Transport Layer: UDP and TCP

CS491G: Computer Networking Lab
V. Arun

Slides adapted from Kurose and Ross

Transport Layer: Outline

1 transport-layer services

2 multiplexing and demultiplexing

3 connectionless transport: UDP

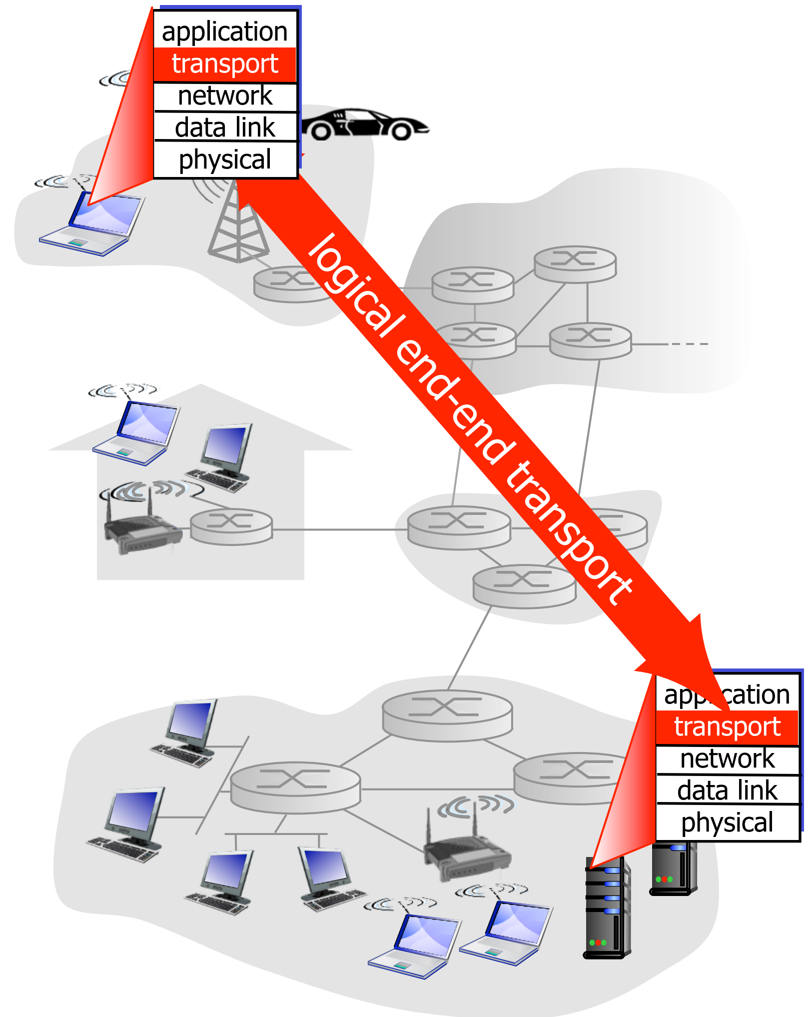
4 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

5 principles of congestion control

6 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
  - recv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Transport vs. network layer

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

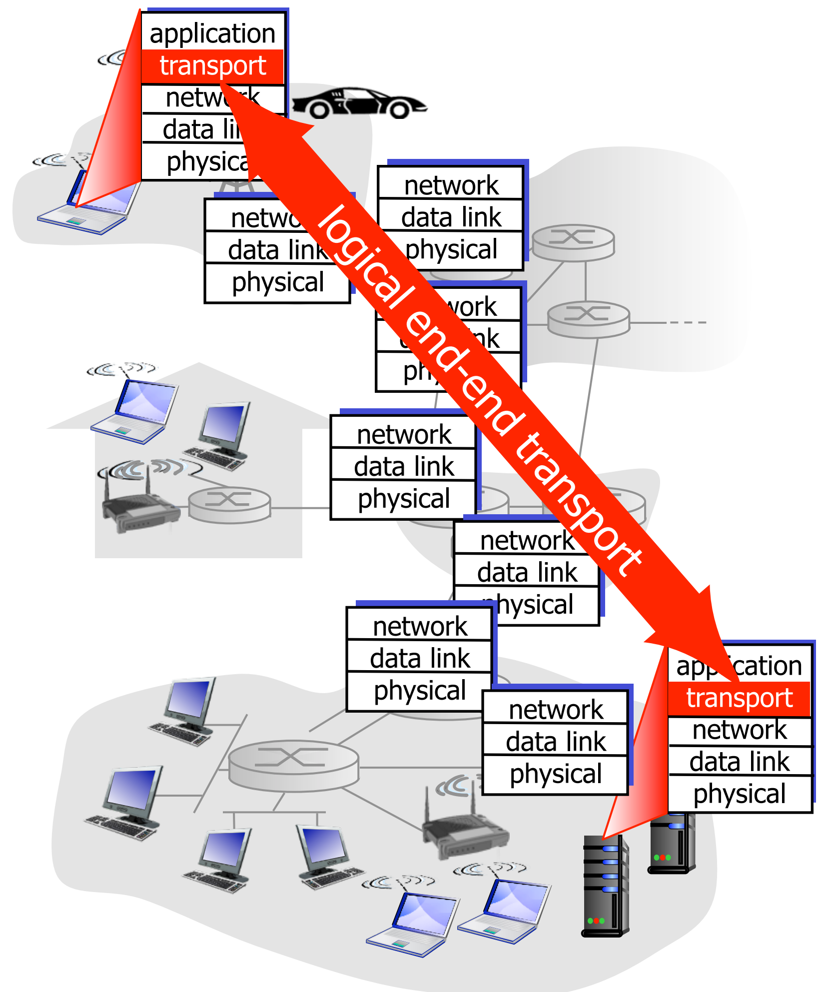
household analogy:

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

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

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



Transport Layer: Outline

1 transport-layer services

2 multiplexing and demultiplexing

3 connectionless transport: UDP

4 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

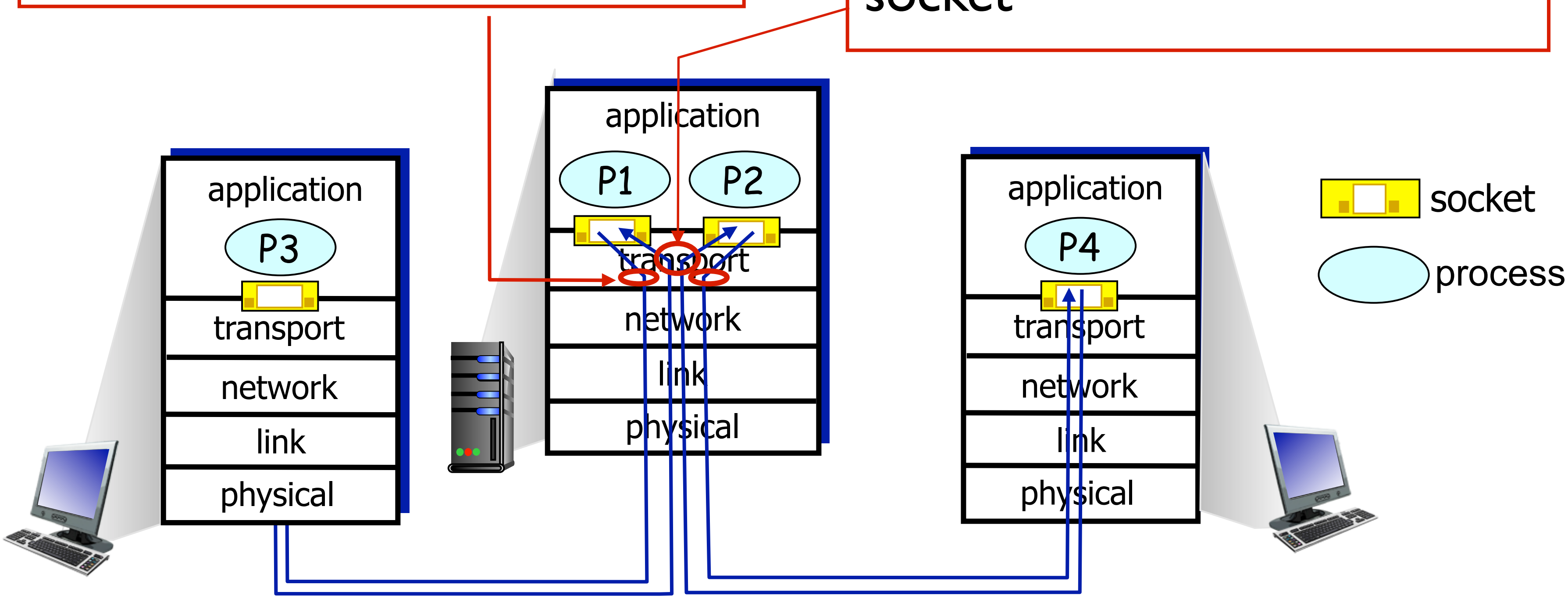
5 principles of congestion control

6 TCP congestion control

Multiplexing/demultiplexing

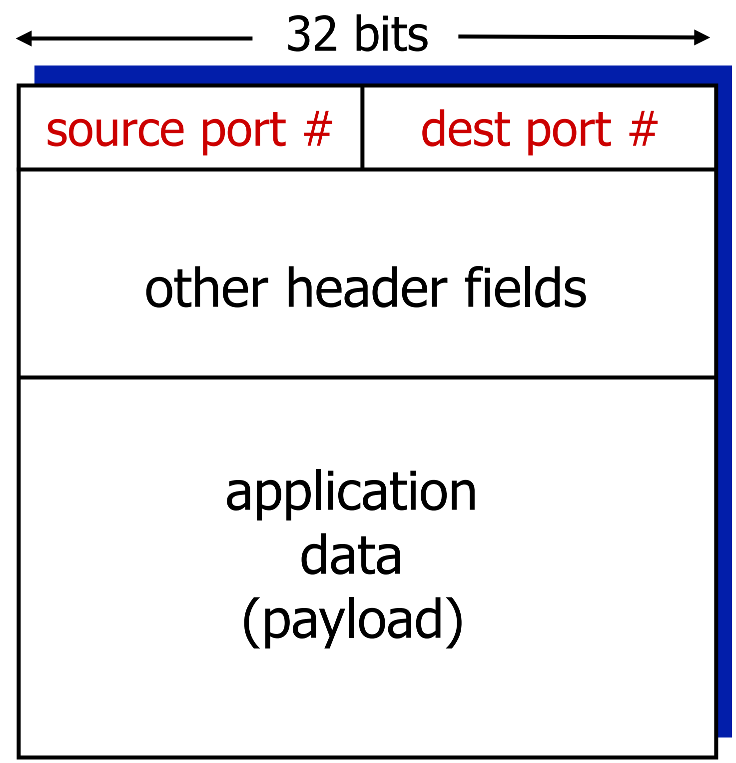
multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

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



How demultiplexing works

- host receives IP datagrams
  - each datagram has source and destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source and destination port number
- host uses *IP addresses & port numbers* to direct segment to right socket



TCP/UDP segment format

Connectionless demultiplexing

- *recall:* created socket has host-local port #:

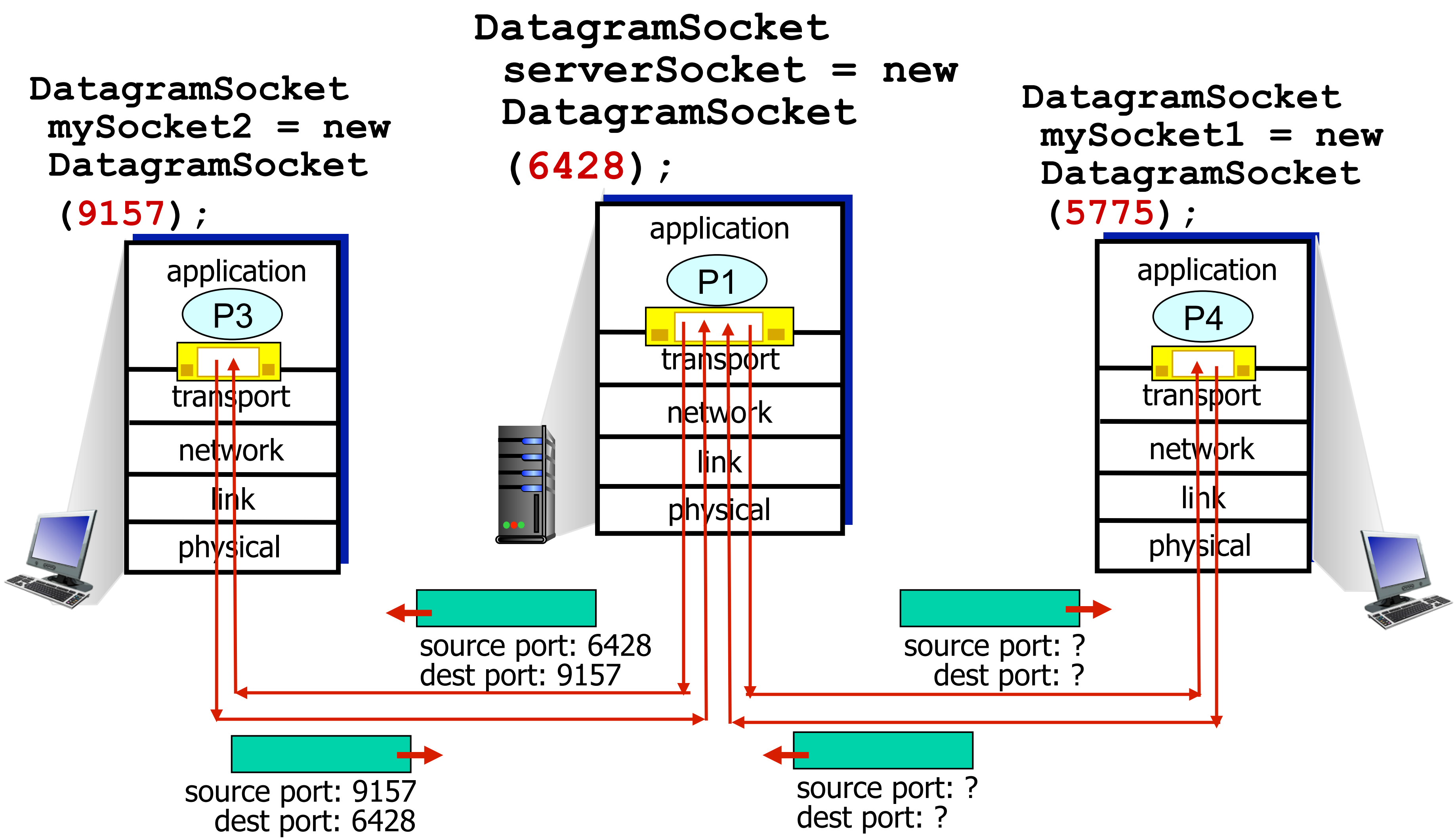
```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

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

- when host receives UDP segment:
  - checks destination IP and port # in segment
  - directs UDP segment to socket bound to that (IP,port)

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

Connectionless demux: example

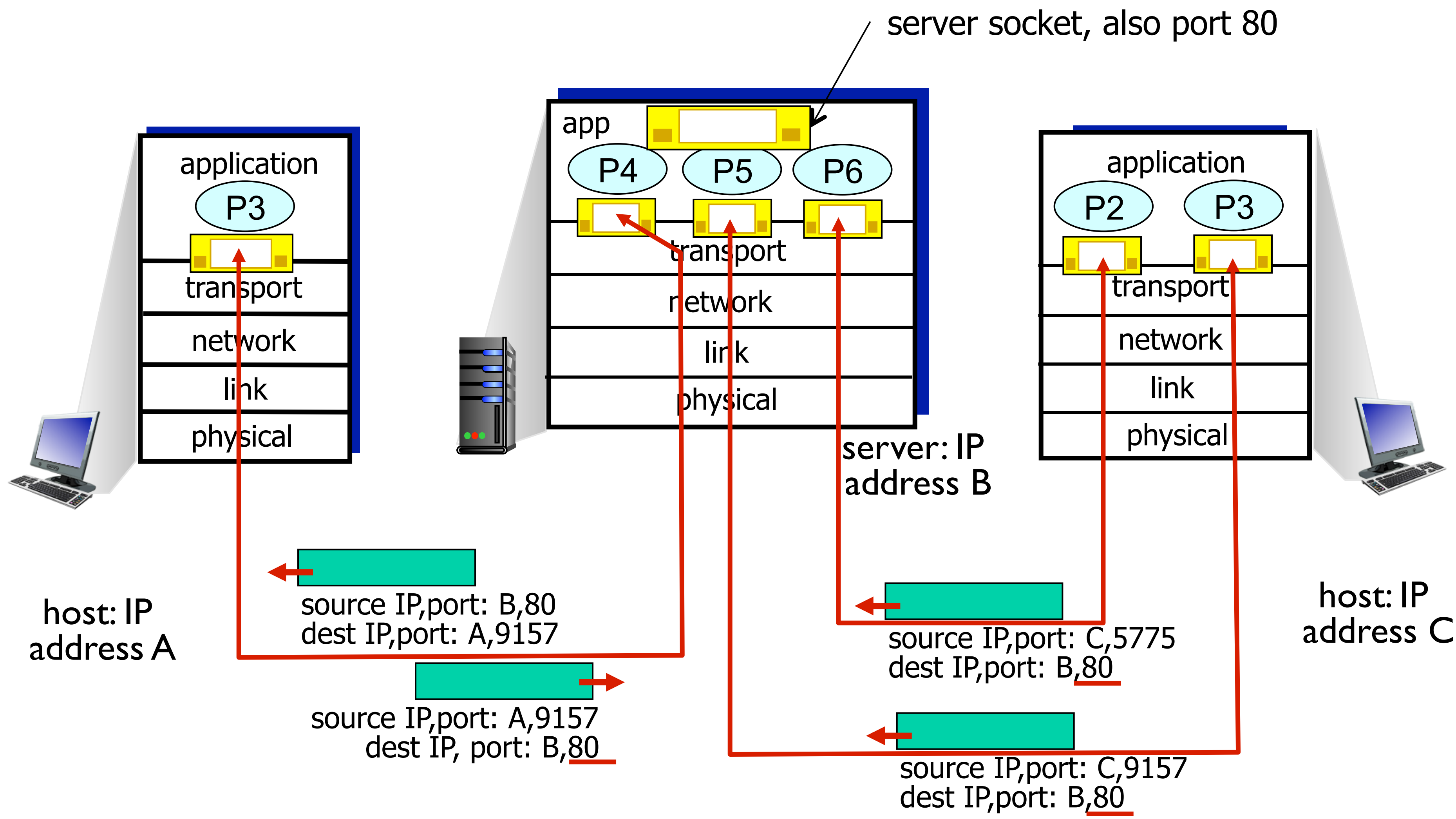


Connection-oriented demux

- 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 right socket

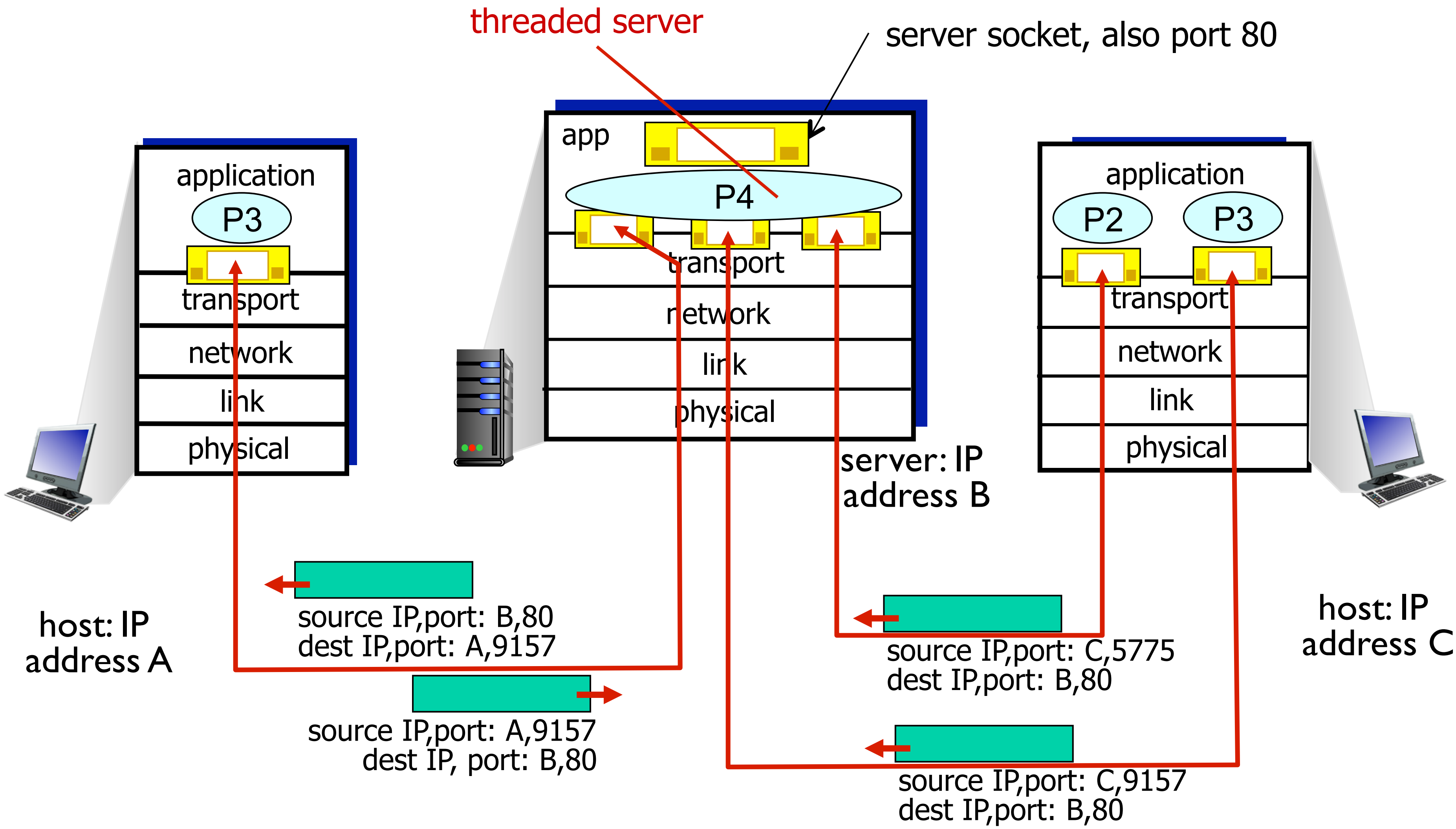
- server host has many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different socket each client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux: example



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

Connection-oriented demux: example



Transport Layer: Outline

1 transport-layer services

2 multiplexing and demultiplexing

3 connectionless transport: UDP

4 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

5 principles of congestion control

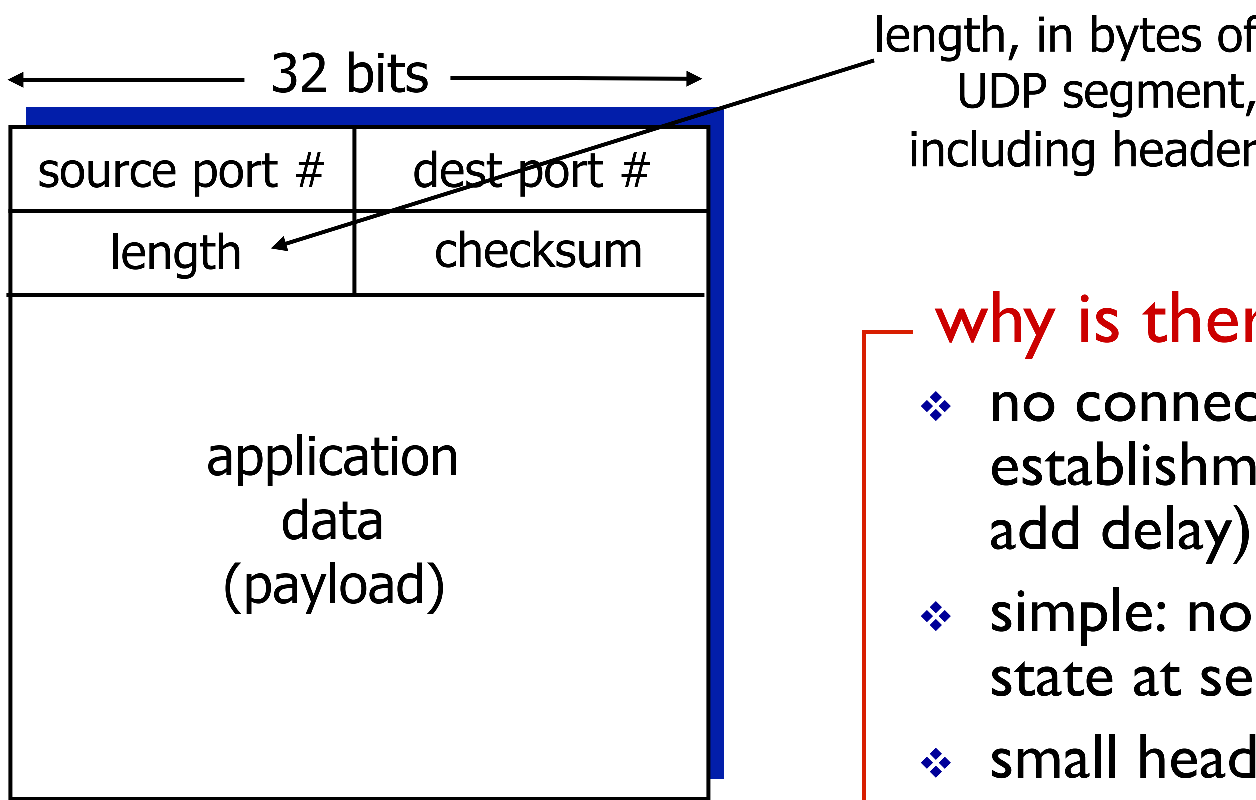
6 TCP congestion control

UDP: User Datagram Protocol [RFC 768]

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

- UDP uses:
  - 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



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" (flipped bits) in segments

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

Q1: Sockets and multiplexing

- TCP uses more information in packet headers in order to demultiplex packets compared to UDP.
    A. True
    B. False

Q2: Sockets UDP

- Suppose we use UDP instead of TCP under HTTP for designing a web server where all requests and responses fit in a single packet. Suppose a 100 clients are simultaneously communicating with this web server. How many sockets are respectively at the server and at each client?
  A. 1,1
  B. 2,1
  C. 200,2
  D. 100,1
  E. 101, 1

Q3: Sockets TCP

- Suppose a 100 clients are simultaneously communicating with (a traditional HTTP/TCP) web server. How many sockets are respectively at the server and at each client?
  A. 1,1
  B. 2,1
  C. 200,2
  D. 100,1
  E. 101, 1

Q4: Sockets TCP

- Suppose a 100 clients are simultaneously communicating with (a traditional HTTP/TCP) web server. Do all of the sockets at the server have the same server-side port number?
  A. Yes
  B. No

Q5: UDP checksums

- Let's denote a UDP packet as (checksum, data) ignoring other fields for this question. Suppose a sender sends (0010, 1110) and the receiver receives (0011,1110). Which of the following is true of the receiver?
  A. Thinks the packet is corrupted and discards the packet.
  B. Thinks only the checksum is corrupted and delivers the correct data to the application.
  C. Can possibly conclude that nothing is wrong with the packet.
  D. A and C

Transport Layer: Outline

1 transport-layer services

2 multiplexing and demultiplexing

3 connectionless transport: UDP

4 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

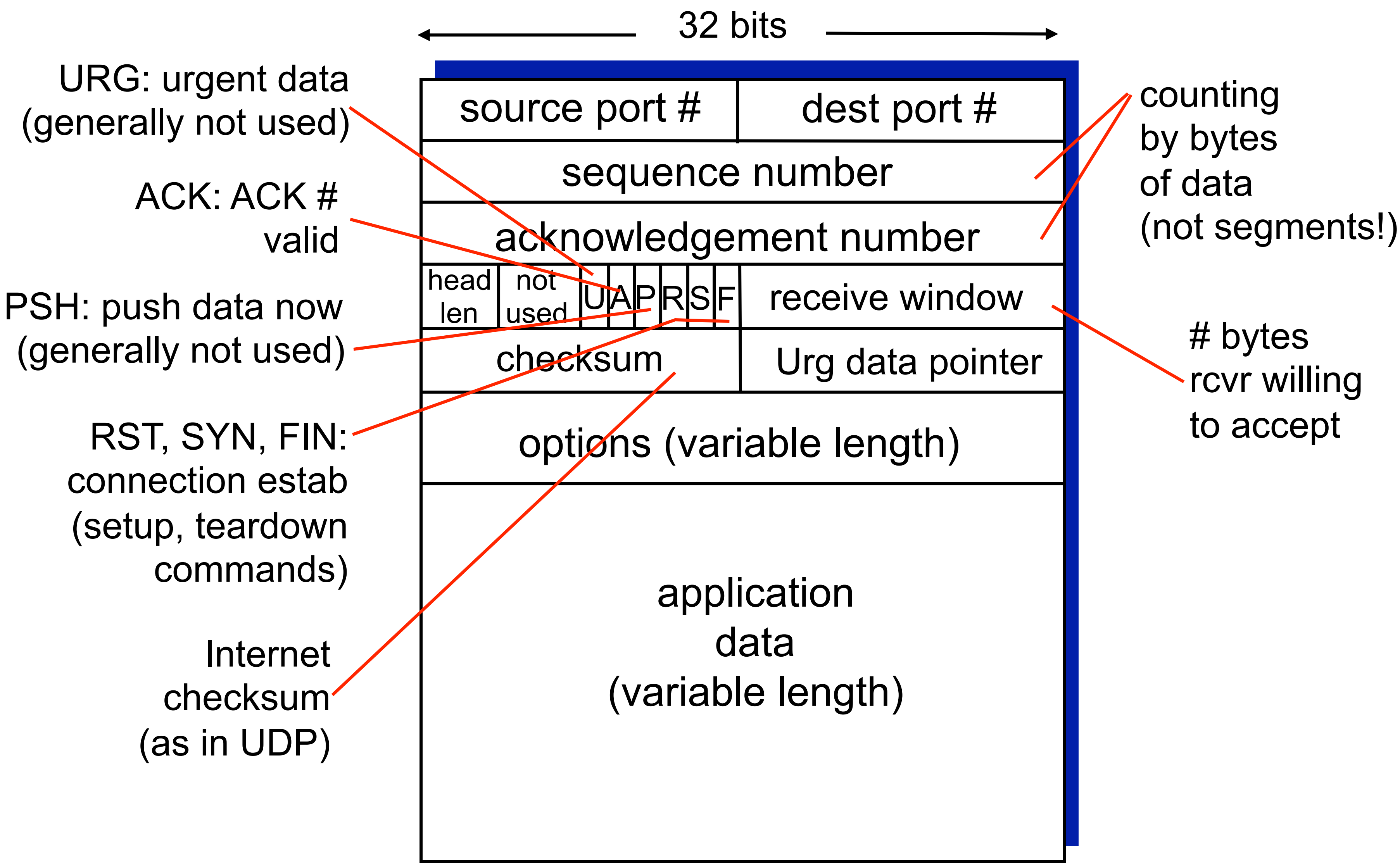
5 principles of congestion control

6 TCP congestion control

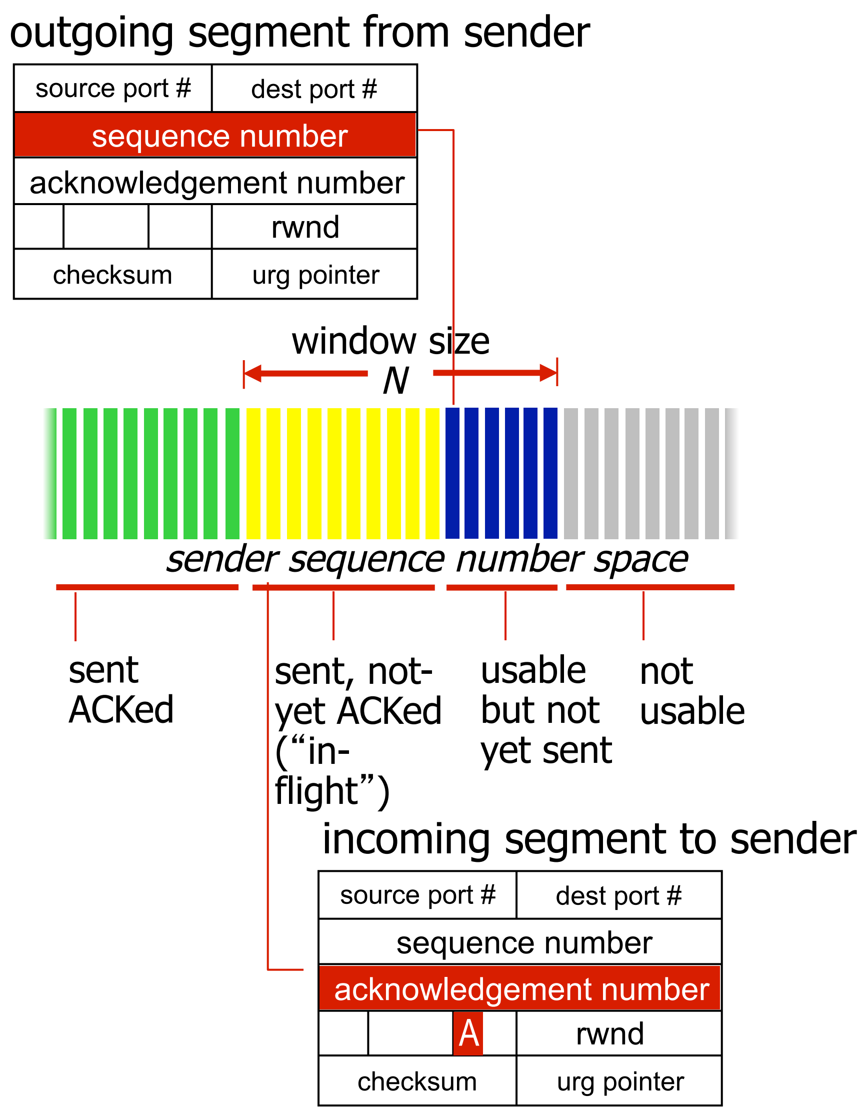
TCP: Overview RFCs: 793,1122,1323, 2018, 2581

- 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



TCP seq. numbers, ACKs



sequence numbers:

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

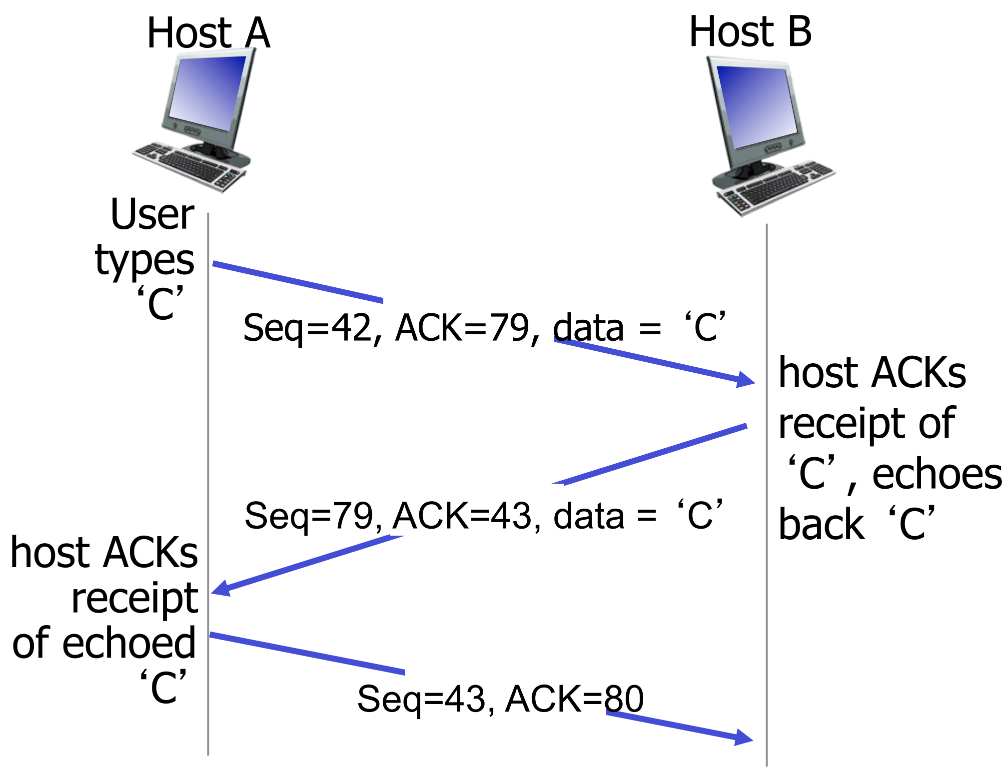
acknowledgements:

- 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

TCP seq. numbers, ACKs



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

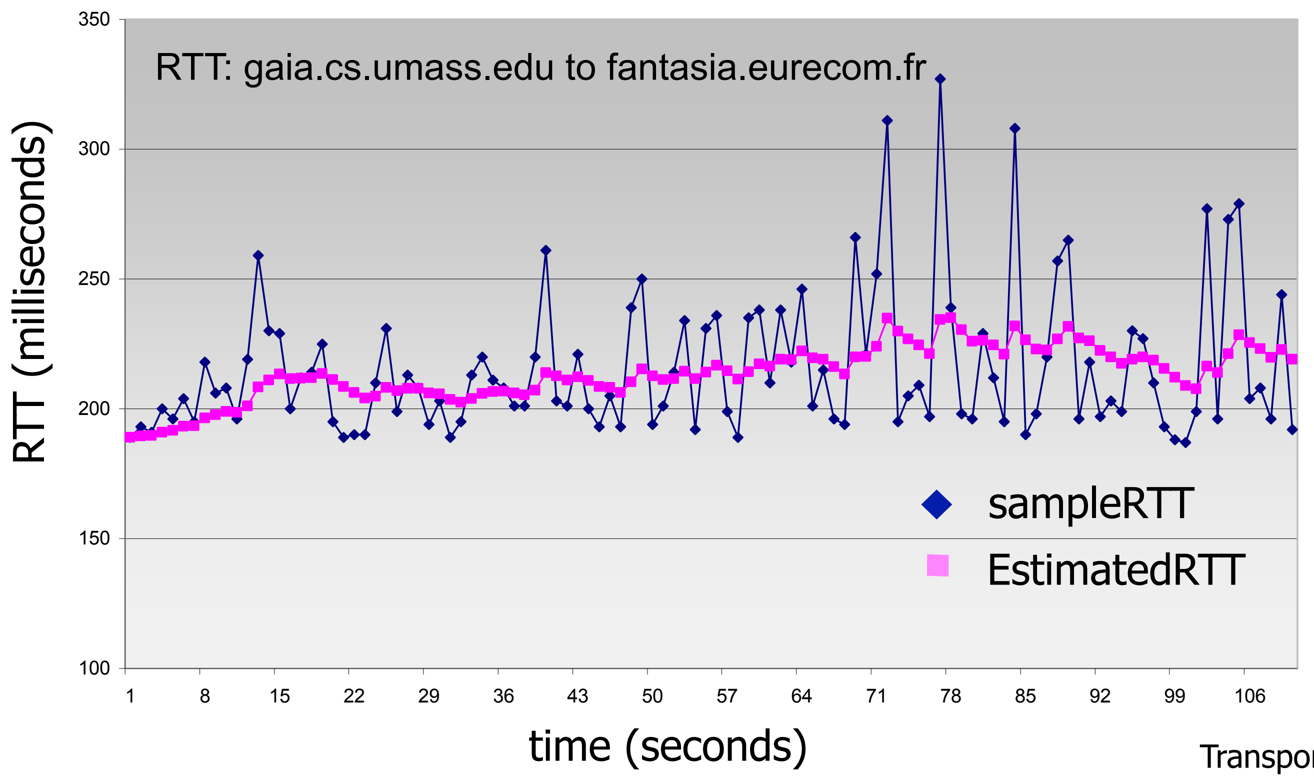
Q: how to estimate RTT?

- **`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`

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



TCP round trip time, timeout

- timeout interval: **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT`** **`->`** larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:

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

             (typically, β = 0.25)
```

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT   "safety margin"

Transport Layer: Outline

1 transport-layer services

2 multiplexing and demultiplexing

3 connectionless transport: UDP

4 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

5 principles of congestion control

6 TCP congestion control

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
  - pipelined segments
  - cumulative acks
    - selective acks often supported as an option
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

let's initially consider simplified TCP sender:
- ignore duplicate acks
- ignore flow control, congestion control

TCP sender events:

data rcvd from app:

- create segment with seq # (= byte-stream number of first data byte in segment)
- start timer if not already running (for oldest unacked segment)
  - **`TimeOutInterval`** = smoothed_RTT + 4*deviation_RTT

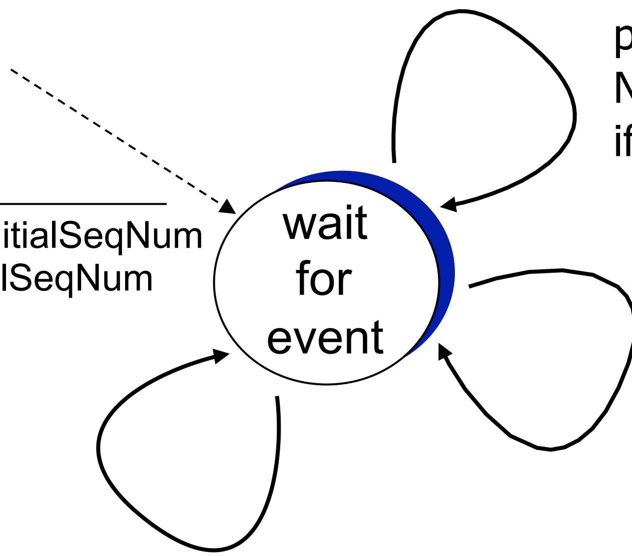
timeout:

- retransmit segment that caused timeout
- restart timer

ack rcvd:

- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - (re-)start timer if still unacked segments

TCP sender (simplified)



Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

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

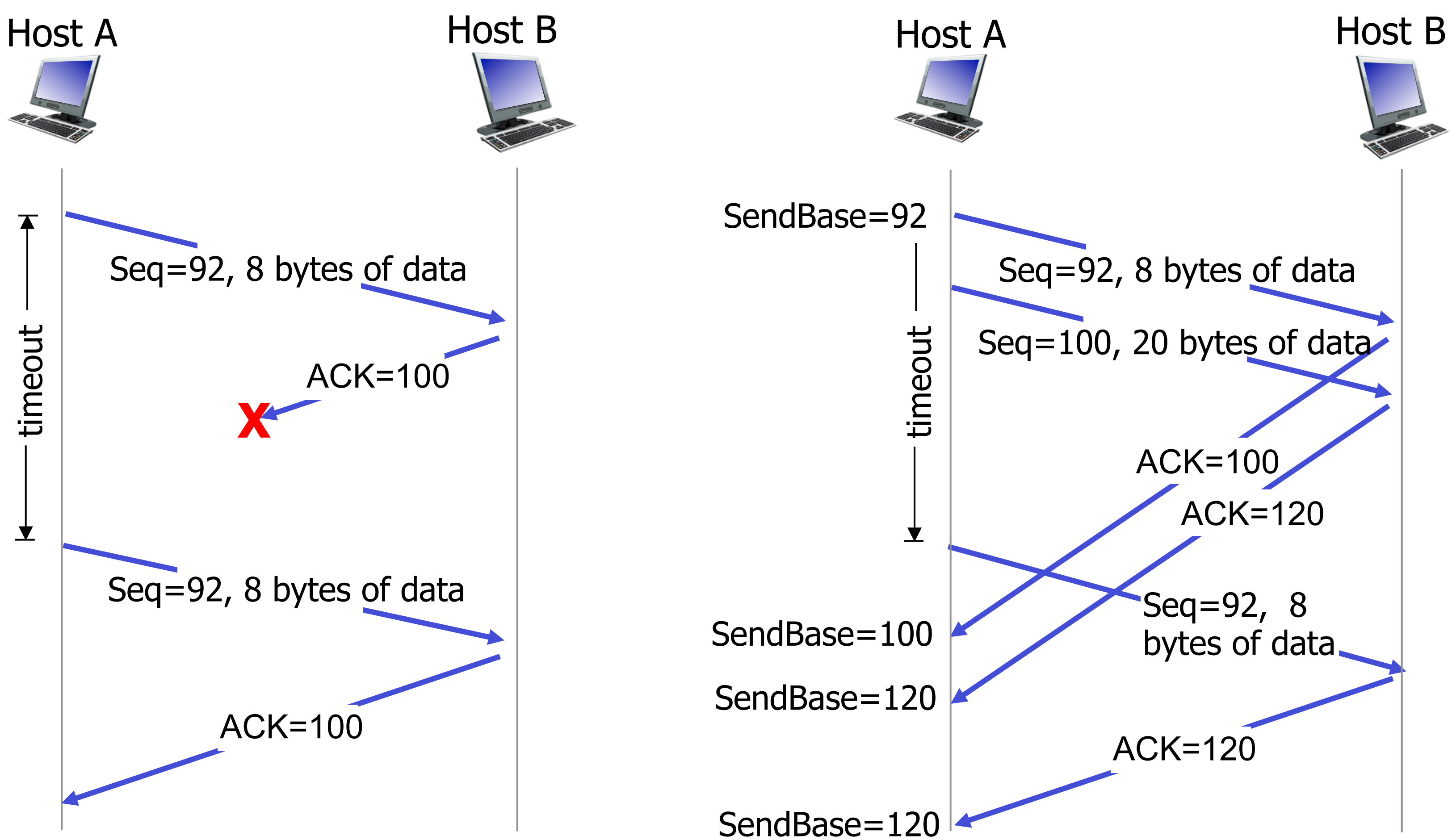
timeout

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)
       (re-)start timer
     else stop timer
  }
```

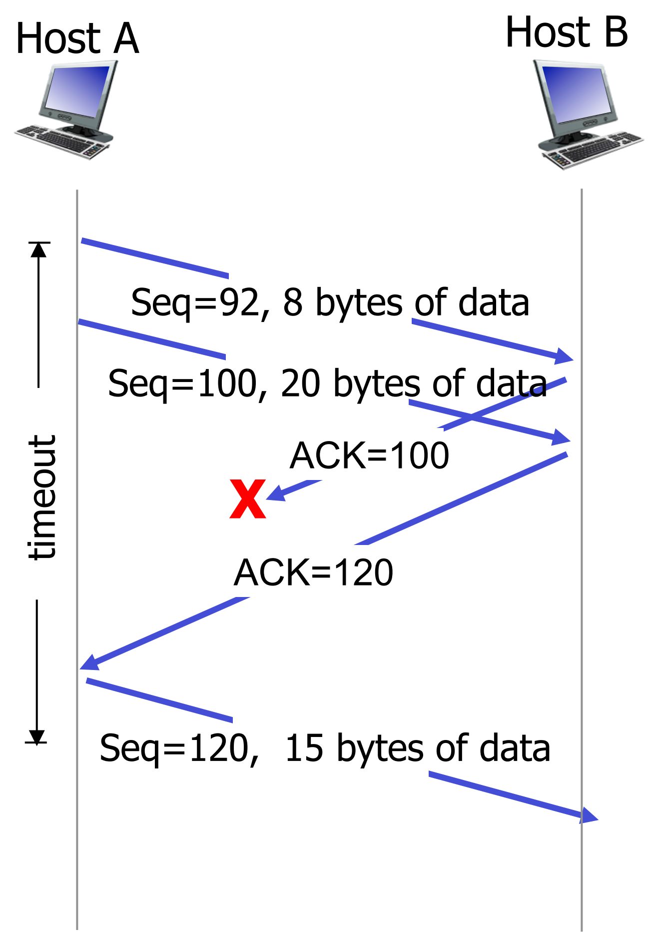
TCP: retransmission scenarios



lost ACK scenario

premature timeout

TCP: retransmission scenarios



cumulative 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

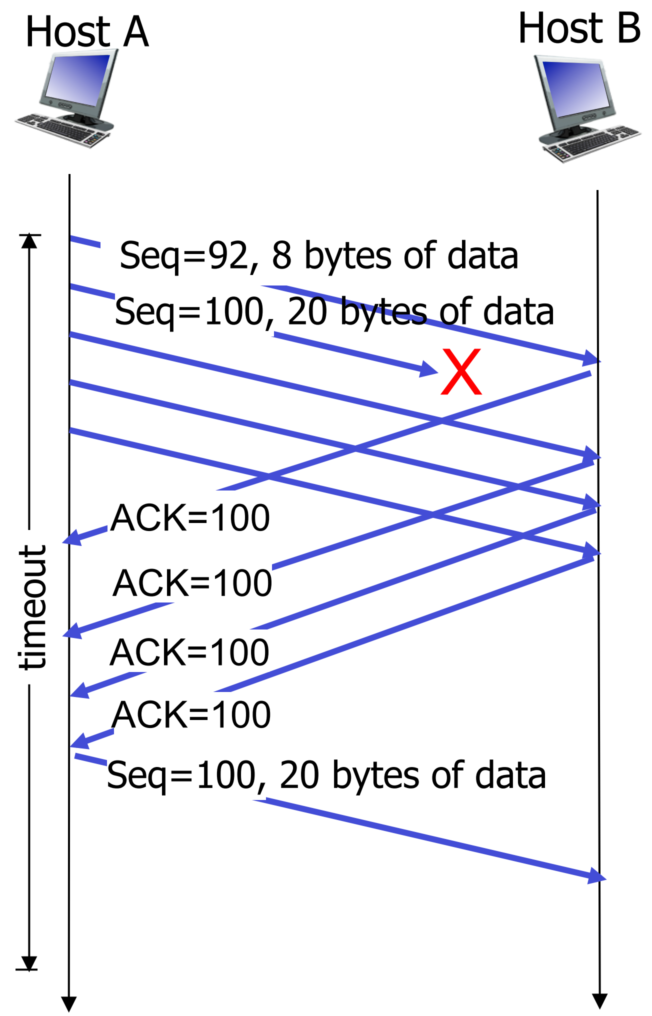
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

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

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

TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

Transport Layer: Outline

1 transport-layer services

2 multiplexing and demultiplexing

3 connectionless transport: UDP

4 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

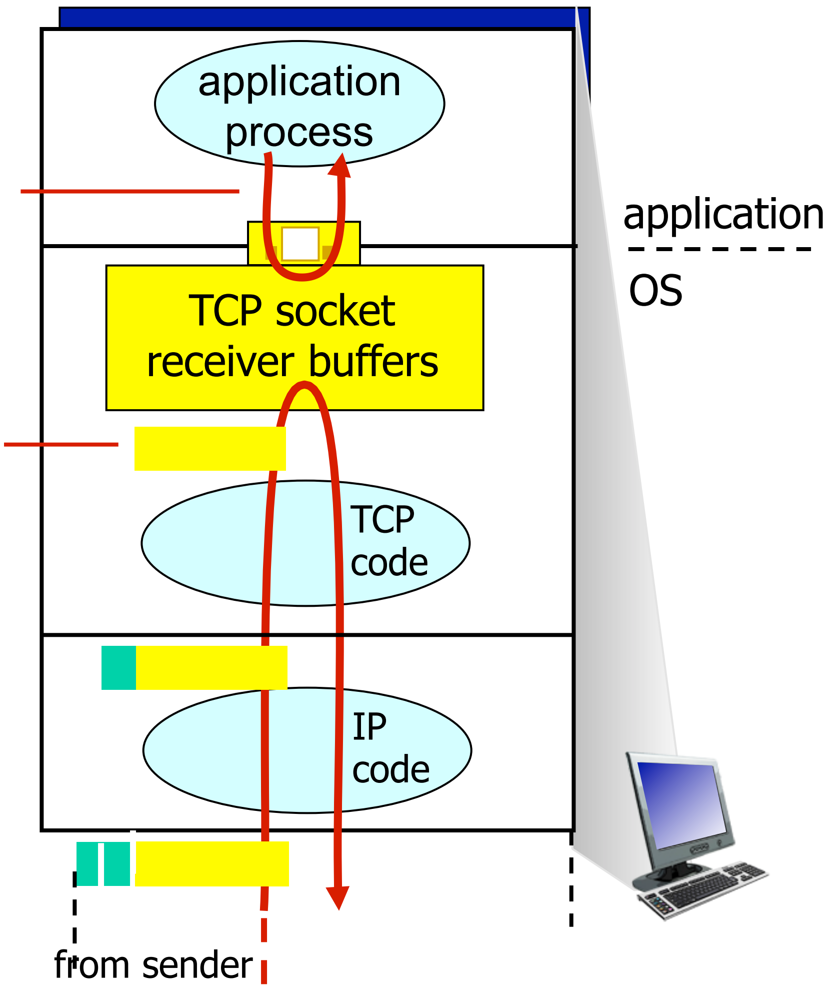
5 principles of congestion control

6 TCP congestion control

TCP flow control

application may remove data from TCP socket buffers ….

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



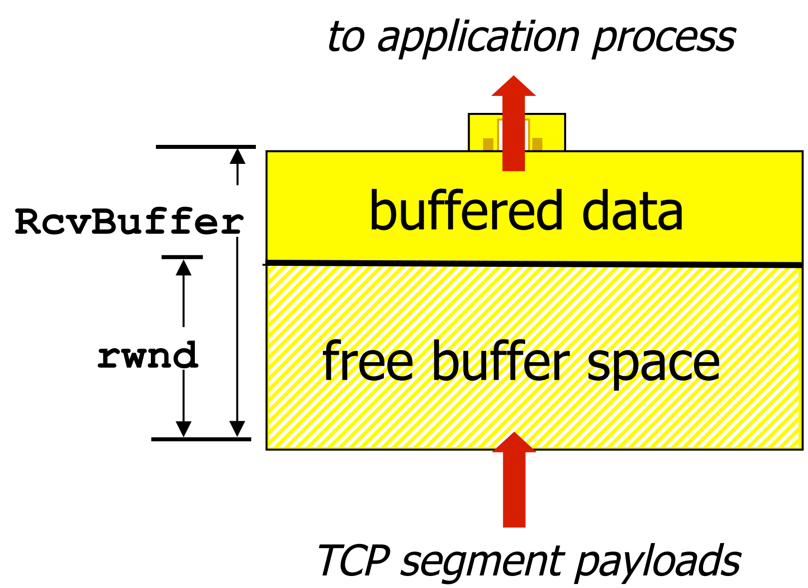
receiver protocol stack

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 can be set via socket options
  - most operating systems auto-adjust **`RcvBuffer`**
- sender limits amount of unacked ("in-flight") data to receiver's **`rwnd`** value to ensure receive buffer will not overflow



receiver-side buffering

Transport Layer: Outline

1 transport-layer services

2 multiplexing and demultiplexing

3 connectionless transport: UDP

4 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

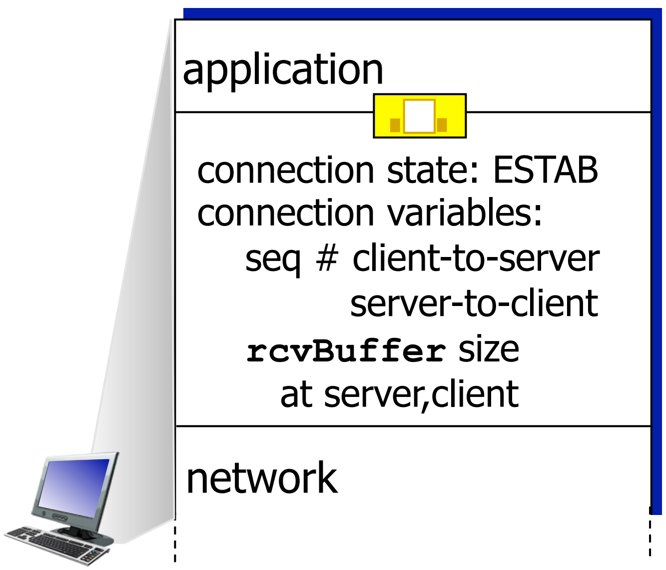
5 principles of congestion control

6 TCP congestion control

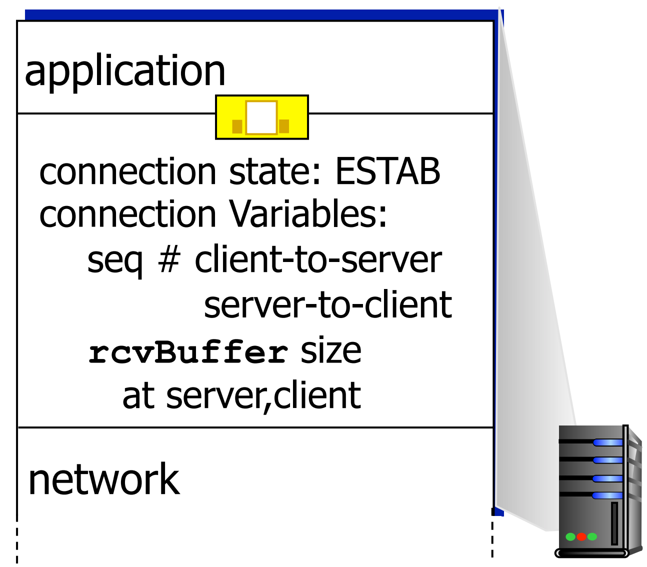
Connection Management

before exchanging data, sender/receiver "handshake":

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters



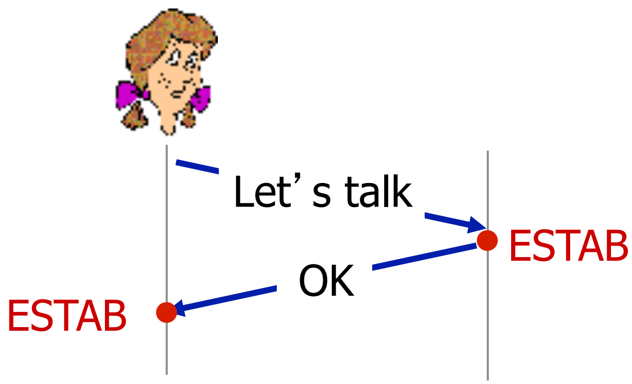

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

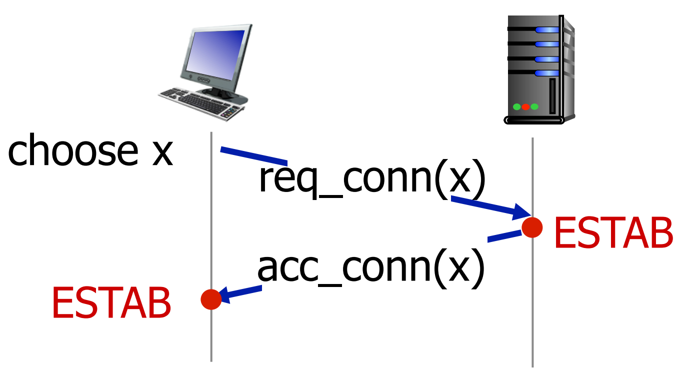



```
Socket connectionSocket =
   welcomeSocket.accept();
```

Agreeing to establish a connection

2-way handshake:



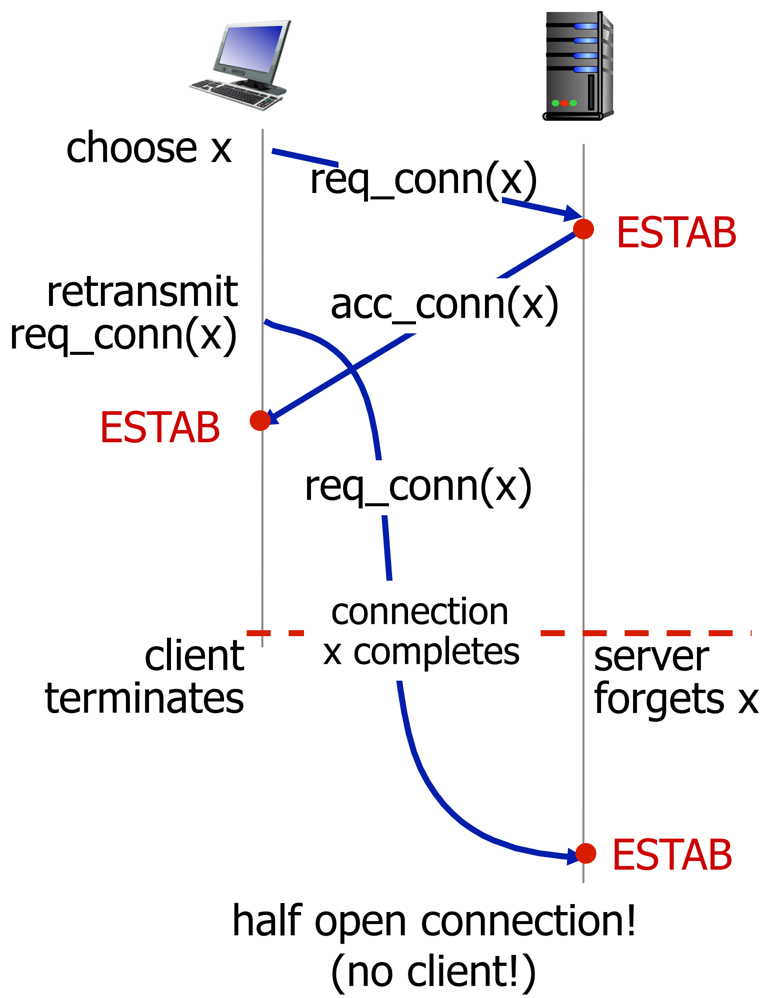


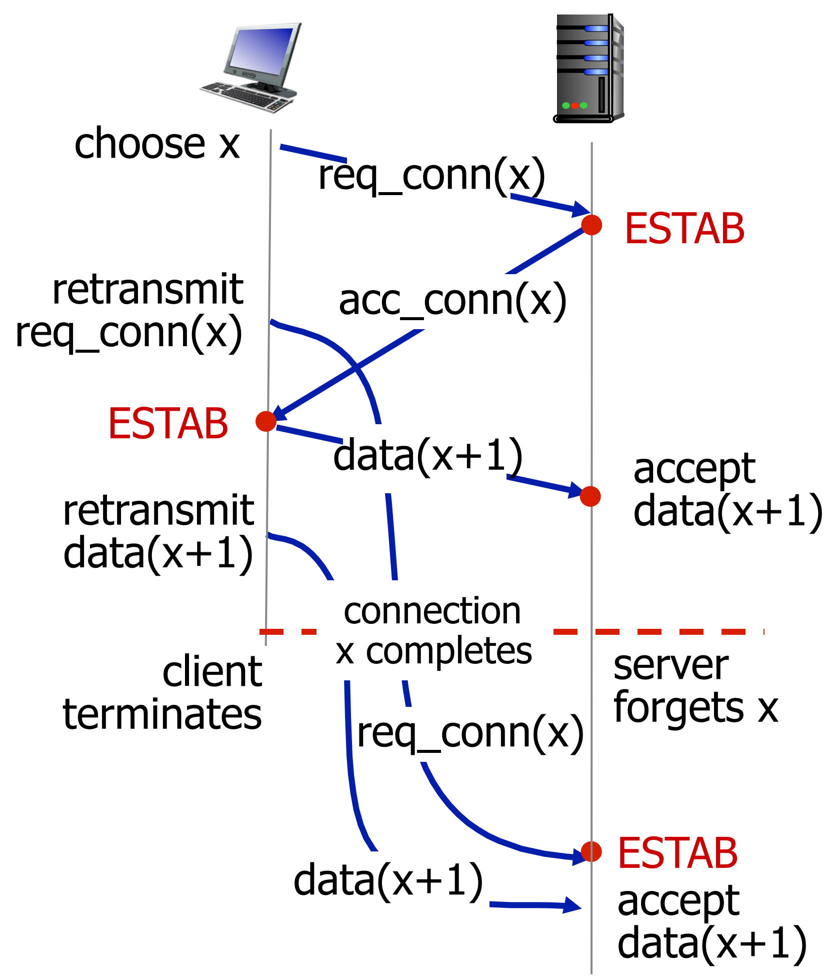
Q: will 2-way handshake always work in network?

- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can't "see" other side

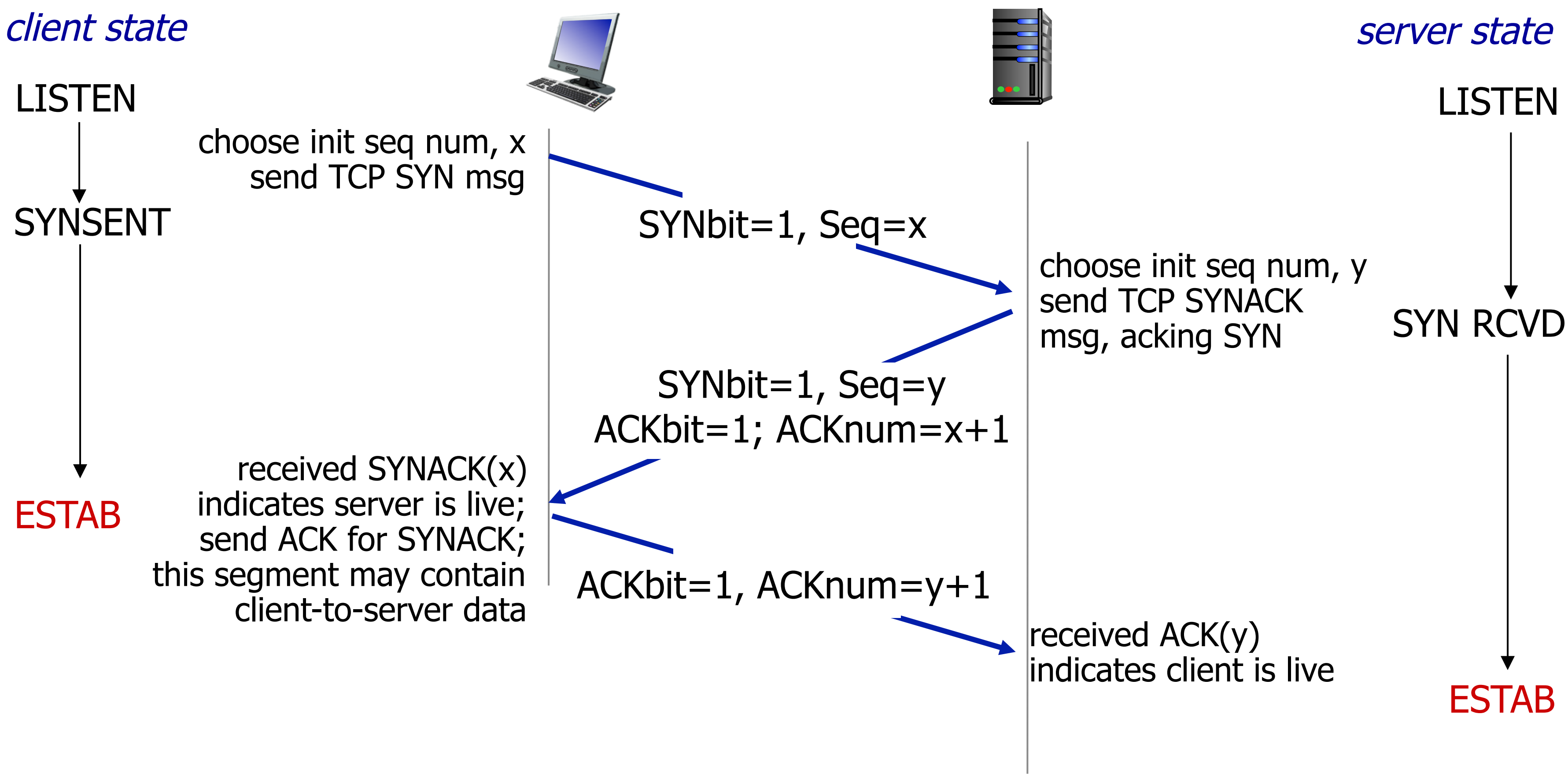
Agreeing to establish a connection

2-way handshake failure scenarios:

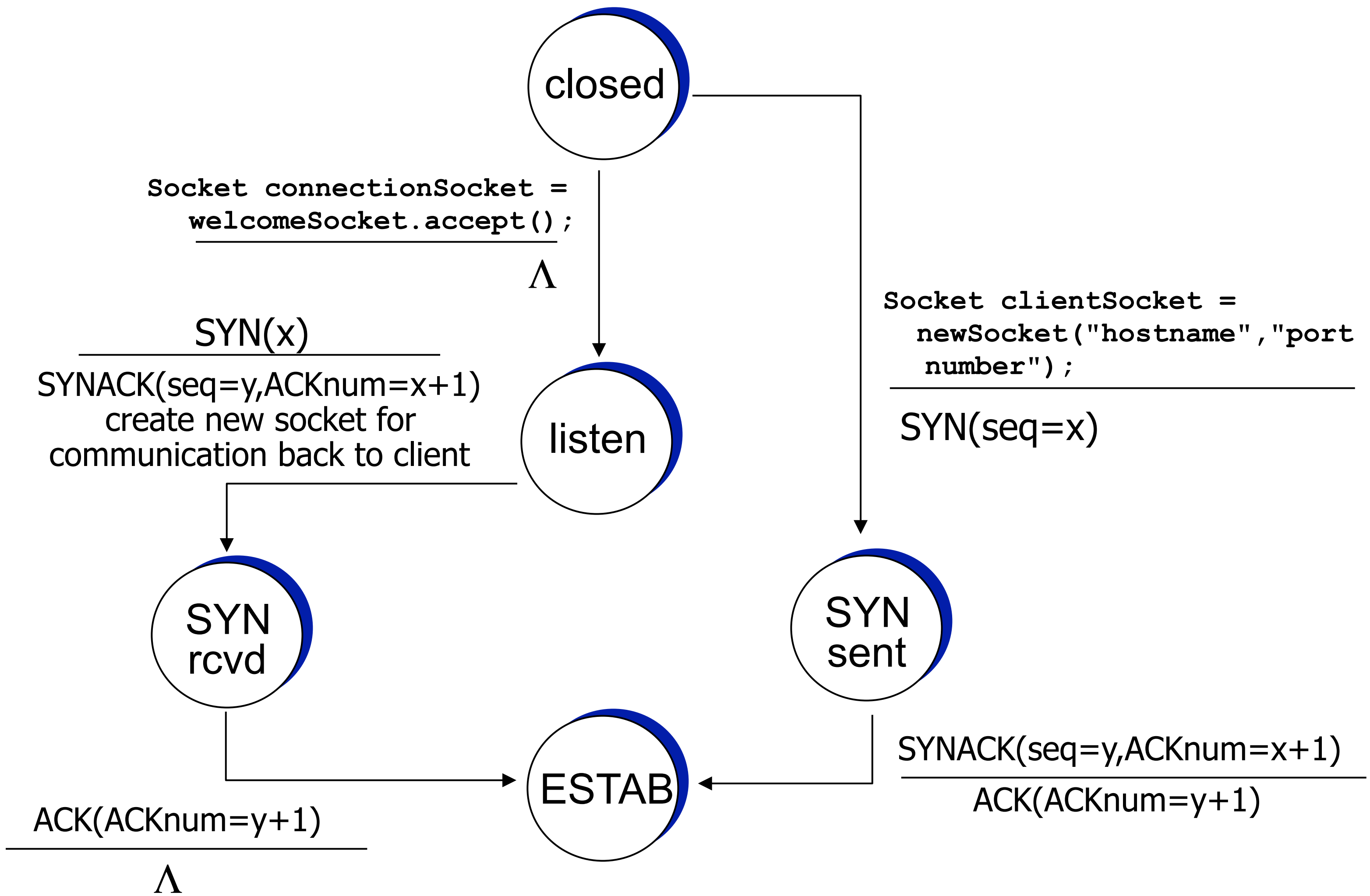




TCP 3-way handshake



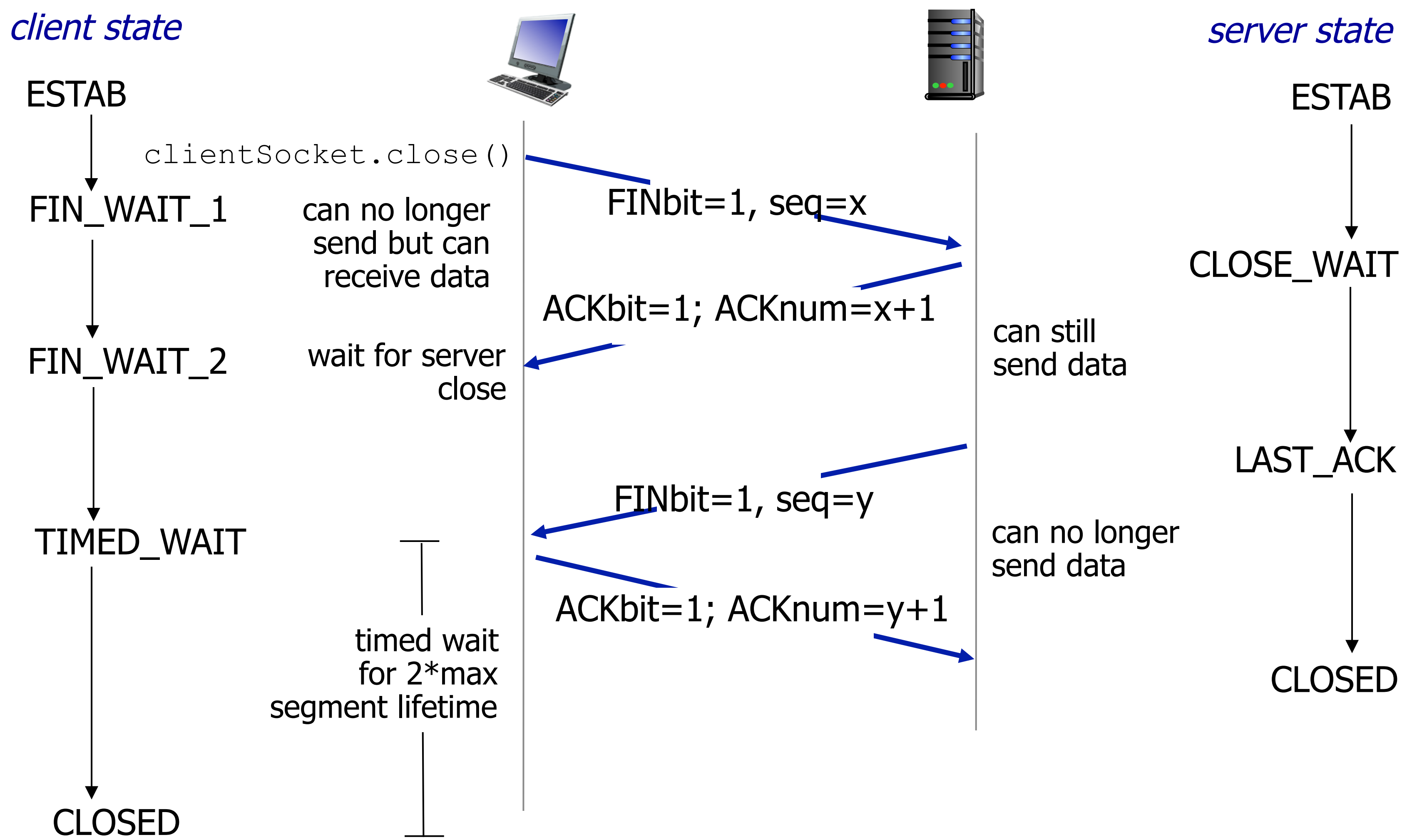
TCP 3-way handshake: FSM



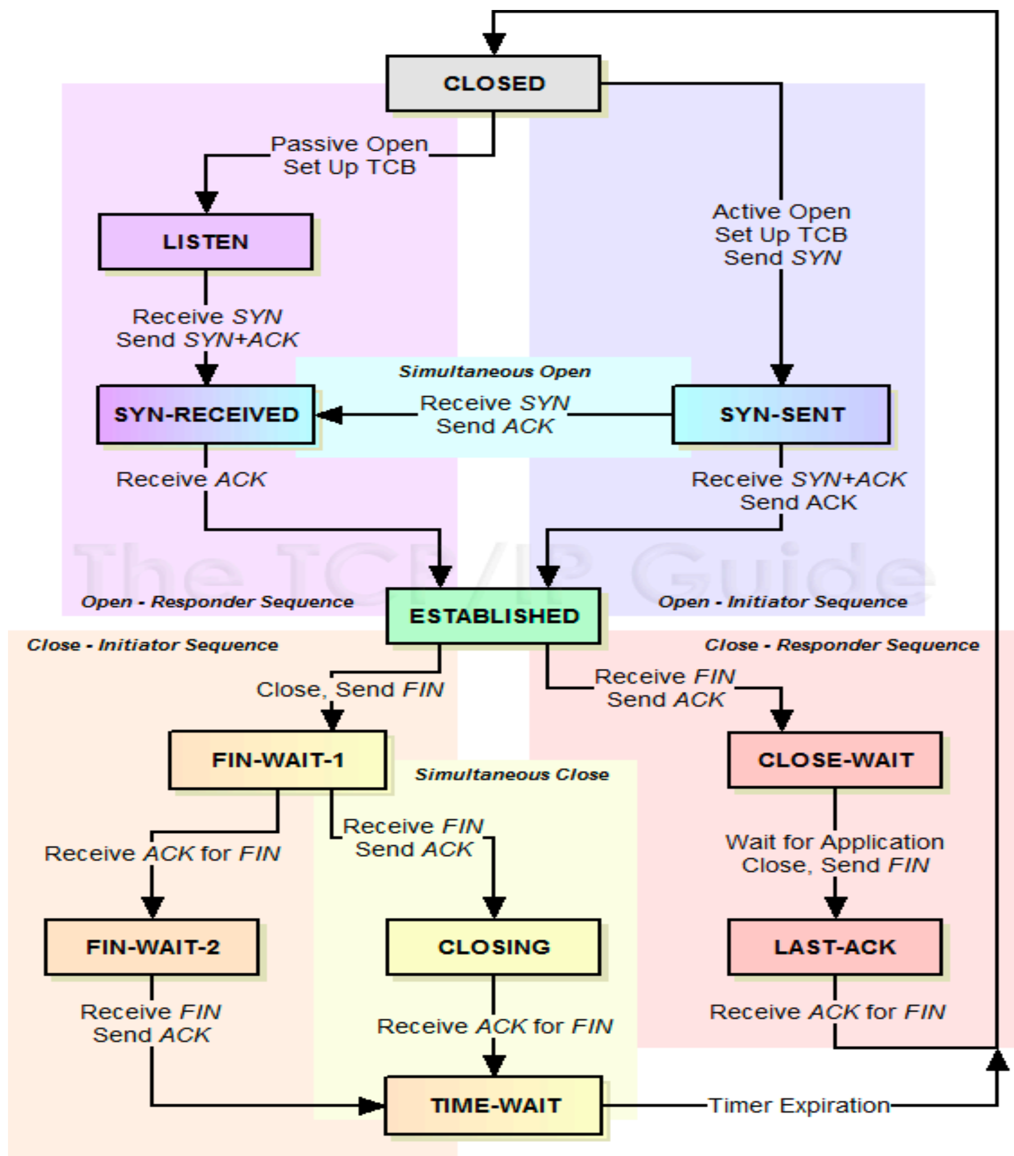
TCP: closing a connection

- 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



TCP: Overall state machine



Transport Layer: Outline

1 transport-layer services

2 multiplexing and demultiplexing

3 connectionless transport: UDP

4 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

5 principles of congestion control

6 TCP congestion control

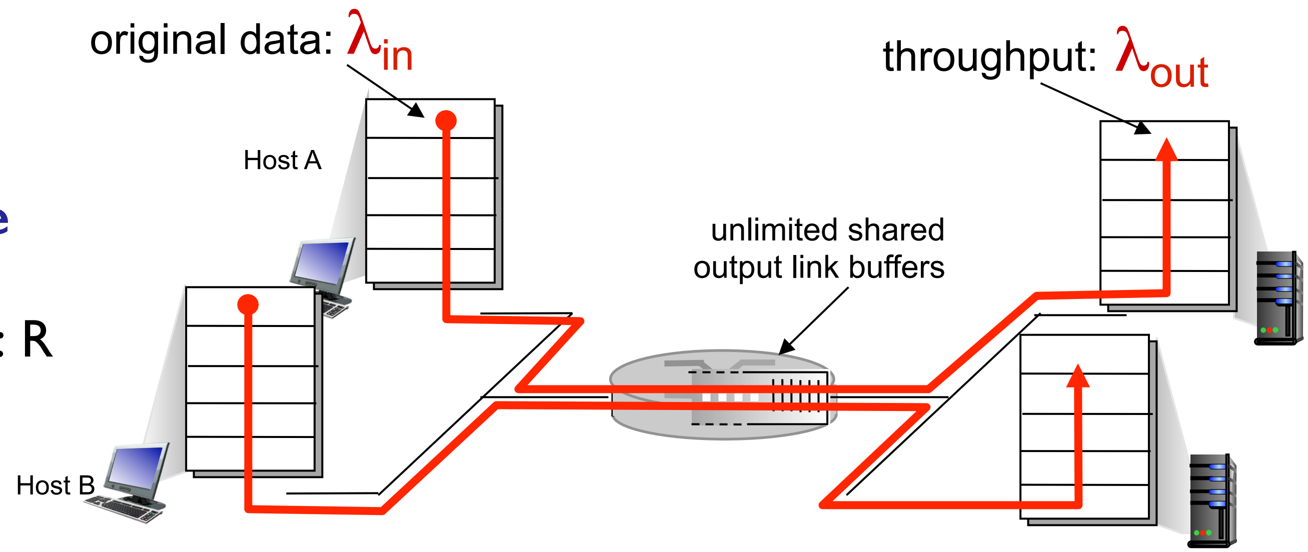
Principles of congestion control

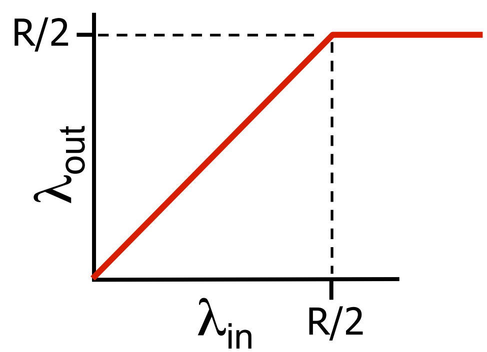
congestion:

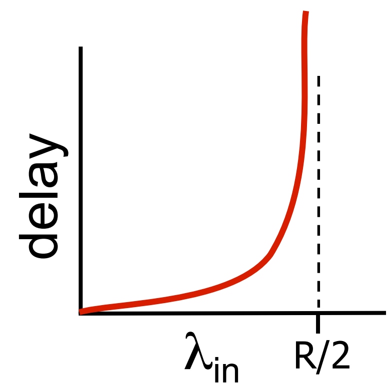
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

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



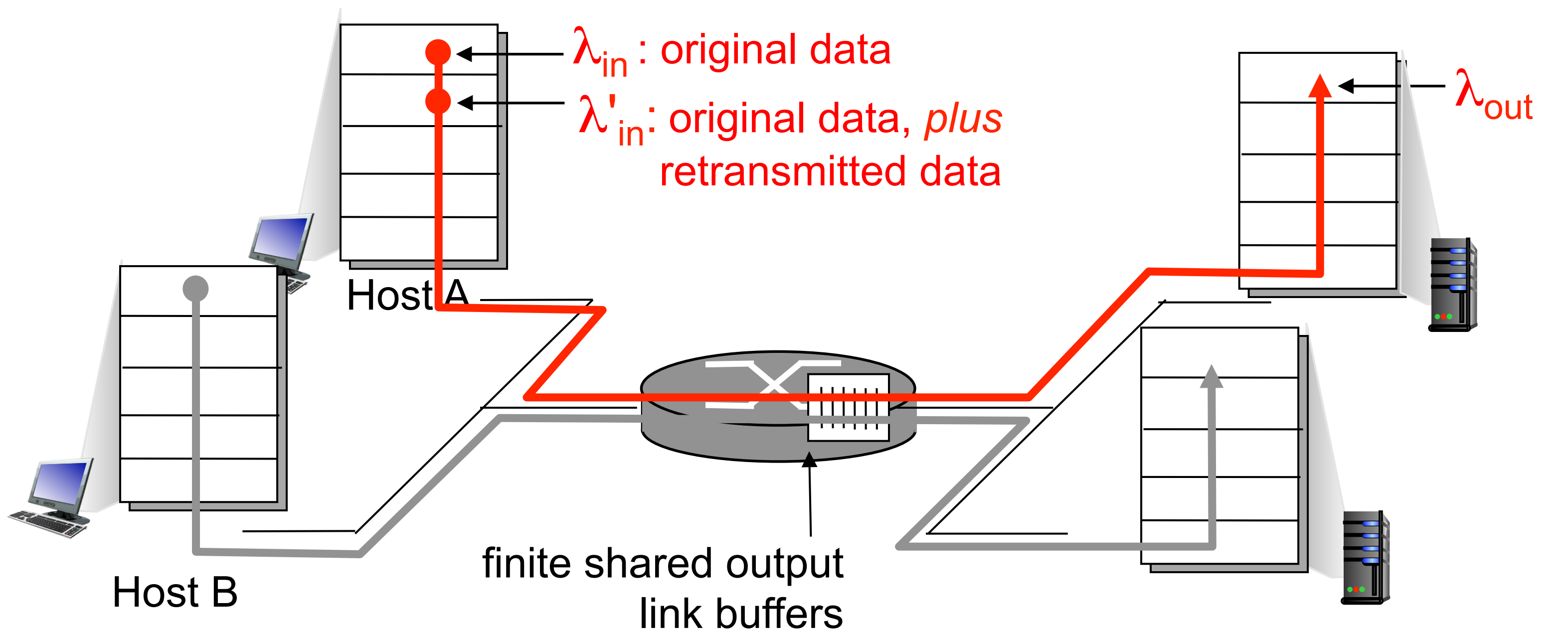




- maximum per-connection throughput: R/2
- large delays as arrival rate, λ_{in}, approaches capacity

Causes/costs of congestion: scenario 2

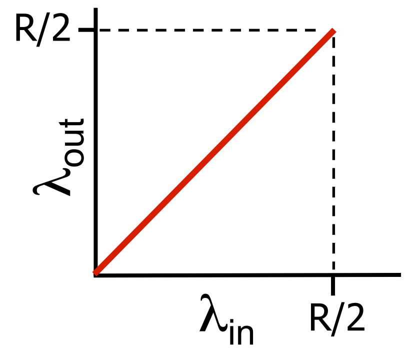
- one router, ***finite*** buffers
- sender retransmission of timed-out packet
  - app-layer input = app-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

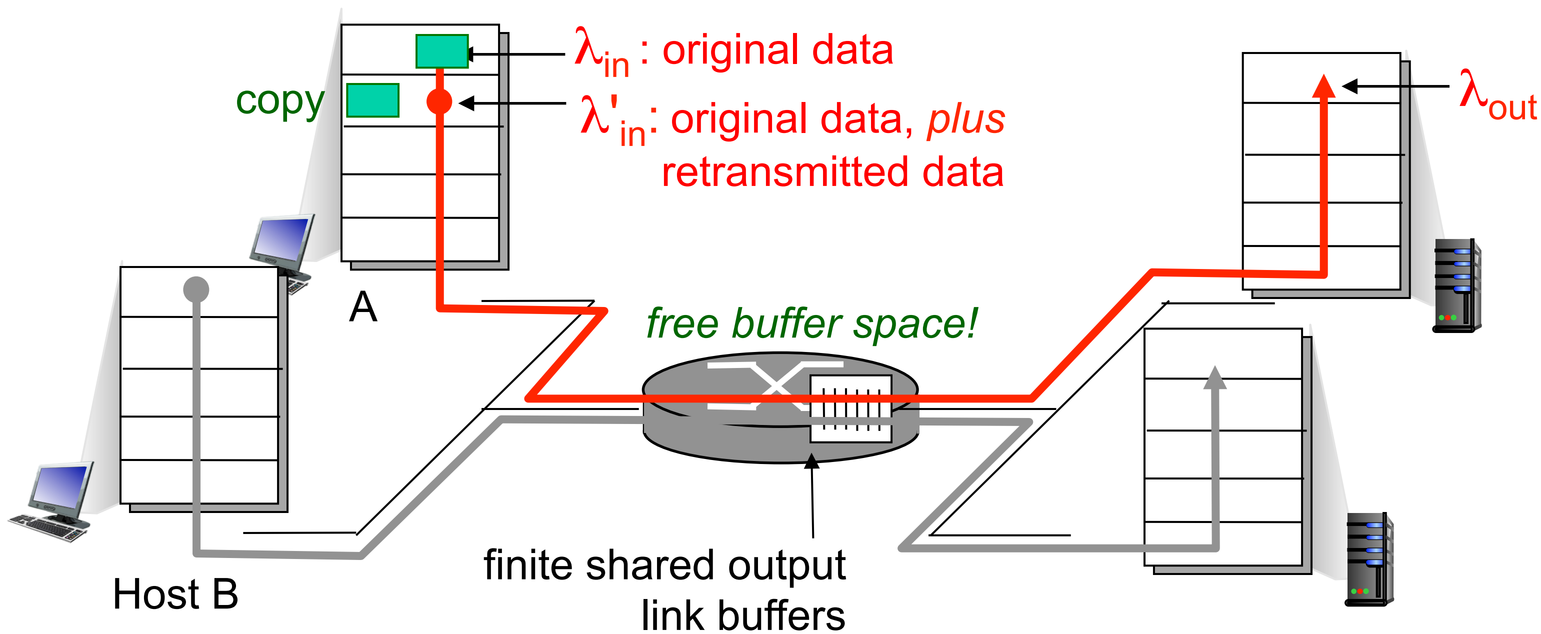


Causes/costs of congestion: scenario 2

idealization: perfect knowledge

- sender sends only when router buffers available

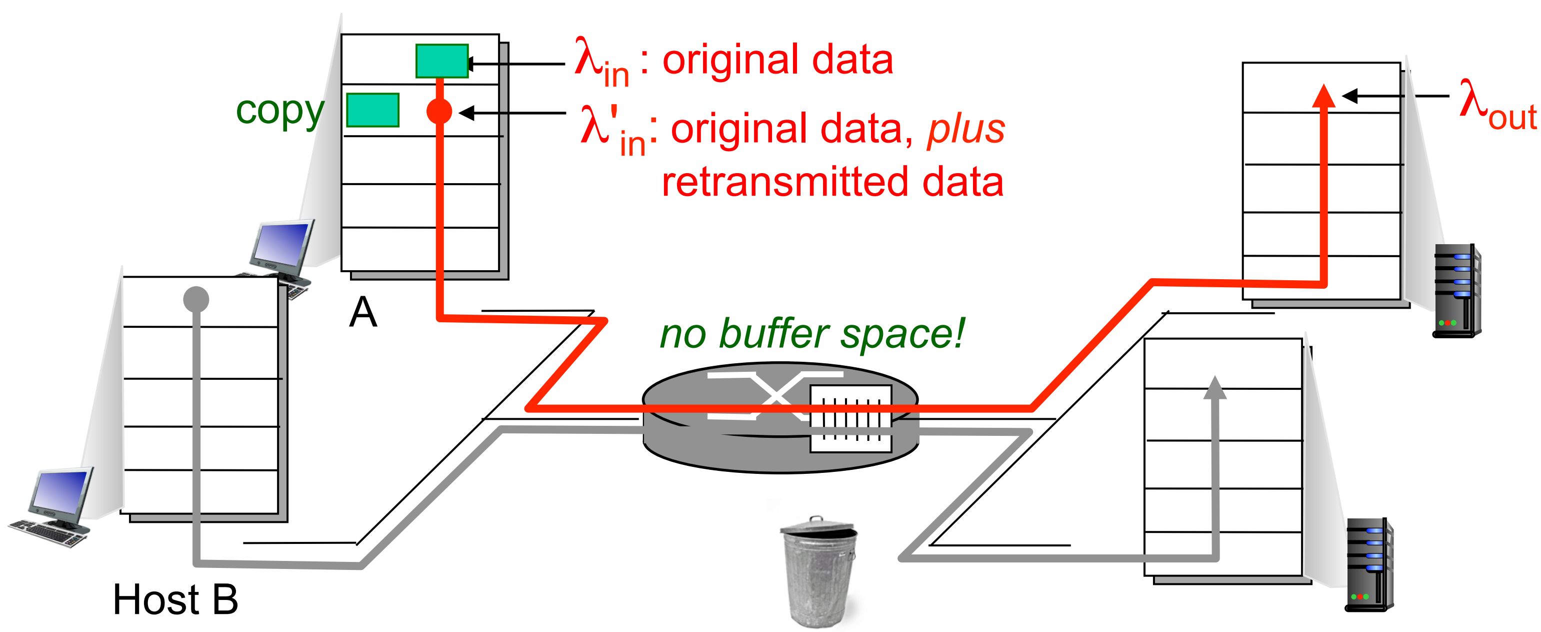




Causes/costs of congestion: scenario 2

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

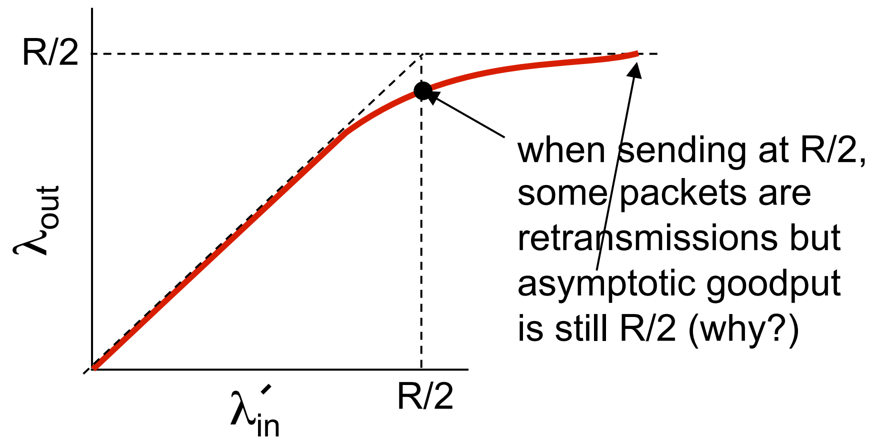
- sender only resends if packet *known* to be lost

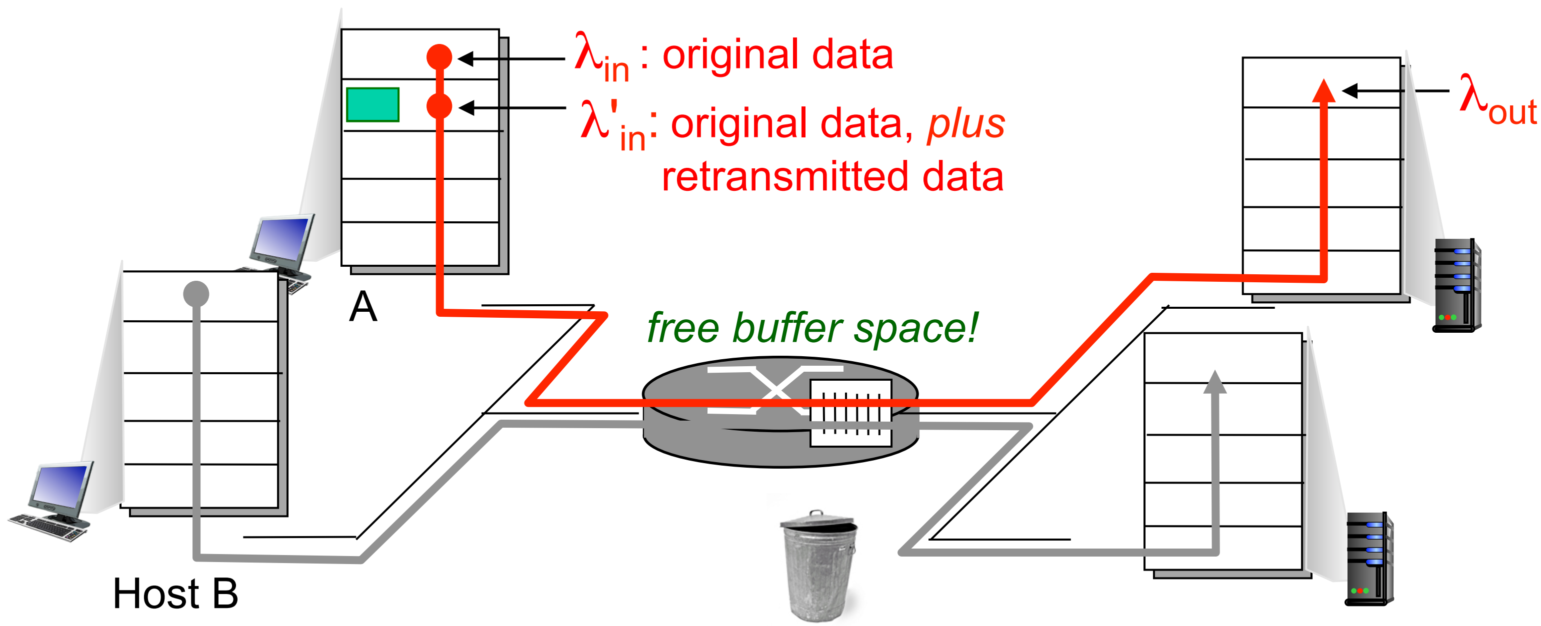


Causes/costs of congestion: scenario 2

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

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

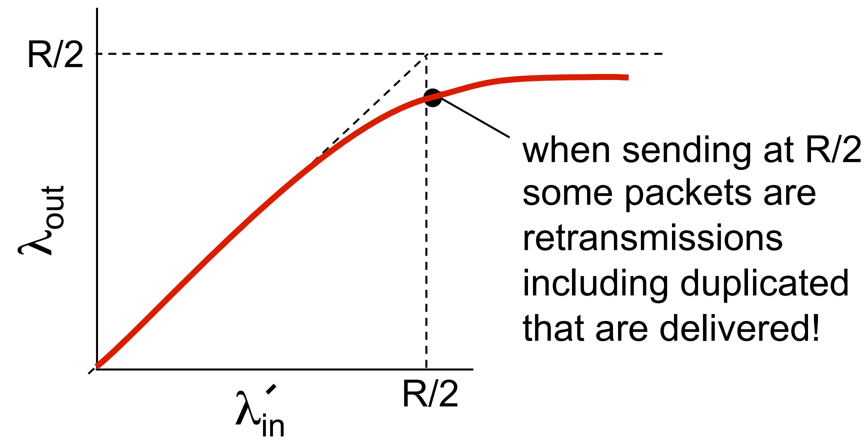


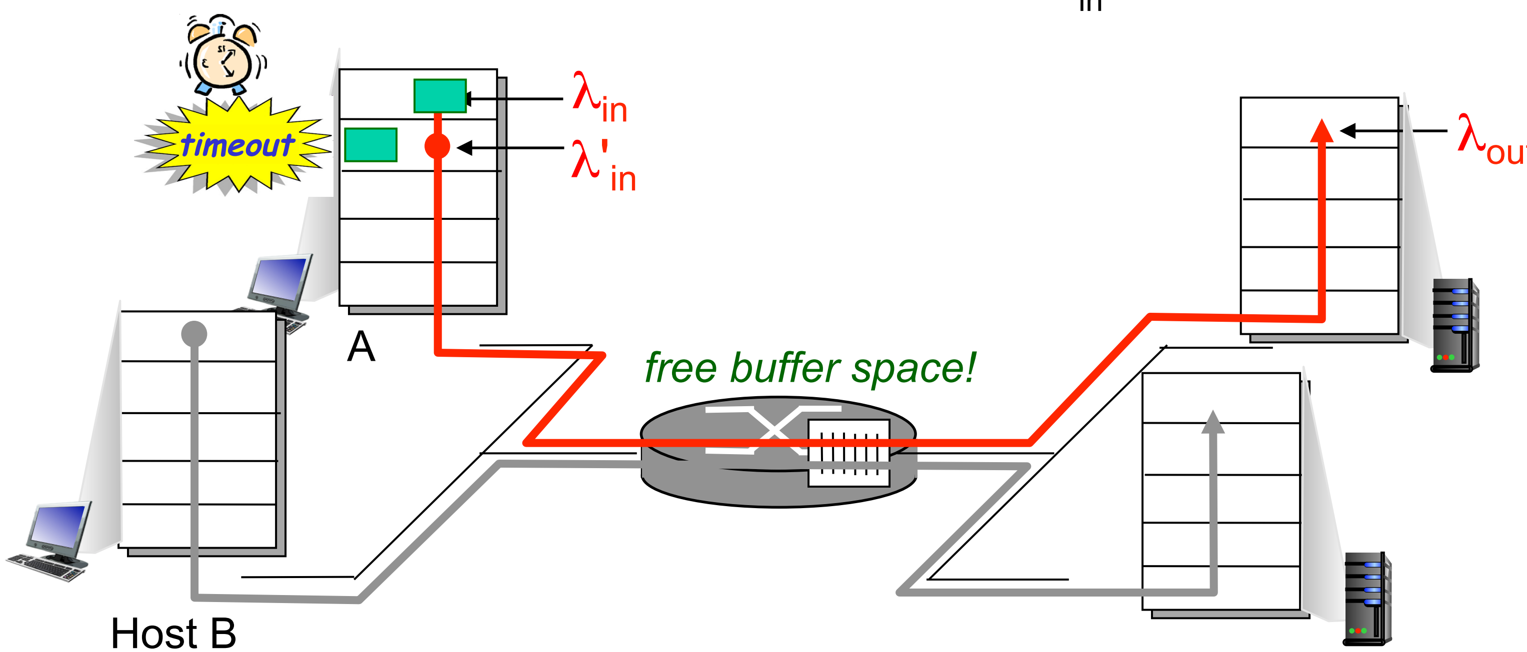


Causes/costs of congestion: scenario 2

Realistic: duplicates

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered

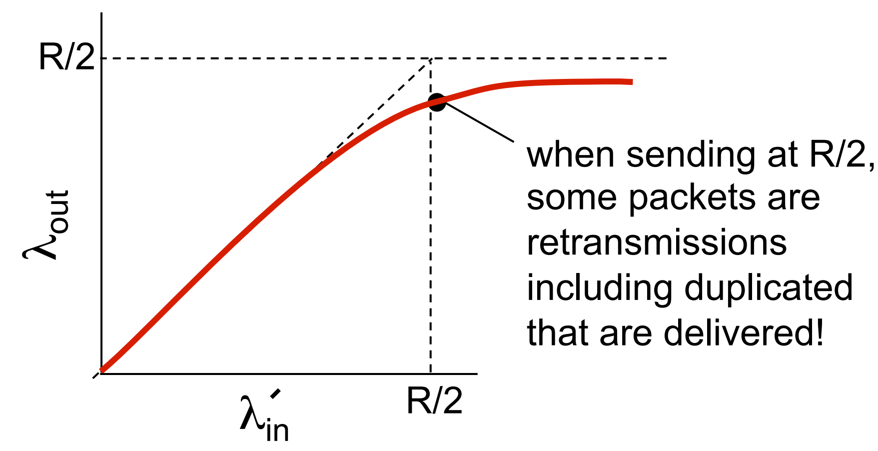




Causes/costs of congestion: scenario 2

Realistic: duplicates

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered



"costs" of congestion:

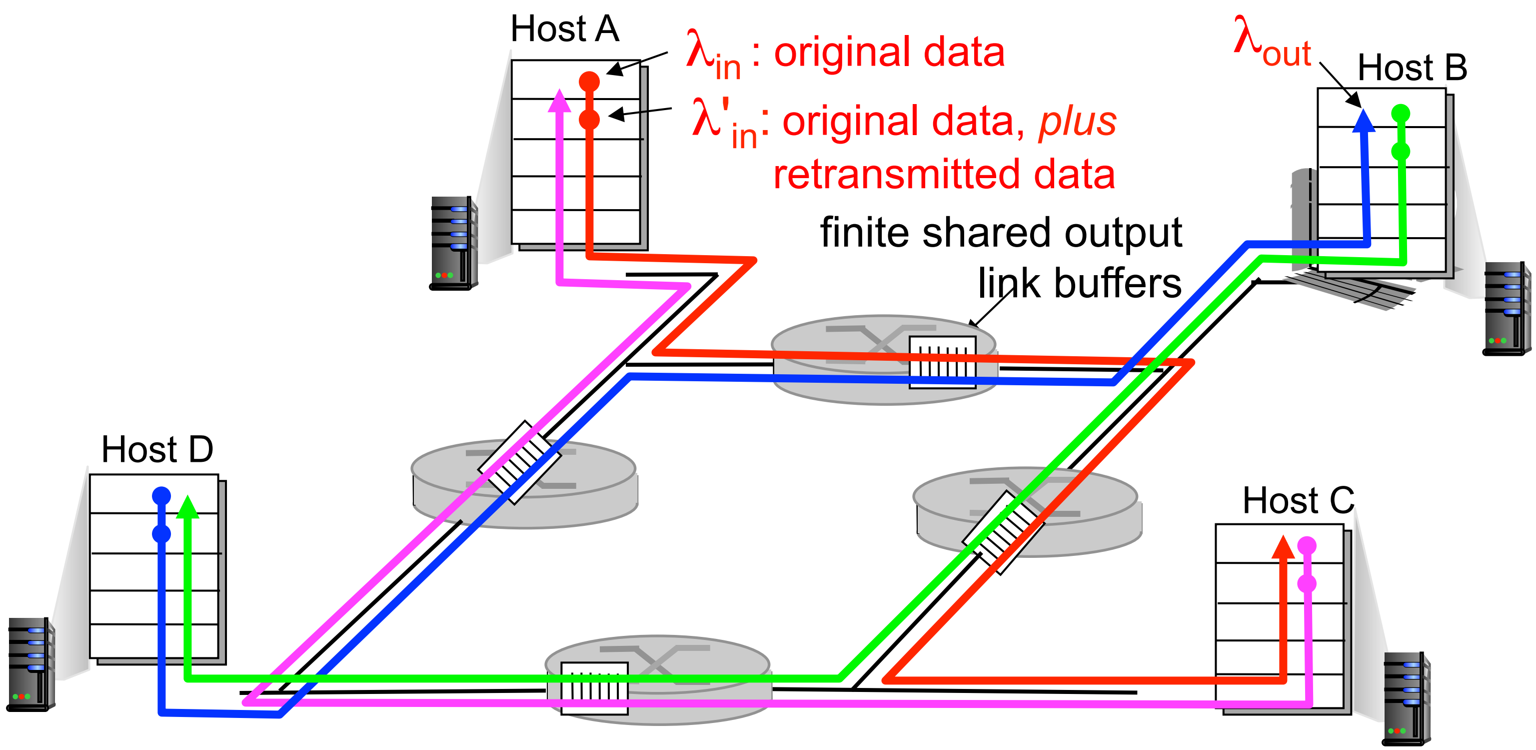
- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

Causes/costs of congestion: scenario 3

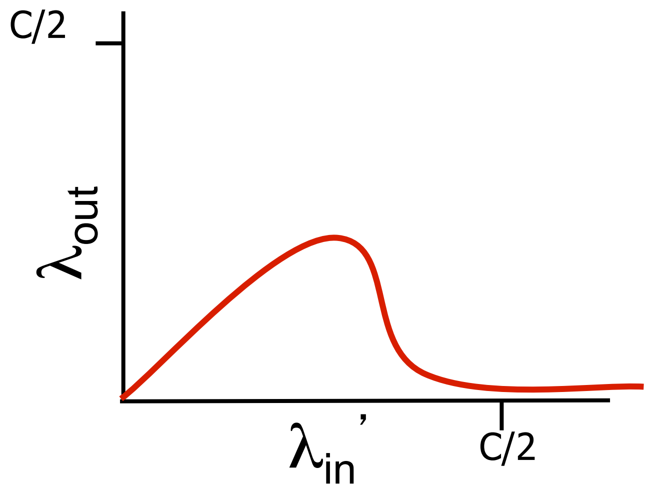
- four senders
- multihop paths
- timeout/retransmit

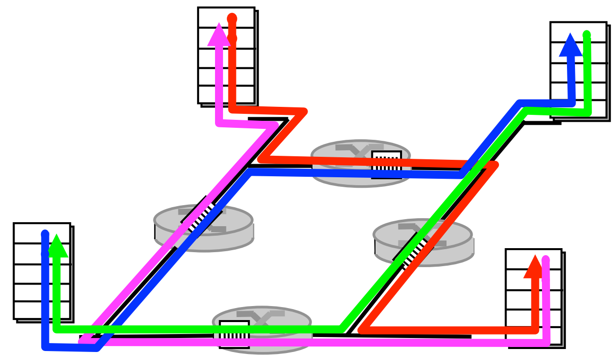
Q: what happens as λ_{in} and λ'_{in} increase ?

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



Causes/costs of congestion: scenario 3





another "cost" of congestion:

- when packet dropped, any "upstream bandwidth used for that packet wasted!

Approaches towards congestion control

two broad approaches towards congestion control:

end-end congestion control:

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

network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate for sender to send at

Case study: ATM ABR congestion control

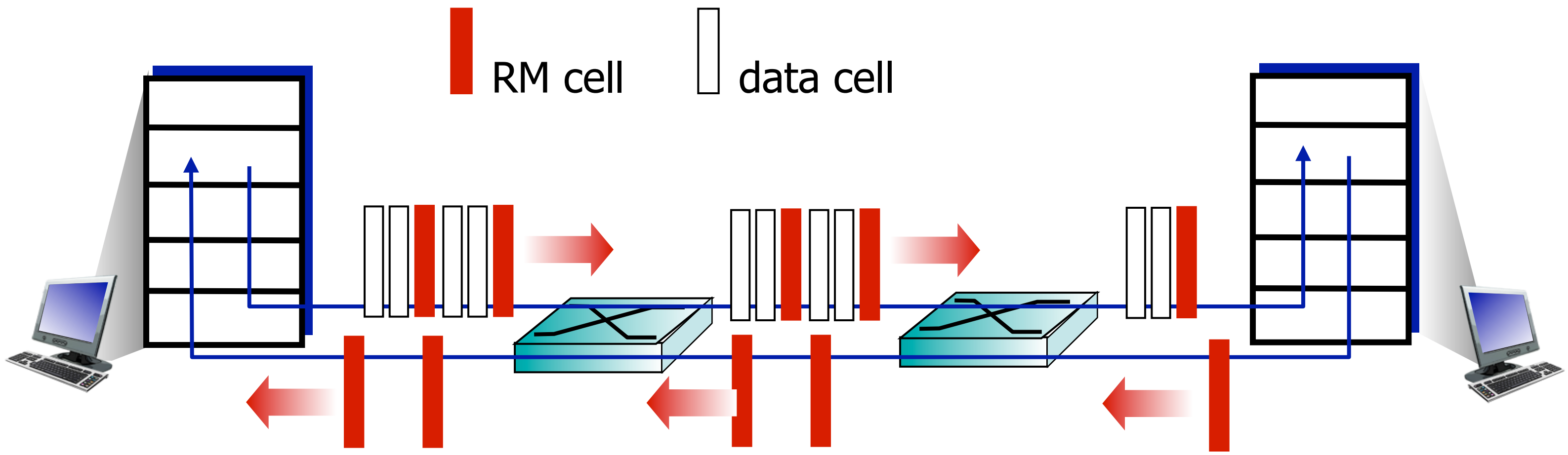
ABR: available bit rate:

- "elastic service"
- if sender's path "underloaded":
  - sender should use available bandwidth
- if sender's path congested:
  - sender throttled to minimum guaranteed rate

RM (resource management) cells:

- sent by sender, interspersed with data cells
- bits in RM cell set by switches ("*network-assisted*")
  - *NI bit:* no increase in rate (mild congestion)
  - *CI bit:* congestion indication
- RM cells returned to sender by receiver, with bits intact

Case study: ATM ABR congestion control



- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - senders' send rate thus max supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, receiver sets CI bit in returned RM cell

Transport Layer: Outline

1 transport-layer services

2 multiplexing and demultiplexing

3 connectionless transport: UDP

4 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

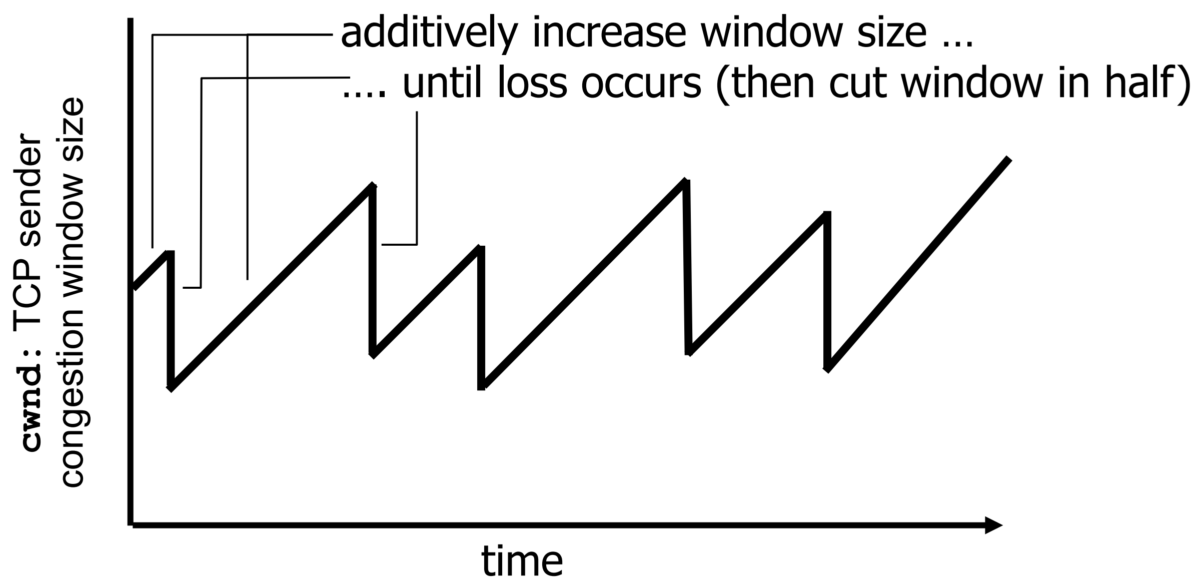
5 principles of congestion control

6 TCP congestion control

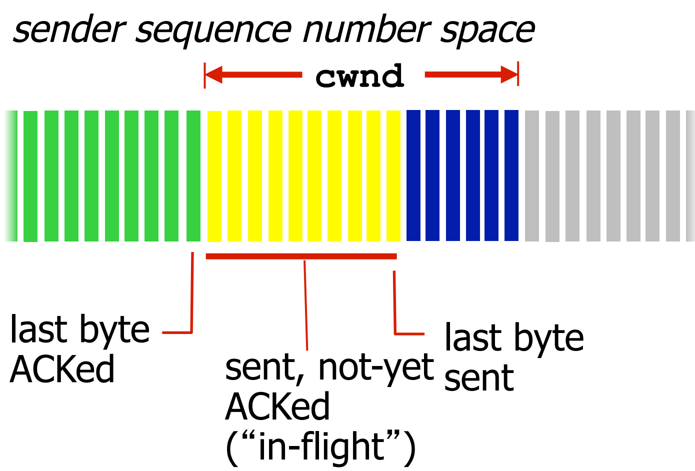
TCP congestion control: additive increase multiplicative decrease

- *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



TCP congestion control window



- sender limits transmission:

`LastByteSent - LastByteAcked` $\leq$ `cwnd`

- **`cwnd`** is dynamic, function of perceived congestion

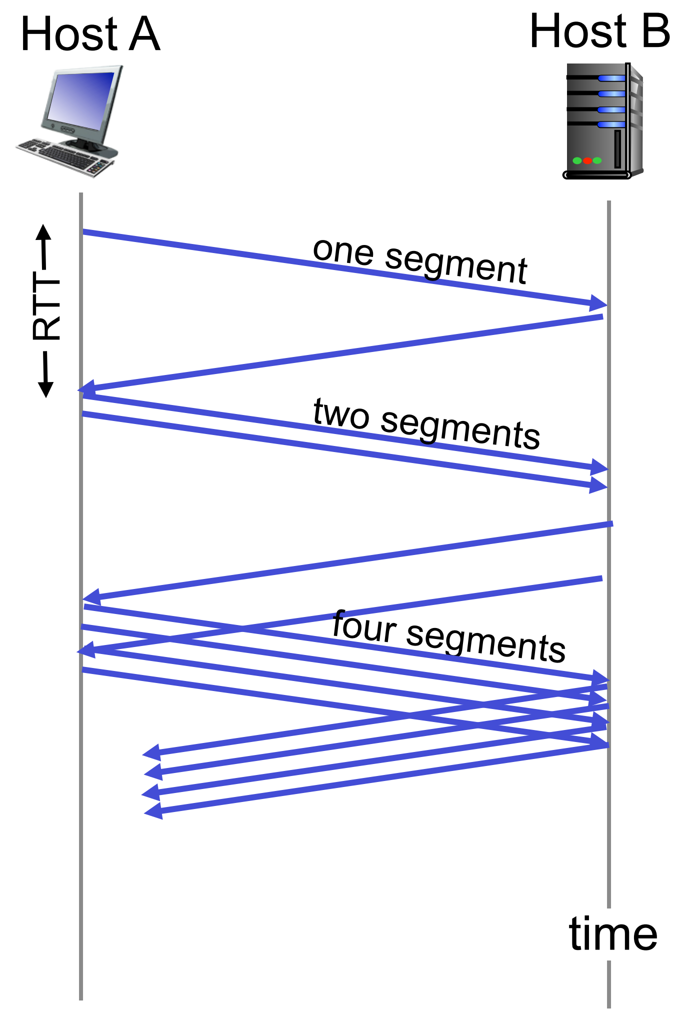
TCP sending rate:

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

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

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`** upon every ACK
- *summary:* initial rate is slow but ramps up exponentially fast



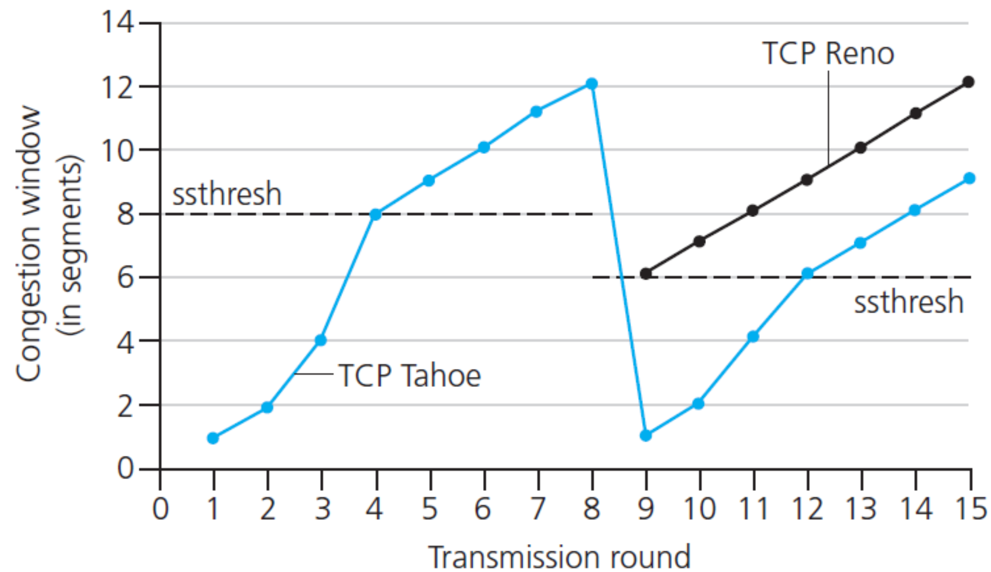
TCP: detecting, reacting to loss

- 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: slow start → cong. avoidance

Q: when should the exponential increase switch to linear?

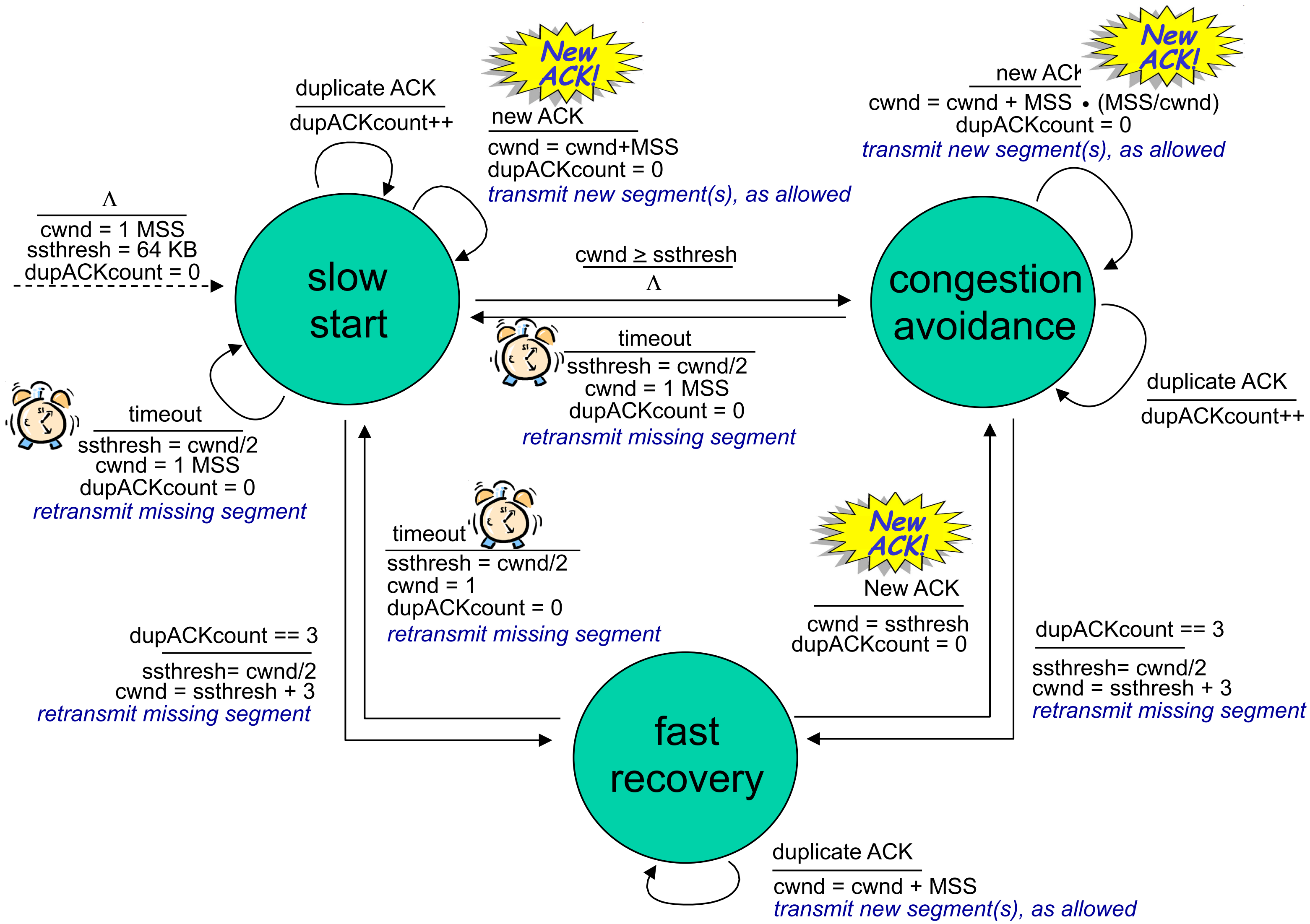
A: when **cwnd** gets to 1/2 of its value before timeout.



Implementation:

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

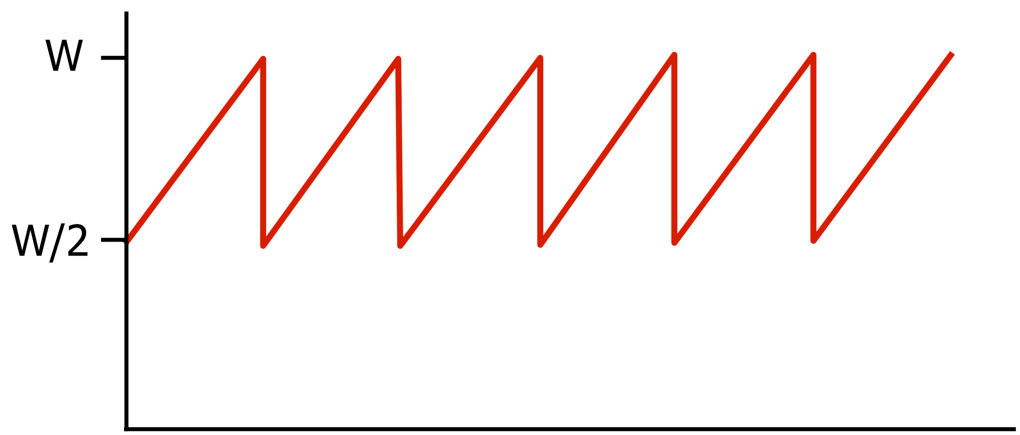
Summary: TCP Congestion Control



TCP throughput: Simplistic model

- avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. throughput is 3/4W per RTT

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



In practice, W not known or fixed, so this model is too simplistic to be useful

TCP throughput: More practical model

- Throughput in terms of segment loss probability, L, round-trip time T, and maximum segment size M [Mathis et al. 1997]:

$$\text{TCP throughput} = \frac{1.22 \cdot M}{T\sqrt{L}}$$

TCP futures: TCP over "long, fat pipes"

- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires W = 83,333 in-flight segments as per the throughput formula

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

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

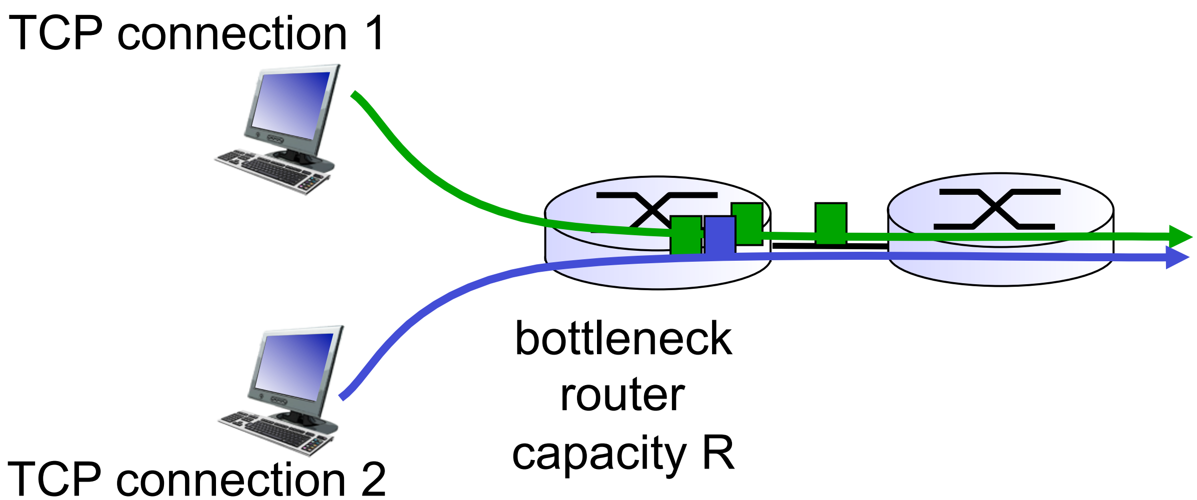
- new versions of TCP for high-speed

TCP throughput wrap-up

- Assume sender window `cwnd`, receiver window `rwnd`, bottleneck capacity C, round-trip time T, path loss rate L, maximum segment size MSS. Then,
  - Instantaneous TCP throughput =
    - min(C, `cwnd/T`,`rwnd/T`)
  - Steady-state TCP throughput =
    - min(C, 1.22M/(T$\sqrt{L}$))

TCP Fairness

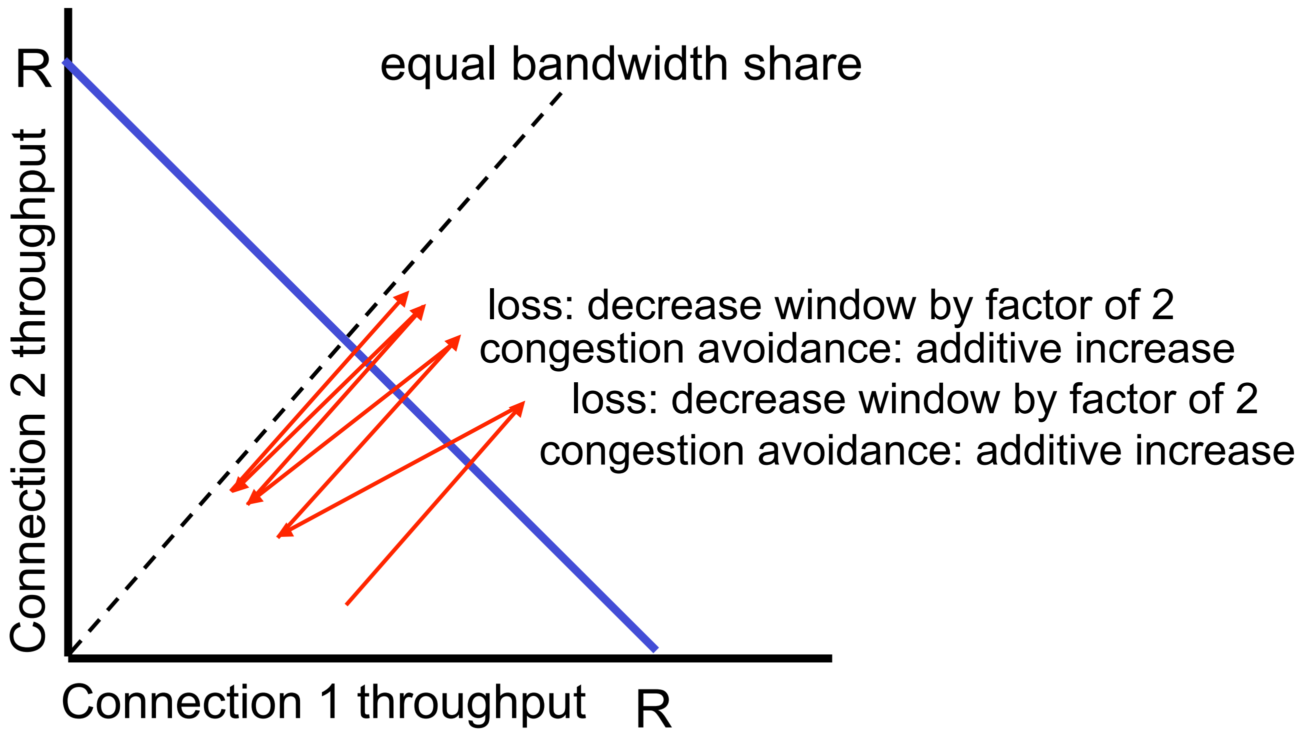
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

two competing sessions:

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



Fairness (more)

Fairness and UDP

- multimedia apps often do not use TCP
  - rate throttling by congestion control can hurt streaming quality
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss

Fairness, parallel TCP connections

- application can open many 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 R/10
  - new app asks for 11 TCPs, gets R/2