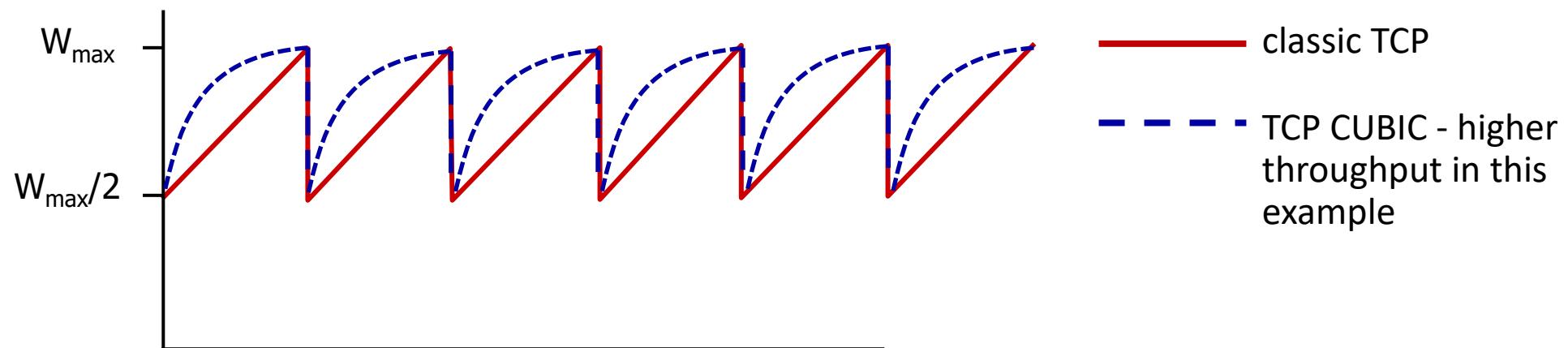
* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Summary: TCP congestion control
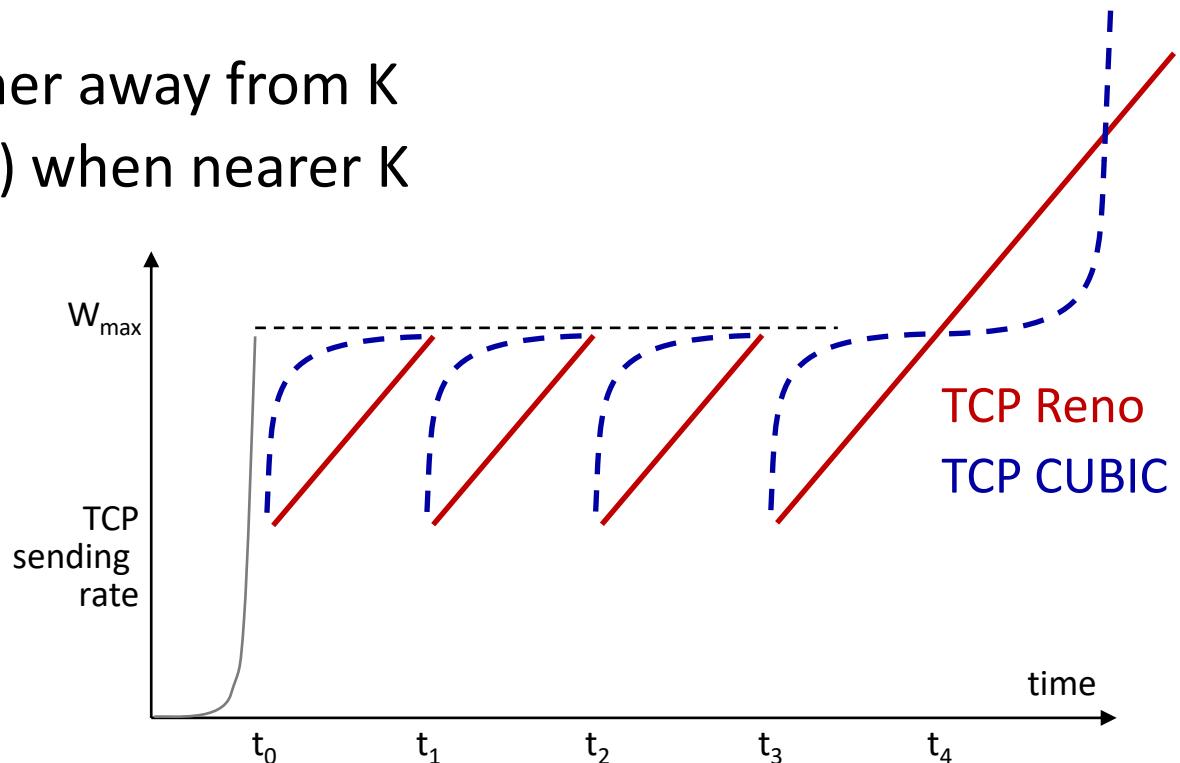
TCP CUBIC

- 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*


classic TCP

TCP CUBIC - higher throughput in this example
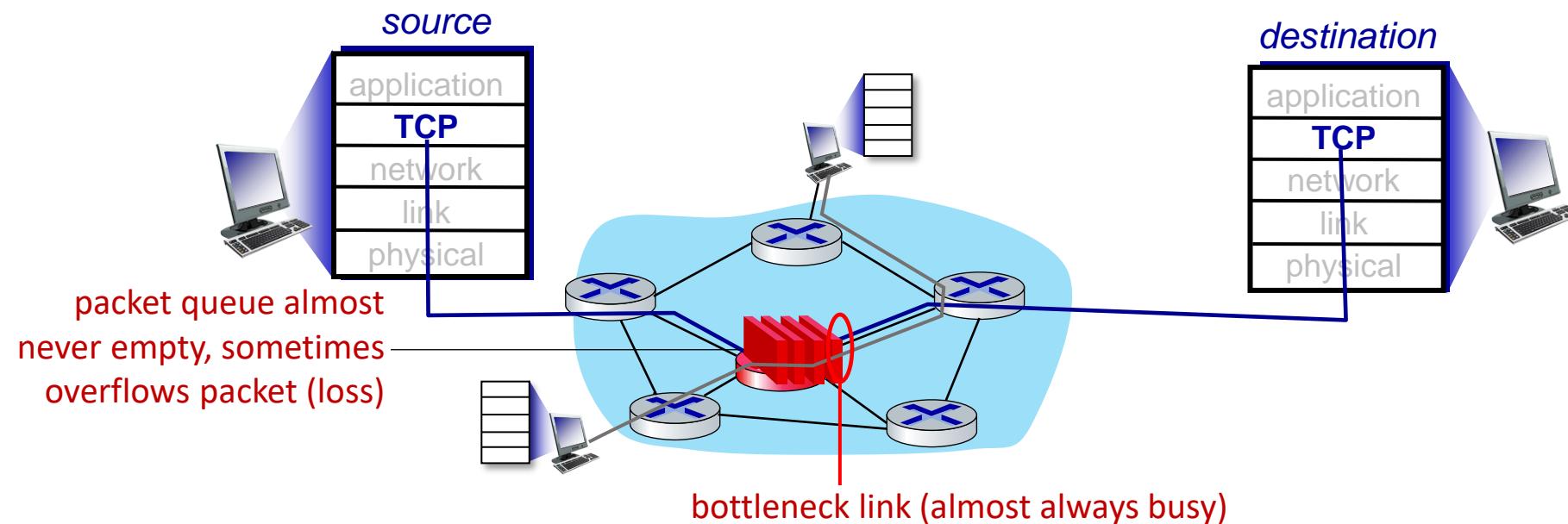
TCP CUBIC

- 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
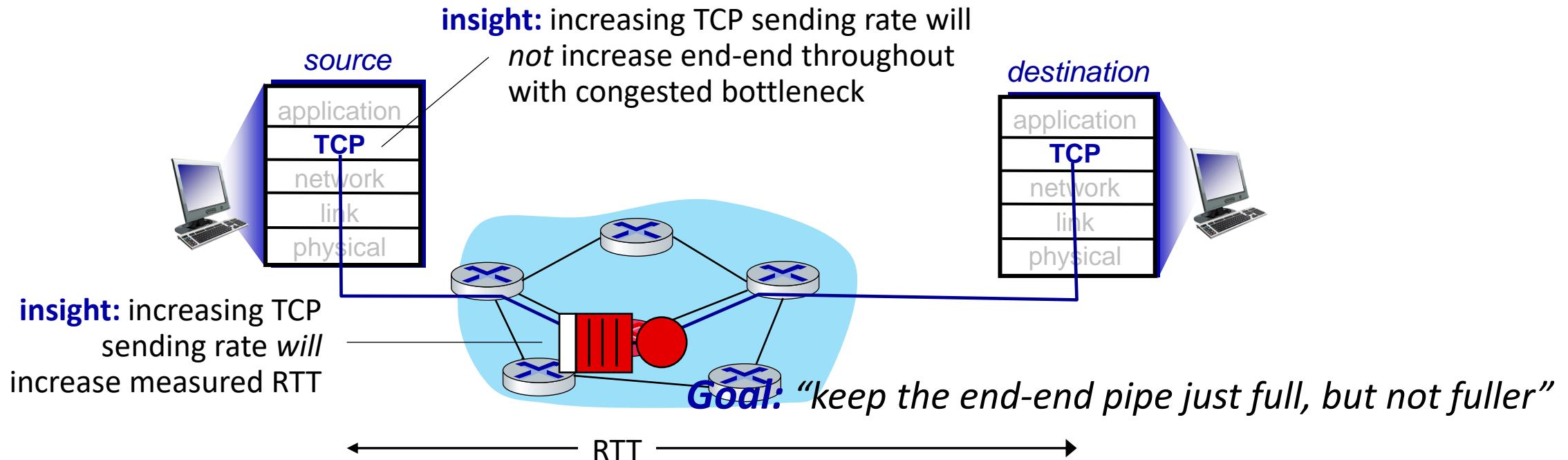
W_{max}

TCP sending rate

TCP Reno

TCP CUBIC

time

t_0 t_1 t_2 t_3 t_4

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*

source

application
TCP
network
link
physical

packet queue almost never empty, sometimes overflows packet (loss)

bottleneck link (almost always busy)

destination

application
**TCP**
network
link
physical
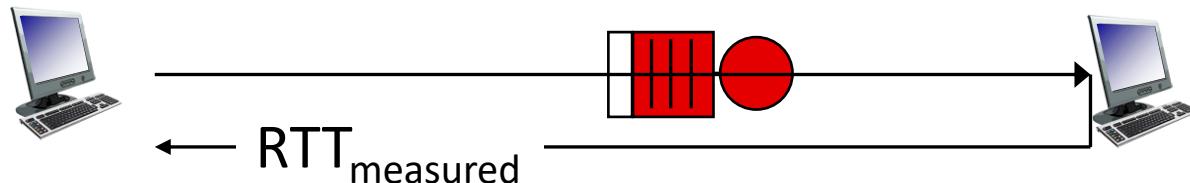
TCP and the congested "bottleneck link"

- 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

insight: increasing TCP sending rate will *not* increase end-end throughout with congested bottleneck

source

application
TCP
network
link
physical

destination

application
TCP
network
link
physical

insight: increasing TCP sending rate *will* increase measured RTT

Goal: "keep the end-end pipe just full, but not fuller"

RTT

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

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

Delay-based approach:

- RTT_{min} - minimum observed RTT (uncongested path)

- uncongested throughput with congestion window `cwnd` is $cwnd/RTT_{min}$

```
if measured throughput "very close" to  uncongested throughput
     increase cwnd linearly         /* since path not congested */
else if measured throughput "far below" uncongested throughout
     decrease cwnd  linearly         /* since path is congested */
```
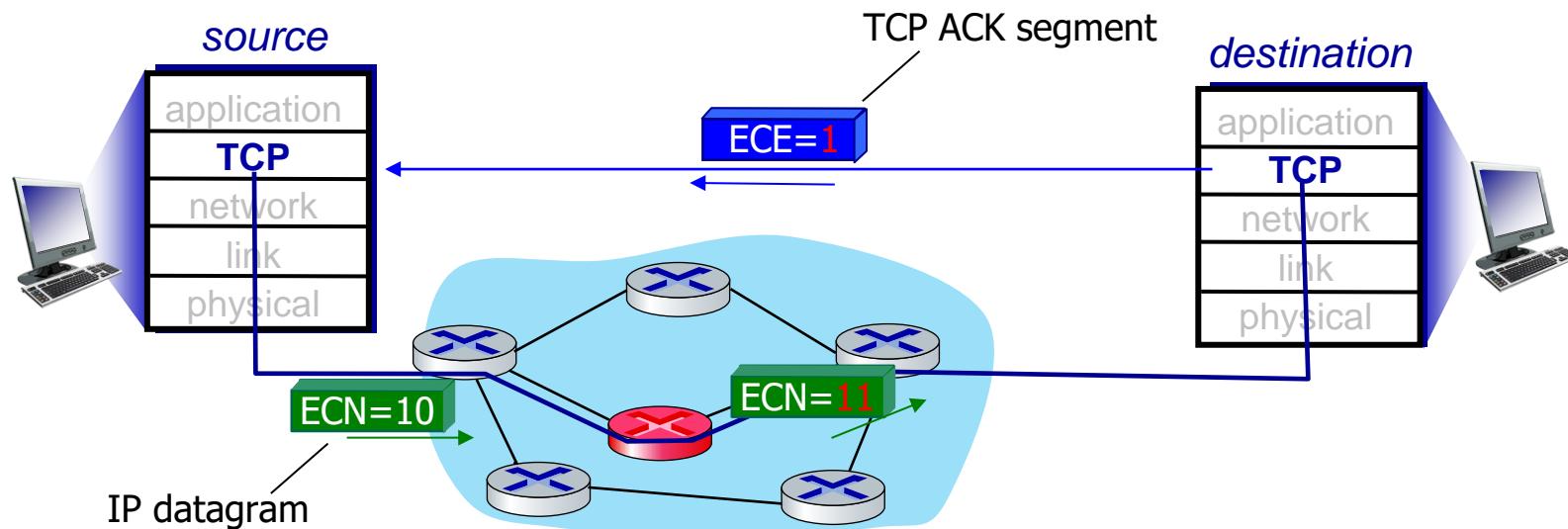
Delay-based TCP congestion control

- congestion control without inducing/forcing loss

- maximizing throughout ("keeping the just pipe full... ") while keeping delay low ("...but not fuller")

- a number of deployed TCPs take a delay-based approach
  - BBR deployed on Google's (internal) backbone network

Explicit congestion notification (ECN)
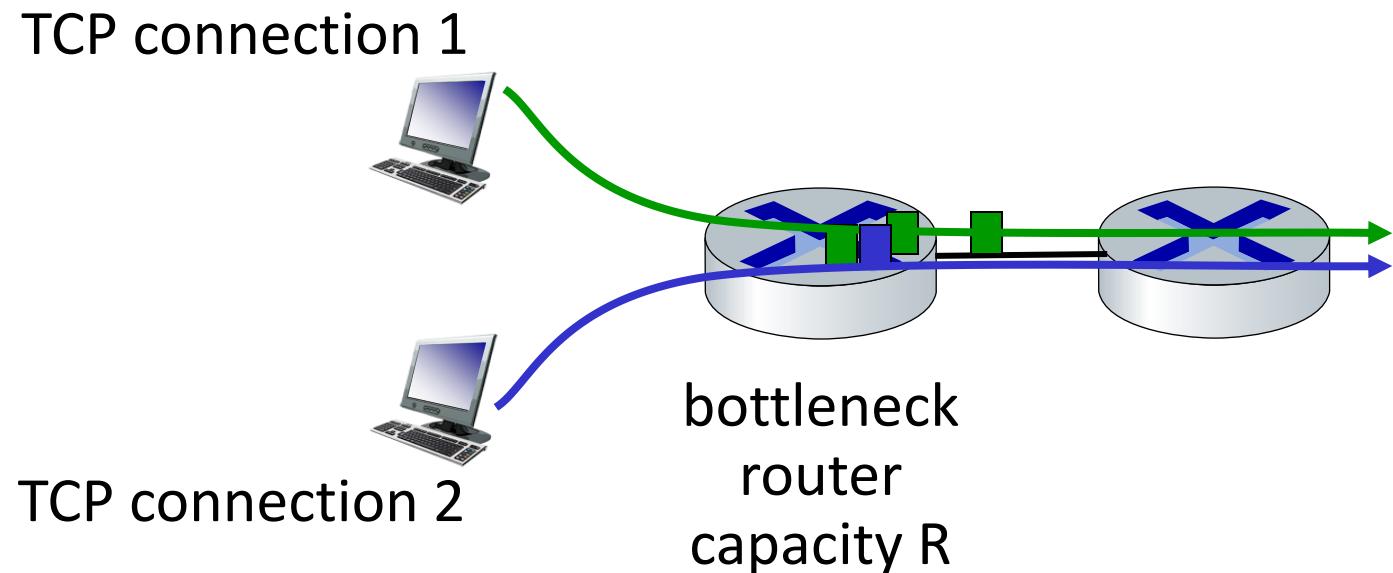
TCP deployments often implement *network-assisted* congestion control:

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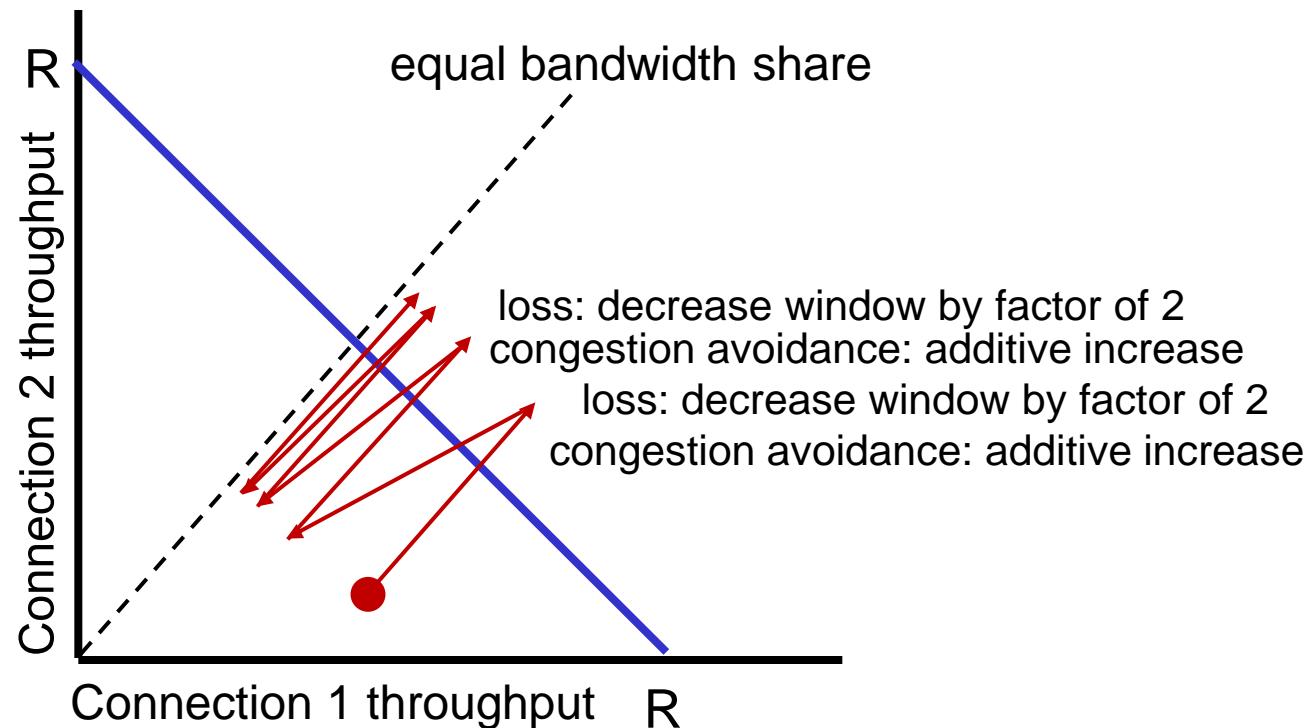
Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*

TCP connection 1

TCP connection 2

bottleneck
router
capacity R

Q: is TCP Fair?

Example: two competing TCP sessions:

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

equal bandwidth share

loss: decrease window by factor of 2
congestion avoidance: additive increase
loss: decrease window by factor of 2
congestion avoidance: additive increase

Connection 2 throughput

Connection 1 throughput

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/2

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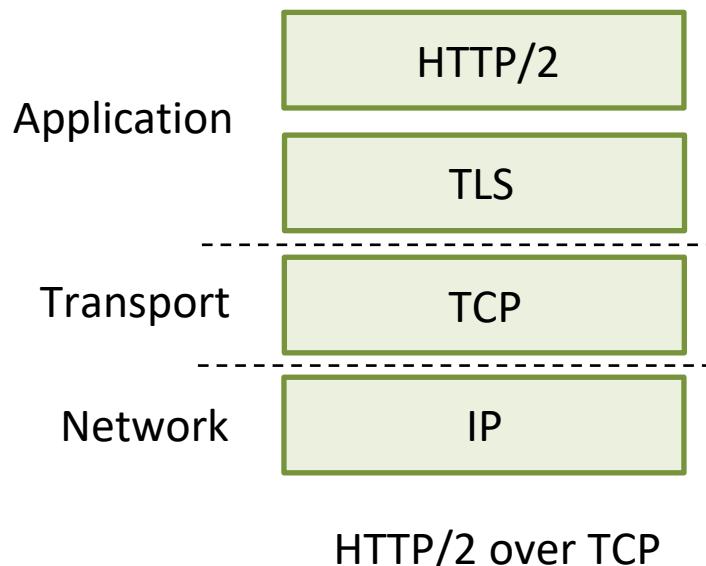
- TCP, UDP: principal transport protocols for 40 years
- different "flavors" of TCP developed, for specific scenarios:

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)

|	
Application	HTTP/2
	TLS
Transport	TCP
Network	IP

HTTP/2 over TCP
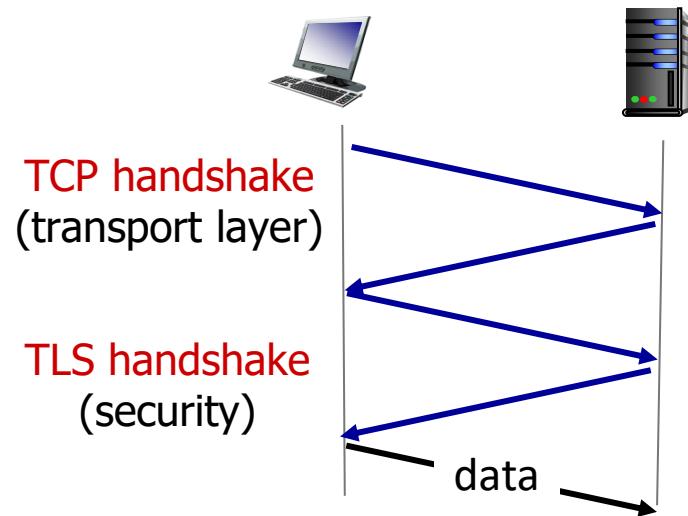
QUIC: Quick UDP Internet Connections

adopts approaches we've studied in this chapter for connection establishment, error control, congestion control
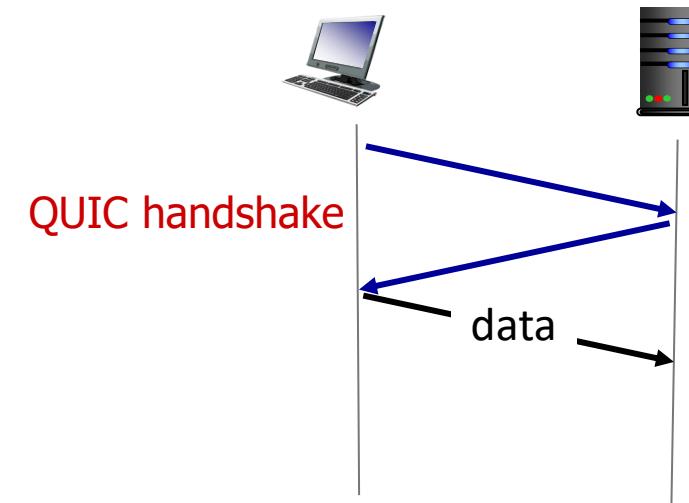
- **error and congestion control:** "Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones." [from QUIC specification]
- **connection establishment:** reliability, congestion control, authentication, encryption, state established 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)
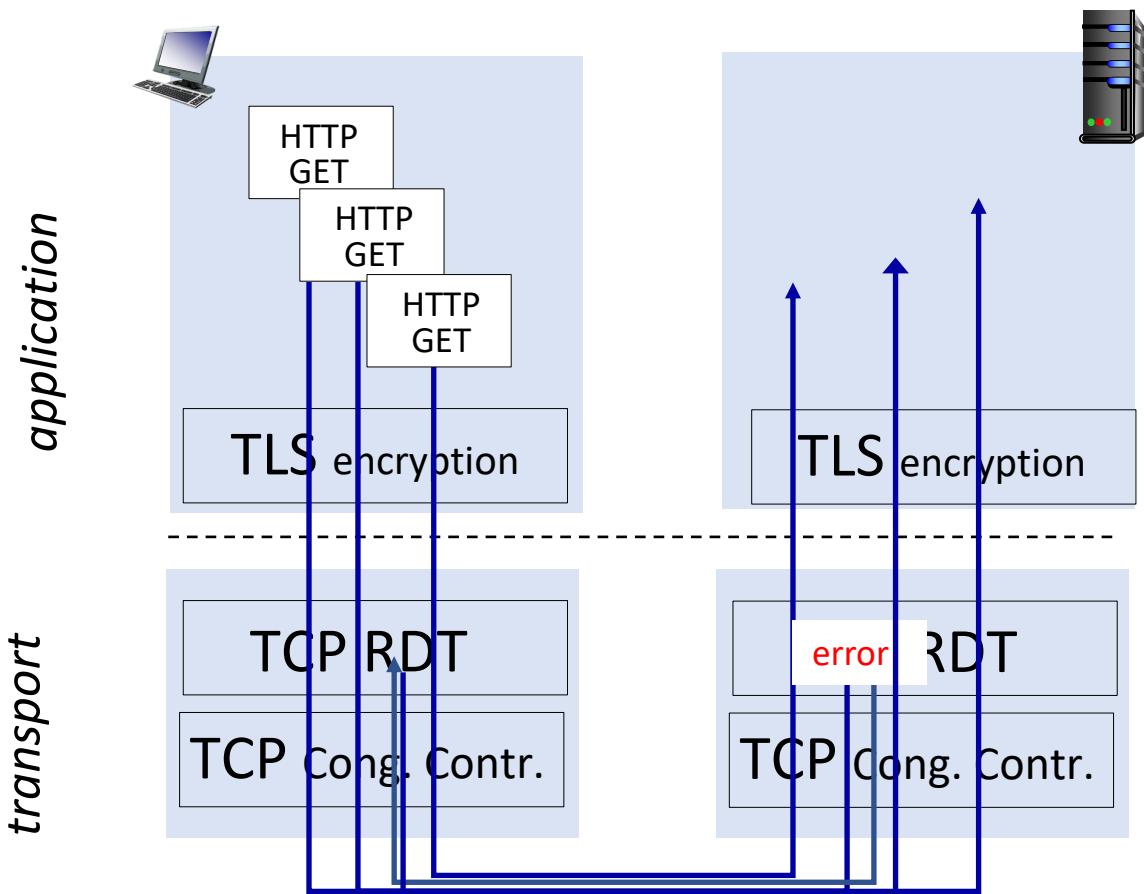
- 2 serial handshakes

QUIC: reliability, congestion control, authentication, crypto state

- 1 handshake

QUIC: streams: parallelism, no HOL blocking

HTTP GET

HTTP GET

HTTP GET

TLS encryption

TLS encryption

application

transport

TCP RDT

error RDT

TCP Cong. Contr.

TCP Cong. Contr.

(a) HTTP 1.1

Chapter 3: summary

- **principles behind transport layer services:**
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- **instantiation, implementation in 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