

Chapter 3

Transport Layer

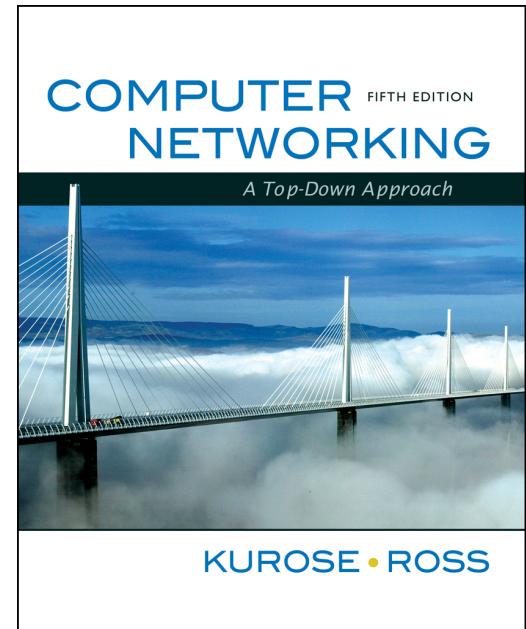
A note on the use of these ppt slides:

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

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

Thanks and enjoy! JFK/KWR

© All material copyright 1996-2011
J.F Kurose and K.W. Ross, All Rights Reserved



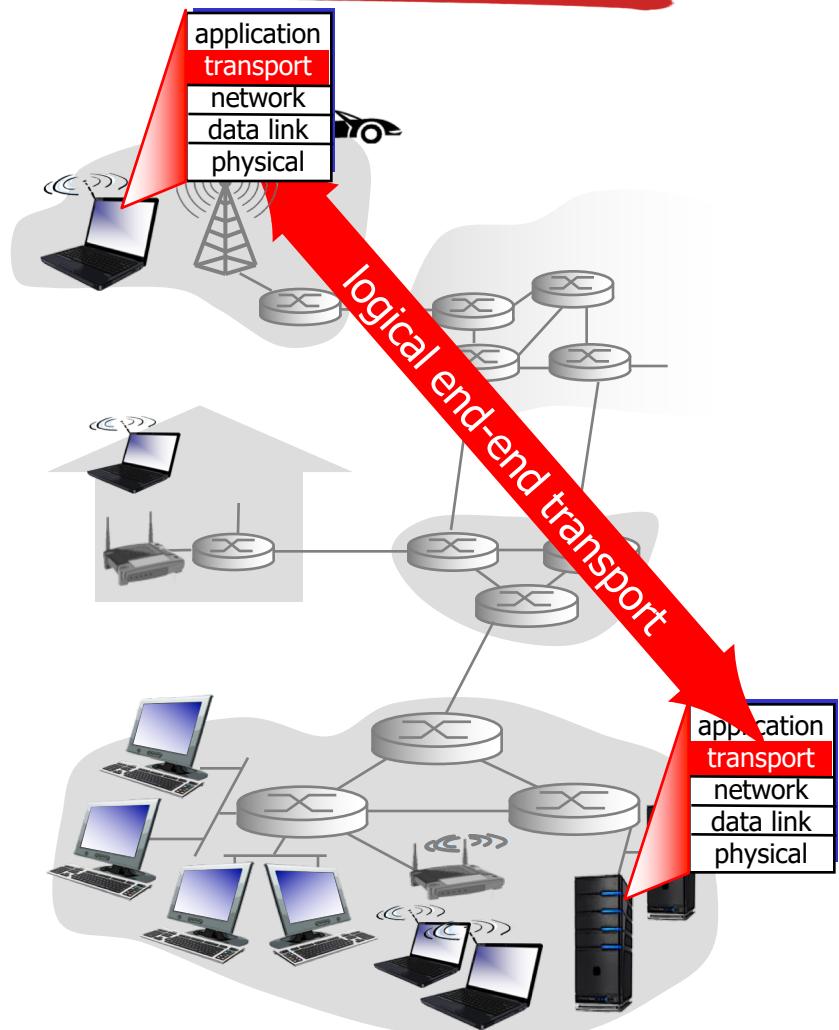
Computer
Networking: A
Top Down
Approach
6th edition
Jim Kurose, Keith Ross
Addison-Wesley
April 2012

Chapter 3: Transport Layer

- ❖ learn about Internet transport layer protocols:
 - UDP: connectionless transport, brief not much to say
 - TCP: connection-oriented reliable transport
 - TCP congestion control

Transport services and protocols

- ❖ provide *logical communication* between app processes running on different hosts
- ❖ transport protocols run in end systems
 - send side: breaks app messages into *segments*, passes to network layer
 - rcv side: reassembles segments into messages, passes to app layer
- ❖ more than one transport protocol available to apps
 - Internet: TCP and UDP
- ❖ no delay/service guarantees



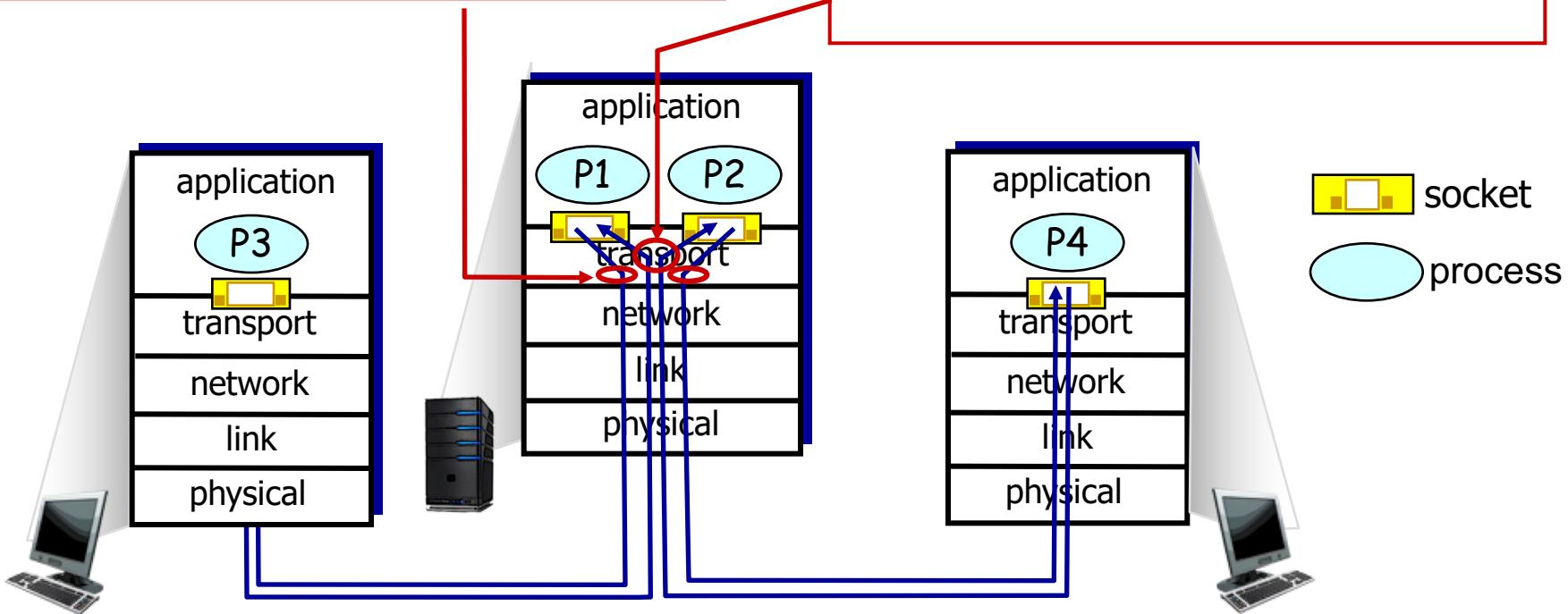
Multiplexing/demultiplexing

multiplexing at sender:

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

demultiplexing at receiver:

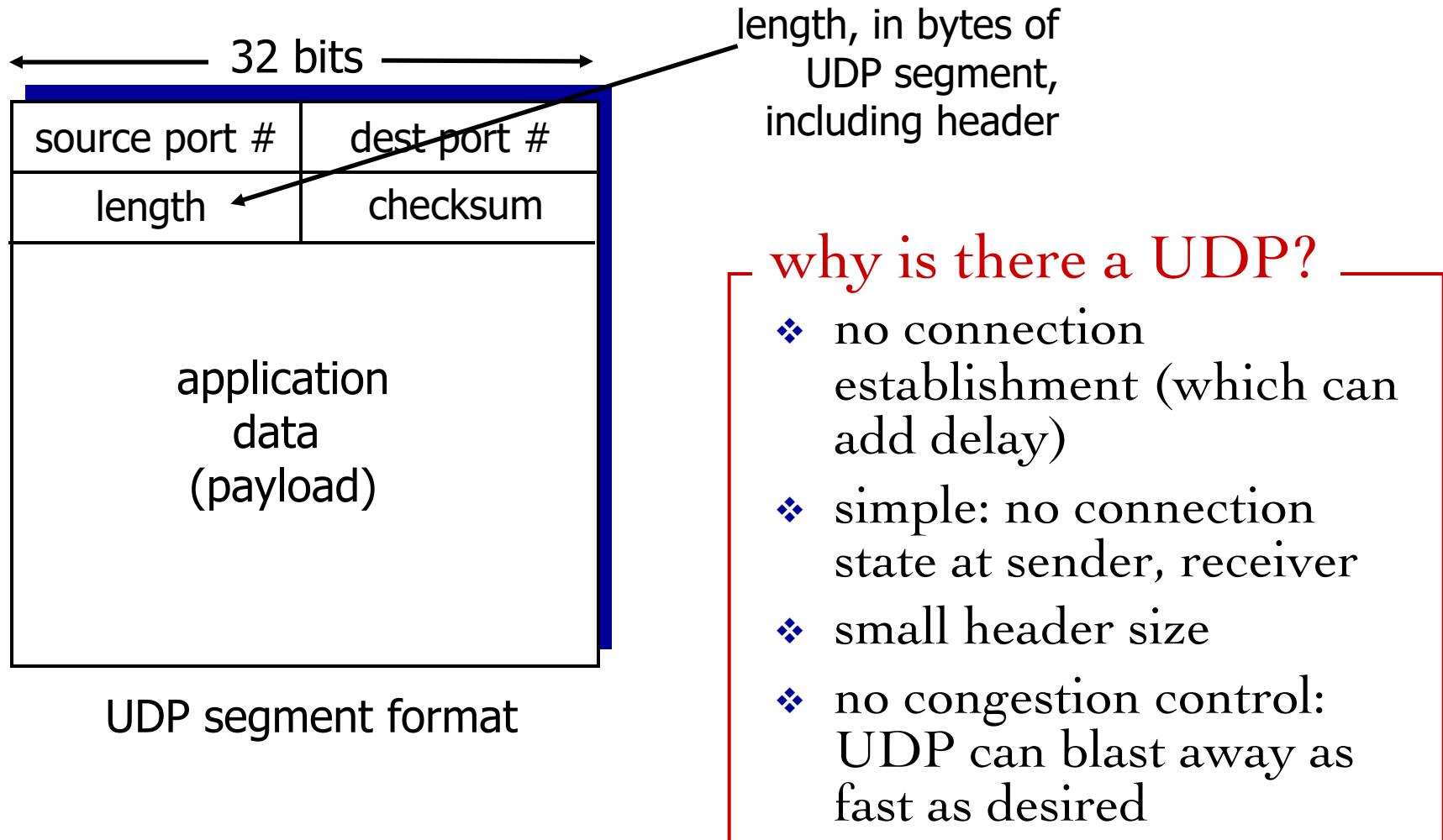
use header info to deliver received segments to correct socket



UDP: User Datagram Protocol [RFC 768]

- ❖ “no frills,” “bare bones” Internet transport protocol
- ❖ “best effort” service, UDP segments may be:
 - lost
 - delivered out-of-order to app
- ❖ *connectionless*:
 - no handshaking between UDP sender, receiver
 - each UDP segment handled independently of others
- ❖ UDP use:
 - streaming multimedia apps (loss tolerant, rate sensitive)
 - DNS
 - SNMP
- ❖ reliable transfer over UDP:
 - add reliability at application layer
 - application-specific error recovery!

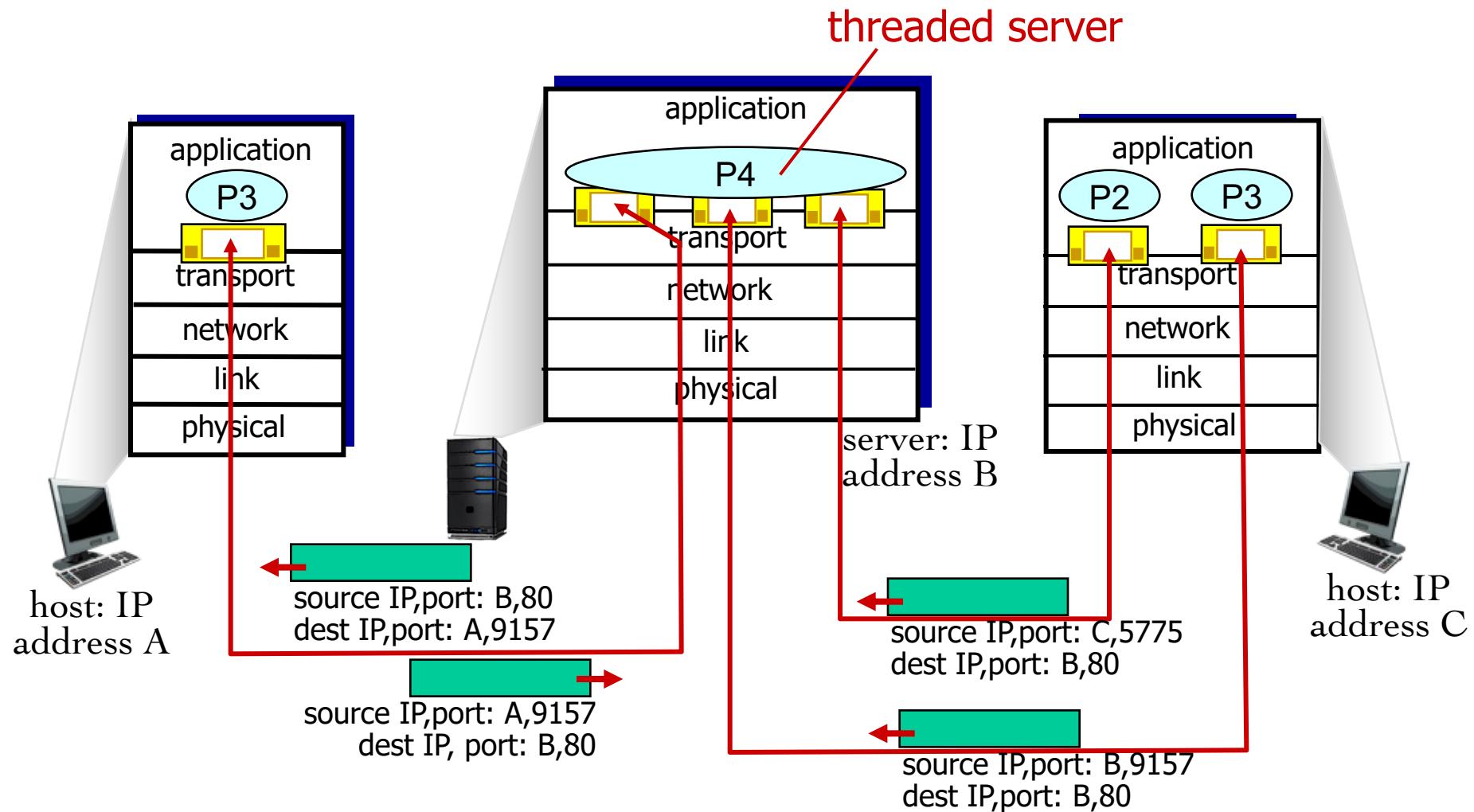
UDP: segment header



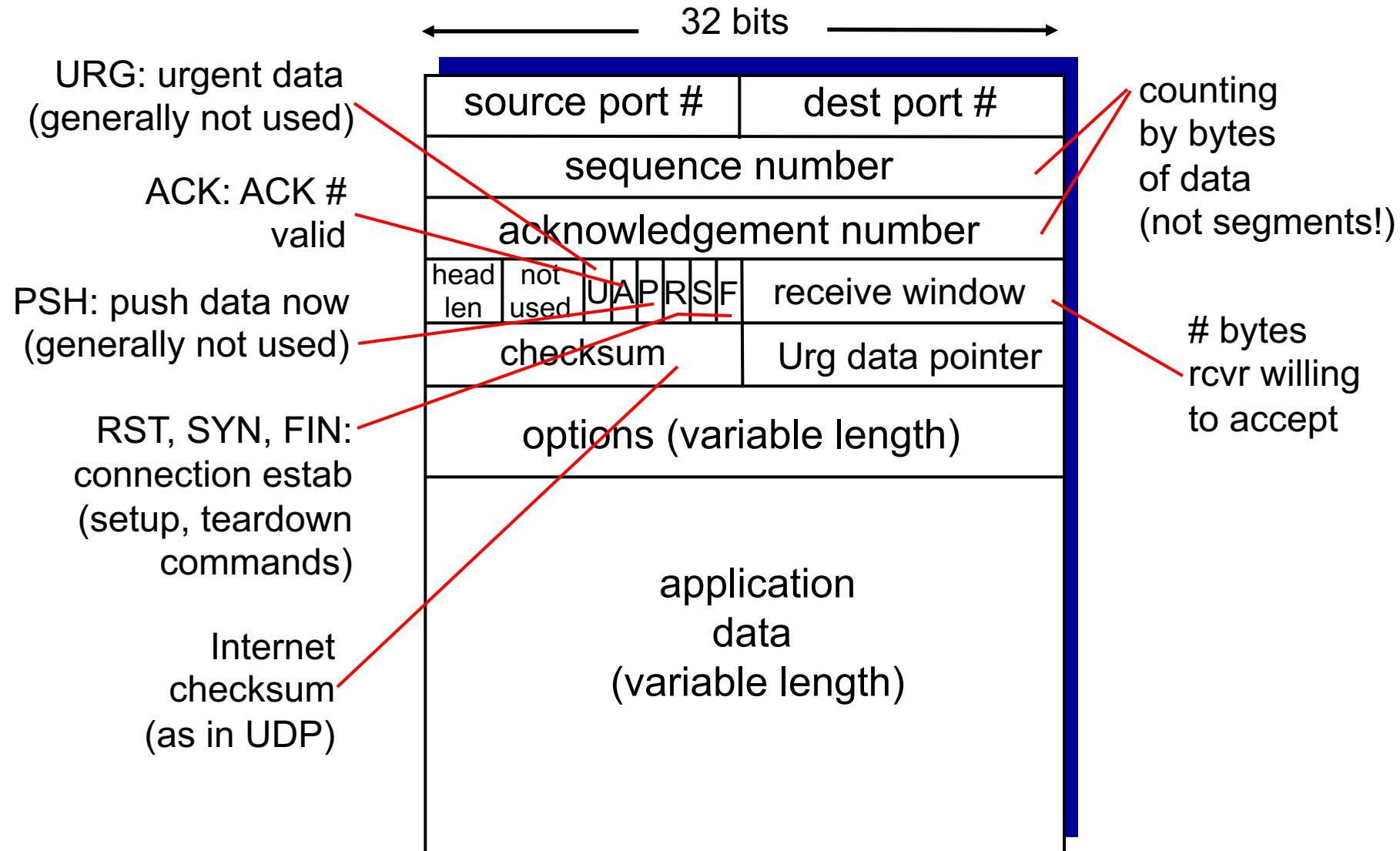
Connection-oriented demux

- ❖ TCP socket identified by 4-tuple:
 - source IP address
 - source port number
 - dest IP address
 - dest port number
- ❖ demux: receiver uses all four values to direct segment to appropriate socket
- ❖ server host may support many simultaneous TCP sockets:
 - each socket identified by its own 4-tuple
- ❖ web servers have different sockets for each connecting client
 - non-persistent HTTP will have different socket for each request

Connection-oriented demux



TCP segment structure



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
 - *Hybrid of go back n and selective repeat*
- ❖ **full duplex data:**
 - bi-directional data flow in same connection
 - MSS: maximum segment size
- ❖ **connection-oriented:**
 - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- ❖ **flow controlled:**
 - sender will not overwhelm receiver

channels with errors *and* loss

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

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

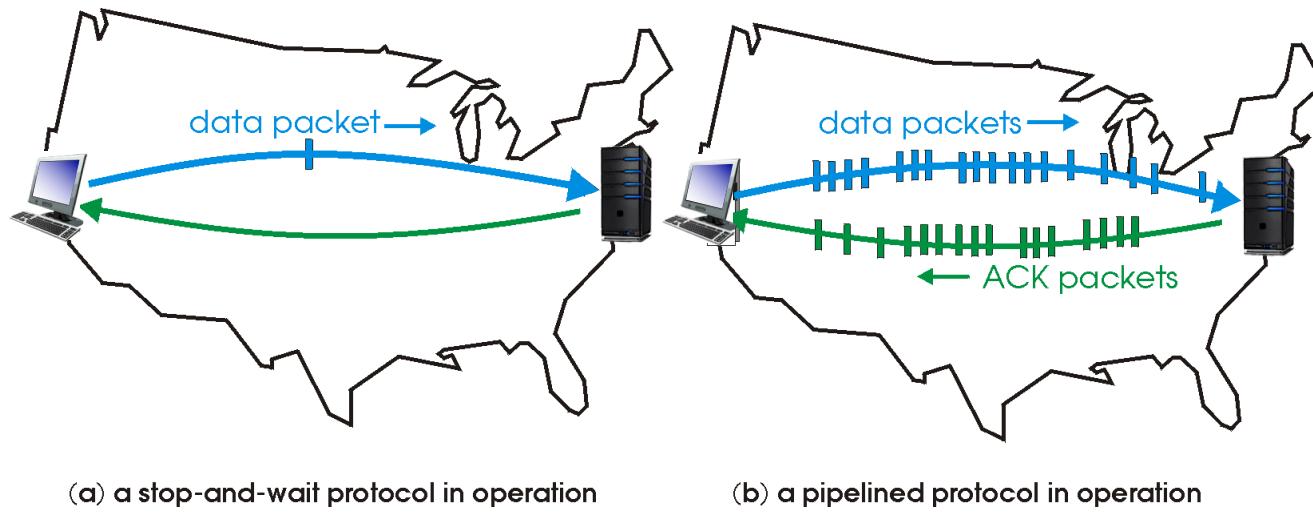
approach: sender waits “reasonable” amount of time for ACK

- ❖ 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

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



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

Pipelined protocols: concepts

Go-back-N:

- ❖ sender can have up to N unacked packets in pipeline
- ❖ receiver only sends *cumulative ack*
 - doesn't ack packet if there's a gap
- ❖ sender has timer for oldest unacked packet
 - when timer expires, retransmit *all* unacked packets

Selective Repeat:

- ❖ sender can have up to N unacked packets in pipeline
- ❖ rcvr sends *individual ack* for each packet
- ❖ sender maintains timer for each unacked packet
 - when timer expires, retransmit only that unacked packet

TCP seq. numbers, ACKs

sequence numbers:

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

acknowledgements:

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

Q: how receiver handles out-of-order segments

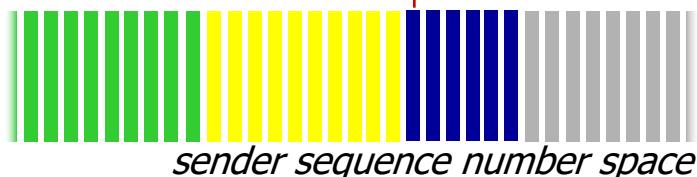
- A: TCP spec doesn’t say,
- up to implementor

outgoing segment from sender

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

window size

N



sent
ACKed

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

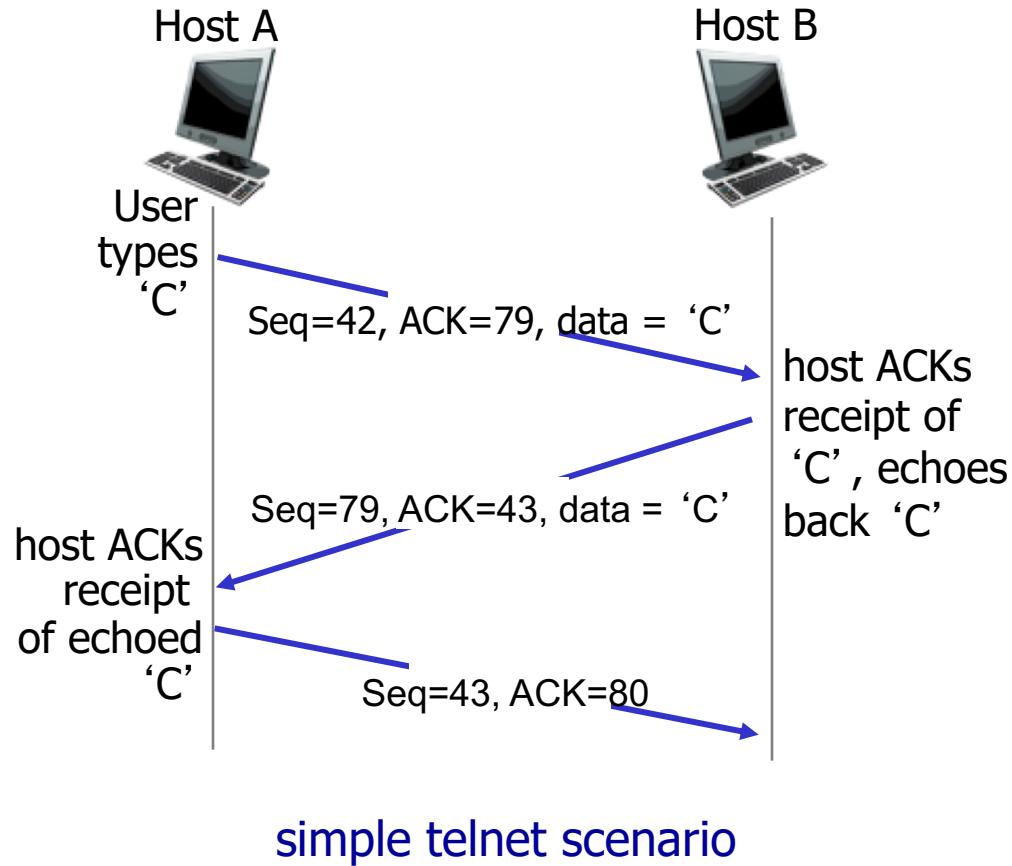
usable
but not
yet sent

not
usable

incoming segment to sender

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

TCP seq. numbers, ACKs



TCP round trip time, timeout

Q: how to set TCP timeout value?

- ❖ longer than RTT
 - but RTT varies
- ❖ *too short*: premature timeout, unnecessary retransmissions
- ❖ *too long*: slow reaction to segment loss

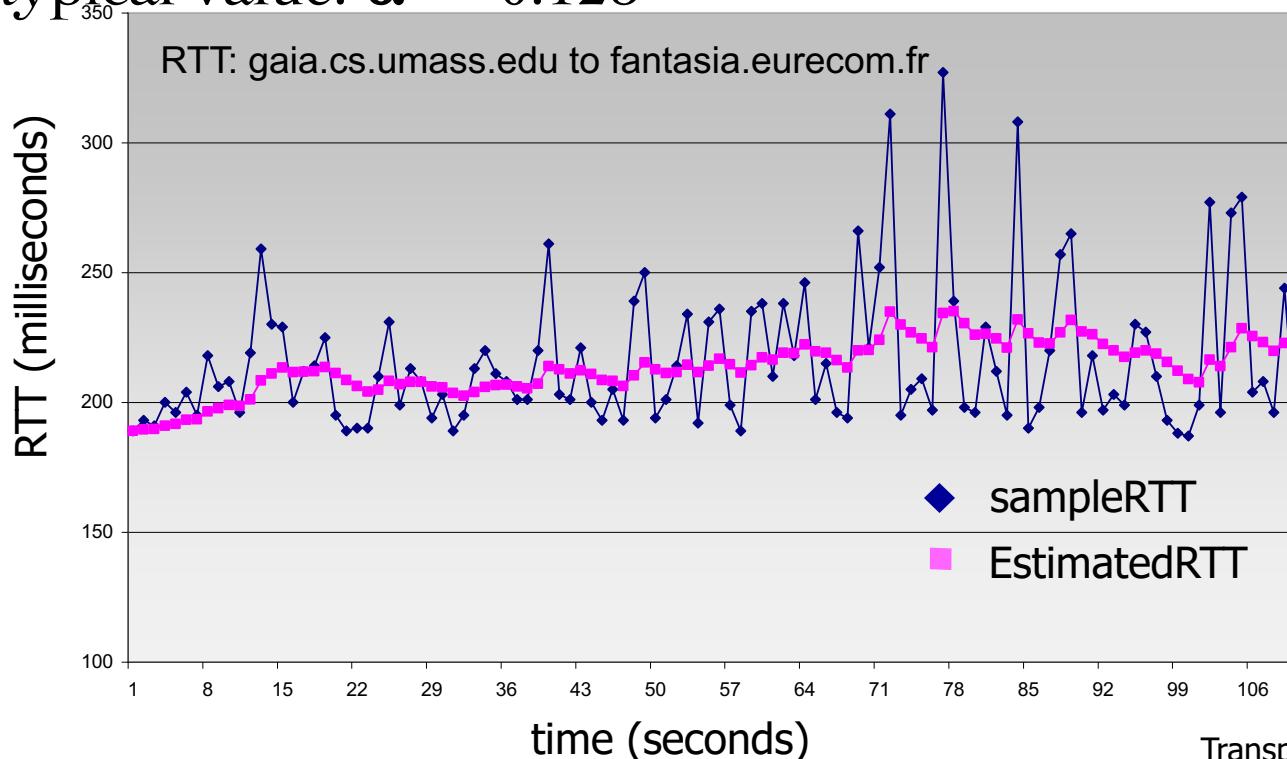
Q: how to estimate RTT?

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

TCP round trip time, timeout

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

- ❖ exponential weighted moving average
- ❖ 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:

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

(typically, $\beta = 0.25$)

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$



↑
estimated RTT

↑
“safety margin”

TCP simple sender (no optimizations, congestion control):

data rcvd from app:

- ❖ 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**

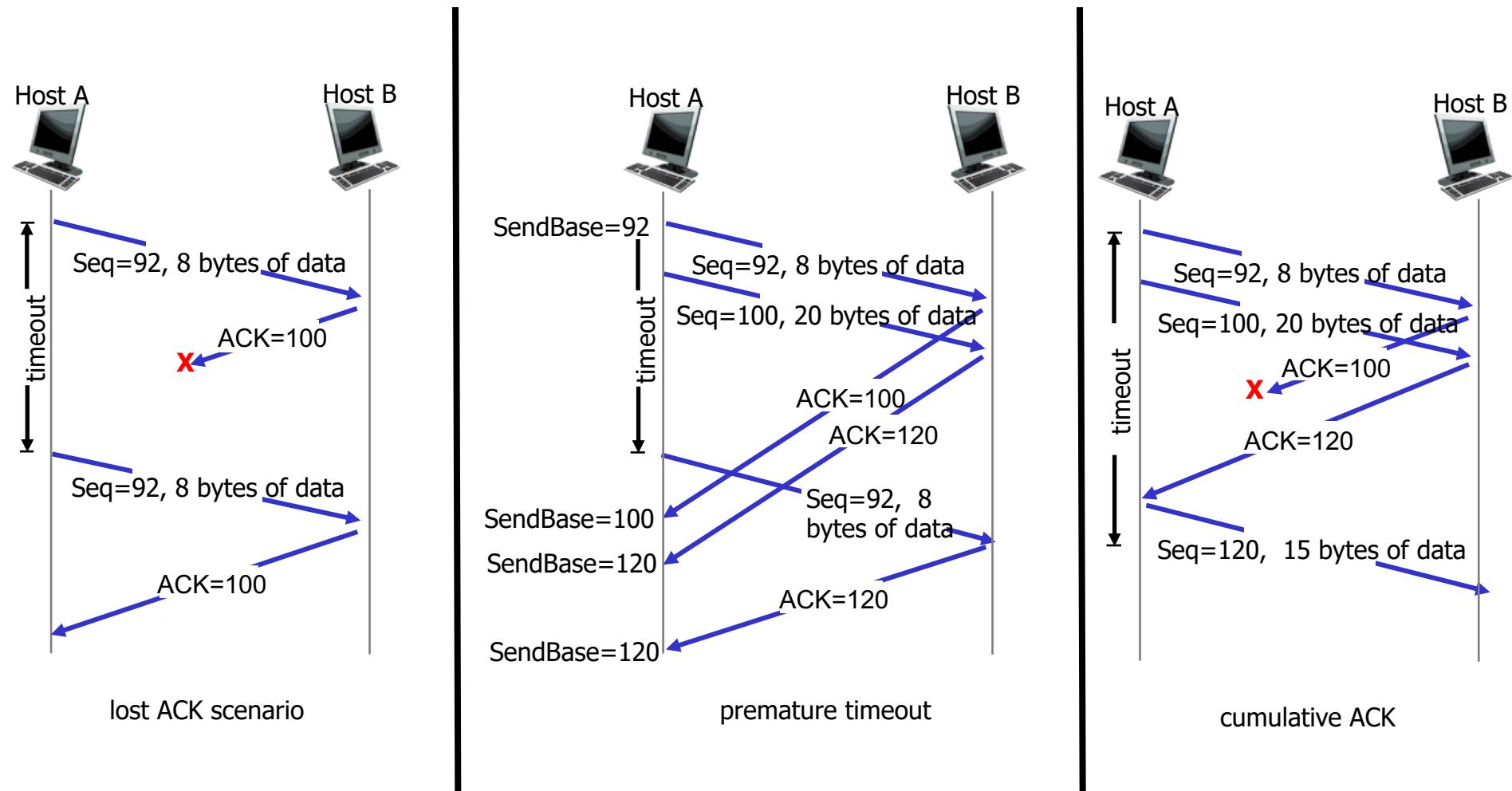
timeout:

- ❖ retransmit segment that caused timeout
- ❖ restart timer

ack rcvd:

- ❖ if ack acknowledges previously unacked segments
 - update what is known to be ACKed
 - start timer if there are still unacked segments

TCP: retransmission scenarios



TCP ACK original generation

[RFC 1122, RFC 2581]

<i>event at receiver</i>	<i>TCP receiver action</i>
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—detect loss before timeout

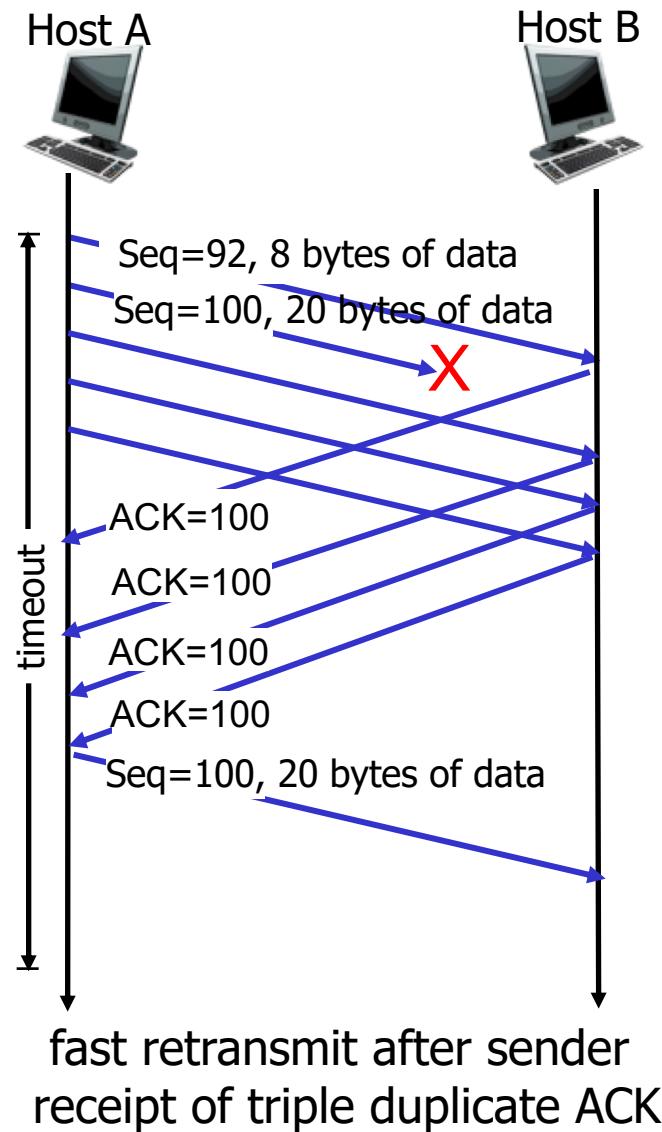
- ❖ time-out period often relatively long:
 - long delay before resending lost packet
- ❖ detect lost segments via duplicate ACKs.
 - sender often sends many segments back-to-back
 - if segment is lost, there will likely be many duplicate ACKs.

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

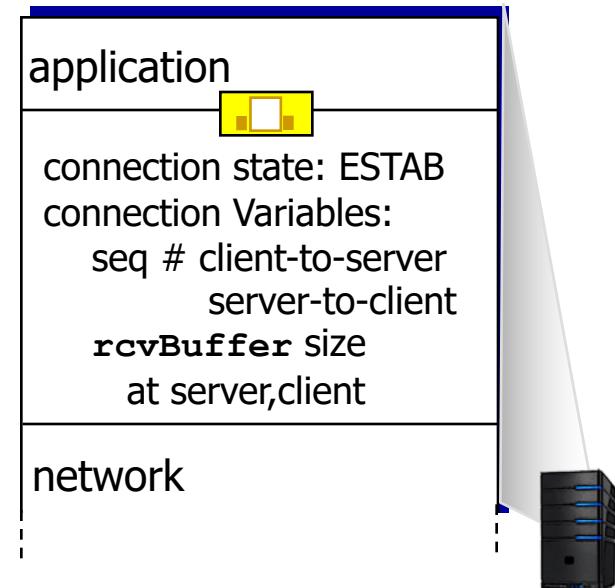
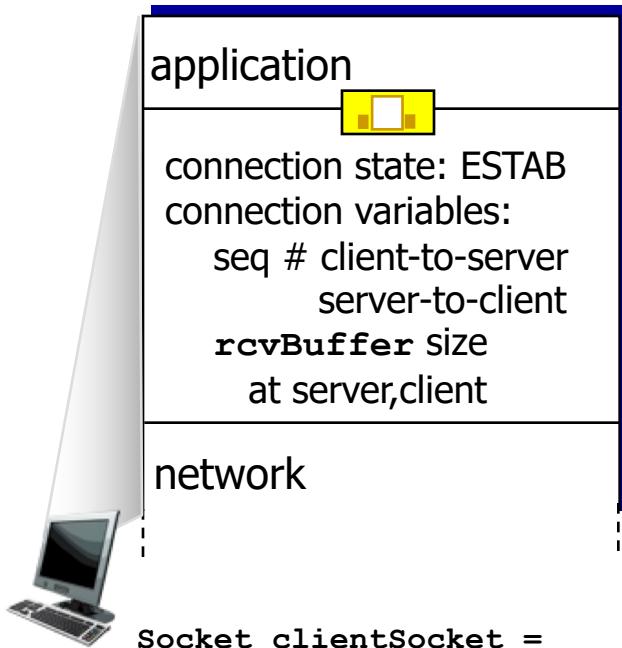
TCP fast retransmit



Connection Management

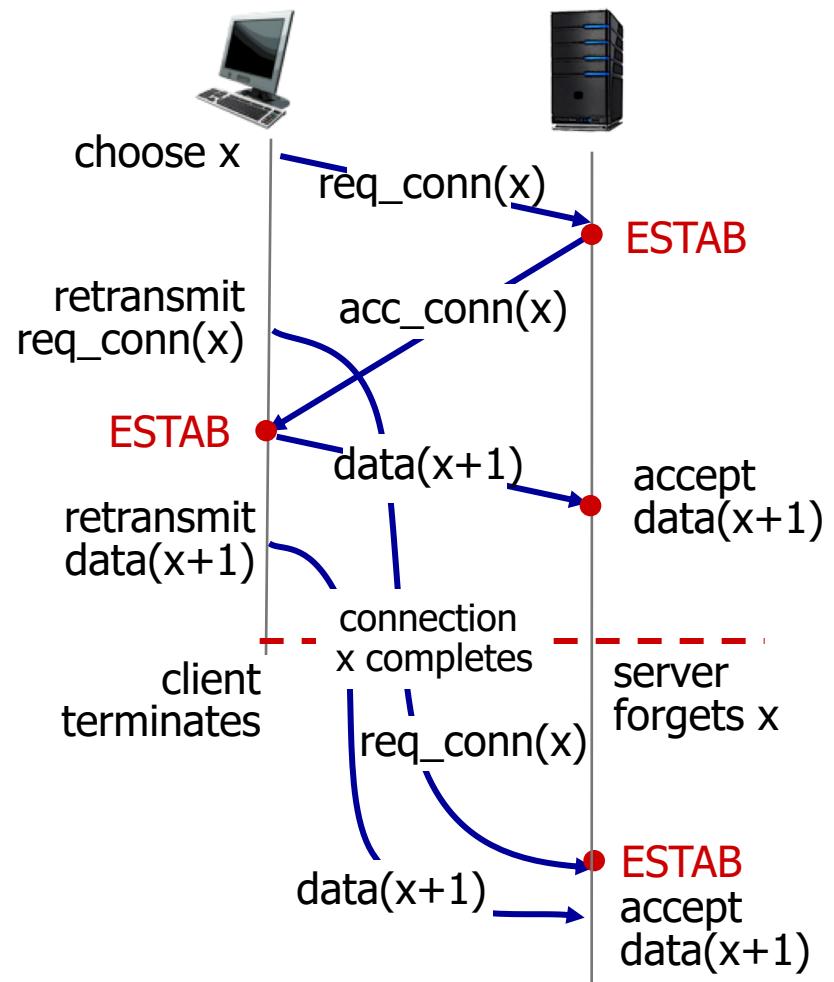
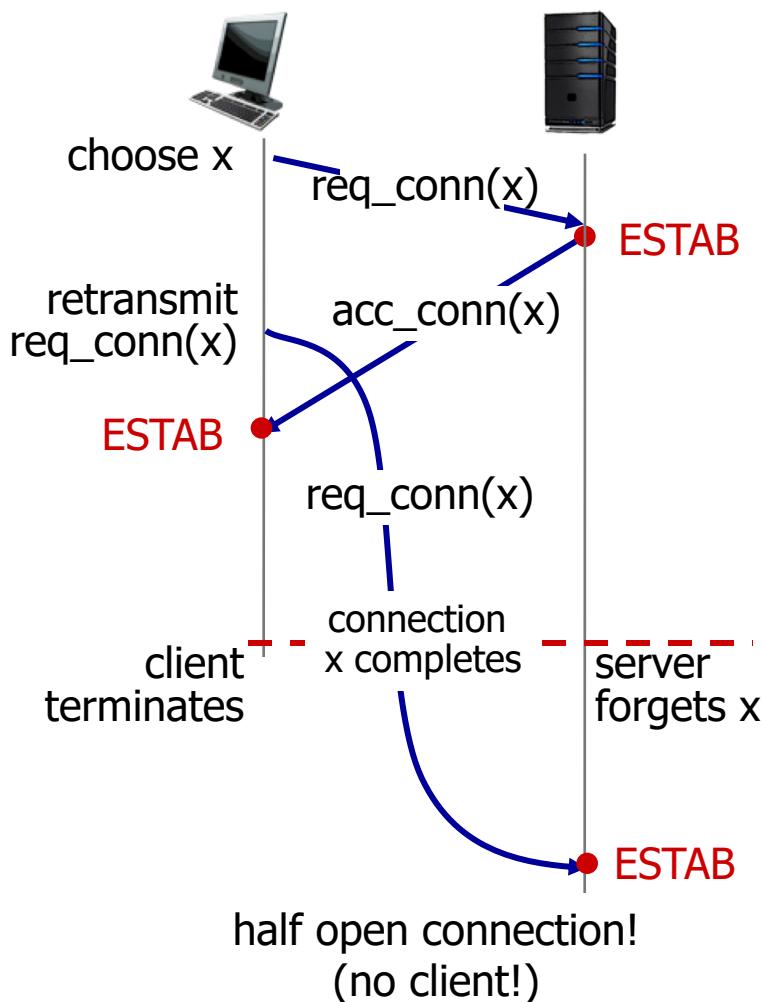
before exchanging data, sender/receiver “handshake”:

- ❖ agree to establish connection (each knowing the other willing to establish connection)
- ❖ agree on connection parameters – sequence number

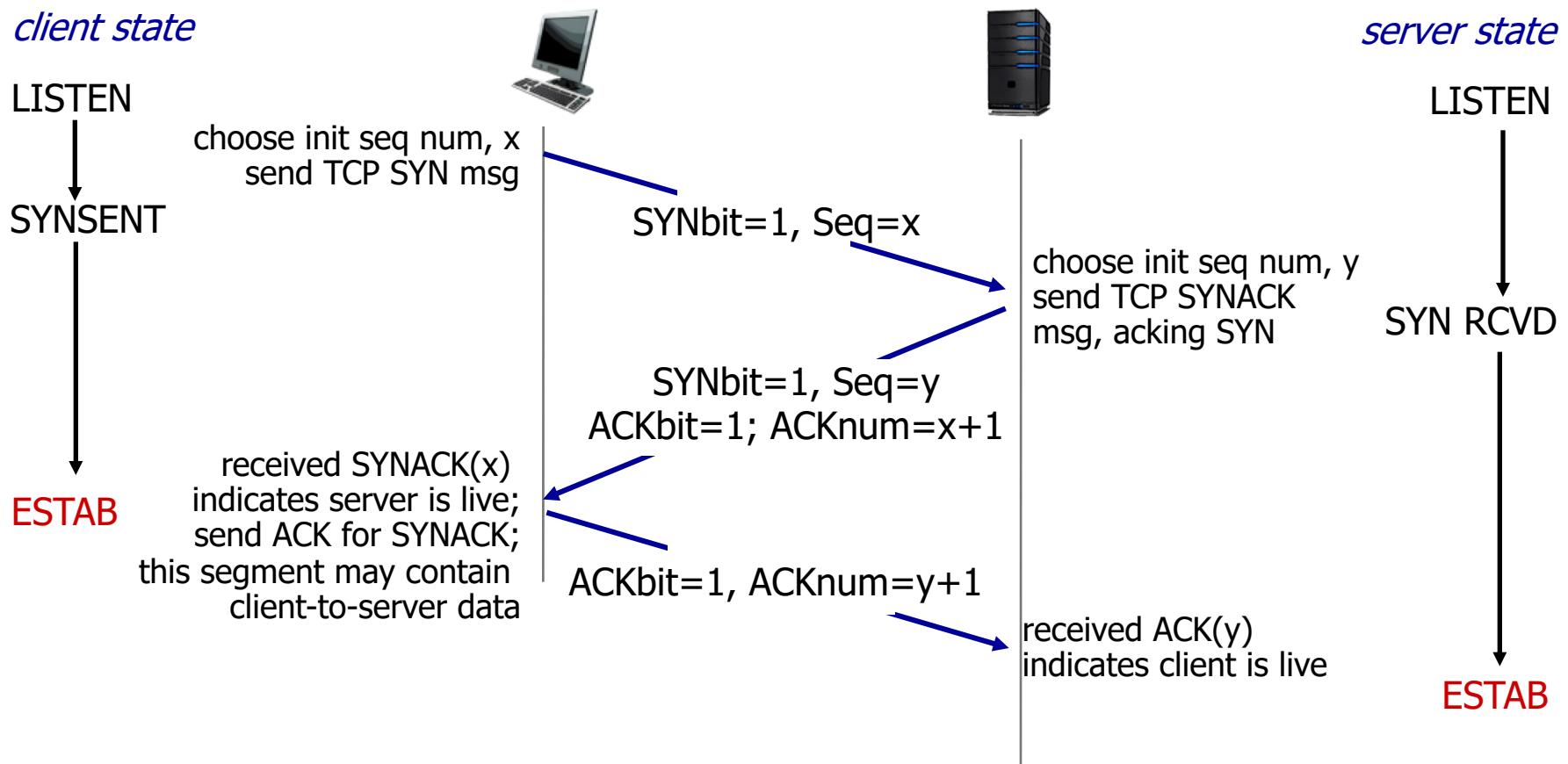


Agreeing to establish a connection

2-way handshake failure scenarios:



TCP 3-way handshake



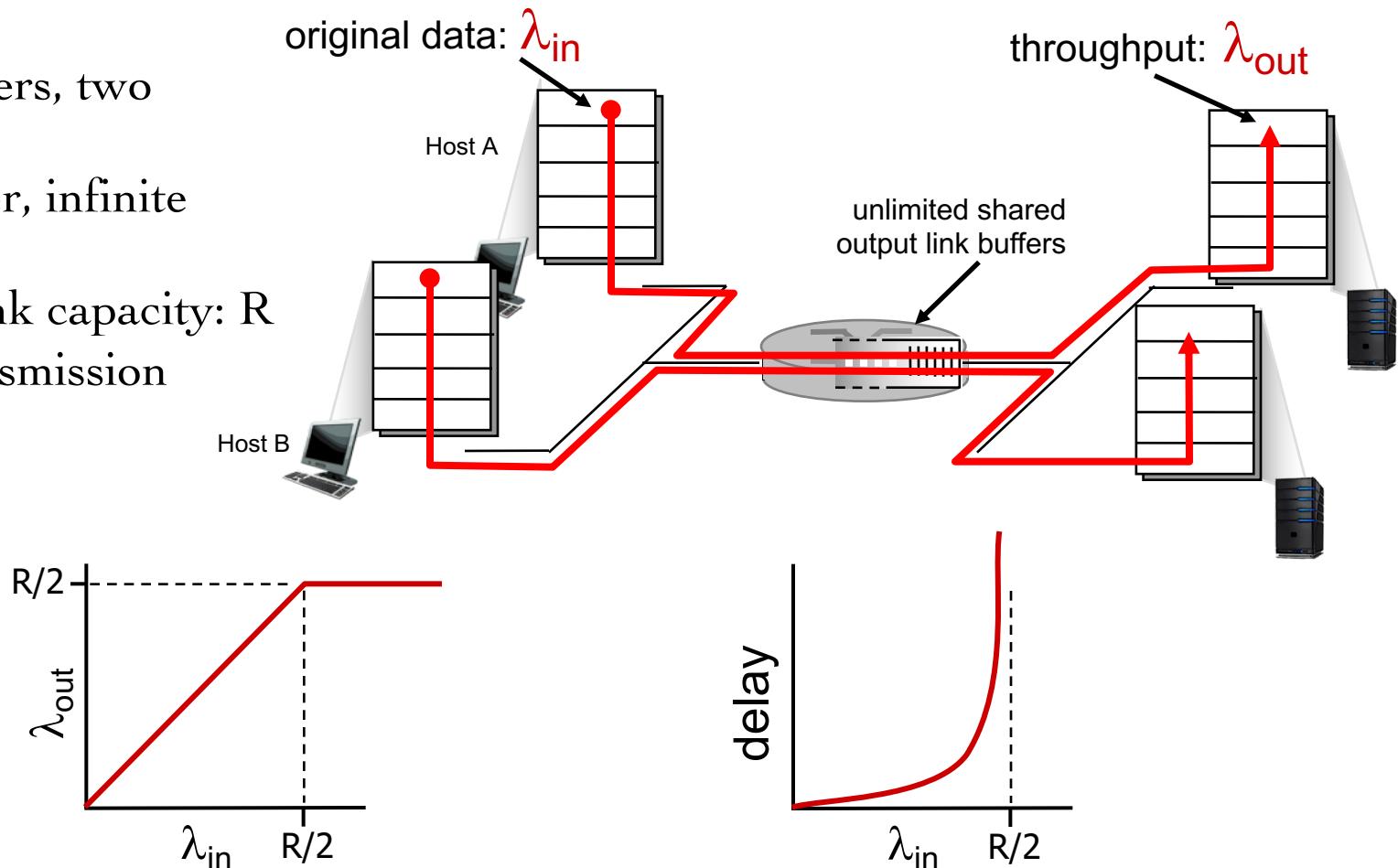
Principles of congestion control

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: delay

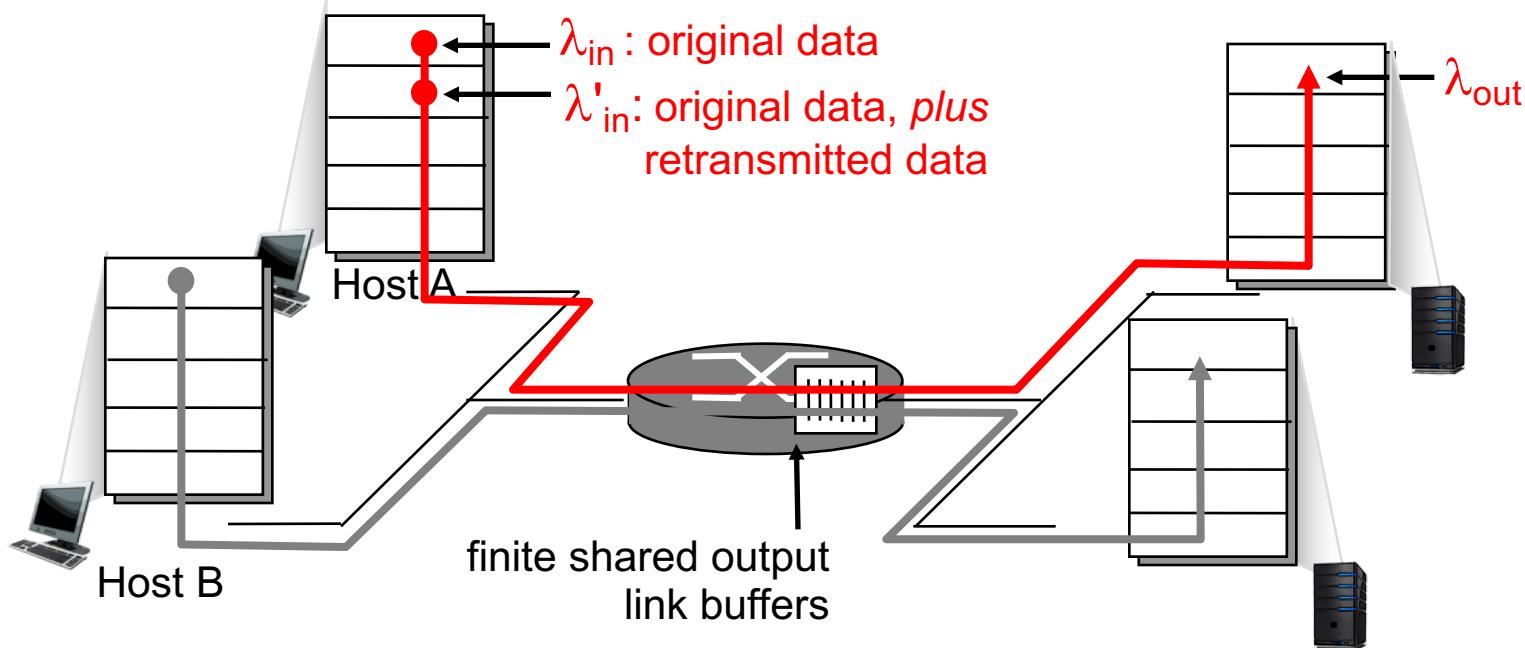
- ❖ two senders, two receivers
- ❖ one router, infinite buffers
- ❖ output link capacity: R
- ❖ no retransmission



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

Causes/costs of congestion: drop + duplicate

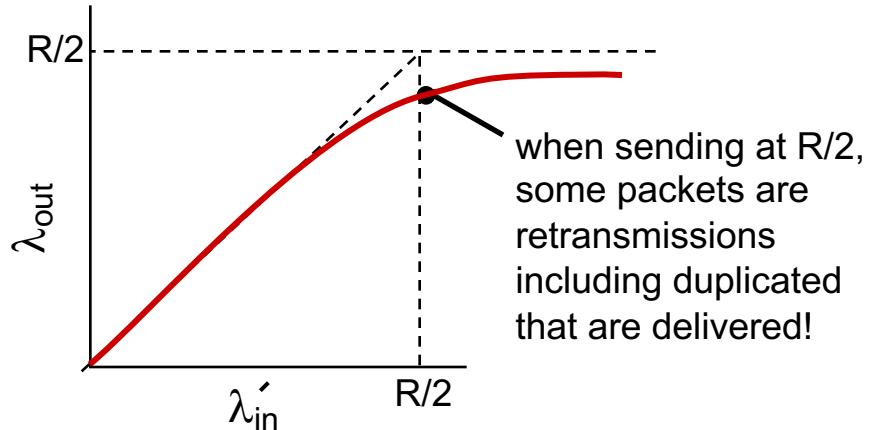
- ❖ one router, *finite* buffers
- ❖ sender retransmission of timed-out packet
 - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
 - transport-layer input includes *retransmissions* : $\lambda'_{in} > \lambda_{in}$



Causes/costs of congestion:

duplicates

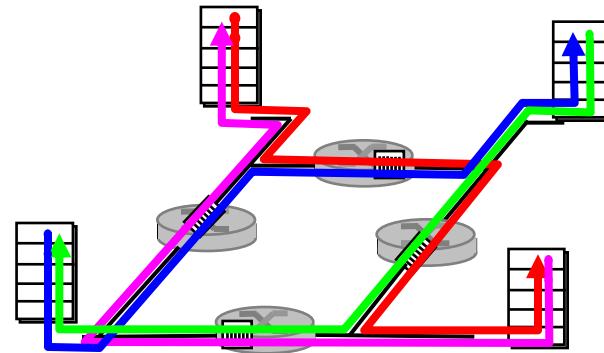
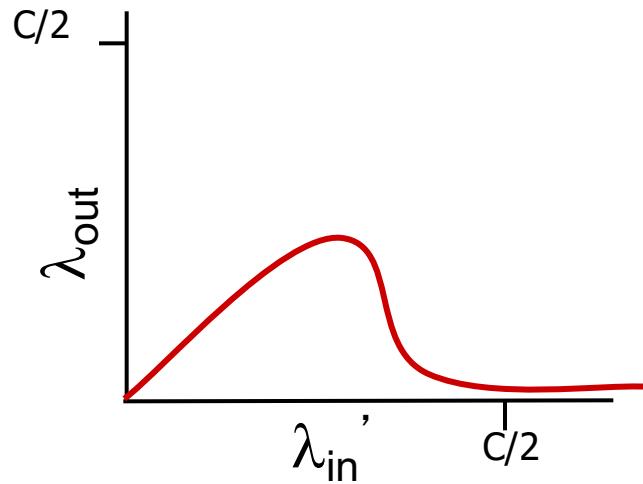
- ❖ packets can be lost, dropped at router due to full buffers
- ❖ sender times out prematurely, sending **two** copies, both of which are delivered



“costs” of congestion:

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

Causes/costs of congestion: multi-hop paths



another “cost” of congestion:

- ❖ when packet dropped, any “upstream transmission capacity used for that packet was 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, Ack signals from receiver
- ❖ approach taken by TCP

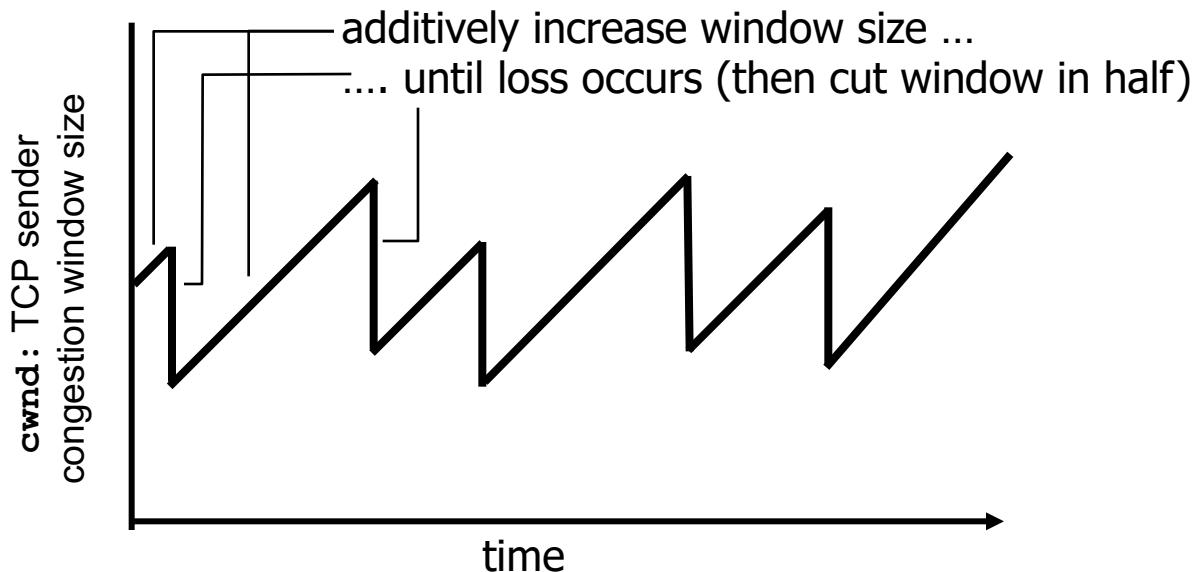
network-assisted congestion control:

- ❖ routers provide feedback to end systems
 - single bit indicating congestion as packet forwarded (SNA, DECbit, **TCP/IP ECN**)

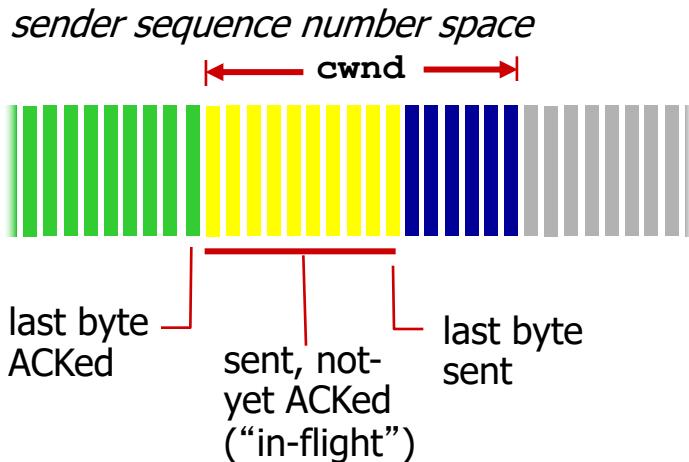
TCP congestion control basics: additive increase multiplicative decrease

- ❖ **approach:** sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
 - **additive increase:** increase **cwnd** by 1 MSS (max segment size, default 536 bytes) every RTT until loss detected
 - **multiplicative decrease:** cut **cwnd** in half after loss

AIMD saw tooth behavior: probing for bandwidth



TCP Congestion Control: window



TCP sending rate:

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

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

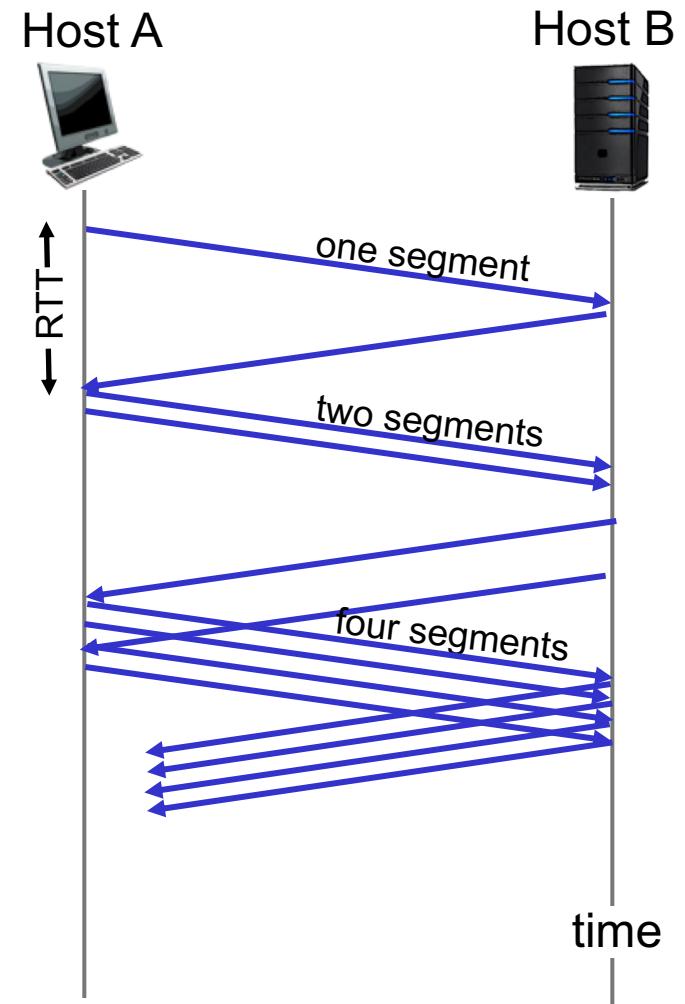
- ❖ sender limits transmission:

$$\frac{\text{LastByteSent} - \text{LastByteAcked}}{\text{cwnd}} \leq 1$$

- ❖ **cwnd** is dynamic, function of perceived network congestion

TCP Slow Start

- ❖ when connection begins, rate (window) small but increase rate (window) exponentially until first loss event:
 - initially **cwnd** = 1 MSS
 - double **cwnd** every RTT
 - done by incrementing **cwnd** for every ACK received
- ❖ summary: initial rate is slow but ramps up exponentially fast



TCP: detecting and responding to loss

- ❖ If loss detected by timeout:
 - **cwnd** set to 1 MSS
 - window initially grows exponentially to threshold (ssthresh), then grows linearly
- ❖ Fast Retransmit (reminder): Faster loss detection signaled by duplicate ACKs from receiver;
 - Receiver adjusted to **send Ack each time packet received** even if it's a duplicate (no advance in seq number)
 - Sender waits for **3 duplicate Acks** (not just one in case slightly out of order delivery)

TCP: reacting to loss

- ❖ If loss detected by timeout **cwnd** set to 1 MSS
- ❖ If lost detected by duplicate ACKS: can be less reactive since ACKs indicate network able to deliver packets to receiver
 - **TCP Reno:** **cwnd** is only cut in half (not set to 1); then window grows linearly
 - **TCP New Reno:** improves retransmit during fast recovery given wireless link loss is not usually due to network congestion
 - for every ACK that advances sequence space, send next packet beyond the ACKed sequence number as well

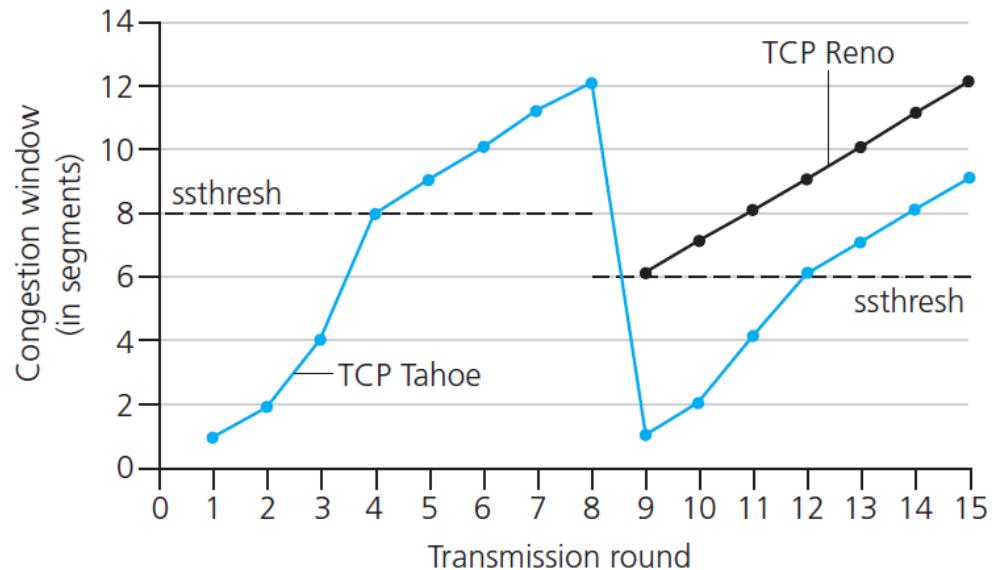
TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

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

Implementation:

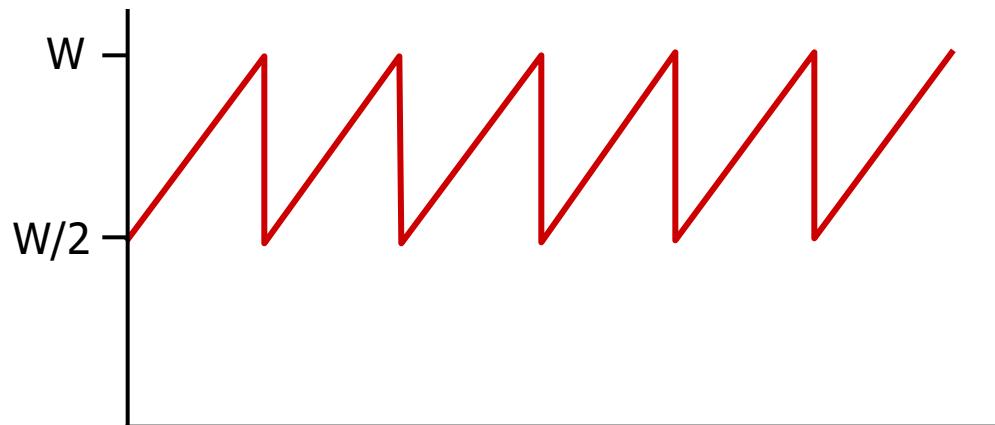
- ❖ variable **ssthresh**
- ❖ on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event



TCP throughput

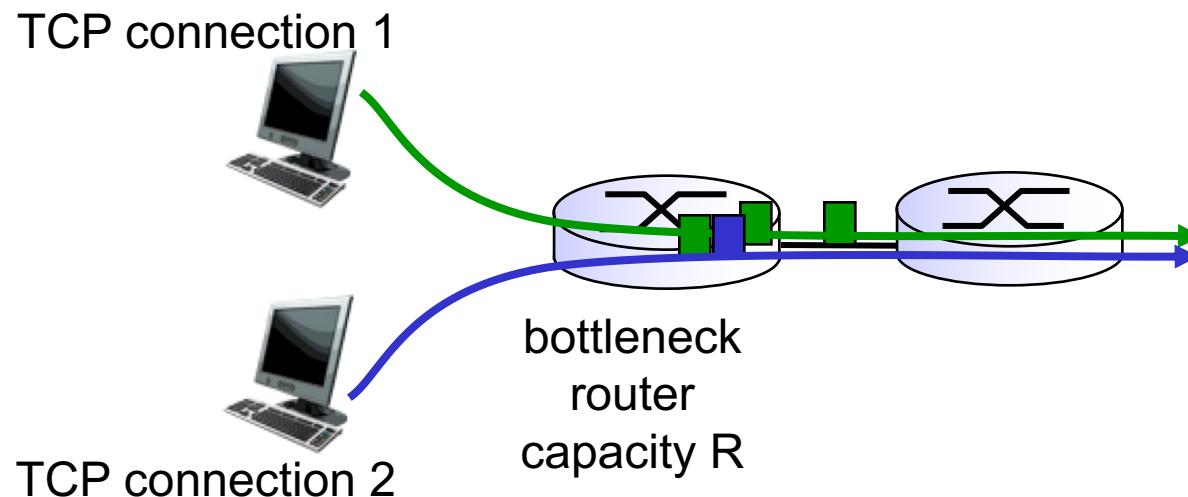
- ❖ avg. TCP thruput as function of window size, RTT?
 - ignore transient of slow start, assume always data to send
- ❖ W: window size (measured in bytes) where loss occurs
 - avg. window size (# in-flight bytes) is $\frac{3}{4} W$
 - avg. thruput is $\frac{3}{4}W$ per RTT

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



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, in concept?

two competing sessions:

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

