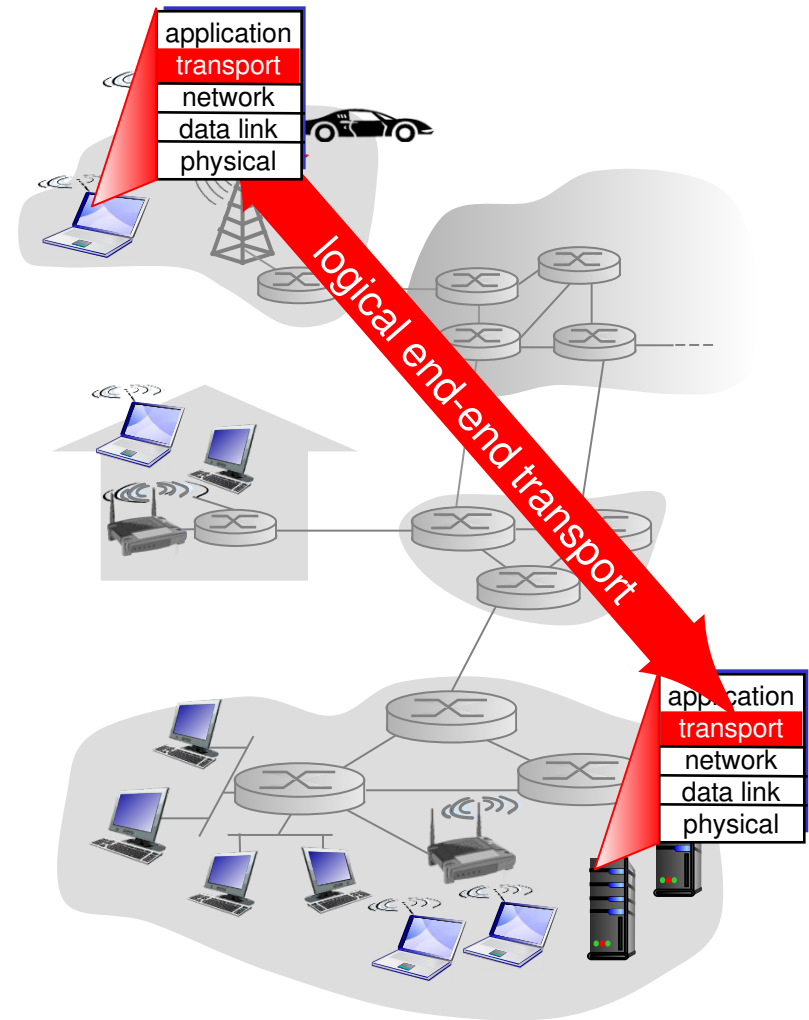# Transport Layer

our goals:

❖ understand principles behind transport layer services:

- multiplexing, demultiplexing
- reliable data transfer
- flow control
- congestion control

❖ learn about Internet transport layer protocols:

- UDP: connectionless transport
- TCP: connection-oriented reliable transport
- TCP congestion control

Transport services and protocols

- provide *logical communication* between app processes running on different hosts

- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer

- more than one transport protocol available to apps
  - Internet: TCP and UDP

Internet transport-layer protocols

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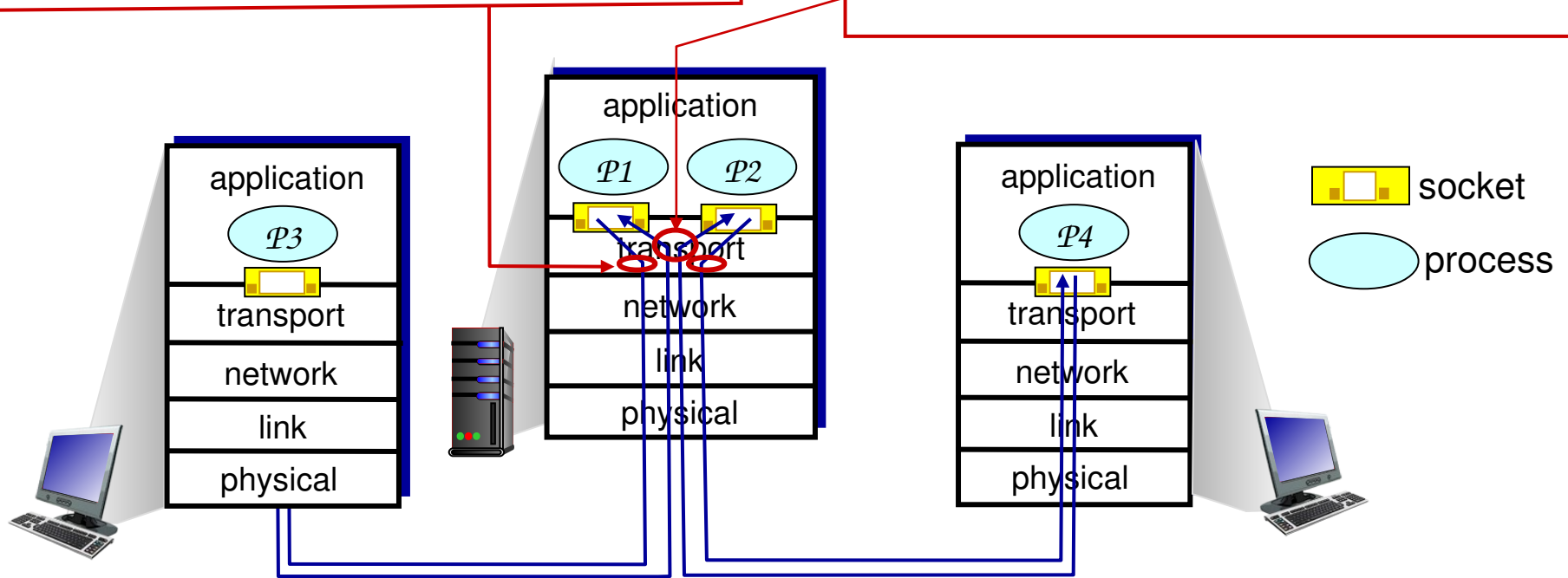 ▪ delay guarantees
  ▪ bandwidth guarantees
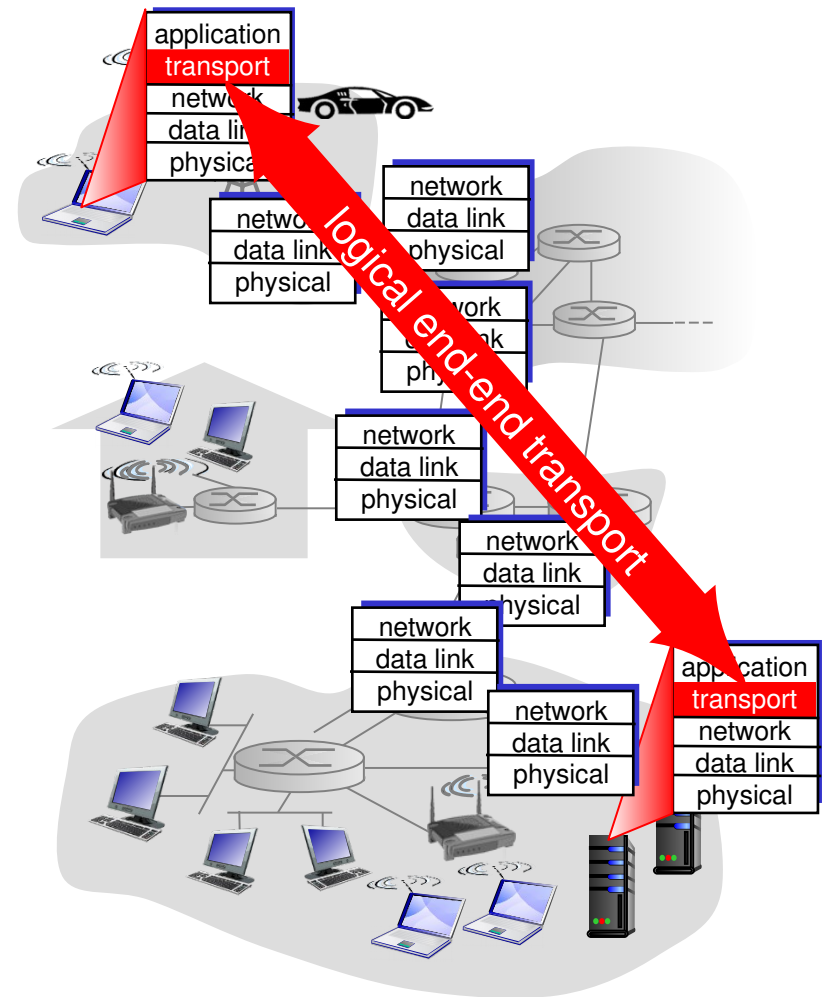
Multiplexing/demultiplexing

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

← 32 bits →

source port #	dest port #
other header fields	
application data (payload)	
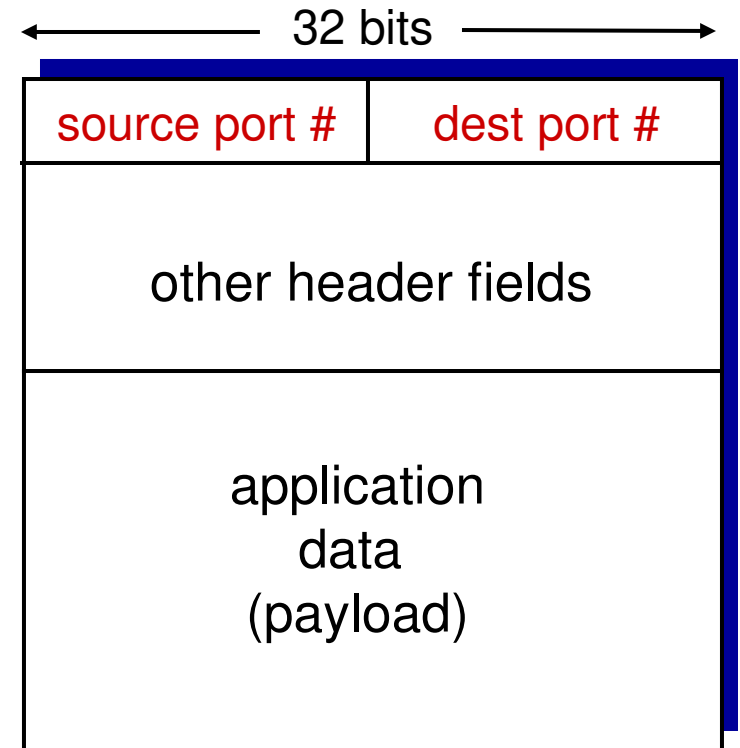
TCP/UDP segment format

Connectionless demultiplexing

- created socket has host-local port #:

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

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

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

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

Connectionless demux: example

`DatagramSocket serverSocket = new DatagramSocket (6428);`

`DatagramSocket mySocket2 = new DatagramSocket (9157);`

`DatagramSocket mySocket1 = new DatagramSocket (5775);`

application

P3

transport

network

link

physical

application

P1

transport

network

link

physical

application

P4

transport

network

link

physical

source port: 6428
dest port: 9157

source port: ?
dest port: ?

source port: 9157
dest port: 6428

source port: ?
dest port: ?
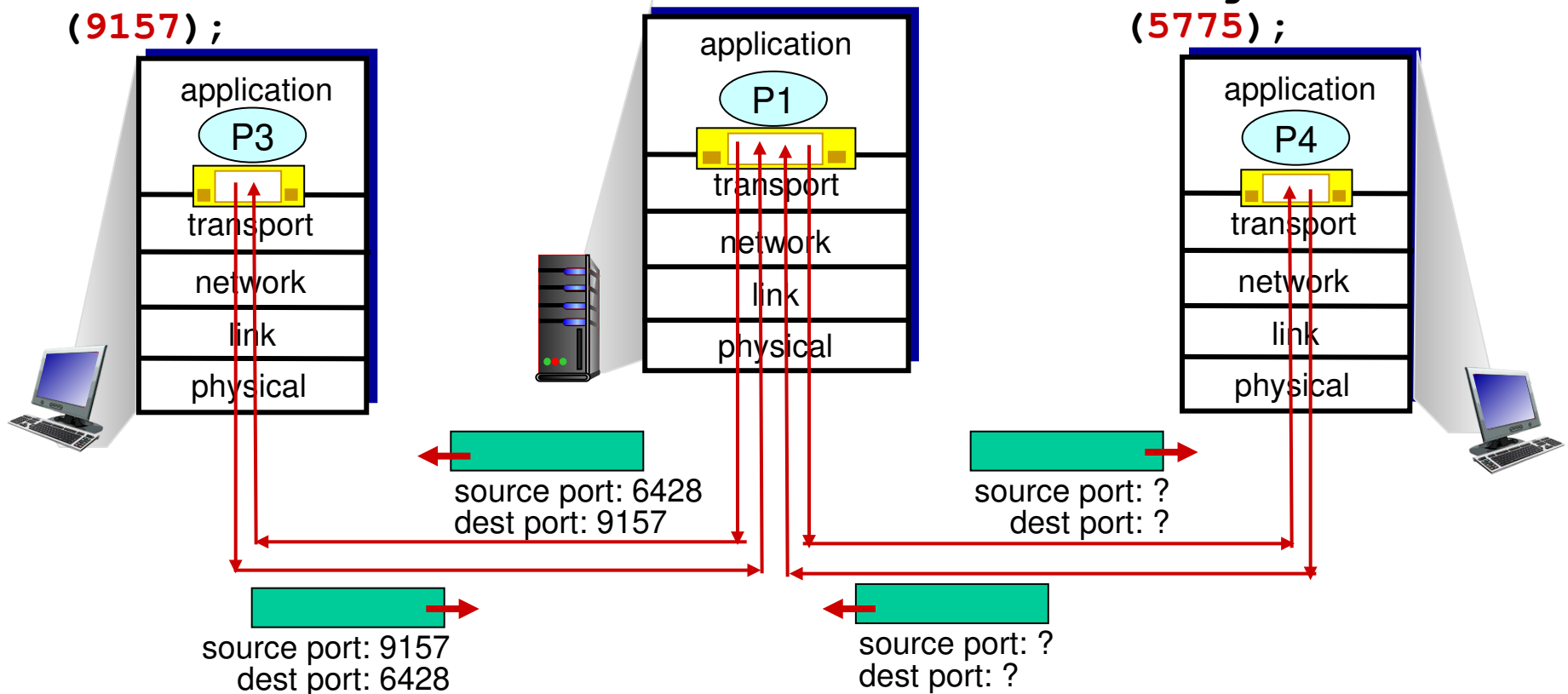
Connection-oriented demux

❖ TCP socket identified by 4-tuple:
  ▪ source IP address
  ▪ source port number
  ▪ dest IP address
  ▪ dest port number
❖ demux: receiver uses all four values to direct segment to appropriate socket

❖ server host may support many simultaneous TCP sockets:
  ▪ each socket identified by its own 4-tuple
❖ web servers have different sockets for each connecting client
  ▪ non-persistent HTTP will have different socket for each request

Connection-oriented demux: example

application

P4   P5   P6

transport

network

link

physical

application

P3

transport

network

link

physical

application

P2   P3

transport

network

link

physical

server:
IP
address
B

host: IP
address
A

host: IP
address
C

source IP,port: B,80
dest IP,port: A,9157

source IP,port: A,9157
dest IP, port: B,80
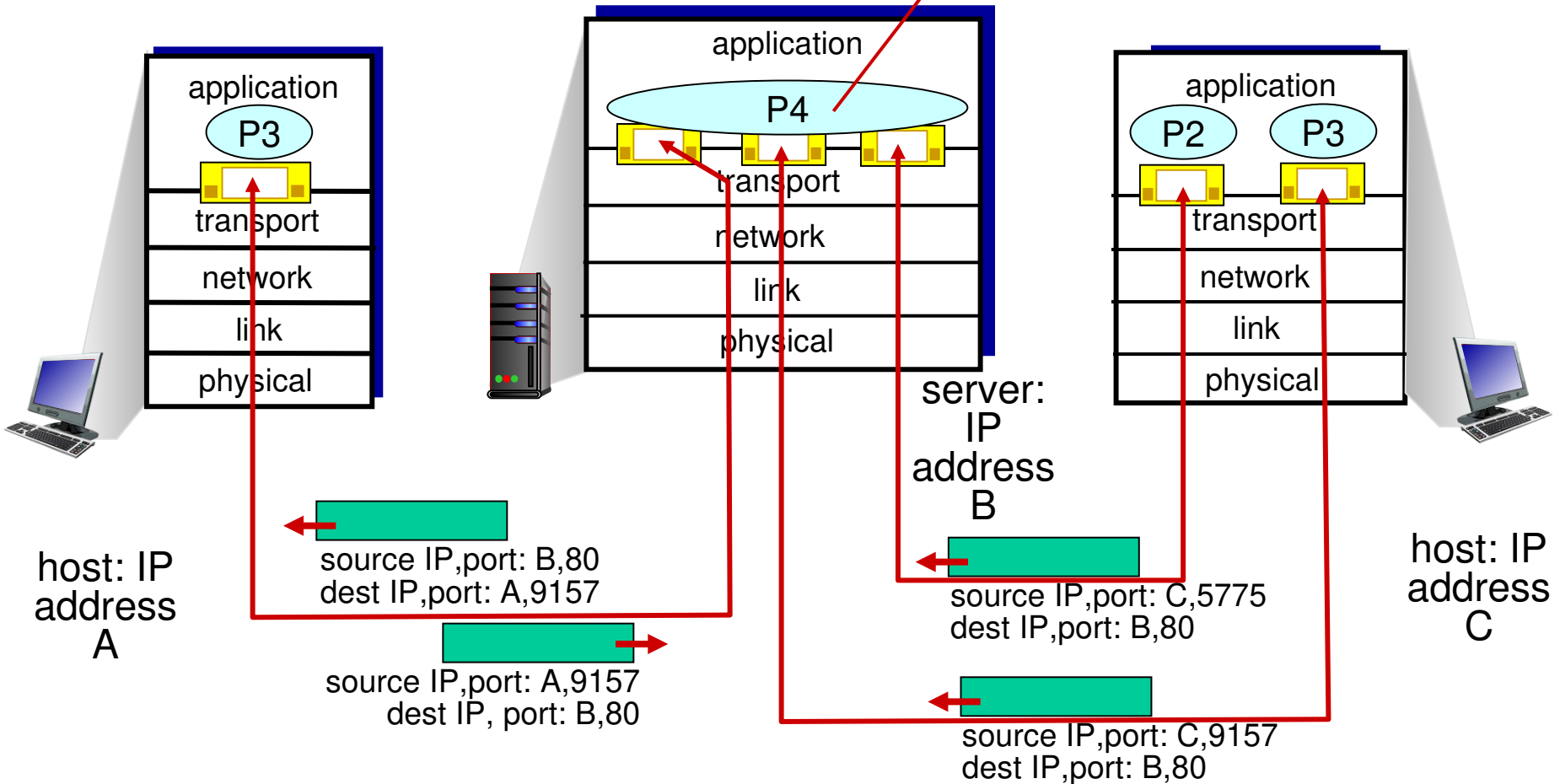
source IP,port: C,5775
dest IP,port: B,80

source IP,port: C,9157
dest IP,port: B,80
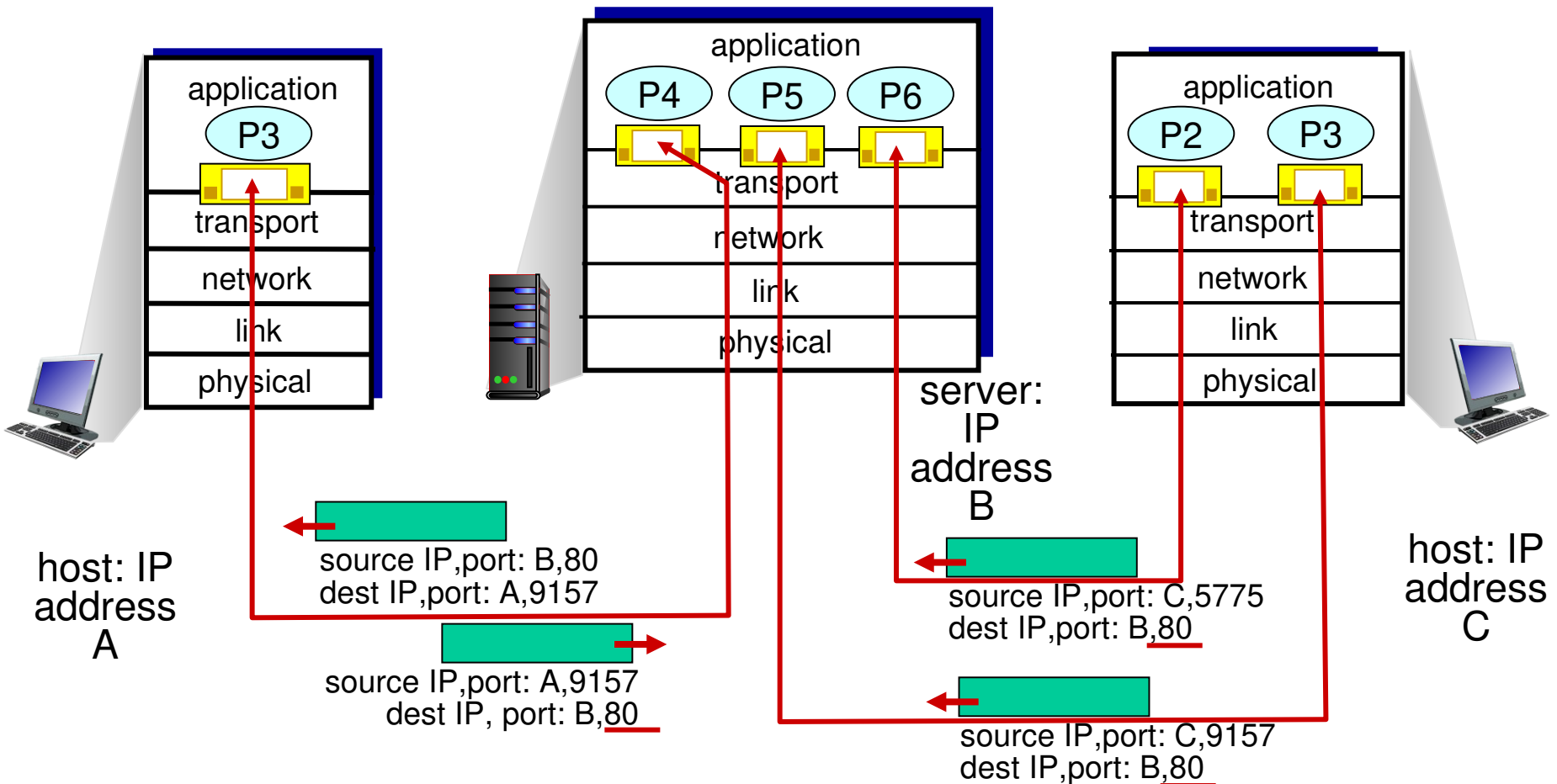
three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

Connection-oriented demux: example

threaded server

application
P4

application
P3

application
P2   P3

transport
network
link
physical

server:
IP
address
B

source IP,port: B,80
dest IP,port: A,9157

source IP,port: A,9157
dest IP, port: B,80

source IP,port: C,5775
dest IP,port: B,80

source IP,port: C,9157
dest IP,port: B,80

host: IP
address
A

host: IP
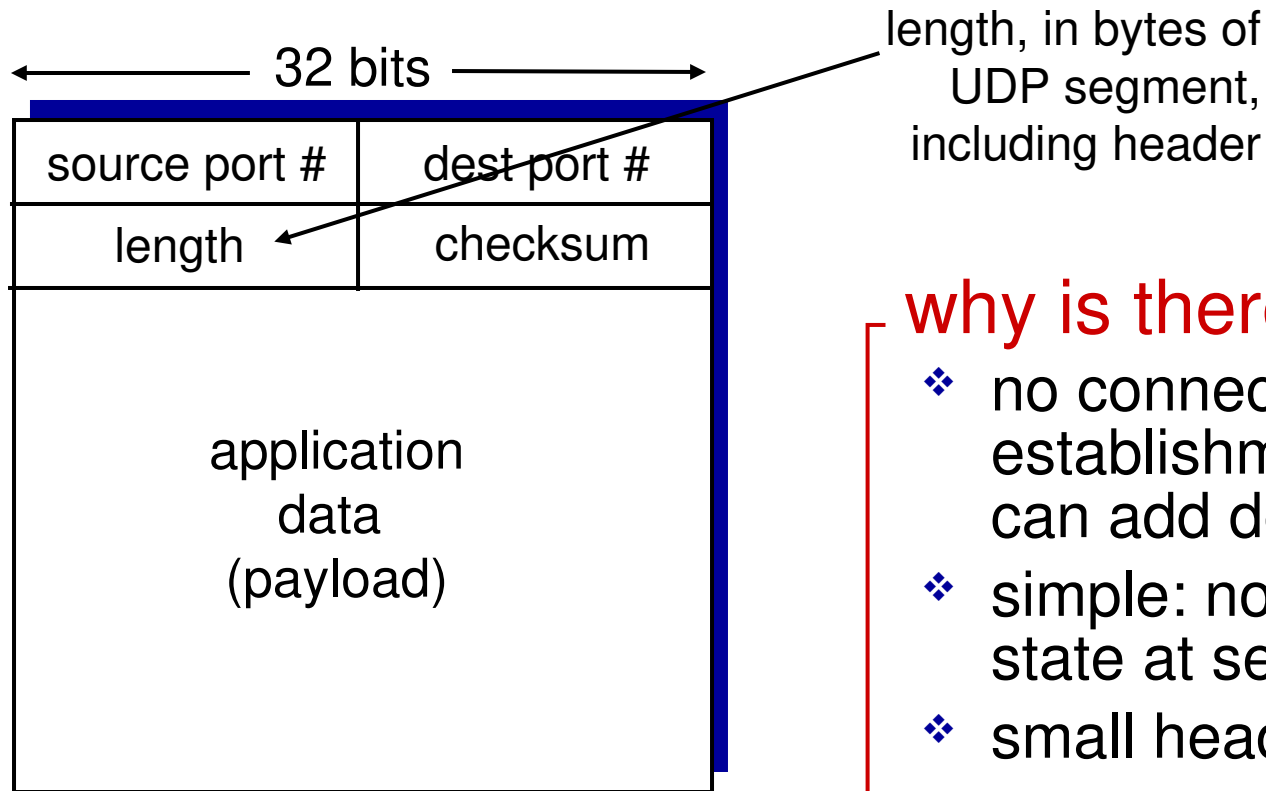address
C

UDP: User Datagram Protocol [RFC 768]

- ❖ "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

- ▪ UDP use:
  - ▪ streaming multimedia apps (loss tolerant, rate sensitive)
  - ▪ DNS
  - ▪ SNMP
- ▪ reliable transfer over UDP:
  - ▪ add reliability at application layer
  - ▪ application-specific error recovery!

UDP: segment header

32 bits

length, in bytes of
UDP segment,
including header

source port #	dest port #
length	checksum

application
data
(payload)

UDP segment format

why is there a UDP?

- ❖ no connection establishment (which can add delay)
- ❖ simple: no connection state at sender, receiver
- ❖ small header size
- ❖ no congestion control: UDP can blast away as fast as desired

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

sender:

❖ treat segment contents, including header fields, as sequence of 16-bit integers

❖ checksum: addition (one's complement sum) of segment contents

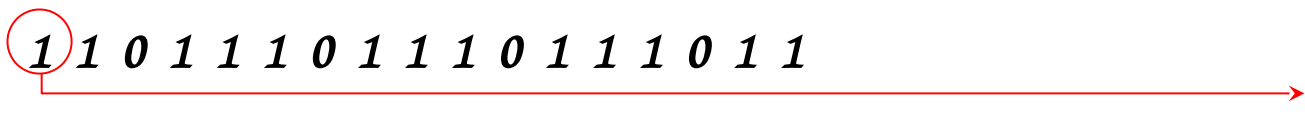❖ sender puts checksum value into UDP checksum field

receiver:

❖ compute checksum of received segment

❖ check if computed checksum equals checksum field value:

- NO - error detected
- YES - no error detected. *But maybe errors nonetheless?* More later ….

Internet checksum: example

example: add two 16-bit integers

$$1\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0$$
$$1\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1$$

wraparound **(1)** 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1

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

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result
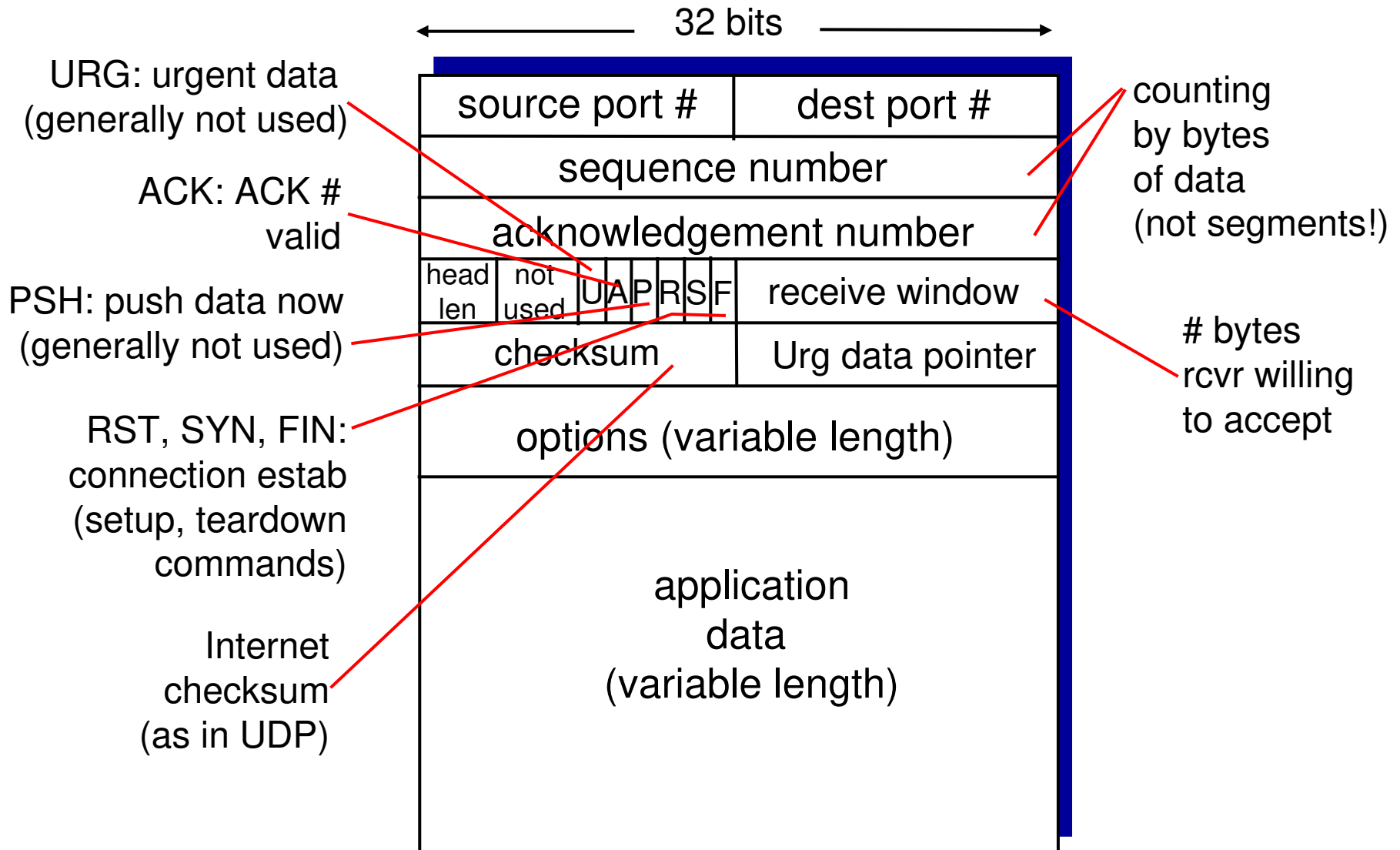
* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

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

- ❖ point-to-point:
  - one sender, one receiver
- ❖ reliable, in-order *byte steam:*
  - no "message boundaries"
- ❖ pipelined:
  - TCP congestion and flow control set window size

- ❖ full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- ❖ connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- ❖ flow controlled:
  - sender will not overwhelm receiver

TCP segment structure

URG: urgent data
(generally not used)

ACK: ACK #
valid

PSH: push data now
(generally not used)

RST, SYN, FIN:
connection estab
(setup, teardown
commands)

Internet
checksum
(as in UDP)

32 bits

source port #	dest port #
sequence number	
acknowledgement number	

head len	not used	U A P R S F	receive window
checksum			Urg data pointer

options (variable length)

application
data
(variable length)

counting
by bytes
of data
(not segments!)

bytes
rcvr willing
to accept

TCP seq. numbers, ACKs

sequence numbers:
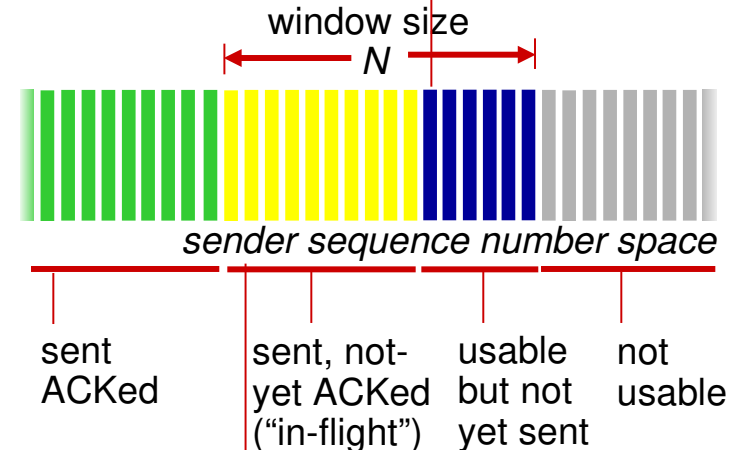- byte stream "number" of first byte in segment's data

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

Q: how receiver handles out-of-order segments
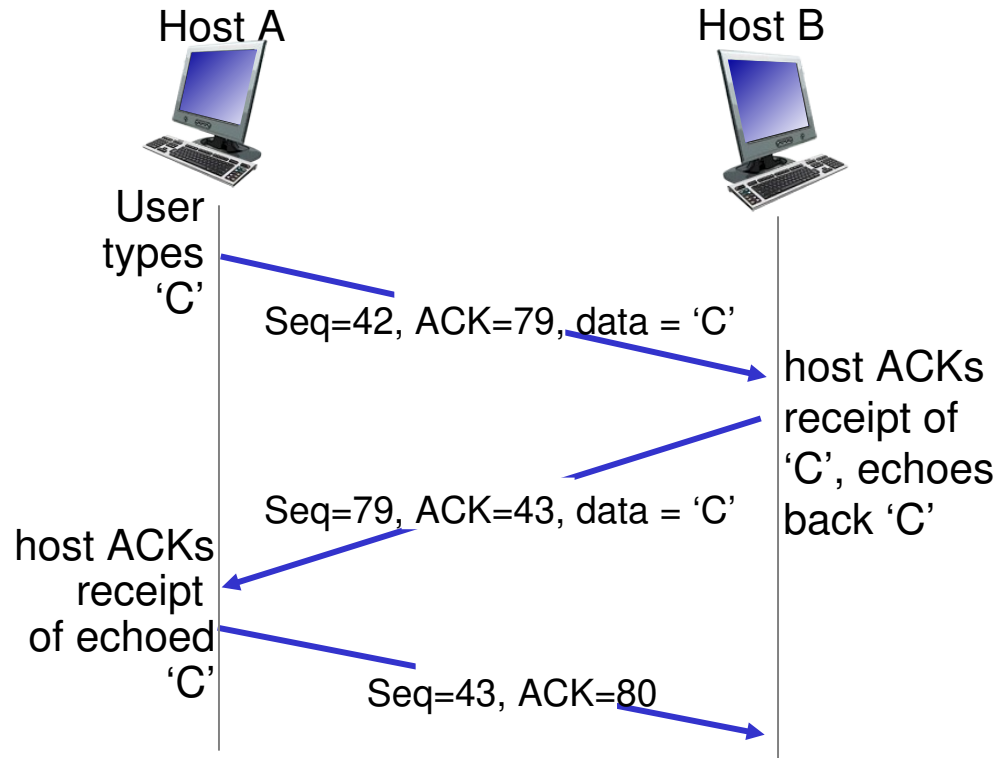- A: TCP spec doesn't say, - up to implementor

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 |

incoming segment to sender

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

TCP seq. numbers, ACKs

Host A   Host B

User
types
'C'
                Seq=42, ACK=79, data = 'C'
                                          host ACKs
                                          receipt of
                                          'C', echoes
                Seq=79, ACK=43, data = 'C'   back 'C'
host ACKs
receipt
of echoed
'C'
                Seq=43, ACK=80

simple telnet scenario

TCP: retransmission scenarios

Host A   Host B   Host A   Host B

Seq=92, 8 bytes of data

timeout

ACK=100

X

Seq=92, 8 bytes of data

ACK=100

lost ACK scenario

SendBase=92

Seq=92, 8 bytes of data

timeout

Seq=100, 20 bytes of data

ACK=100

ACK=120

Seq=92, 8
bytes of data

SendBase=100

SendBase=120

ACK=120

SendBase=120
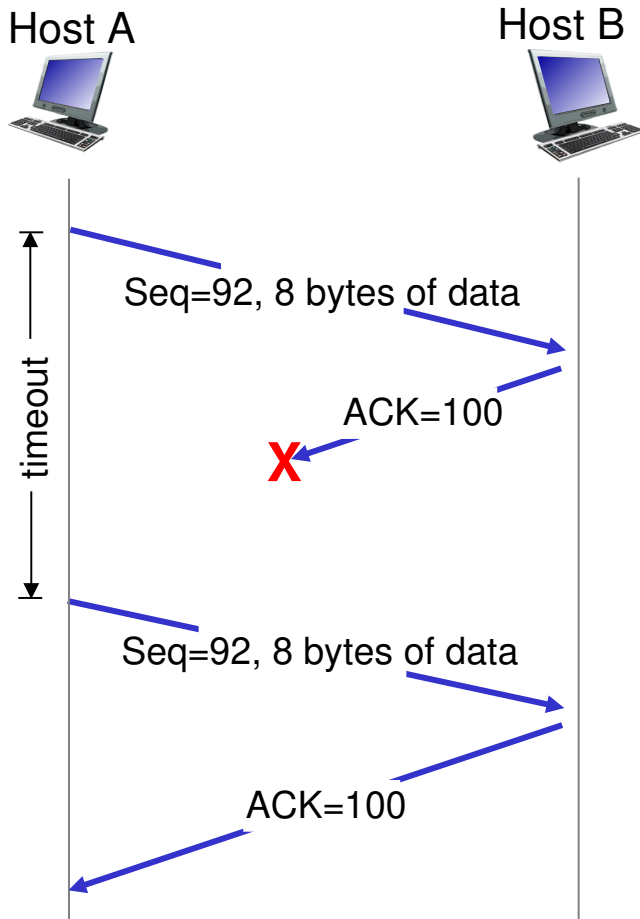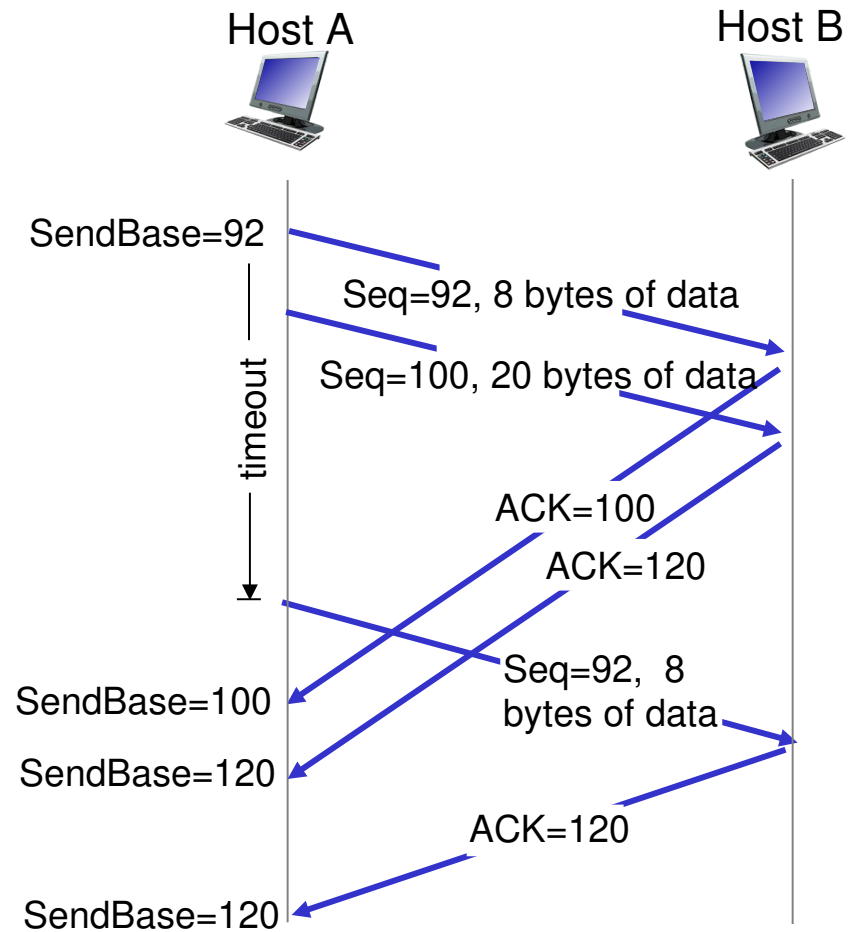
premature timeout

TCP: retransmission scenarios

Seq=92, 8 bytes of data

Seq=100, 20 bytes of data

ACK=100

X

ACK=120

timeout

Seq=120, 15 bytes of data
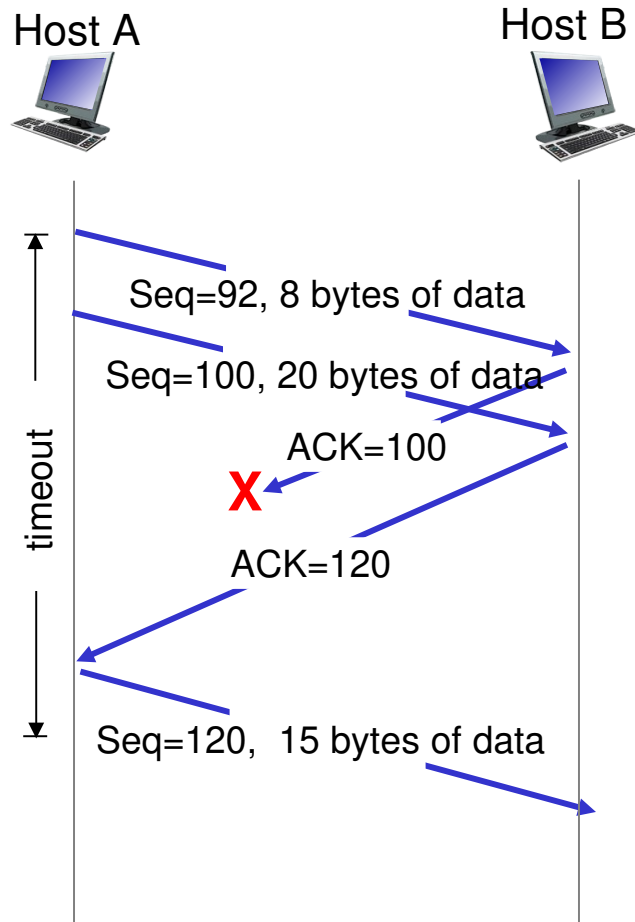
cumulative ACK

TCP flow control

application may
remove data from
TCP socket buffers ….

application

OS

… slower than TCP
receiver is delivering
(sender is sending)

application
process

TCP socket
receiver buffers

TCP
code

IP
code

from sender

receiver protocol stack

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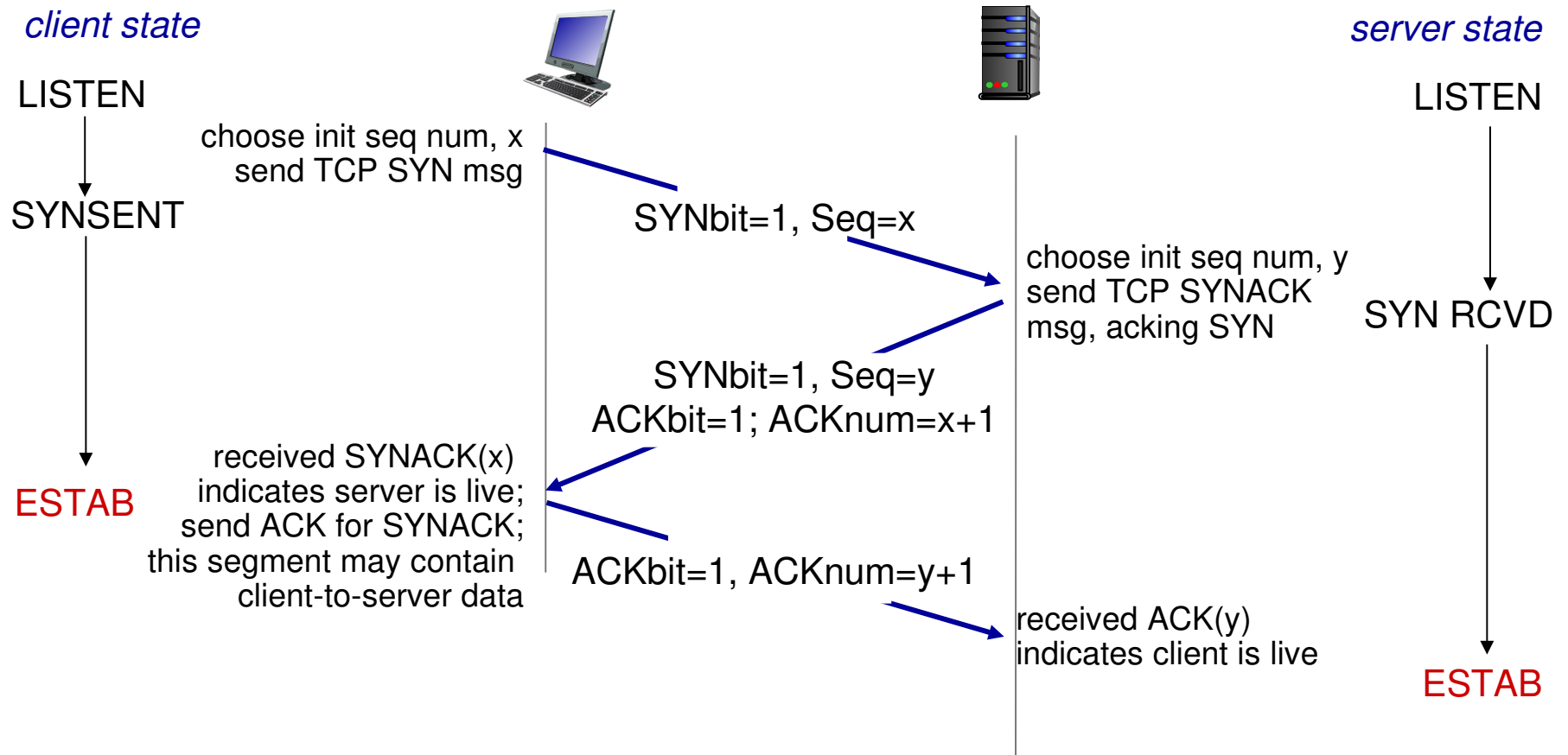receiver's buffer by
transmitting too much, too fast
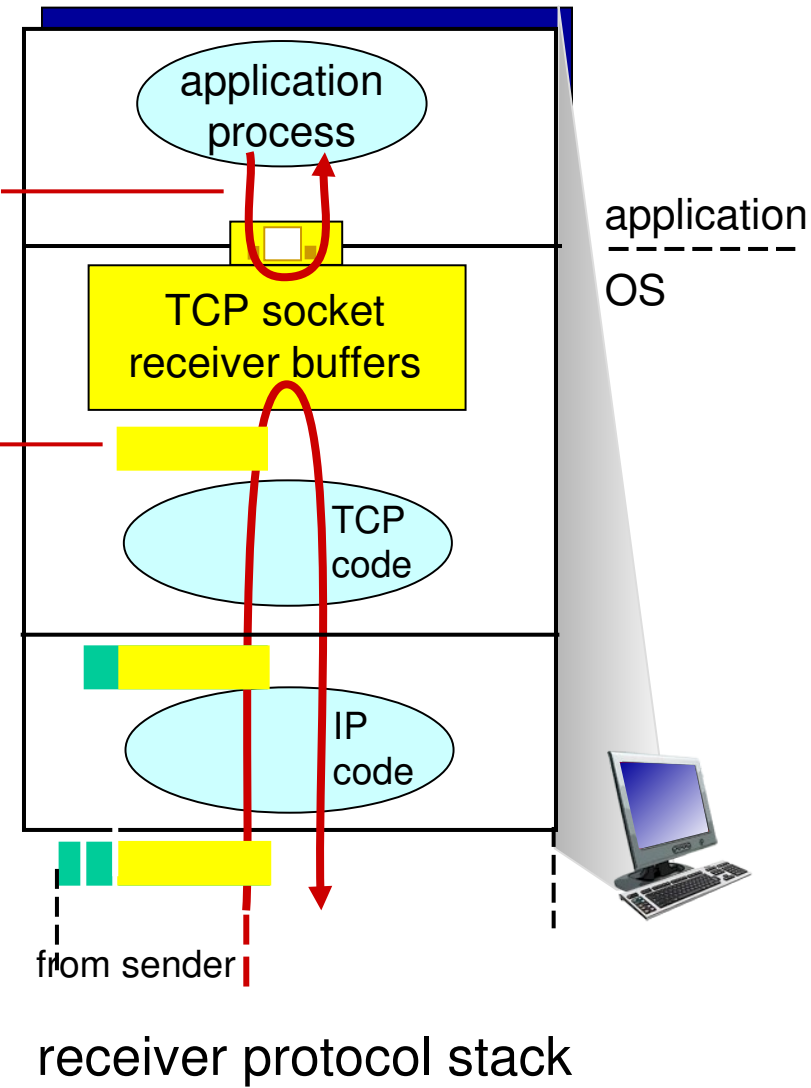
TCP flow control

❖ receiver "advertises" free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments

  ▪ **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 receiver's **rwnd** value

❖ guarantees receive buffer will not overflow

to application process

RcvBuffer

buffered data

rwnd

free buffer space

TCP segment payloads

receiver-side buffering

TCP 3-way handshake

client state

LISTEN

SYNSENT

ESTAB

choose init seq num, x
send TCP SYN msg

SYNbit=1, Seq=x

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

ACKbit=1, ACKnum=y+1

server state

LISTEN

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

SYN RCVD

received ACK(y)
indicates client is live

ESTAB

TCP: closing a connection

❖ client, server each close their side of connection
  ▪ send TCP segment with FIN bit = 1

❖ respond to received FIN with ACK
  ▪ on receiving FIN, ACK can be combined with own FIN

❖ simultaneous FIN exchanges can be handled

TCP: closing a connection

client state

server state

ESTAB

ESTAB

`clientSocket.close()`

FIN_WAIT_1   can no longer
send but can
receive data

FINbit=1, seq=x

CLOSE_WAIT

ACKbit=1; ACKnum=x+1

FIN_WAIT_2   wait for server
close

can still
send data

FINbit=1, seq=y

LAST_ACK

TIMED_WAIT

can no longer
send data

ACKbit=1; ACKnum=y+1

timed wait
for 2*max
segment lifetime

CLOSED

CLOSED

Principles of congestion control

congestion:

❖ informally: "too many sources sending too much data too fast for *network* to handle"

❖ different from flow control!

❖ manifestations:
- lost packets (buffer overflow at routers)
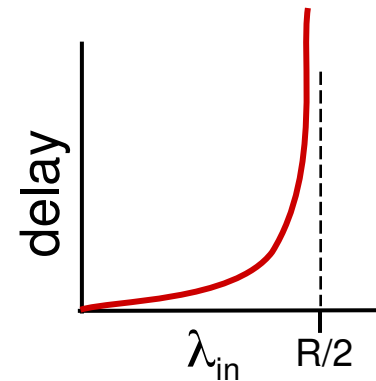- long delays (queueing in router buffers)

❖ a top-10 problem!

Causes/costs of congestion: scenario 1

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

original data: λ_{in}

throughput: λ_{out}

Host A

Host B

unlimited shared output link buffers
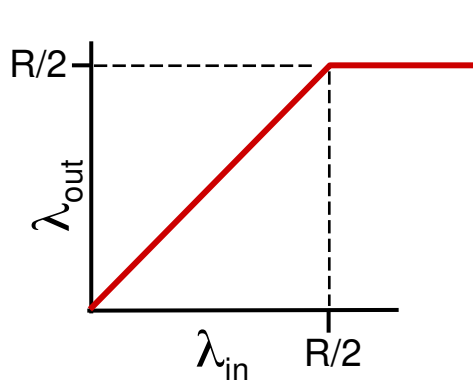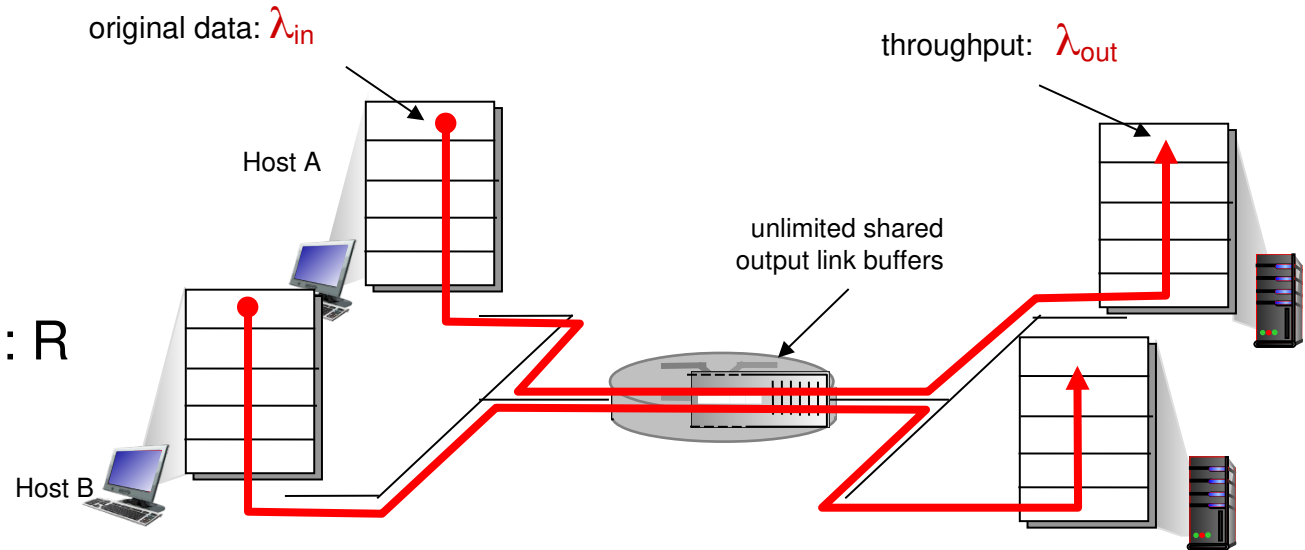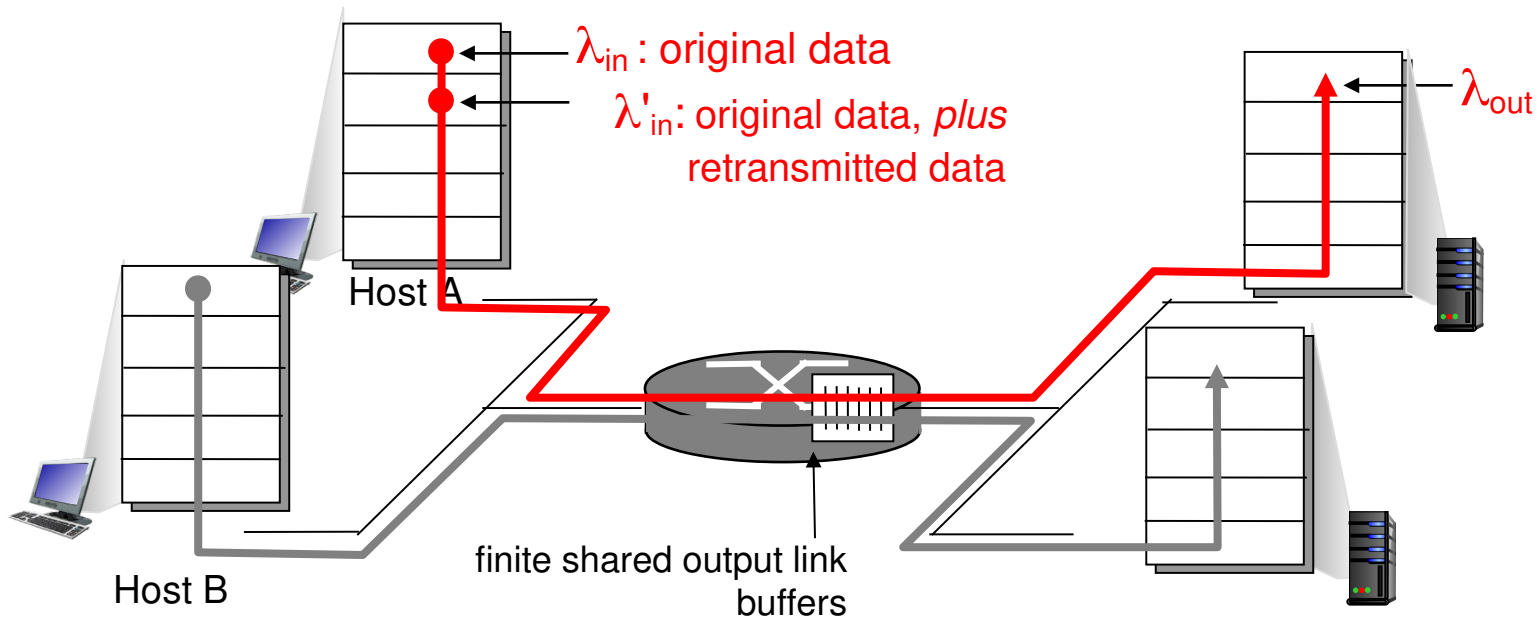

❖ maximum per-connection throughput: R/2

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

Causes/costs of congestion: scenario 2

❖ one router, *finite* buffers
❖ sender retransmission of timed-out packet
  ▪ application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
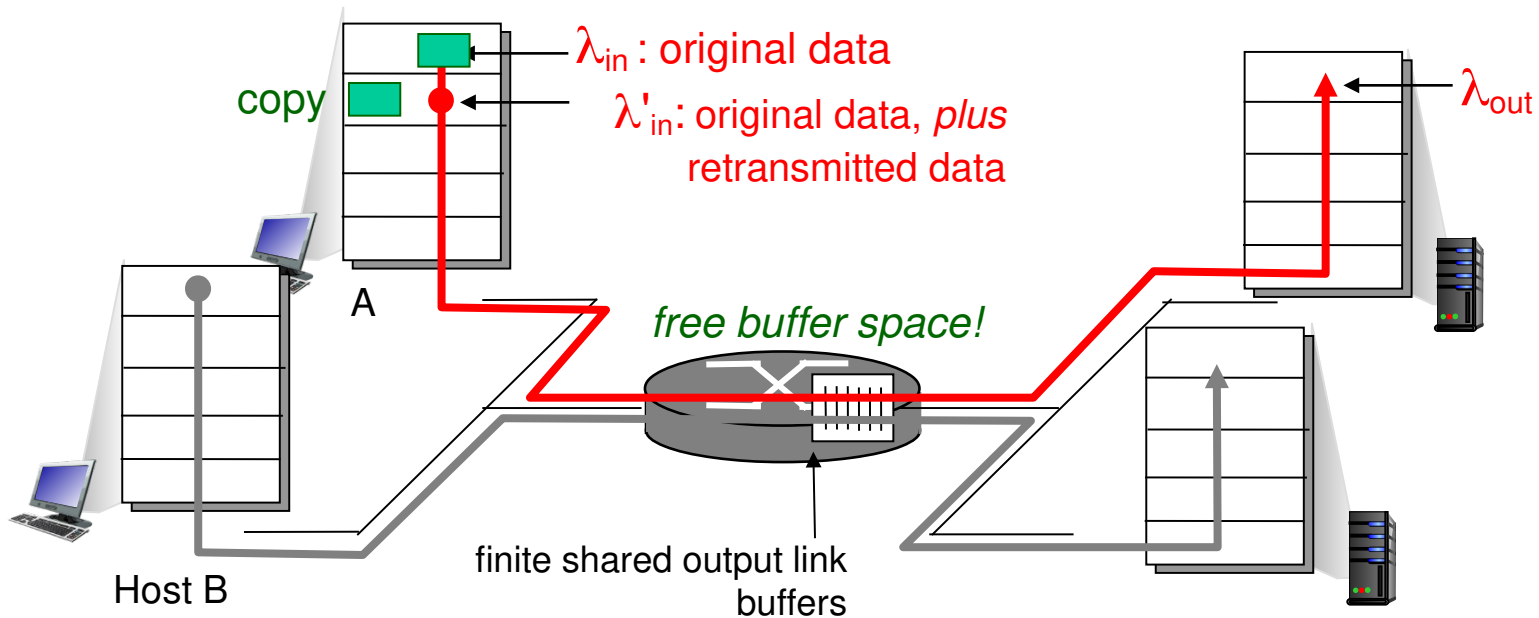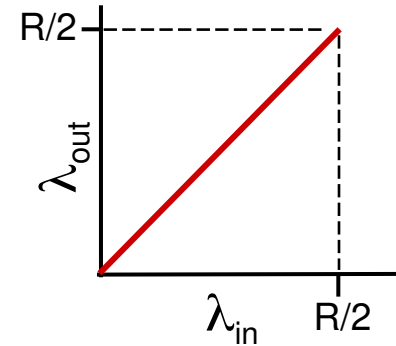 ▪ transport-layer input includes *retransmissions* : λ_{in} λ_{in}

$$\geq$$

λ_{in} : original data

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

λ_{out}

Host A

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

λ_{out}

copy

A

Host B

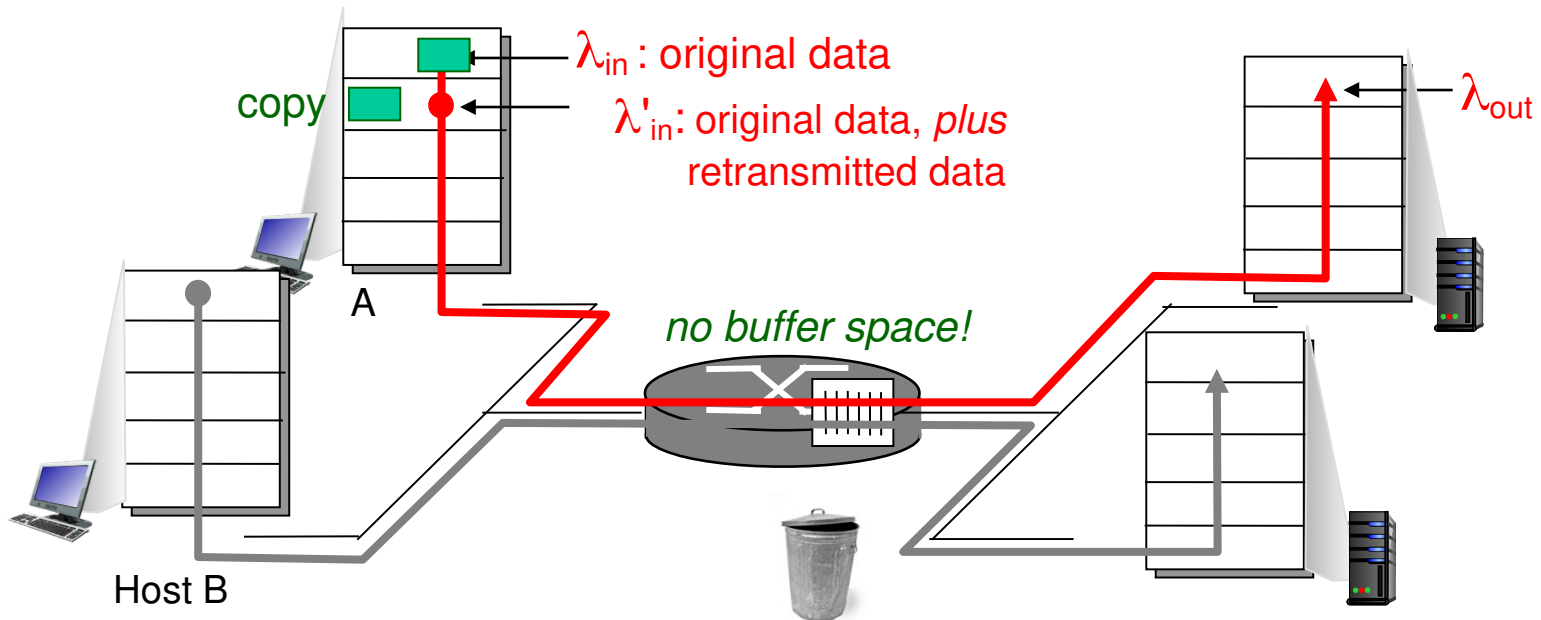*free buffer space!*

finite shared output link buffers

Causes/costs of congestion: scenario 2

Idealization: known loss
packets can be lost,
dropped at router due to
full buffers

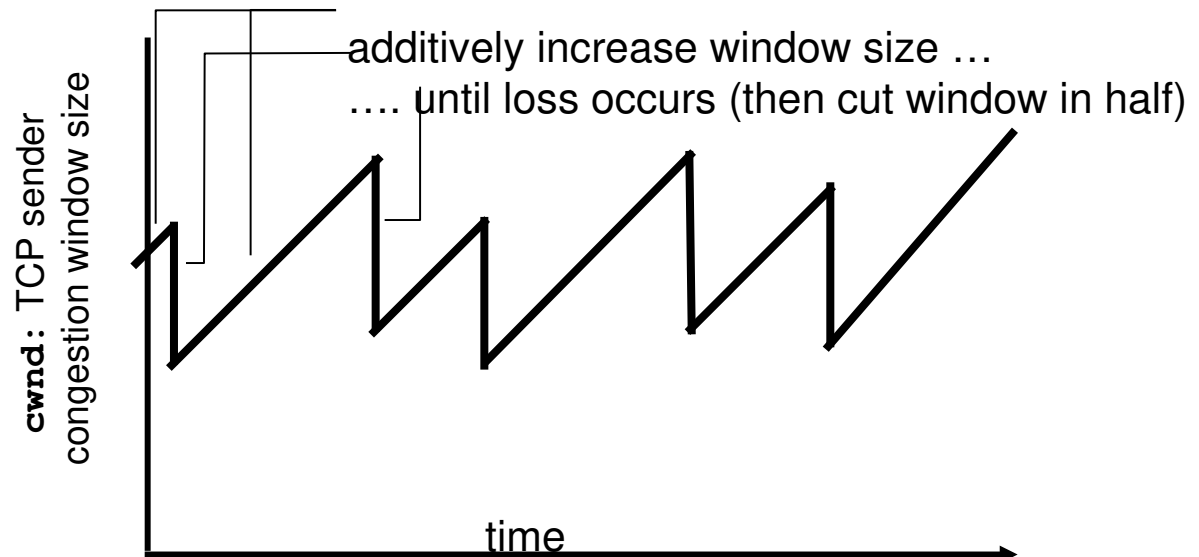❖ sender only resends if
packet *known* to be lost

λ_{in} : original data

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

copy

λ_{out}

A

no buffer space!

Host B

TCP congestion control: additive increase multiplicative decrease

- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase `cwnd` by 1 MSS every RTT until loss detected
  - *multiplicative decrease*: cut `cwnd` in half after loss

AIMD saw tooth behavior: probing for bandwidth

additively increase window size …

…. until loss occurs (then cut window in half)

`cwnd`: TCP sender congestion window size

time

Summary

❖ principles behind transport layer services:

  ▪ multiplexing, demultiplexing

  ▪ reliable data transfer

  ▪ flow control

  ▪ congestion control

❖ instantiation, implementation in the Internet

  ▪ UDP

  ▪ TCP

# 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.

Thanks and enjoy! JFK/KWR
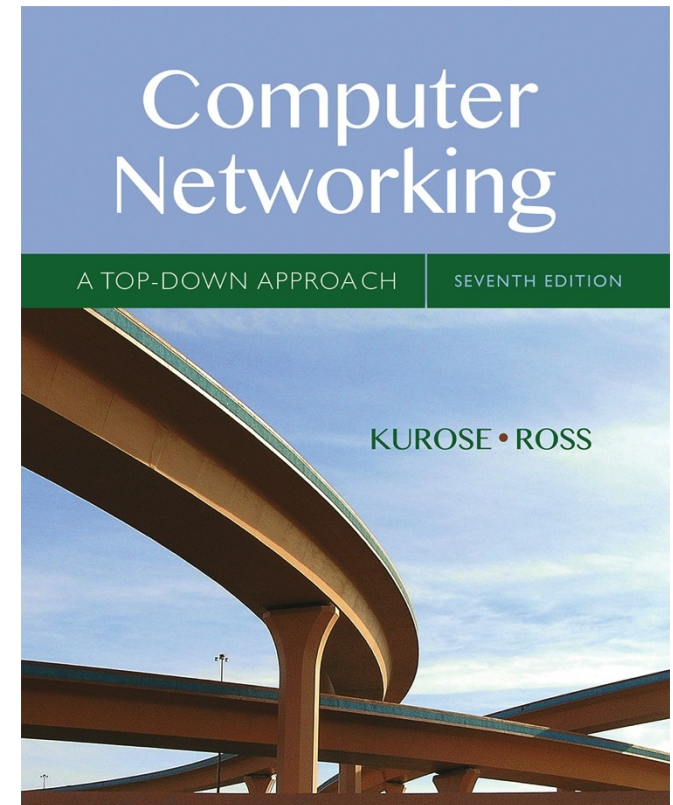
*Computer
Networking: A Top
Down Approach*

7th edition
Jim Kurose, Keith Ross
Pearson/Addison Wesley
April 2016