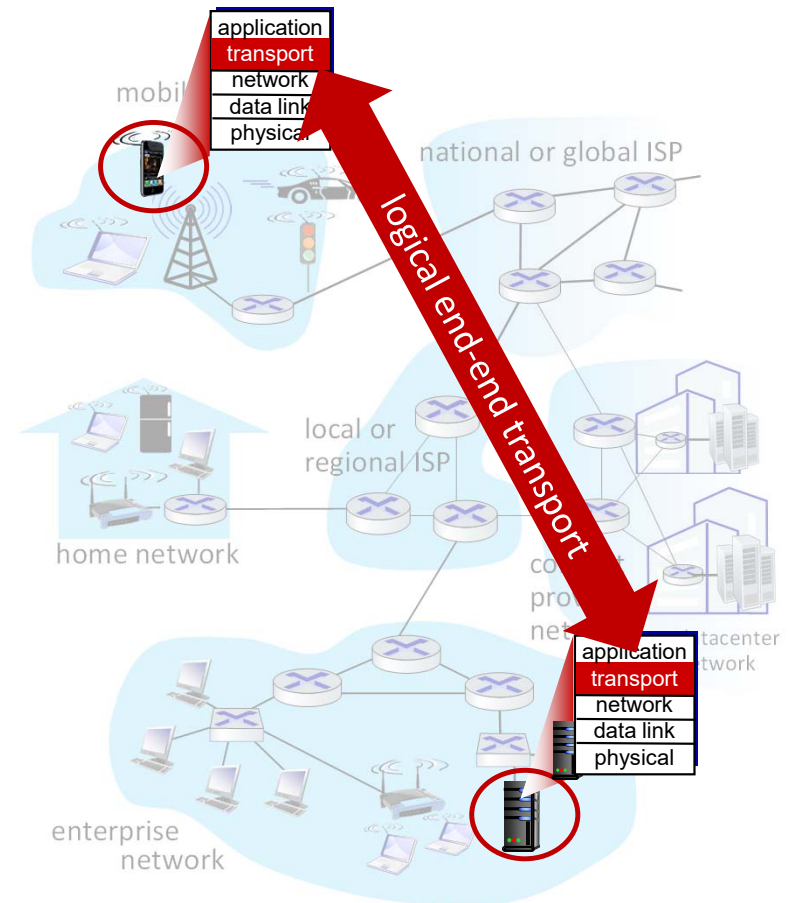# Transport layer: overview

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 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

- network layer: logical communication between *hosts*

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

Two principal Internet transport protocols

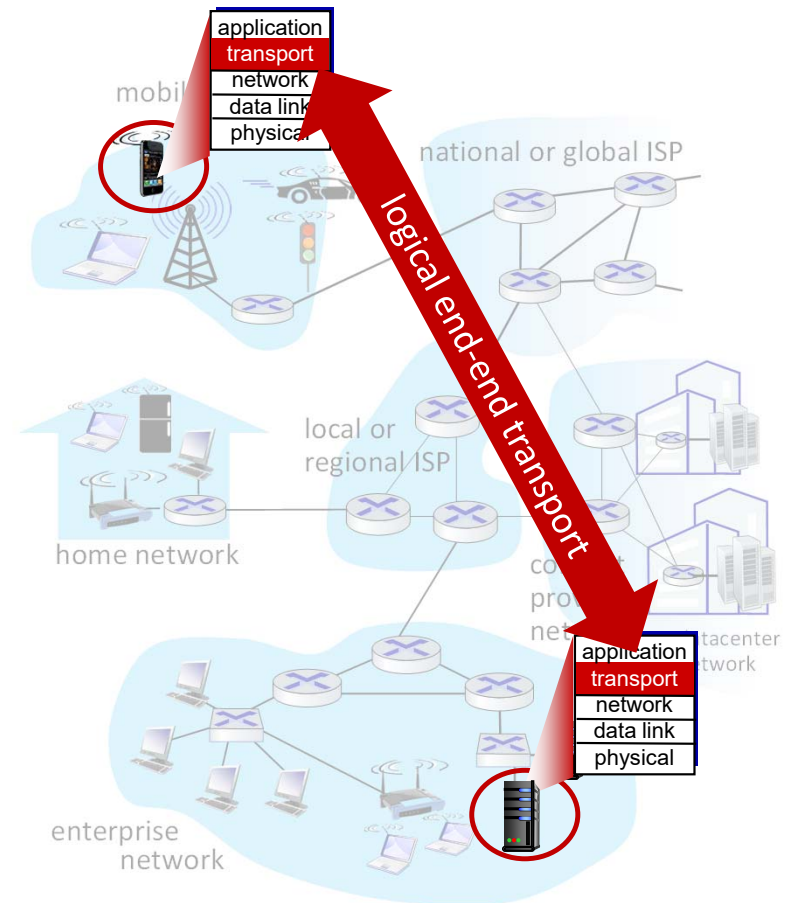- **TCP:** Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
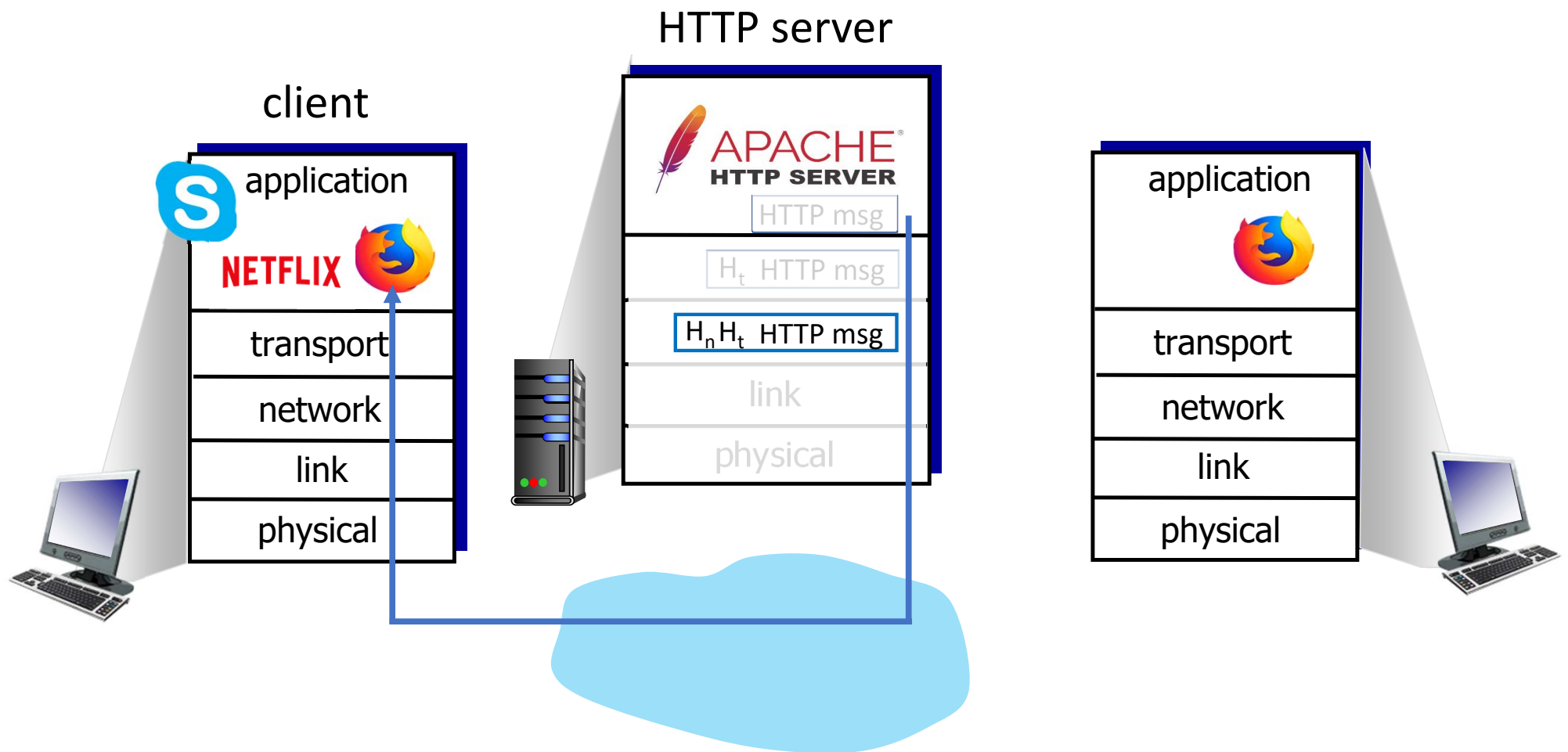 - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees

client

HTTP server

application

application

transport

transport

network

network

link

link

physical

physical

HTTP msg

H_t HTTP msg

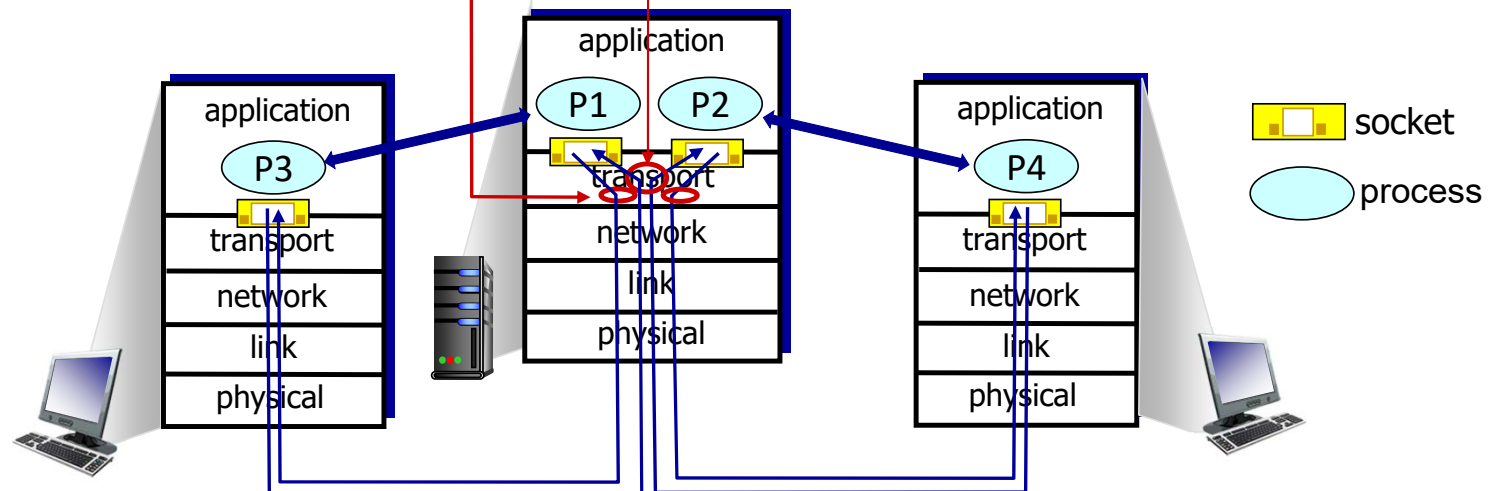$H_n H_t$ HTTP msg

link

physical

Multiplexing/demultiplexing

multiplexing at sender:

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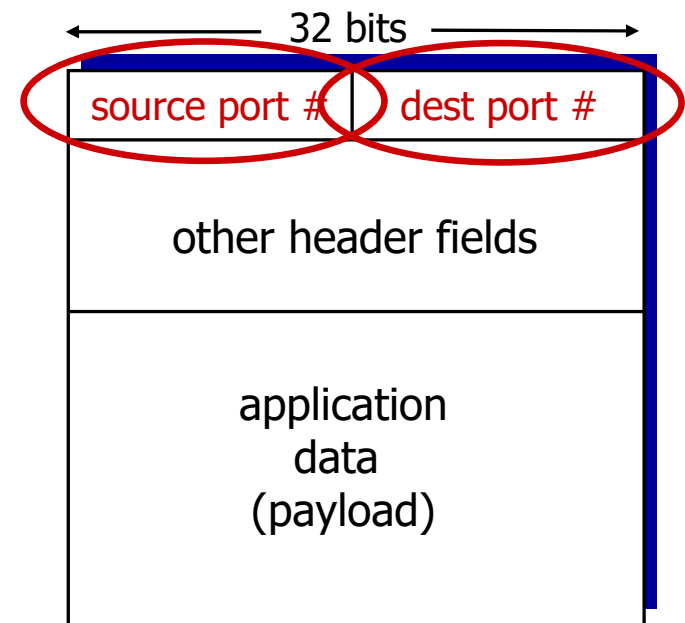
demultiplexing at receiver:

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

Recall:

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

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

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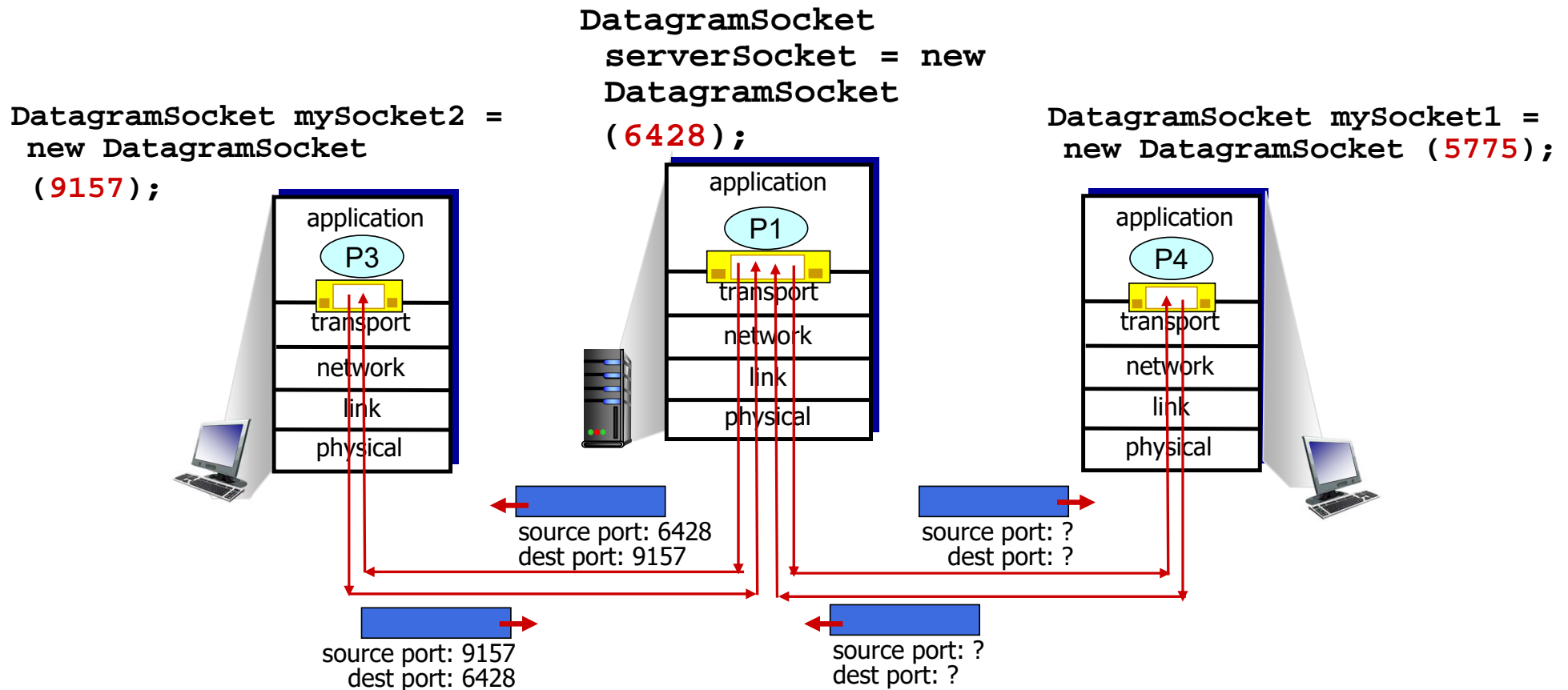 - destination IP address
  - destination port #

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

```
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: ?
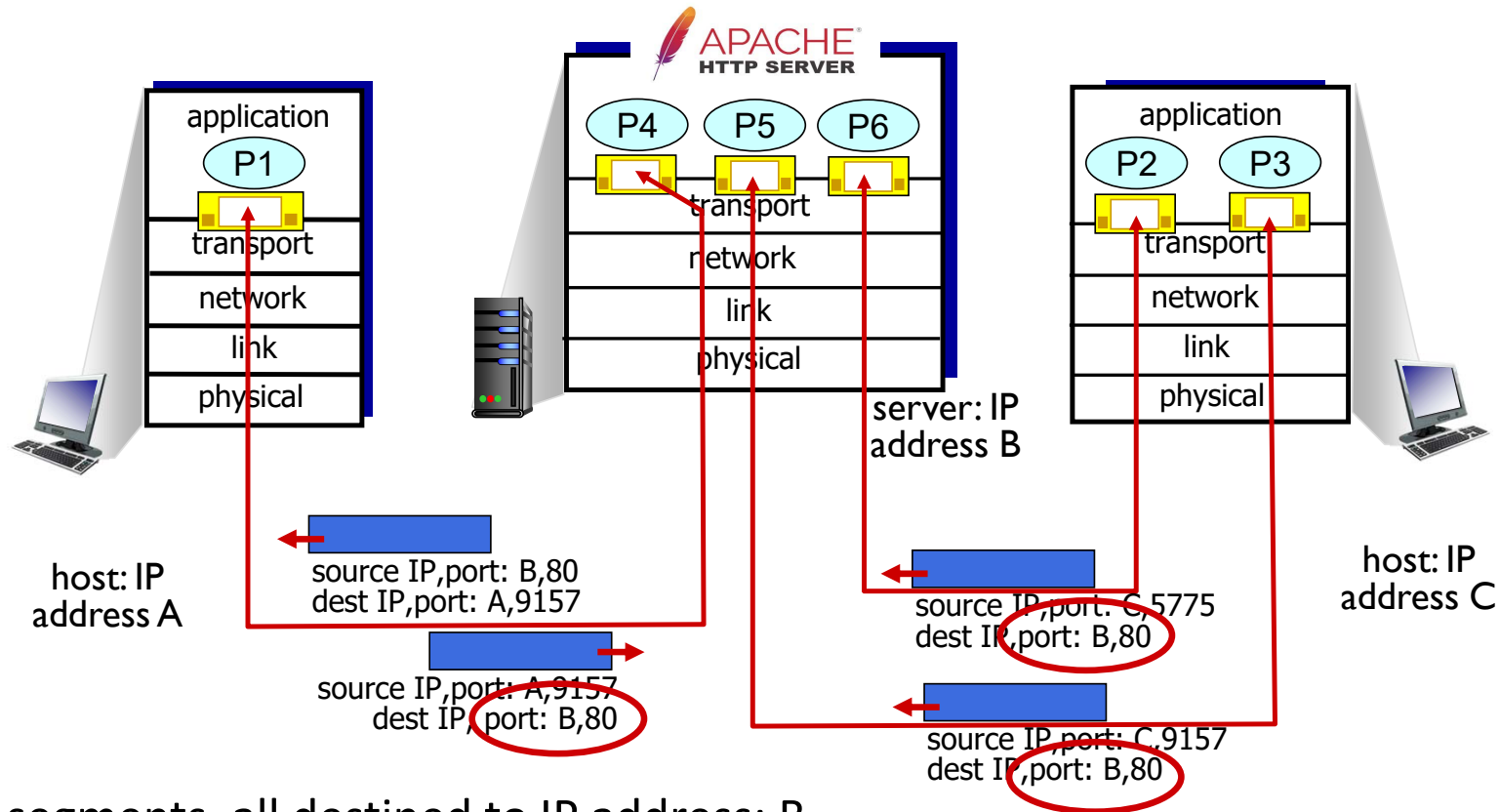
Connection-oriented demultiplexing

- 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

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

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

host: IP
address A

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

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

host: IP
address C

server: IP
address B

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

Chapter 3: roadmap

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

UDP: User Datagram Protocol

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

- no connection establishment (which can add RTT delay)
- simple: no connection state at sender, receiver
- 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 ...
     +---------------- ...
```
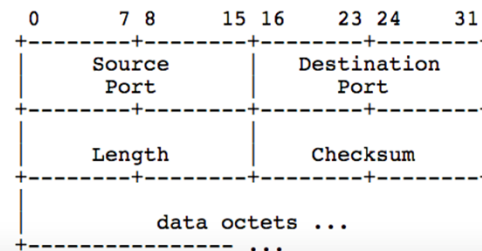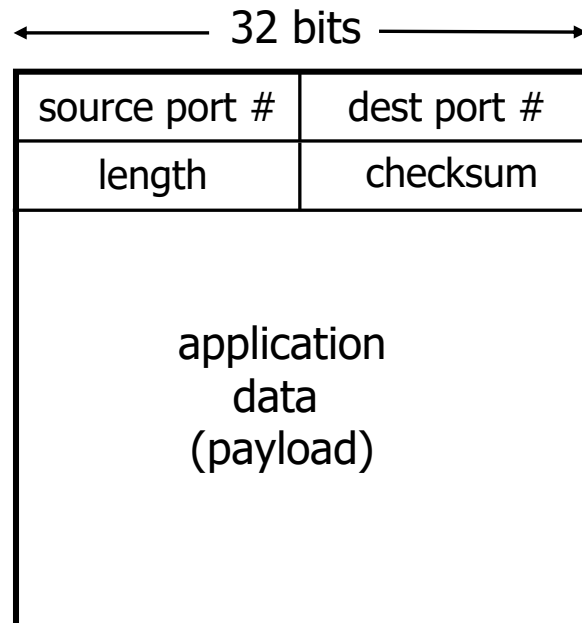
UDP segment header

32 bits

source port #	dest port #
length	checksum

application
data
(payload)

UDP segment format

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 (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
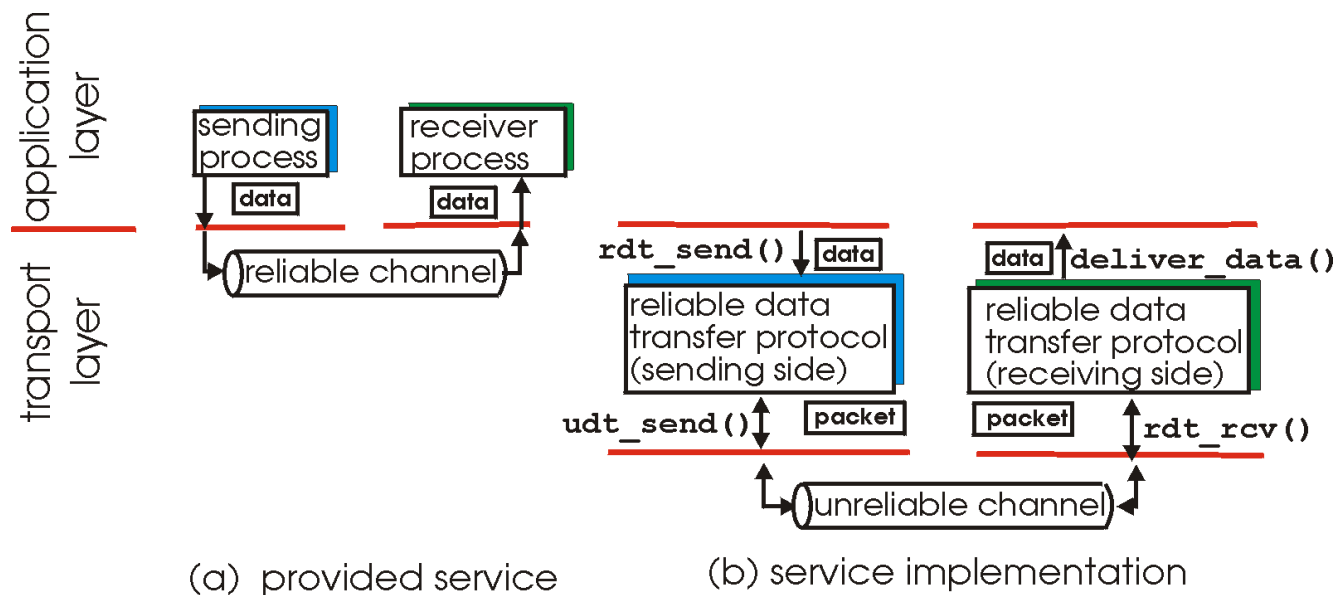
Internet checksum: weak protection!

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 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

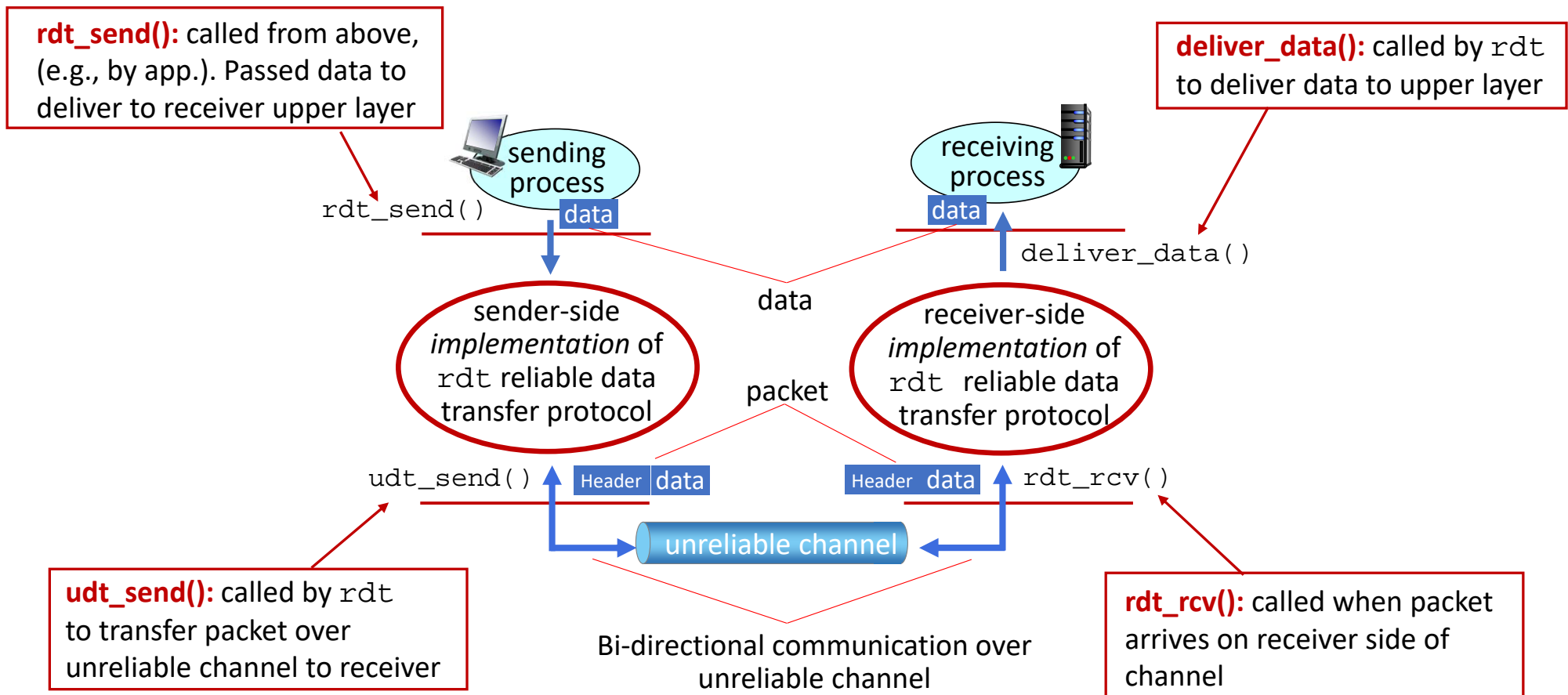Even though numbers have changed (bit flips), *no* change in checksum!

Principles of reliable data transfer

- **important in application, transport, link layers**
  - top-10 list of important networking topics!

transport   application
layer   layer

sending process   receiver process

data   data

reliable channel

`rdt_send()` data   data `deliver_data()`

reliable data transfer protocol (sending side)

reliable data transfer protocol (receiving side)

`udt_send()` packet   packet `rdt_rcv()`

unreliable channel

(a) provided service   (b) service implementation

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer protocol (rdt): interfaces

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

`rdt_send()`

sending process

data

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

`udt_send()`

Header | data

data

packet

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

unreliable channel

Bi-directional communication over unreliable channel

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

receiving process

data

`deliver_data()`

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

Header | data

`rdt_rcv()`

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

3-21

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

Problems!

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

Discussion

sender:

- seq # added to pkt
- two seq. #s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted

receiver:

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

NAK-free protocol

- same functionality, 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

Channels with errors *and* loss

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

- checksum, seq. #, ACKs, retransmissions will be of help … but not enough

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
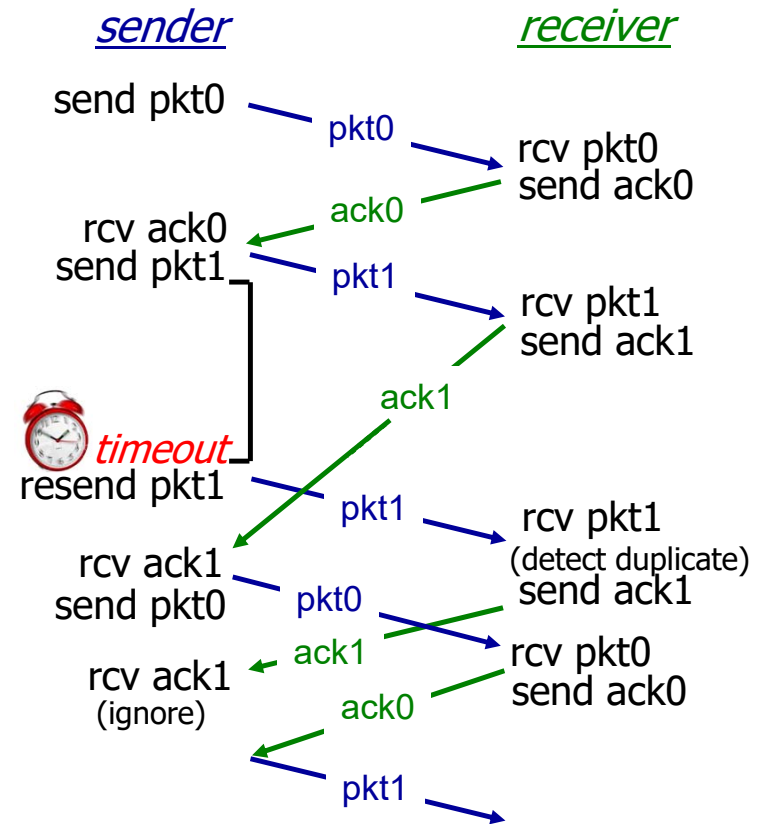 - receiver must specify seq # of pkt being ACKed
- requires countdown timer

Stop & Wait

sender   *receiver*

send pkt0
     pkt0
                  rcv pkt0
                  send ack0
     ack0
rcv ack0
send pkt1
     pkt1
                  rcv pkt1
     ack1   send ack1
rcv ack1
send pkt0
     pkt0
                  rcv pkt0
     ack0   send ack0

(a) no loss

sender   *receiver*

send pkt0
     pkt0
                  rcv pkt0
                  send ack0
     ack0
rcv ack0
send pkt1
     pkt1 **X**
        loss

timeout
resend pkt1
     pkt1
                  rcv pkt1
     ack1   send ack1
rcv ack1
send pkt0
     pkt0
                  rcv pkt0
     ack0   send ack0

(b) packet loss

Stop & Wait

sender   *receiver*

send pkt0   pkt0

                       rcv pkt0
                       send ack0

             ack0

rcv ack0
send pkt1   pkt1

                       rcv pkt1
          ack1   send ack1

           X
          loss

timeout
resend pkt1   pkt1

                       rcv pkt1
                       (detect duplicate)
          ack1   send ack1

rcv ack1
send pkt0   pkt0

                       rcv pkt0
          ack0   send ack0

(c) ACK loss

sender   *receiver*

send pkt0   pkt0

                       rcv pkt0
                       send ack0

rcv ack0   ack0
send pkt1   pkt1

                       rcv pkt1
                       send ack1

          ack1

timeout
resend pkt1

               pkt1   rcv pkt1
rcv ack1   (detect duplicate)
send pkt0   pkt0   send ack1

          ack1
rcv ack1   rcv pkt0
(ignore)   ack0   send ack0

               pkt1

(d) premature timeout/ delayed ACK

3-28

Performance

- Stop & wait is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

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

- U $_{sender}$: *utilization* – fraction of time sender busy sending

$$U_{sender} = \frac{L \ / \ R}{RTT + L \ / \ R} = \frac{.008}{30.008} = 0.00027$$

- if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!

Stop-and-wait operation

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

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

$$= 0.00027$$
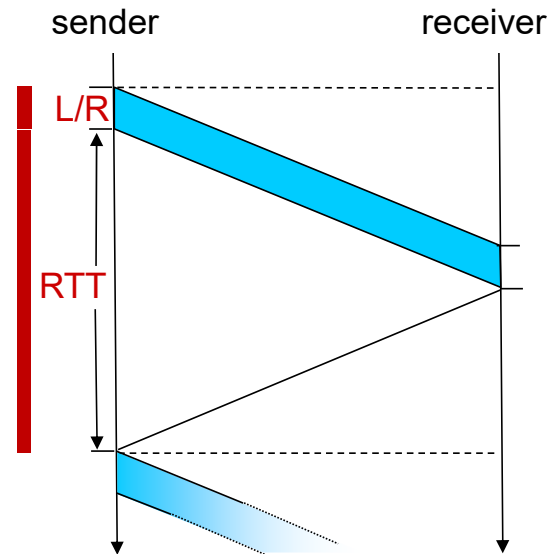
- Stop-and-wait performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

Pipelined protocols

pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver
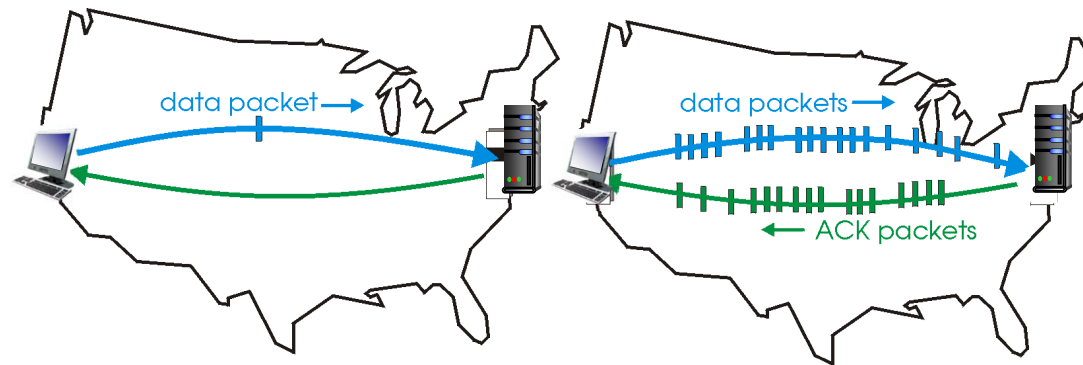
(a) a stop-and-wait protocol in operation   (b) a pipelined protocol in operation

- two generic forms of pipelined protocols: *go-Back-N, selective repeat*

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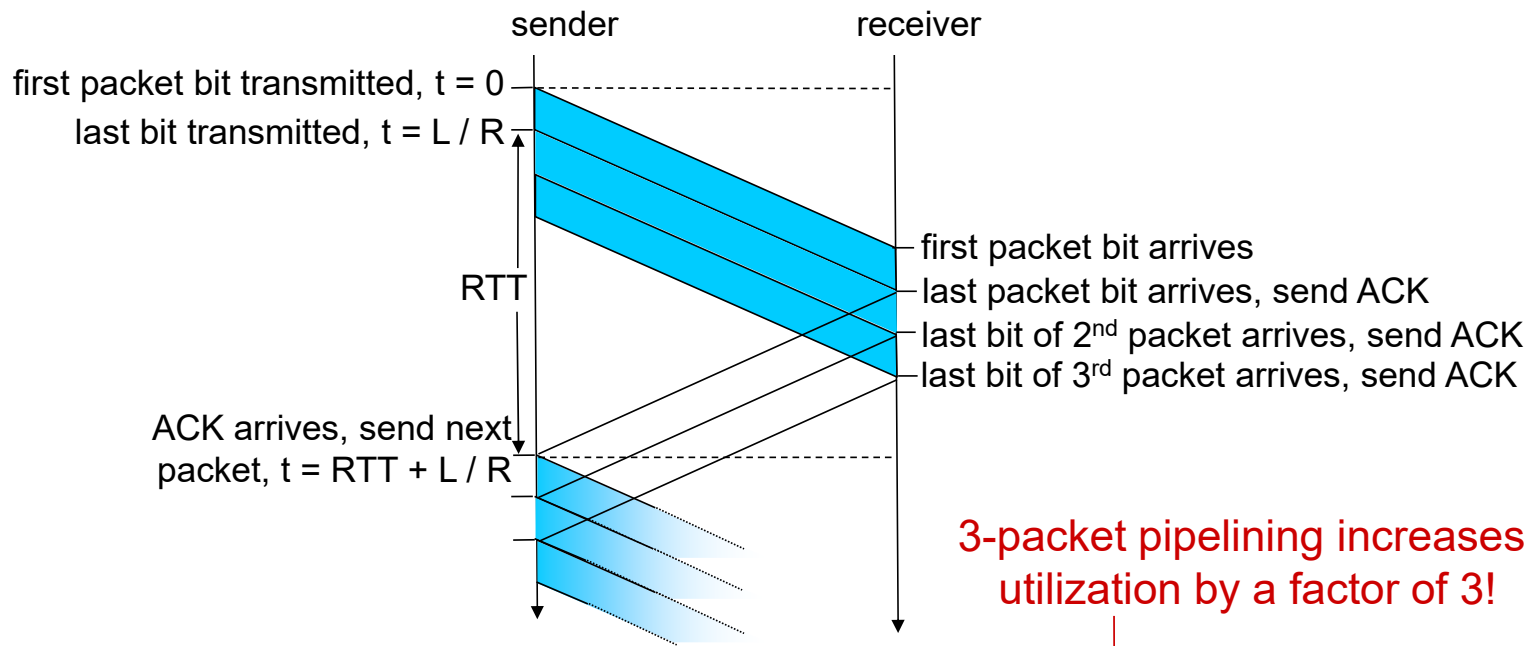
3-packet pipelining increases utilization by a factor of 3!

$$ U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081 $$

Go-Back-N: sender

- sender: "window" of up to N, consecutive transmitted but unACKed pkts
  - k-bit seq # in pkt header
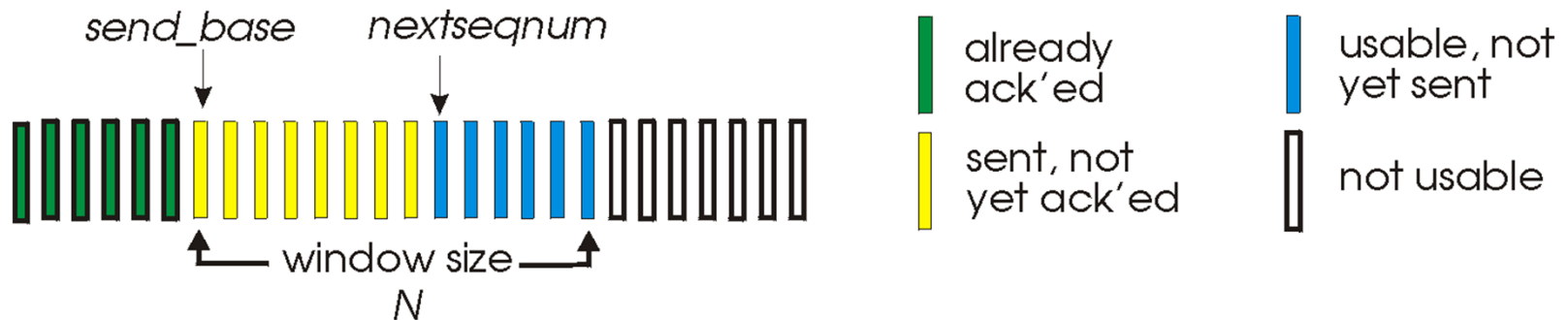
- *cumulative ACK:* ACK(n): ACKs all packets up to, including seq # n
  - on receiving ACK(n): move window forward to begin at $n+1$
- timer for oldest in-flight packet
- *timeout(n):* retransmit packet n and all higher seq # packets in window

Go-Back-N: receiver

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

Receiver view of sequence number space:

received and ACKed

Out-of-order: received but not ACKed

Not received

rcv_base
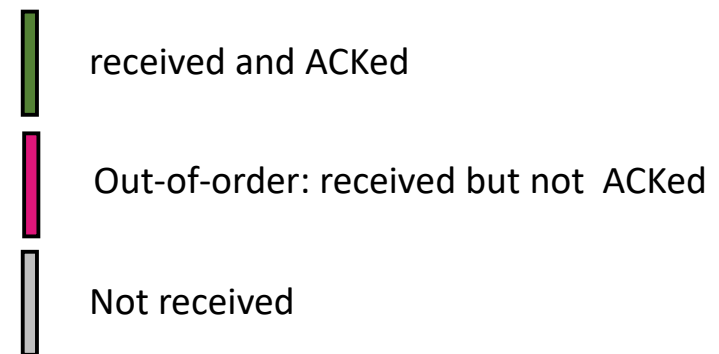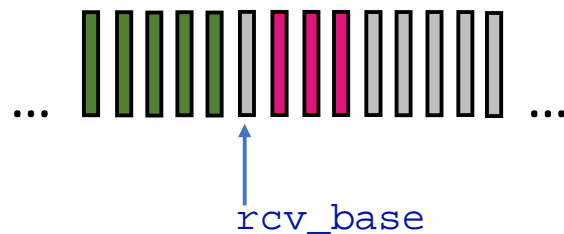
Go-Back-N in action
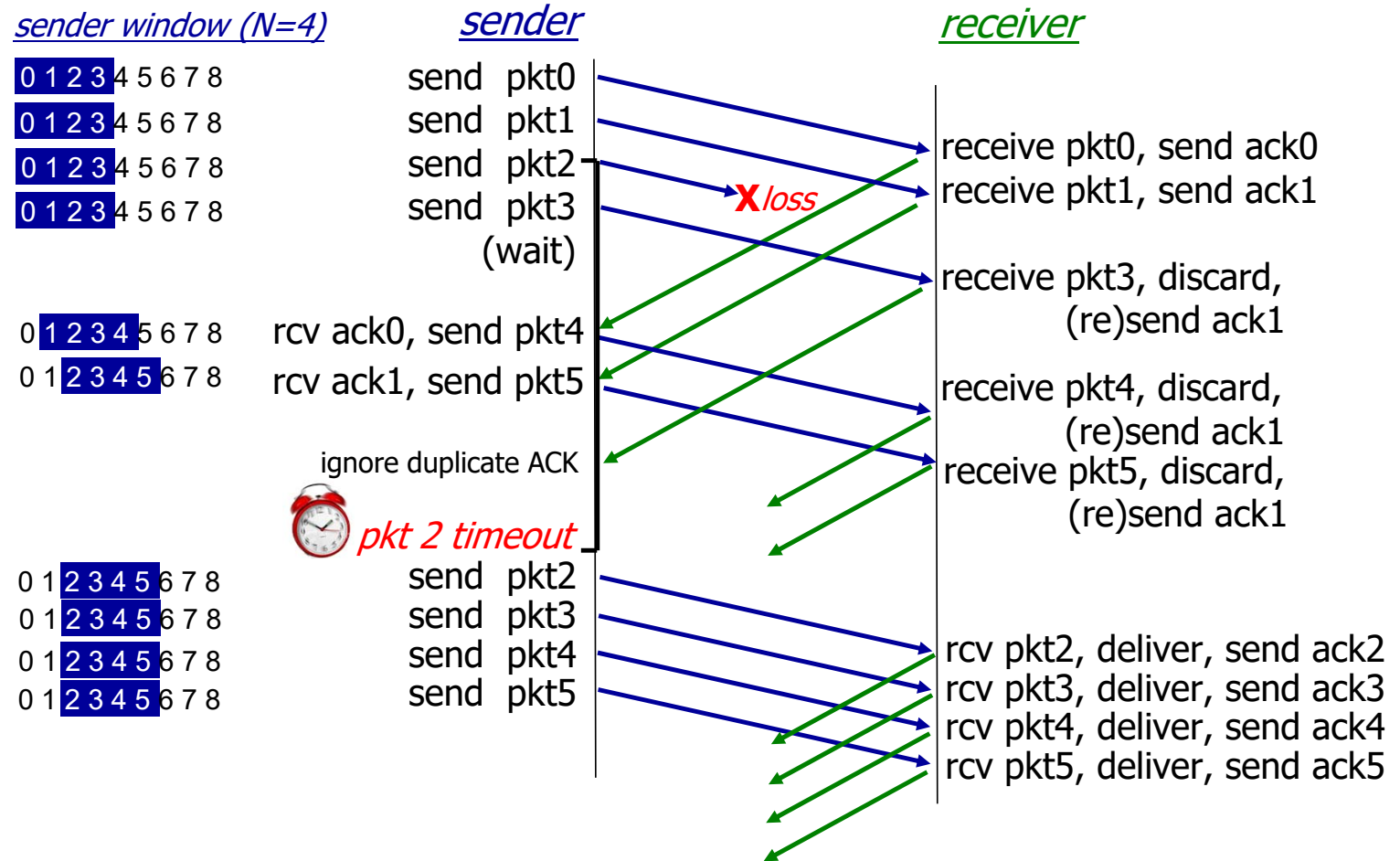
Selective repeat

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

- sender times-out/retransmits individually for unACKed packets
  - sender maintains timer for each unACKed pkt

- sender window
  - *N* consecutive seq #s
  - limits seq #s of sent, unACKed packets

Selective repeat: sender, receiver windows

send_base   nextseqnum

already ack'ed

sent, not yet ack'ed

usable, not yet sent

not usable

window size N

(a) sender view of sequence numbers

out of order (buffered) but already ack'ed

Expected, not yet received

acceptable (within window)

not usable

window size N

rcv_base

(b) receiver view of sequence numbers
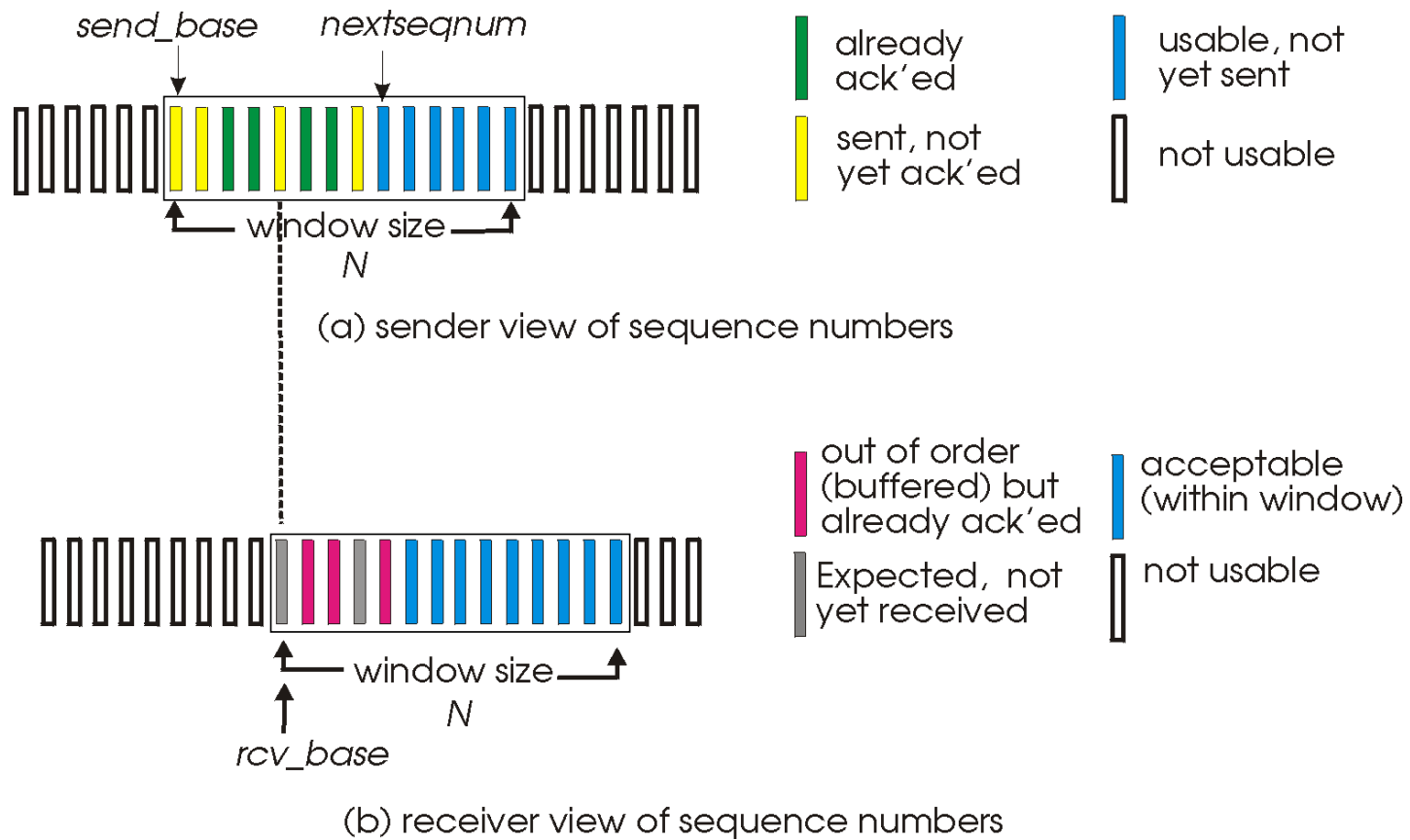
Selective repeat: sender and receiver

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]:

- 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	
0 1 2 3 4 5 6 7 8	
0 1 2 3 4 5 6 7 8	
0 1 2 3 4 5 6 7 8	

send pkt0

send pkt1

send pkt2

send pkt3

(wait)

X *loss*

receive pkt0, send ack0

receive pkt1, send ack1

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
receive pkt5, buffer,
send ack5

record ack3 arrived

pkt 2 timeout

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

send pkt2
(but not 3,4,5)

rcv pkt2; deliver pkt2,
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

sender window
(after receipt)

receiver window
(after receipt)

(a) no problem

will accept packet
with seq number 0

(b) oops!

will accept packet
with seq number 0

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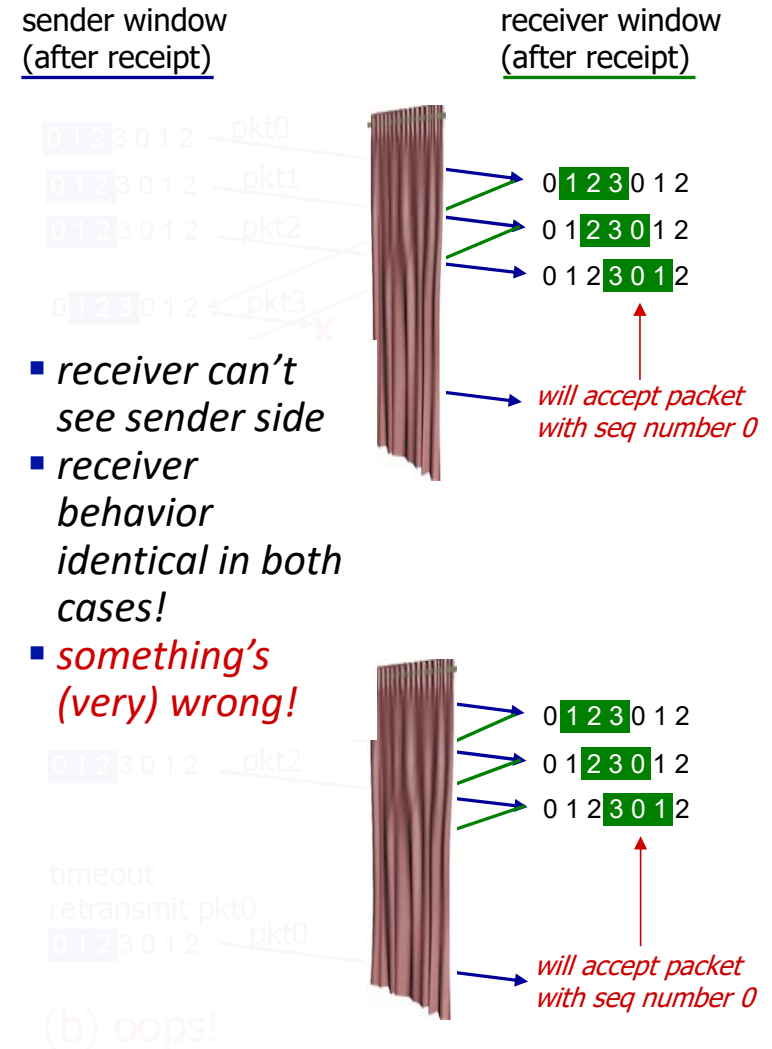
- *receiver can't see sender side*
- *receiver behavior identical in both cases!*
- *something's (very) wrong!*

will accept packet with seq number 0

will accept packet with seq number 0

(b) oops!

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

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)	

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

length (of TCP header)

Internet checksum

C, E: congestion notification

TCP options

RST, SYN, FIN: connection management

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

flow control: # bytes receiver willing to accept

data sent by application into TCP socket

TCP sequence numbers, ACKs

Sequence numbers:

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

Acknowledgements:

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

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
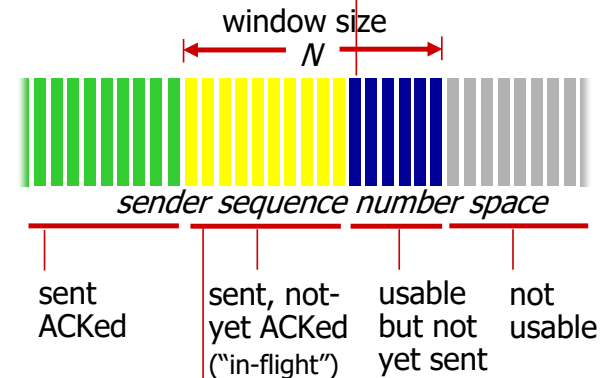
outgoing segment from receiver

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
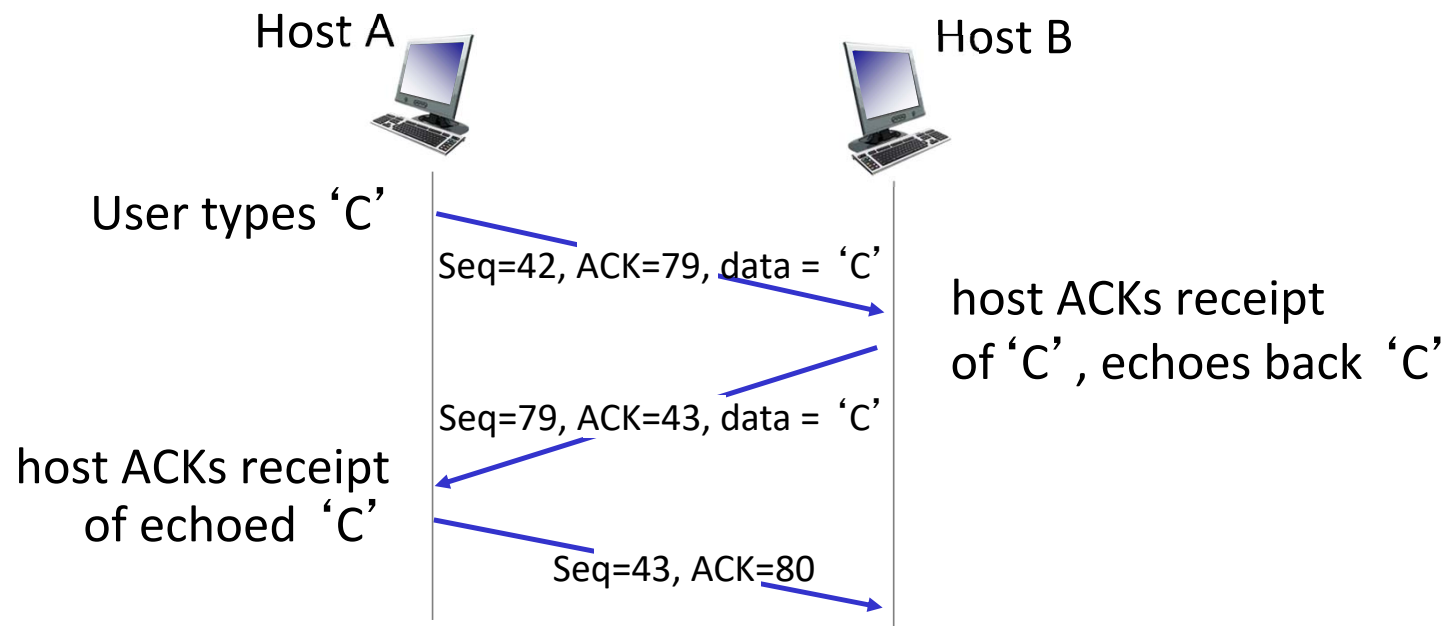
simple telnet scenario

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

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

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

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

◆ sampleRTT
■ EstimatedRTT

RTT (milliseconds) vs time (seconds)
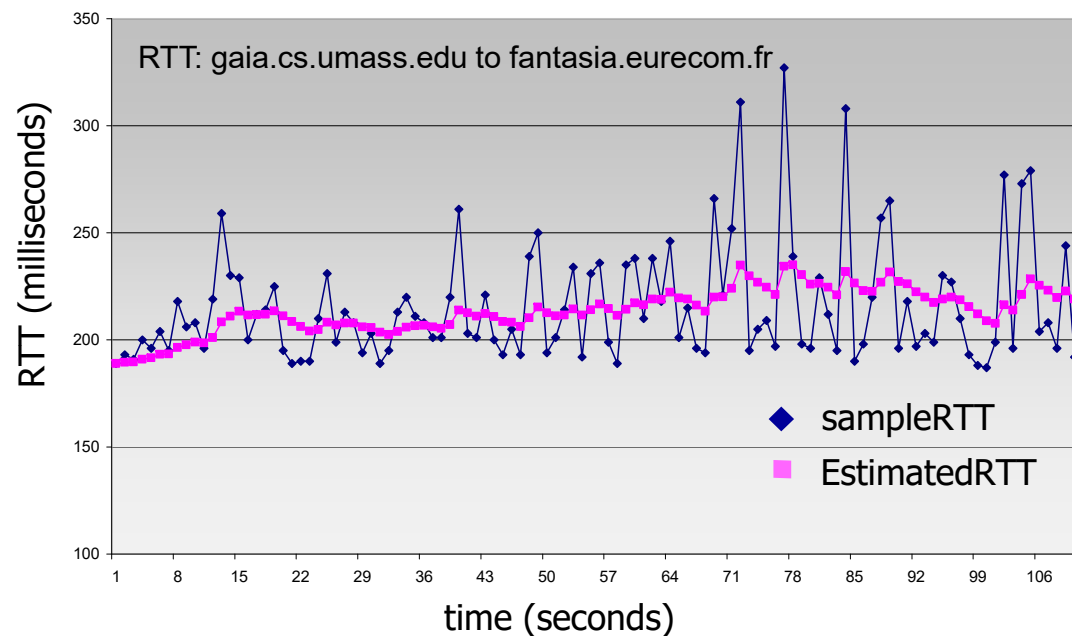
TCP round trip time, timeout

- 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**:

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

(typically, β = 0.25)

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 sender (simplified)

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

Λ
_____
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

wait
for
event

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
   }

3-51

TCP: retransmission scenarios

lost ACK scenario
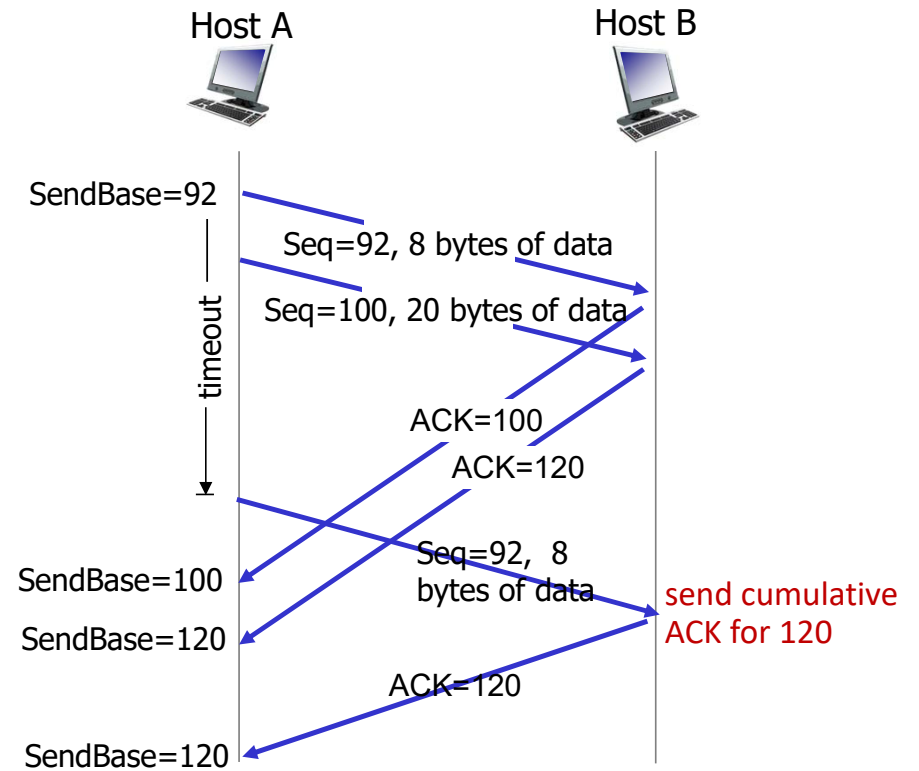
premature timeout

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
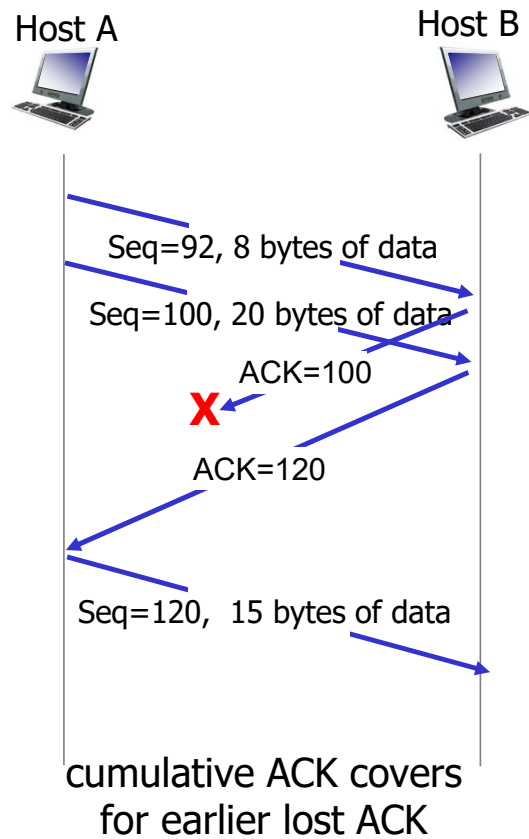
TCP ACK generation [RFC 1122, RFC 2581]

event at receiver	TCP receiver action
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send duplicate ACK, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

TCP fast retransmit

TCP fast retransmit

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

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

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
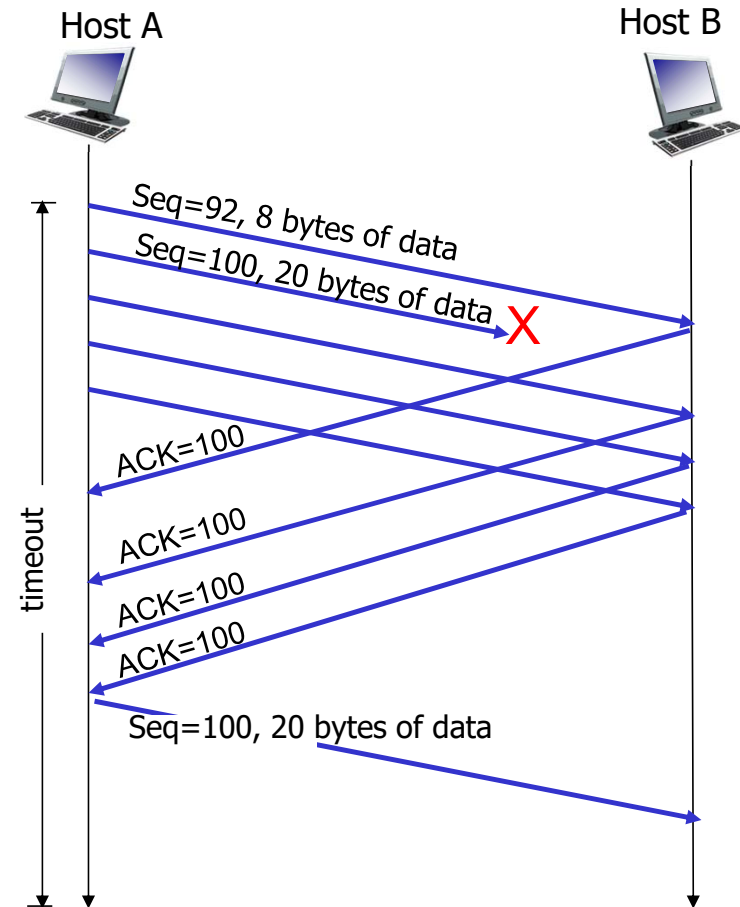
TCP flow control

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 autoadjust **RcvBuffer**

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

- guarantees receive buffer will not overflow

to application process

RcvBuffer

buffered data

rwnd

free buffer space

TCP segment payloads

TCP receiver-side buffering
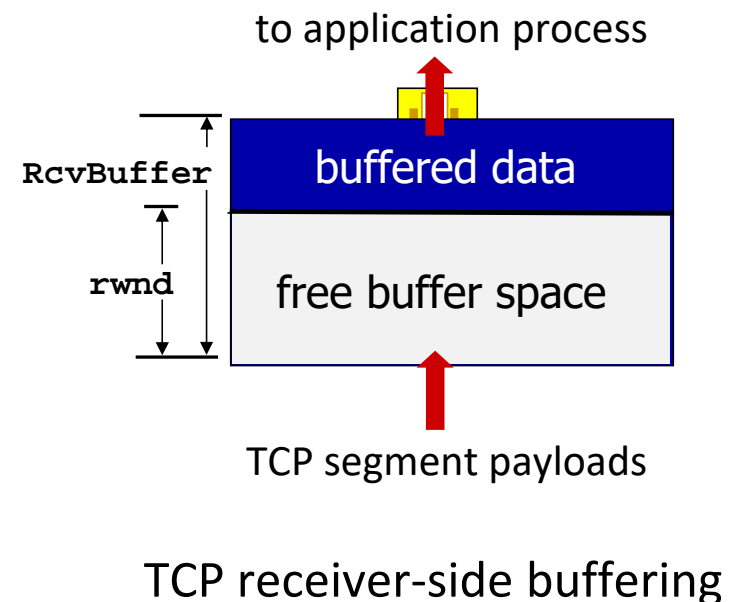
TCP flow control

- 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 autoadjust **RcvBuffer**

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

- guarantees receive buffer will not overflow

flow control: # bytes receiver willing to accept

receive window
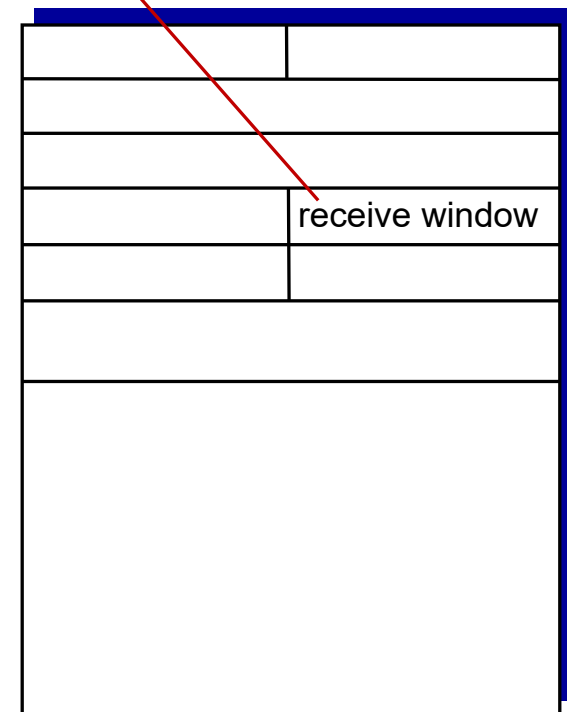
TCP segment format

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
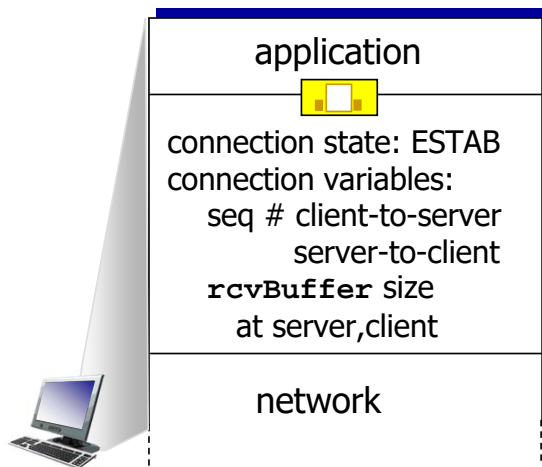
network

```
Socket clientSocket =
   newSocket("hostname","port number");
```
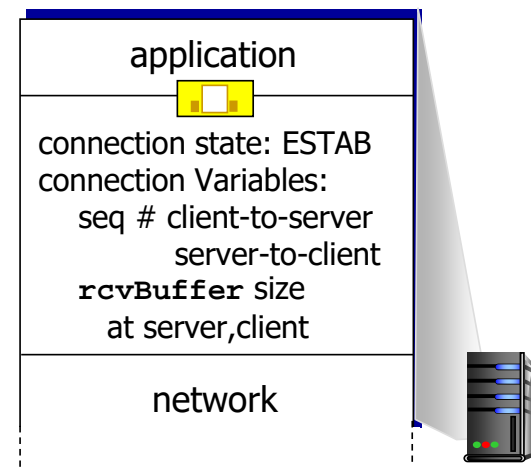
```
Socket connectionSocket =
   welcomeSocket.accept();
```

Agreeing to establish a connection
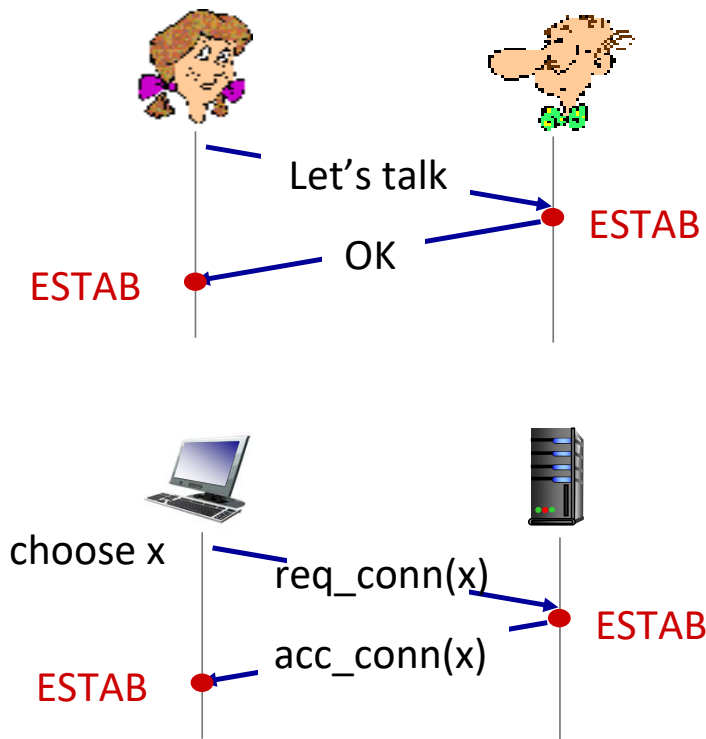
2-way handshake:

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

Agreeing to establish a connection

2-way handshake failure scenarios:

TCP 3-way handshake

Server state

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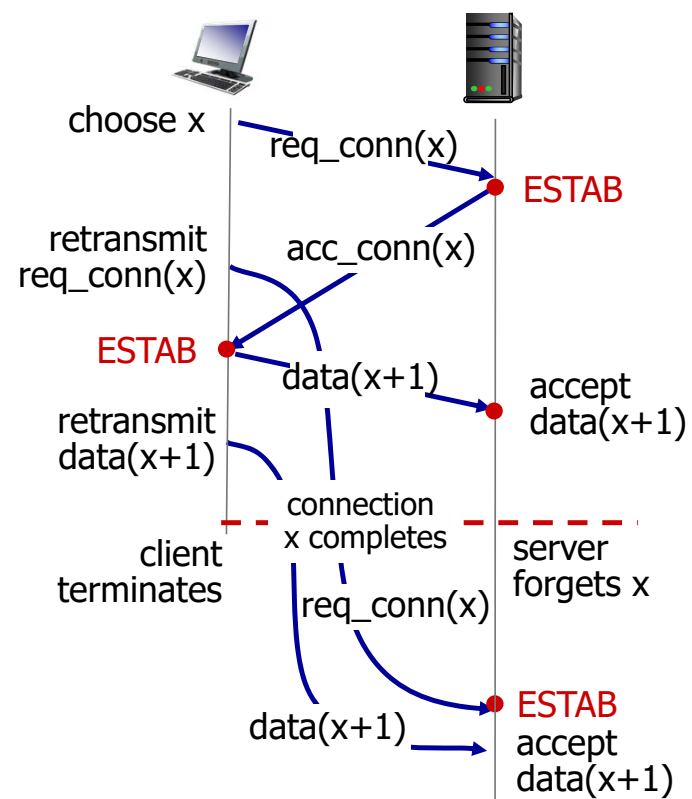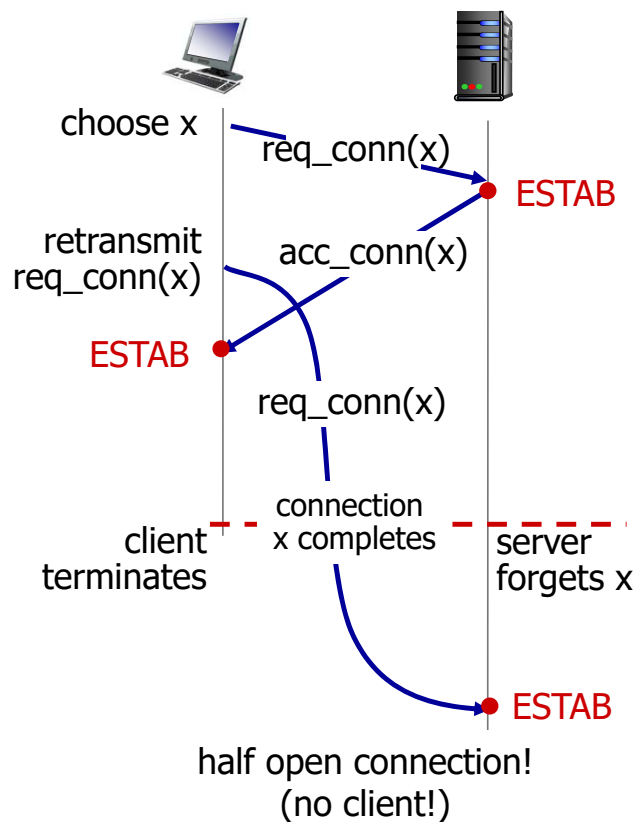
Client state

```
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

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

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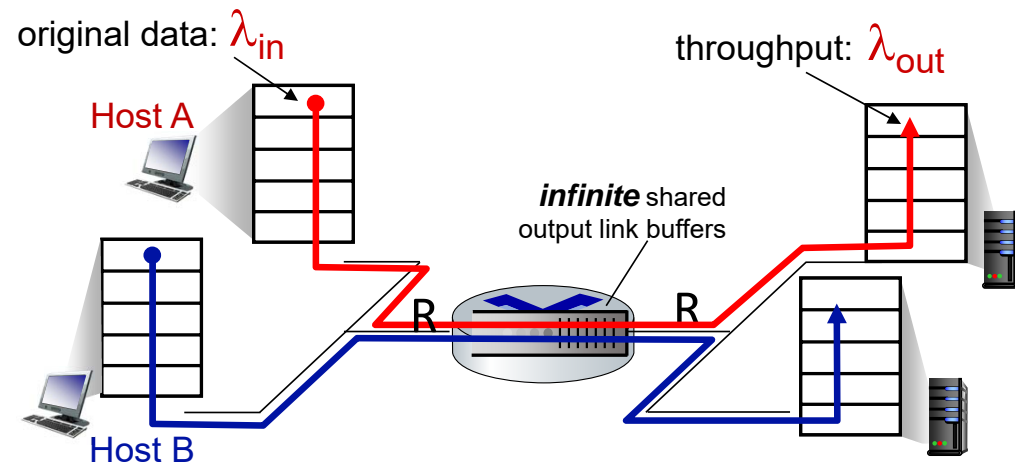
Q: What happens as arrival rate λ_{in} approaches R/2?

throughput: λ_{out}

R/2

λ_{in} R/2
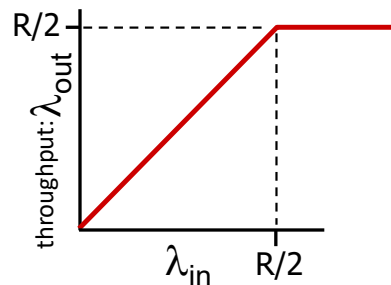
maximum per-connection throughput: R/2

delay

λ_{in} R/2
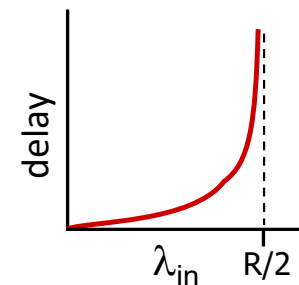
large delays as arrival rate λ_{in} approaches capacity

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}

R

R

Host B

finite shared output link buffers
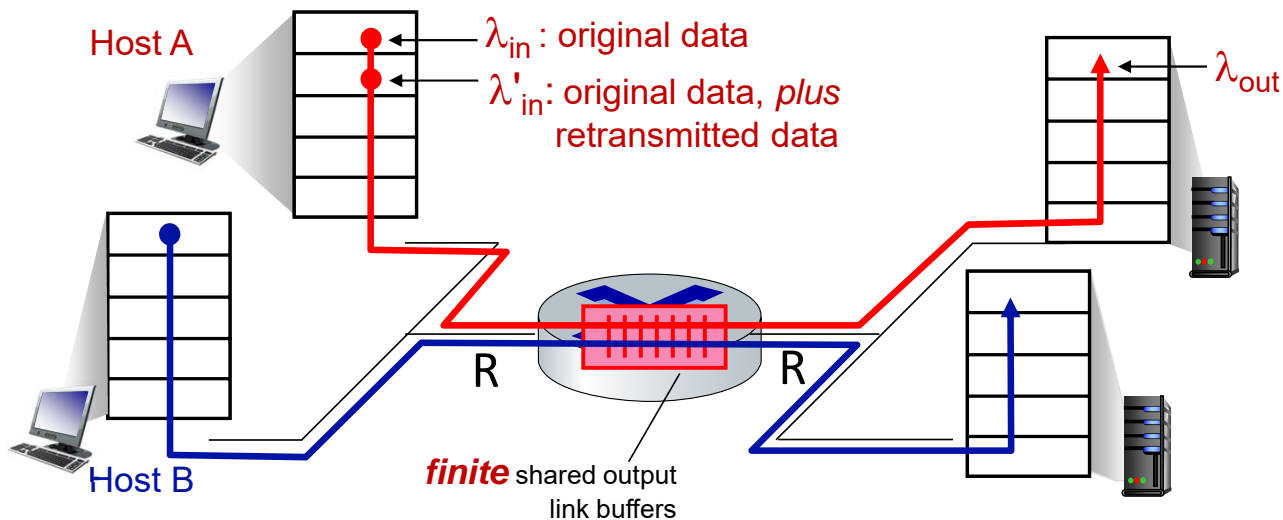
Causes/costs of congestion: scenario 2

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

throughput: λ_{out}

R/2

λ_{in}

R/2

Causes/costs of congestion: scenario 2
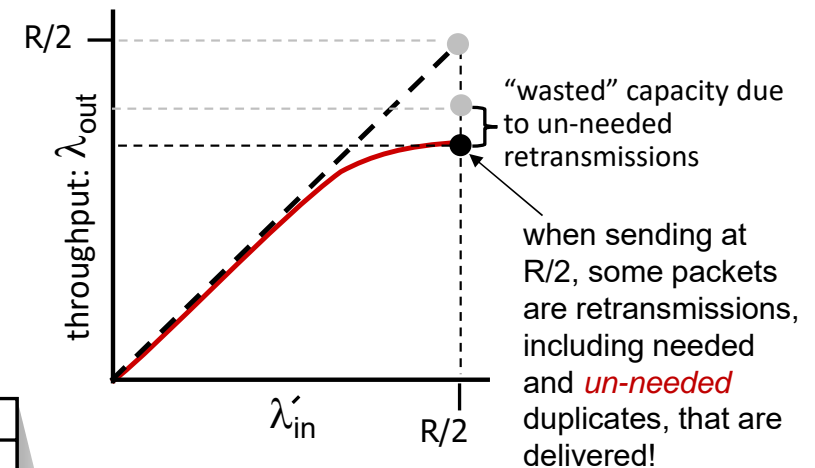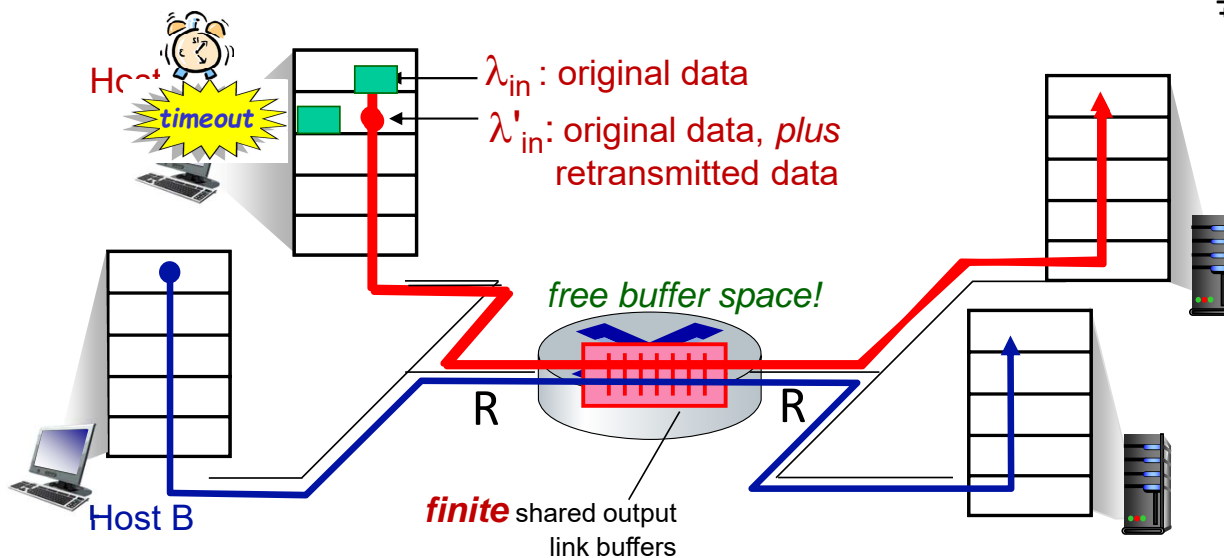
Idealization: *some* perfect knowledge

- 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

no buffer space!

R   R

Host B

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

R/2 — "wasted" capacity due to retransmissions

throughput: λ_{out}

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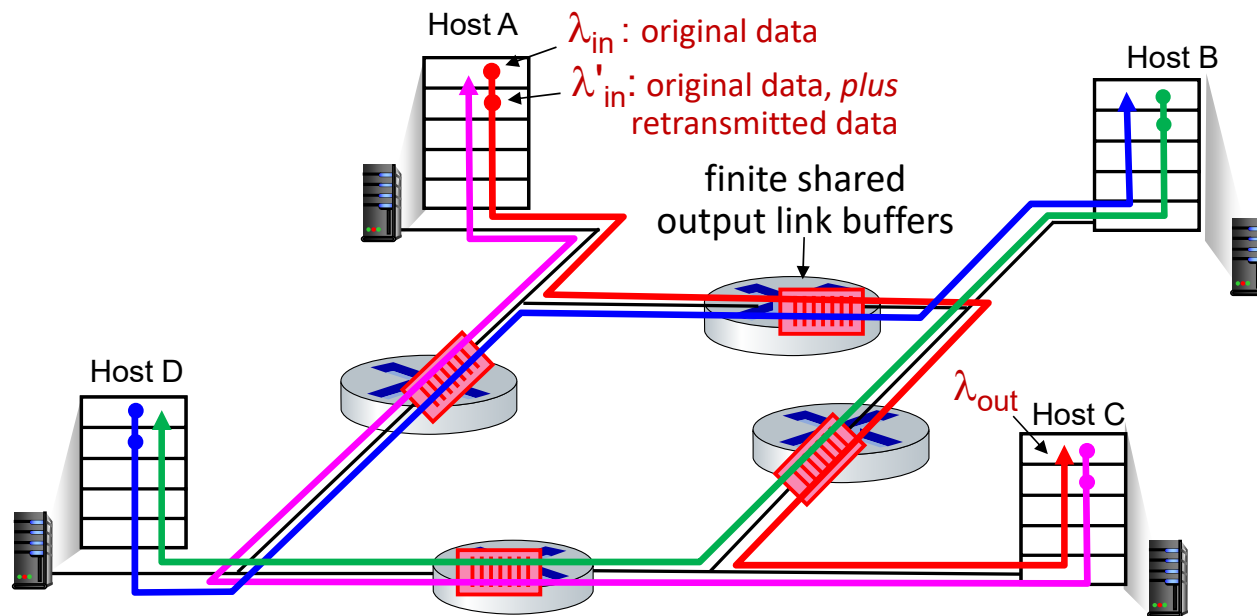- but sender times can time out prematurely, sending *two* copies, *both* of which are delivered

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

when sending at R/2, some packets are retransmissions, including needed and *un-needed* duplicates, that are delivered!

λ'_{in} R/2
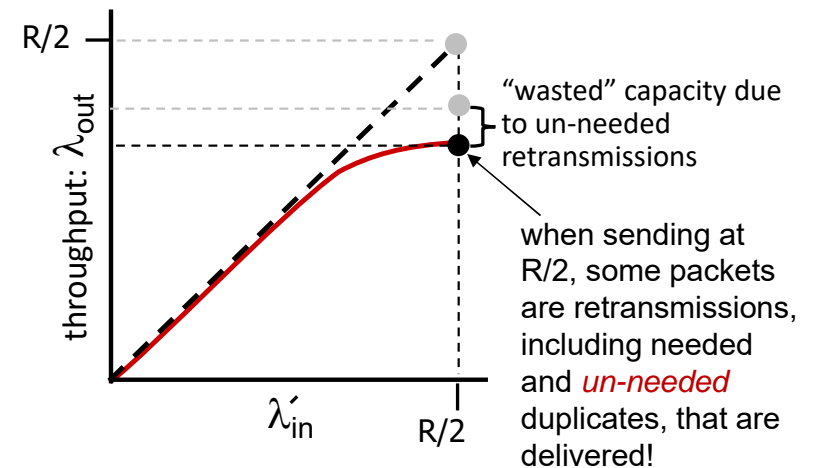
"costs" of congestion:

- 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 $\rightarrow 0$

Host A

λ_{in} : original data

λ_{in}' : original data, *plus* retransmitted data

Host B

finite shared output link buffers

Host D

λ_{out} Host C

3-72

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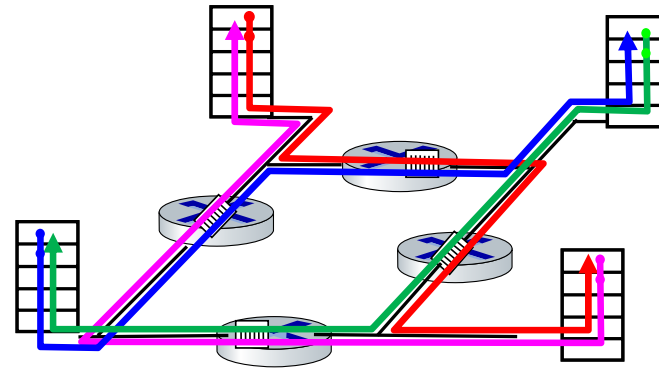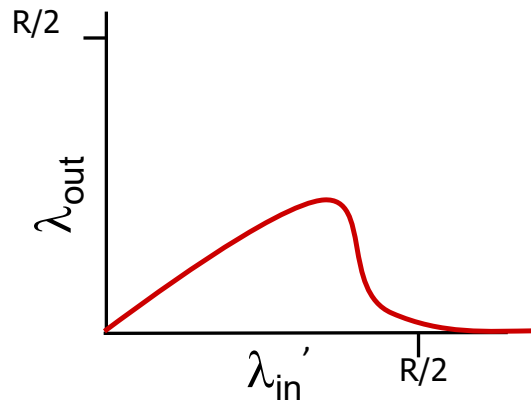
another "cost" of congestion:

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

Approaches towards congestion control

End-end congestion control:

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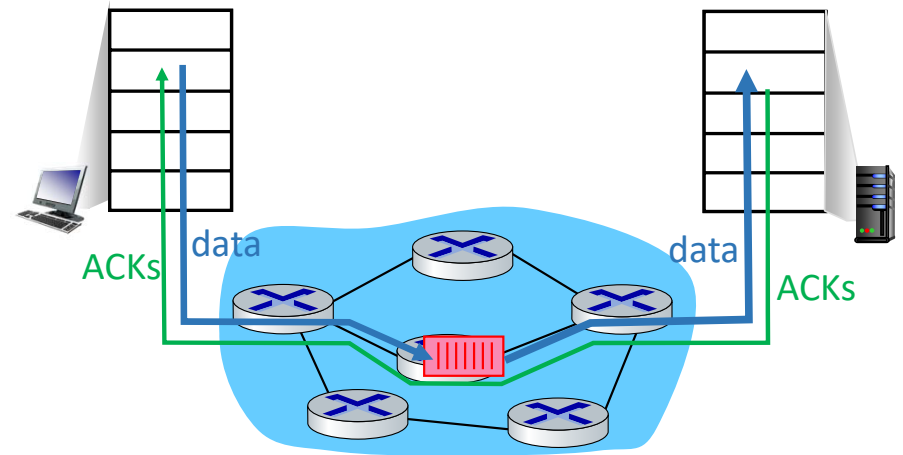
Approaches towards congestion control

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
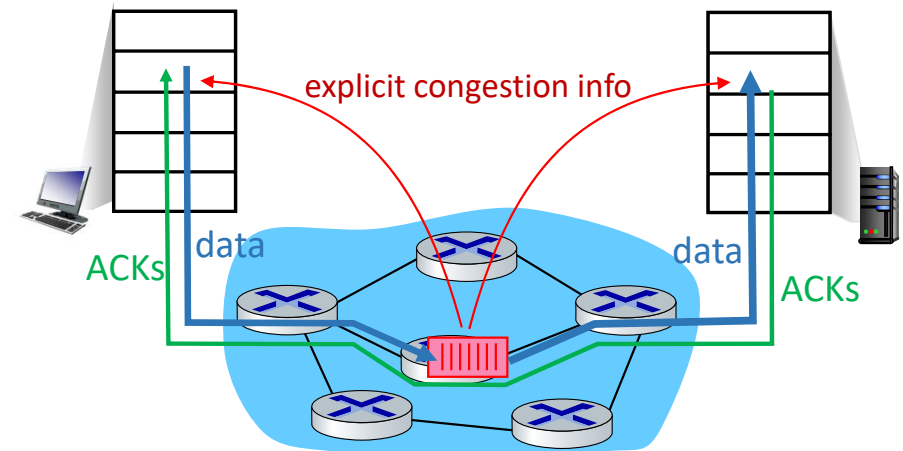
explicit congestion info

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

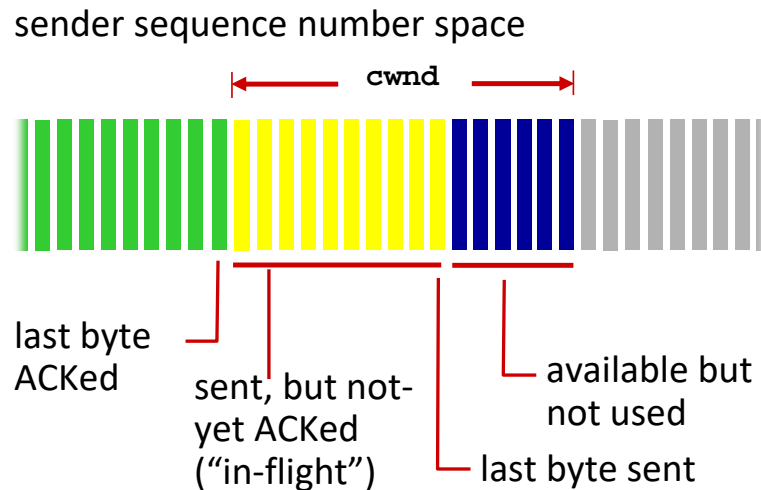increase sending rate by 1 maximum segment size every RTT until loss detected

Multiplicative Decrease

cut sending rate in half at each loss event
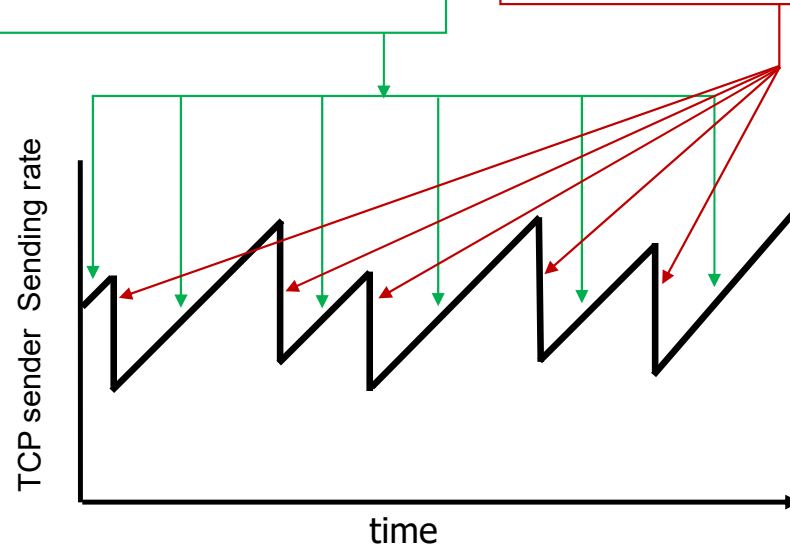
AIMD sawtooth behavior: *probing* for bandwidth

TCP sender Sending rate

time

TCP congestion control: details

sender sequence number space

cwnd

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 $\approx \dfrac{\text{cwnd}}{\text{RTT}}$ 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

Host A   Host B

one segment

two segments

four segments
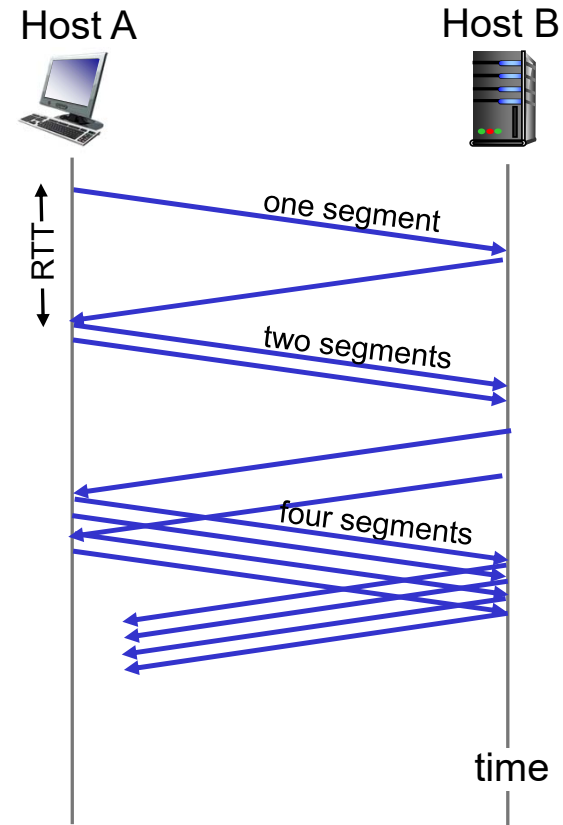
time

TCP: detecting, reacting to loss

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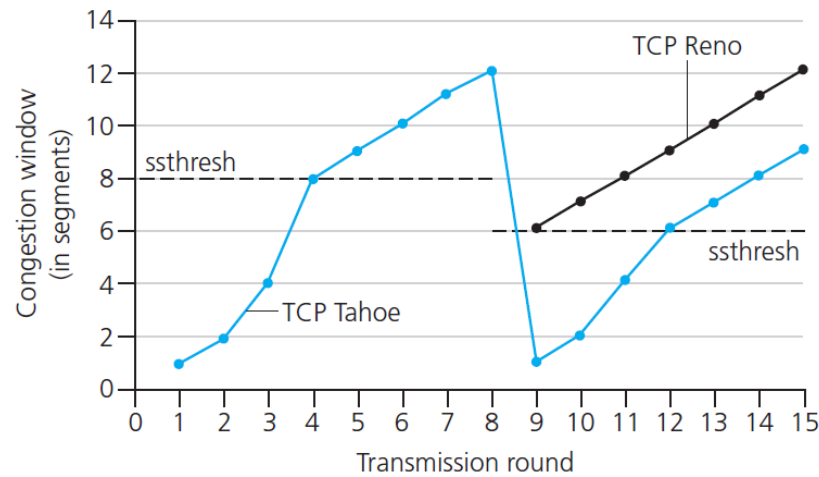- TCP Tahoe always sets **cwnd** to 1 (timeout or 3 duplicate acks)

TCP: switching from slow start to CA

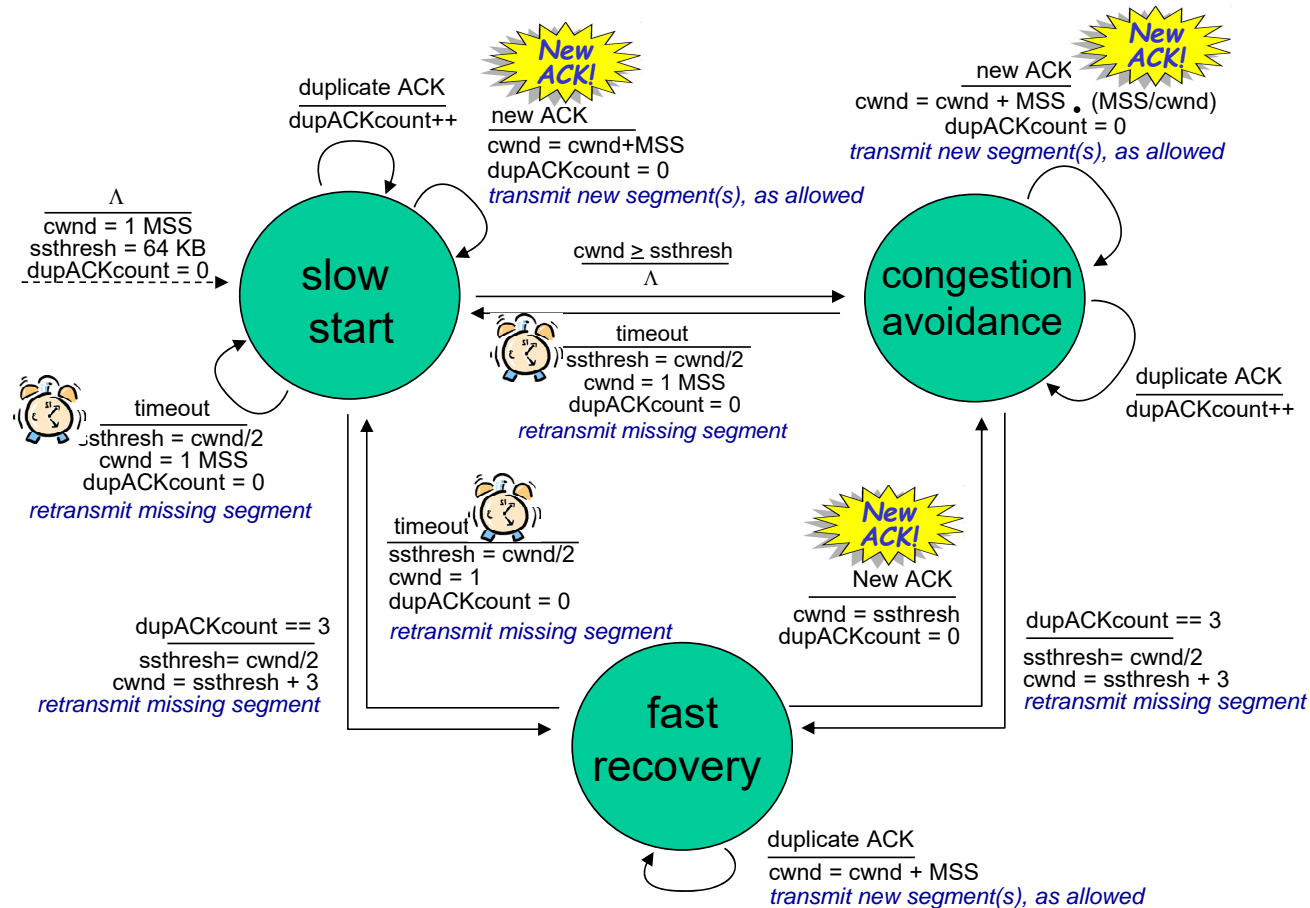Q: when should the exponential increase switch to linear?

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

Implementation:

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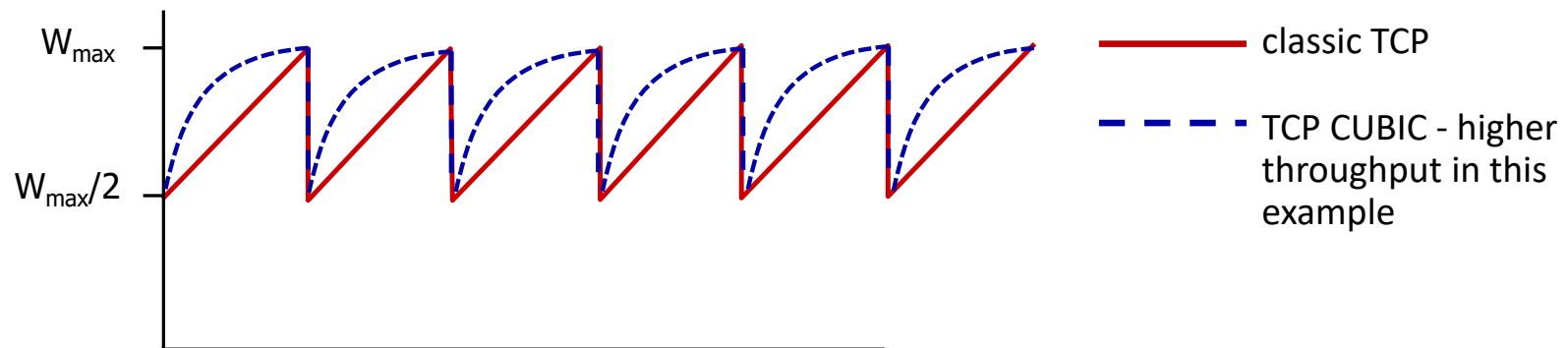- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event

Summary: TCP congestion control

New ACK!

duplicate ACK
———————
dupACKcount++

new ACK
———————
cwnd = cwnd+MSS
dupACKcount = 0
transmit new segment(s), as allowed

Λ
———————
cwnd = 1 MSS
ssthresh = 64 KB
dupACKcount = 0

slow start

timeout
———————
ssthresh = cwnd/2
cwnd = 1 MSS
dupACKcount = 0
retransmit missing segment

cwnd ≥ ssthresh
———————
Λ

timeout
———————
ssthresh = cwnd/2
cwnd = 1 MSS
dupACKcount = 0
retransmit missing segment

New ACK!

new ACK
———————
cwnd = cwnd + MSS · (MSS/cwnd)
dupACKcount = 0
transmit new segment(s), as allowed

congestion avoidance

duplicate ACK
———————
dupACKcount++

timeout
———————
ssthresh = cwnd/2
cwnd = 1
dupACKcount = 0
retransmit missing segment

dupACKcount == 3
———————
ssthresh= cwnd/2
cwnd = ssthresh + 3
retransmit missing segment

New ACK!

New ACK
———————
cwnd = ssthresh
dupACKcount = 0

dupACKcount == 3
———————
ssthresh= cwnd/2
cwnd = ssthresh + 3
retransmit missing segment

fast recovery

duplicate ACK
———————
cwnd = cwnd + MSS
transmit new segment(s), as allowed

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*
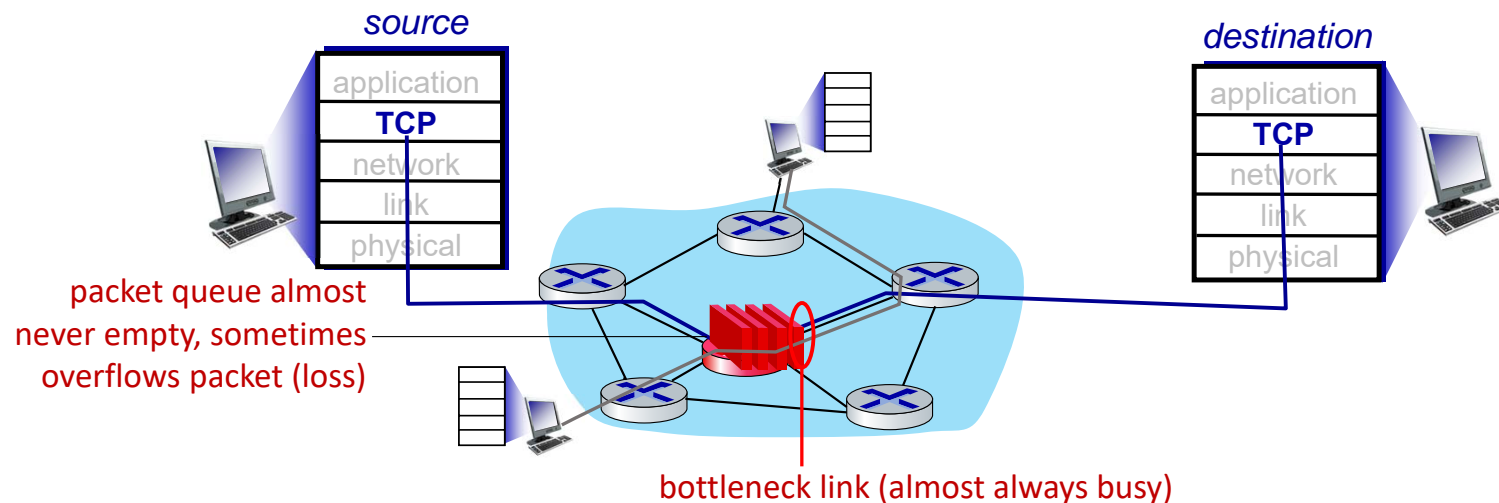

W_{max}

$W_{max}/2$

—— 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 tuneable
- increase W as a function of the *cube* of the distance between current time and K
  - larger increases when further away from K
  - smaller increases (cautious) when nearer K

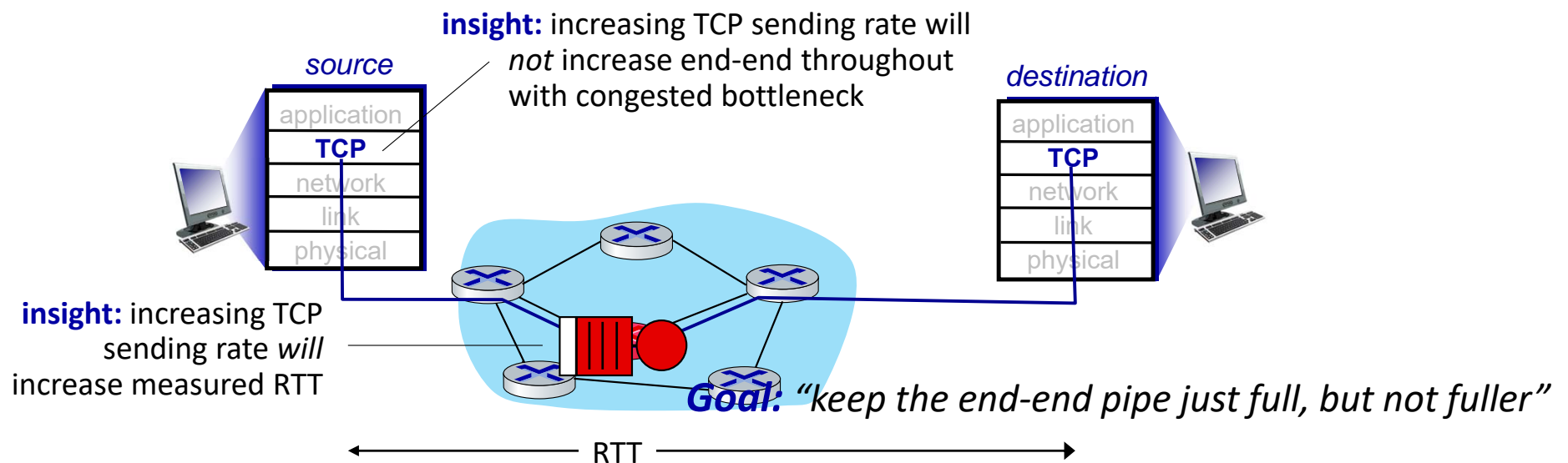- TCP CUBIC default in Linux, most popular TCP for popular Web servers
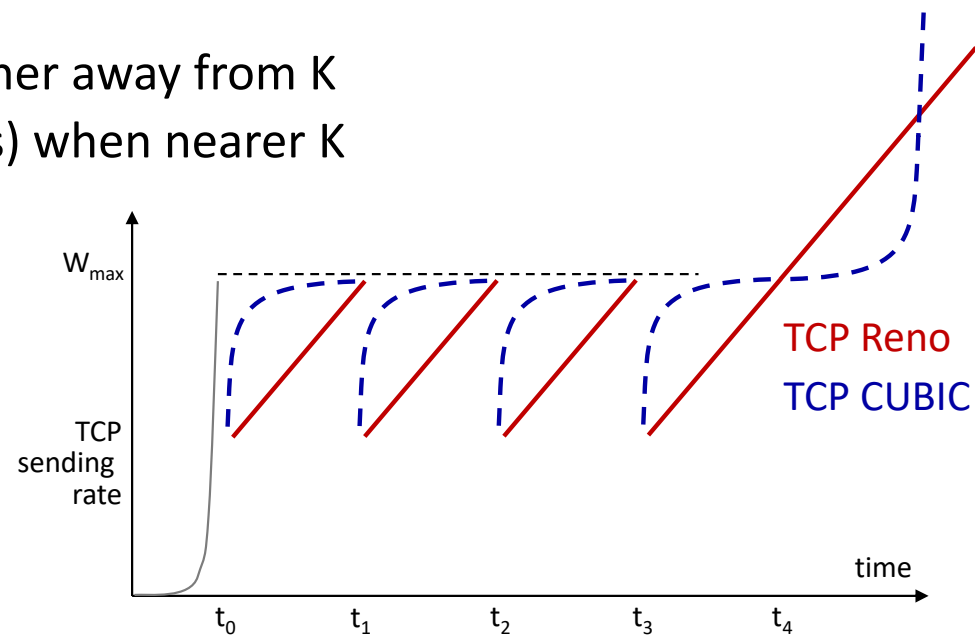
W_{max}

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

destination

application
TCP
network
link
physical

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

bottleneck link (almost always busy)

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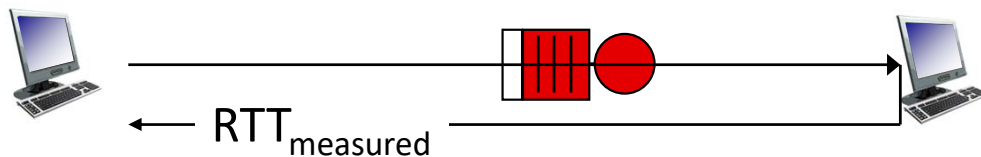
$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{RTT_{measured}}$$

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
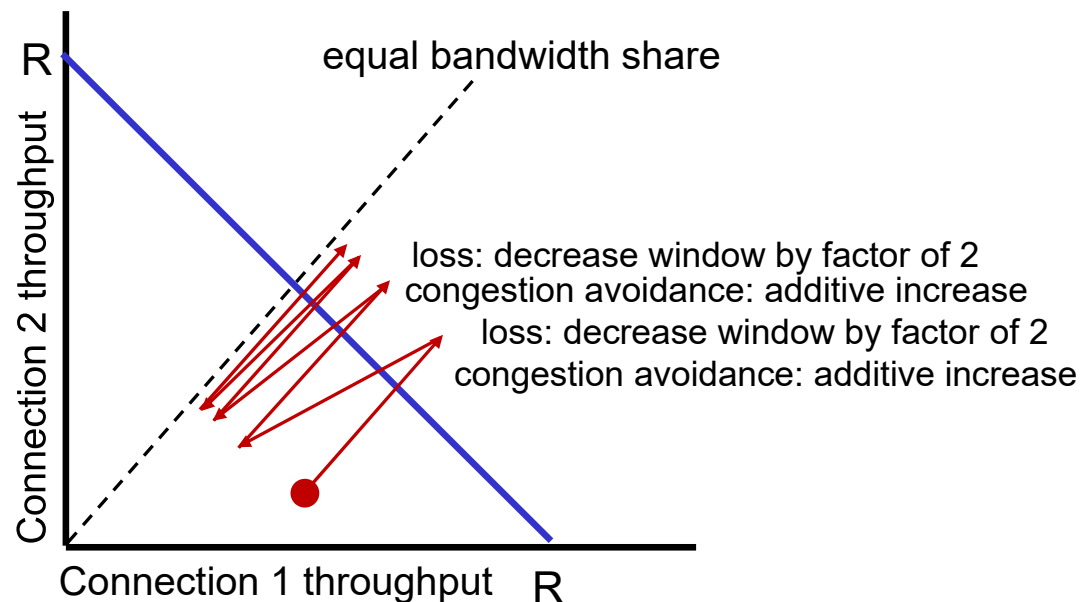- involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)
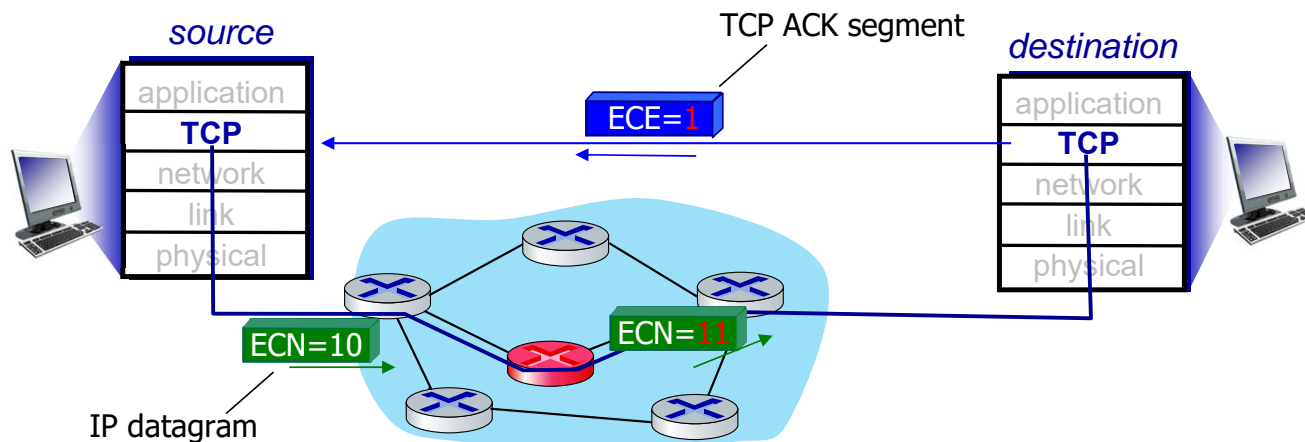
TCP fairness

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
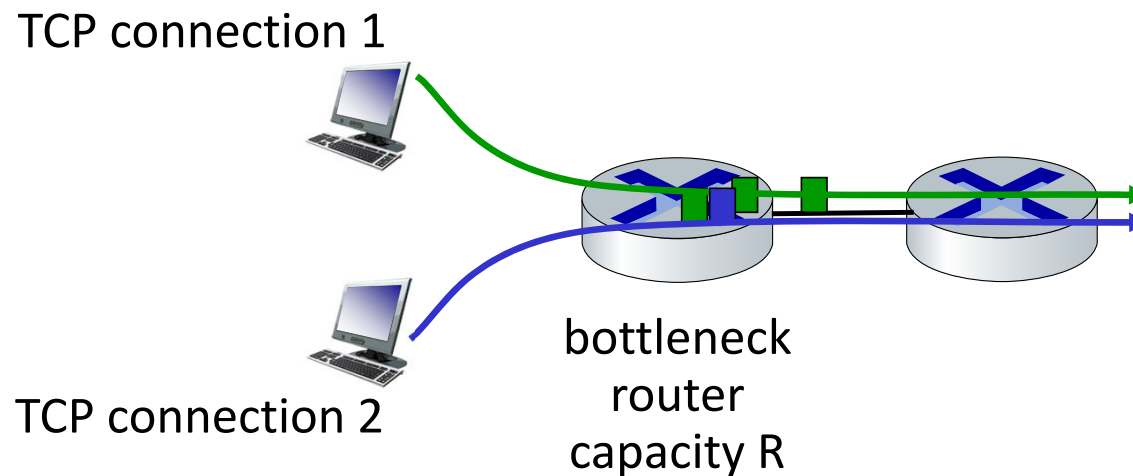
Q: is TCP Fair?

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   R

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

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)

|	
	HTTP/2
Application	TLS
Transport	TCP
Network	IP

HTTP/2 over TCP
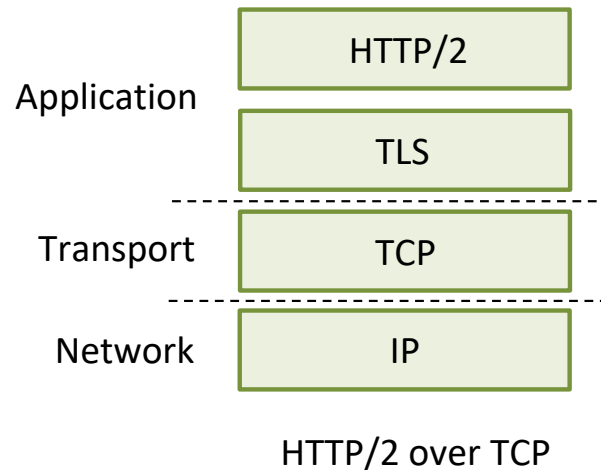
QUIC: Quick UDP Internet Connections

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 handshake
(transport layer)

TLS handshake
(security)

data

QUIC handshake

data
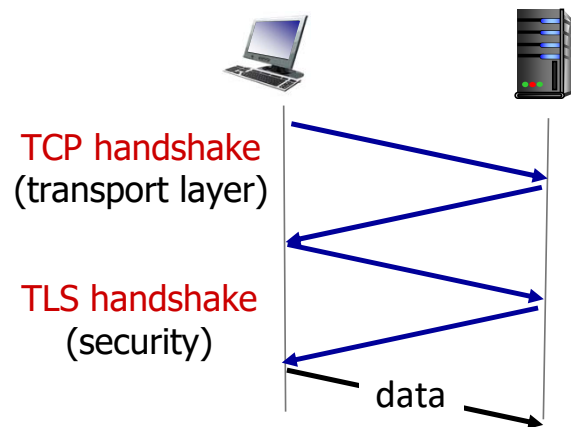
TCP (reliability, congestion control state) + TLS (authentication, crypto state)
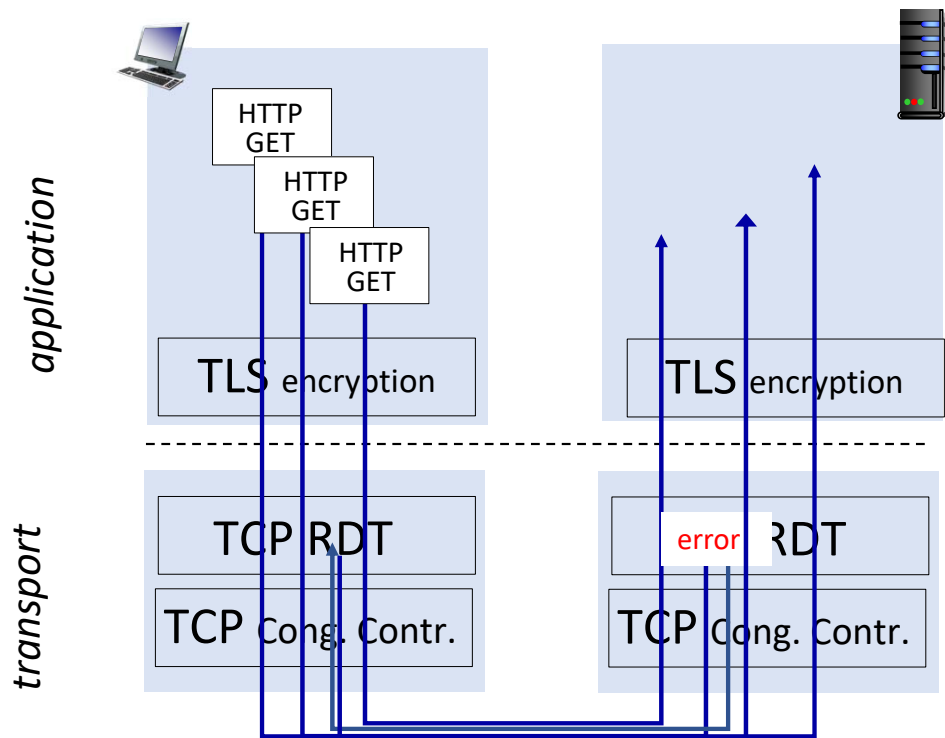
- 2 serial handshakes
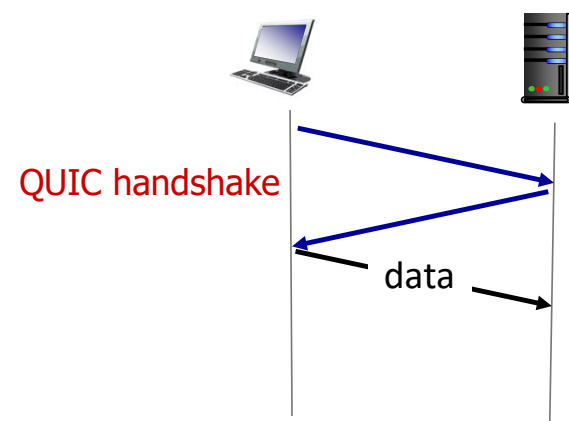
QUIC: reliability, congestion control, authentication, crypto state

- 1 handshake

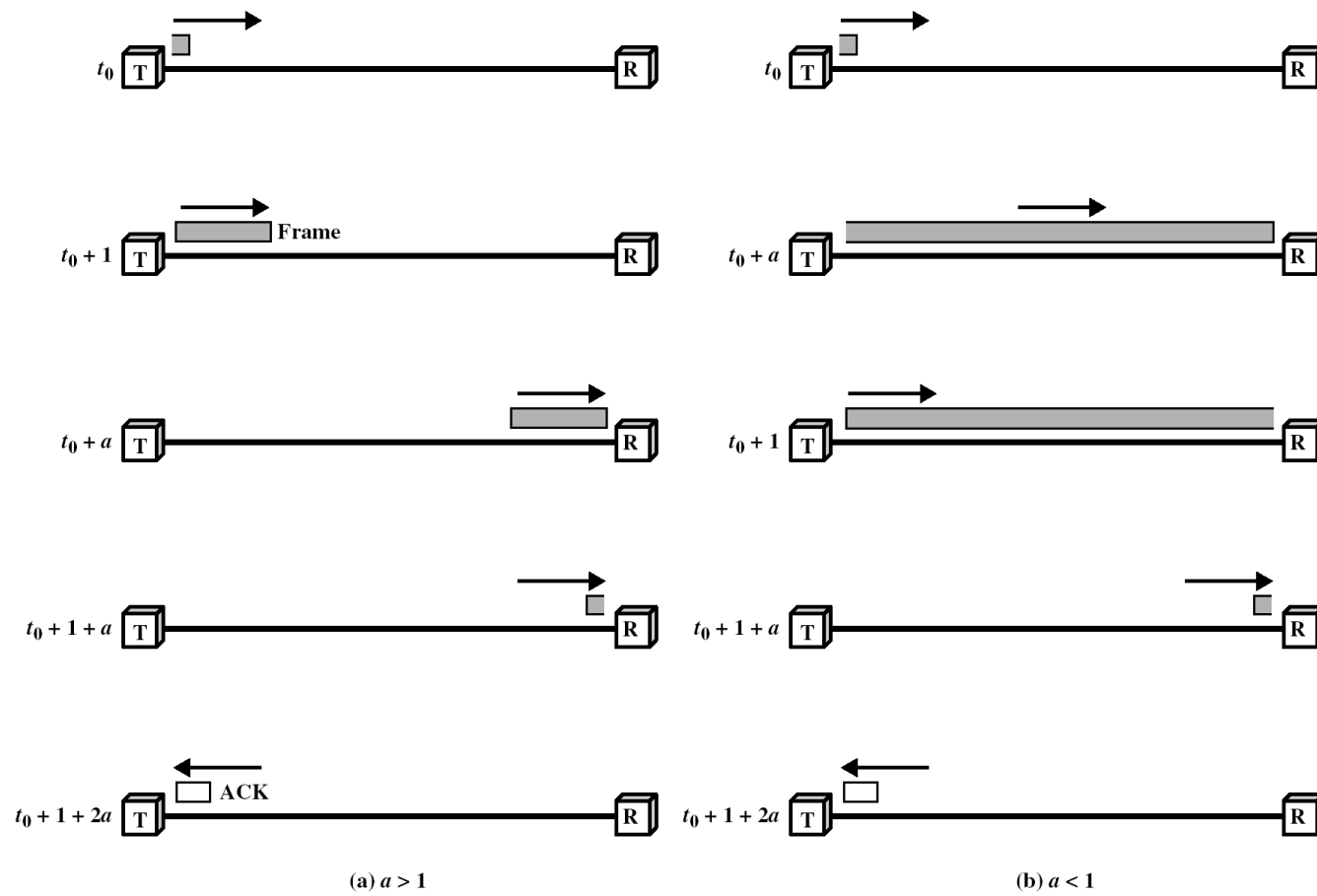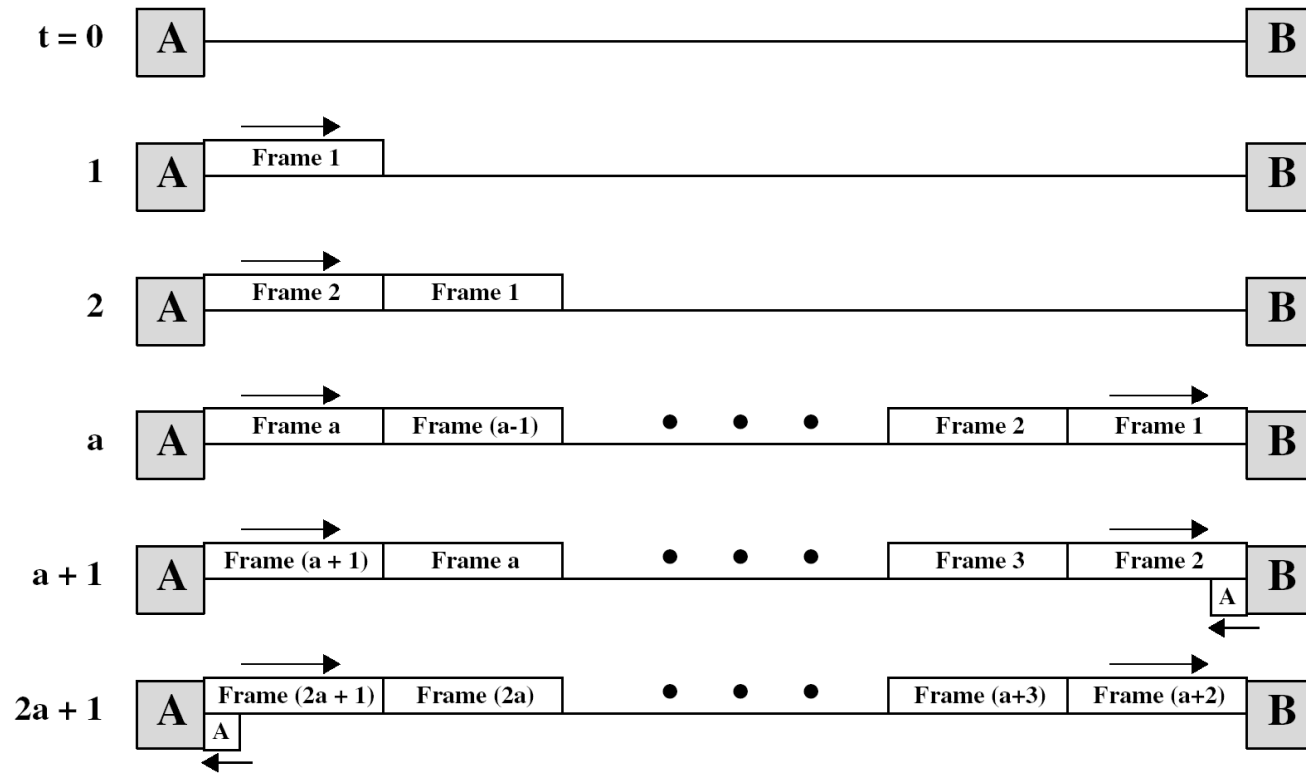# QUIC: streams: parallelism, no HOL blocking

(a) HTTP 1.1

Chapter 3: summary

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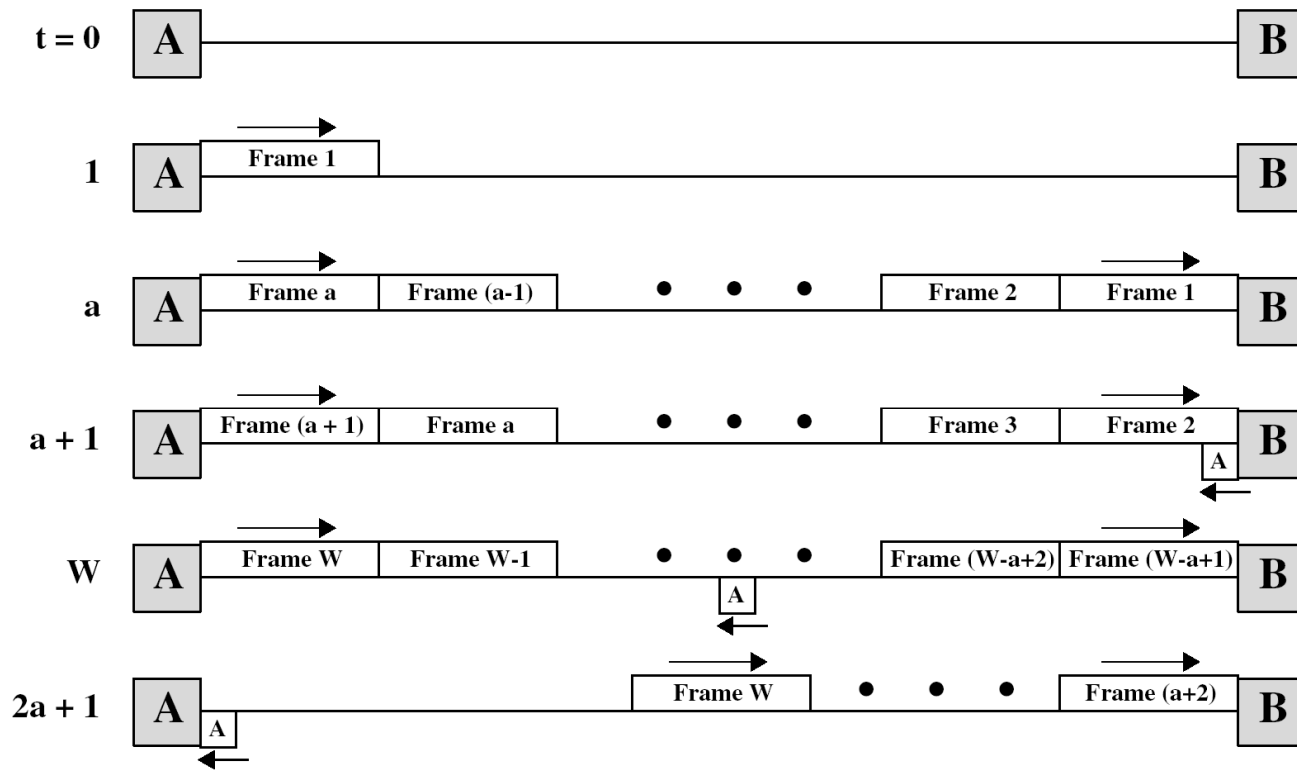 - UDP
  - TCP

Up next:

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

t_0 | T ——————————————— R

$t_0 + 1$ | T ———— Frame ——————————— R

$t_0 + a$ | T ——————————————— R

$t_0 + 1 + a$ | T ——————————————— R

$t_0 + 1 + 2a$ | T ——— ACK ——————————— R

(a) $a > 1$

t_0 | T ——————————————— R

$t_0 + a$ | T ——————————————— R

$t_0 + 1$ | T ——————————————— R

$t_0 + 1 + a$ | T ——————————————— R

$t_0 + 1 + 2a$ | T ——————————————— R

(b) $a < 1$

(a) W ≥ 2a + 1

(b) W < 2a + 1

Delay modeling

Q: How long does it take to receive an object from a Web server after sending a request?

Ignoring congestion, delay is influenced by:

- TCP connection establishment
- data transmission delay
- slow start

Notation, assumptions:

- Assume one link between client and server of rate R
- S: MSS (bits)
- O: object size (bits)
- no retransmissions (no loss, no corruption)
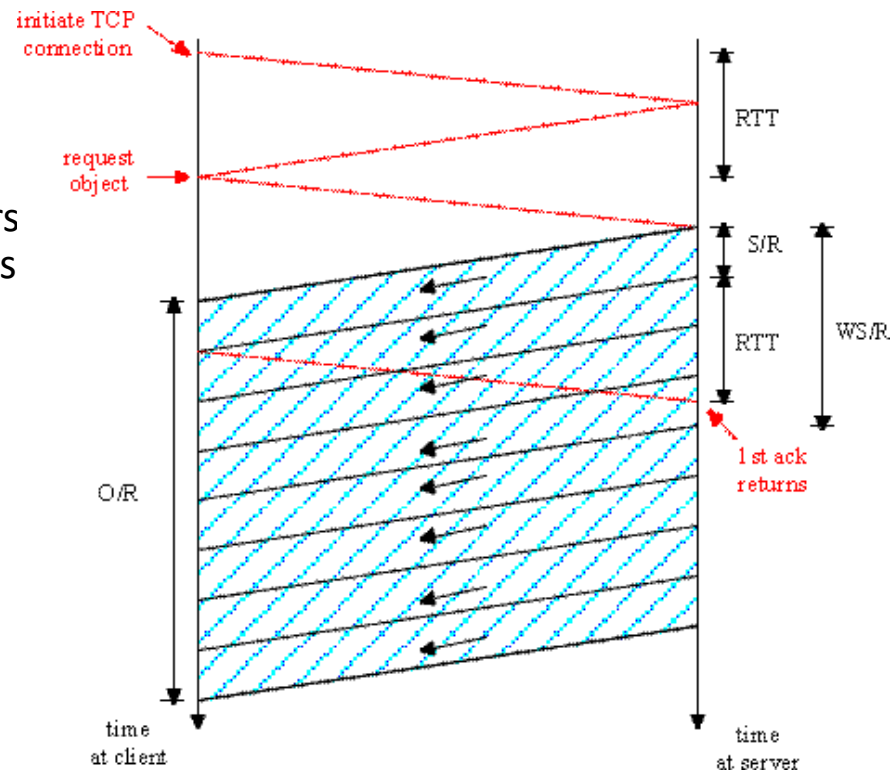
Window size:

- First assume: fixed congestion window, W segments
- Then dynamic window, modeling slow start

Fixed congestion window (1)

<u>First case:</u>

WS/R > RTT + S/R: ACK for firs
 segment in window returns
 before window's worth of
 data sent

delay = 2RTT + O/R

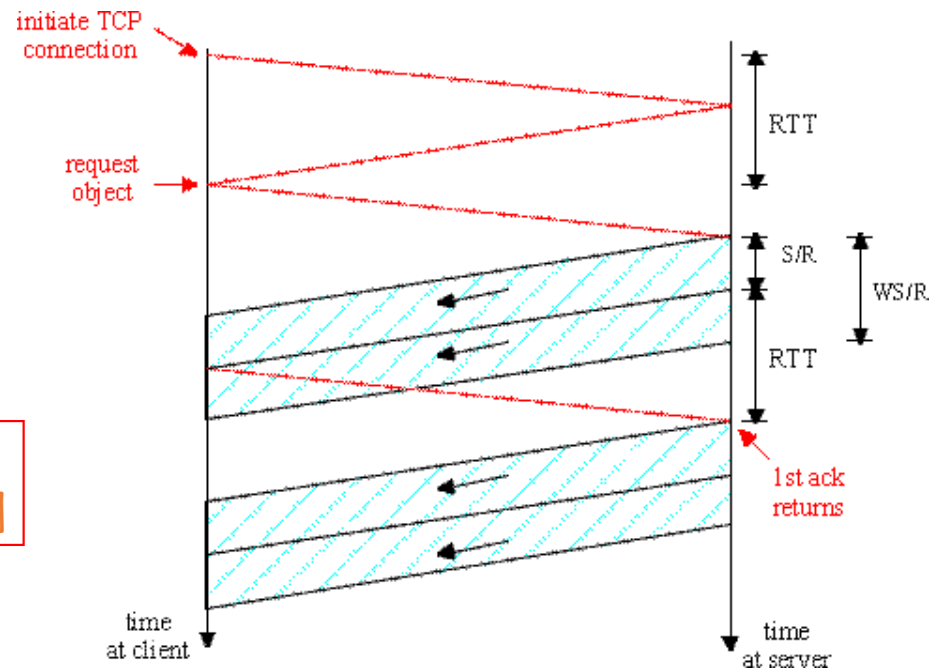# Fixed congestion window (2)

Second case:

- WS/R < RTT + S/R: wait for ACK after sending window's worth of data sent

delay = 2RTT + O/R
+ (K-1)[S/R + RTT - WS/R]

initiate TCP connection

request object

RTT

S/R

WS/R

RTT

1st ack returns

time at client

time at server

TCP Delay Modeling: Slow Start (1)

Now suppose window grows according to slow start

Will show that the delay for one object is:

$$Latency = 2RTT + \frac{O}{R} + P\left[RTT + \frac{S}{R}\right] - (2^P - 1)\frac{S}{R}$$

where *P* is the number of times TCP idles at server:

$$P = \min\{Q, K - 1\}$$

- where Q is the number of times the server idles
  if the object were of infinite size.

- and K is the number of windows that cover the object.

TCP Delay Modeling: Slow Start (2)

Delay components:
- 2 RTT for connection estab and request
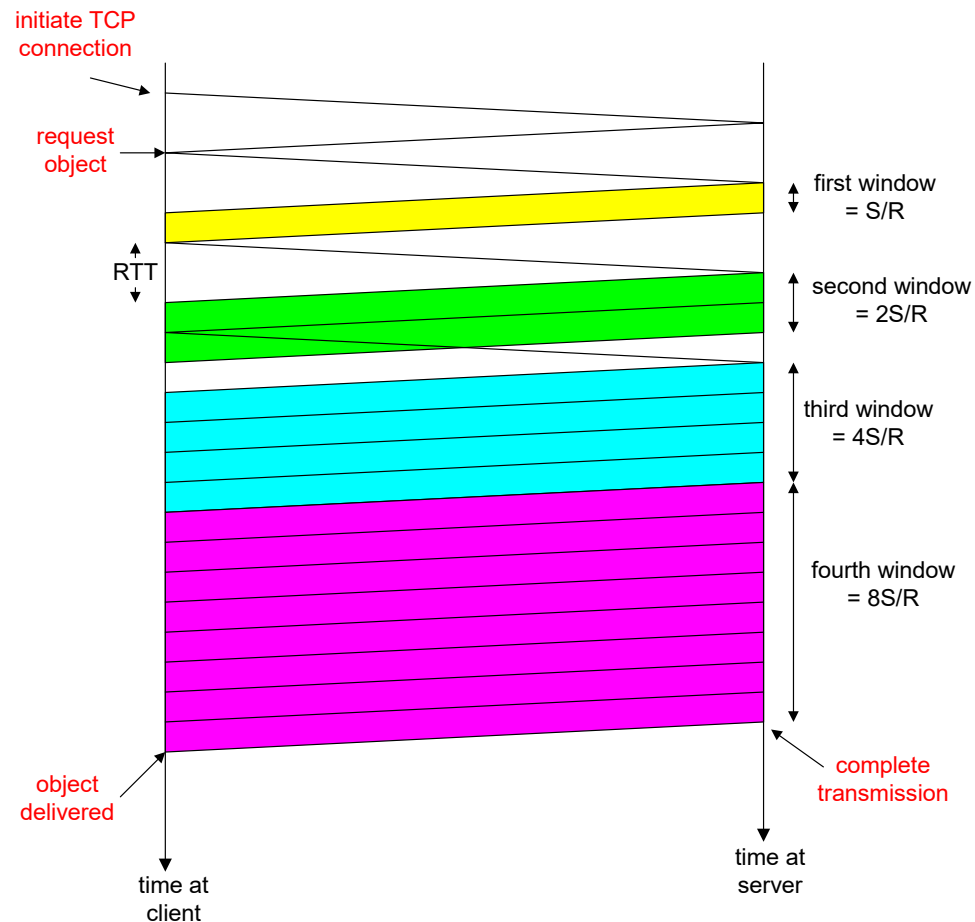- O/R to transmit object
- time server idles due to slow start

Server idles:
$P = \min\{K-1,Q\}$ times

Example:
- O/S = 15 segments
- K = 4 windows
- Q = 2
- $P = \min\{K-1,Q\} = 2$

Server idles P=2 times

initiate TCP connection
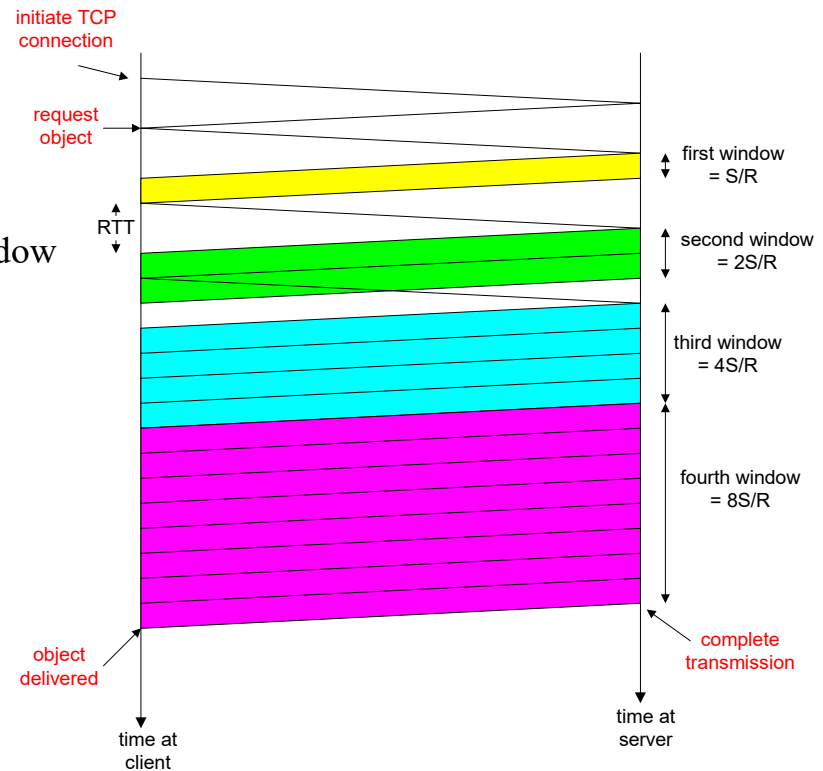
request object

RTT

first window = S/R

second window = 2S/R

third window = 4S/R

fourth window = 8S/R

object delivered

complete transmission

time at client

time at server

TCP Delay Modeling (3)

$$\frac{S}{R} + RTT = \text{time from when server starts to send segment}$$

$$\text{until server receives acknowledgement}$$

$$2^{k-1}\frac{S}{R} = \text{time to transmit the kth window}$$

$$\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]^{+} = \text{idle time after the } k\text{th window}$$

$$\text{delay} = \frac{O}{R} + 2RTT + \sum_{p=1}^{P} idleTime_p$$

$$= \frac{O}{R} + 2RTT + \sum_{k=1}^{P}\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]$$

$$= \frac{O}{R} + 2RTT + P\left[RTT + \frac{S}{R}\right] - (2^P - 1)\frac{S}{R}$$

initiate TCP
connection

request
object

RTT

first window
= S/R

second window
= 2S/R

third window
= 4S/R

fourth window
= 8S/R

object
delivered

complete
transmission

time at
client

time at
server

TCP Delay Modeling (4)

Recall K = number of windows that cover object

How do we calculate K ?

$$K = \min\{k : 2^0 S + 2^1 S + \cdots + 2^{k-1} S \geq O\}$$
$$= \min\{k : 2^0 + 2^1 + \cdots + 2^{k-1} \geq O/S\}$$
$$= \min\{k : 2^k - 1 \geq \frac{O}{S}\}$$
$$= \min\{k : k \geq \log_2(\frac{O}{S} + 1)\}$$
$$= \left\lceil \log_2(\frac{O}{S} + 1) \right\rceil$$