

# Chapter 3

## Transport Layer

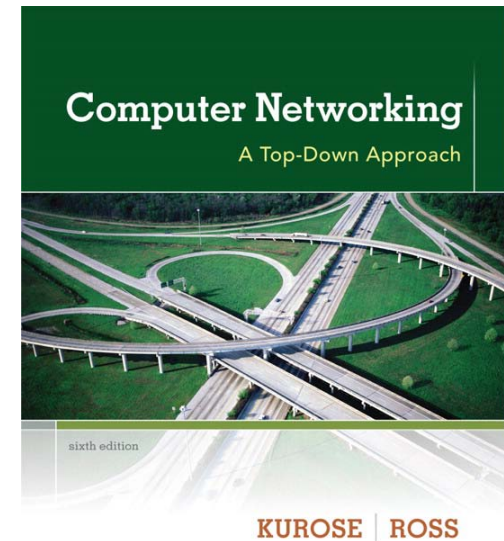
### A note on the use of these ppt slides:

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 1996-2012  
J.F Kurose and K.W. Ross, All Rights Reserved



*Computer  
Networking: A Top  
Down Approach*  
6<sup>th</sup> edition  
Jim Kurose, Keith Ross  
Addison-Wesley  
March 2012

# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

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

3.6 principles of congestion control

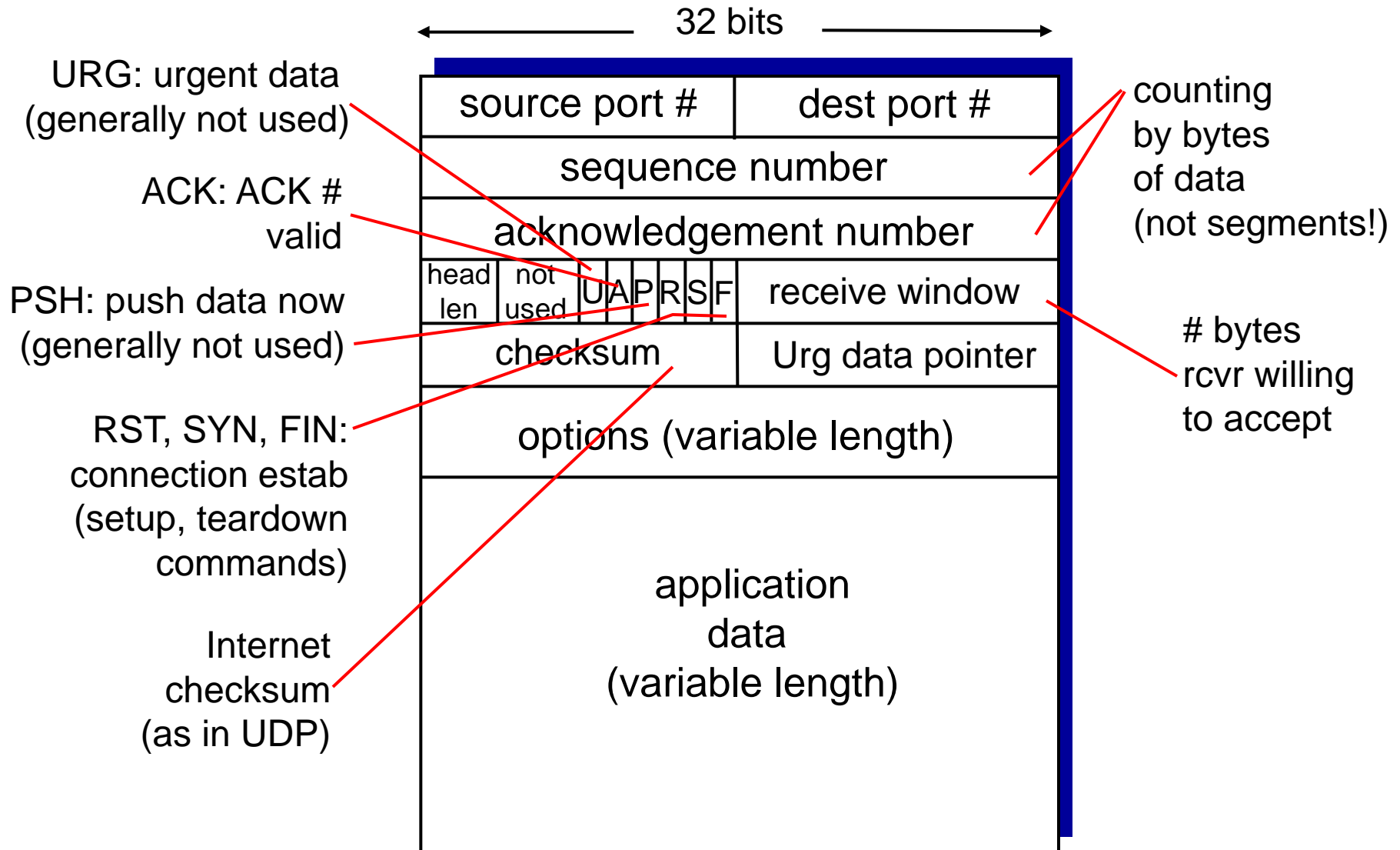
3.7 TCP congestion control

# TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- ❖ **point-to-point:**
  - one sender, one receiver
- ❖ **reliable, in-order *byte stream*:**
  - 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) initializes 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:

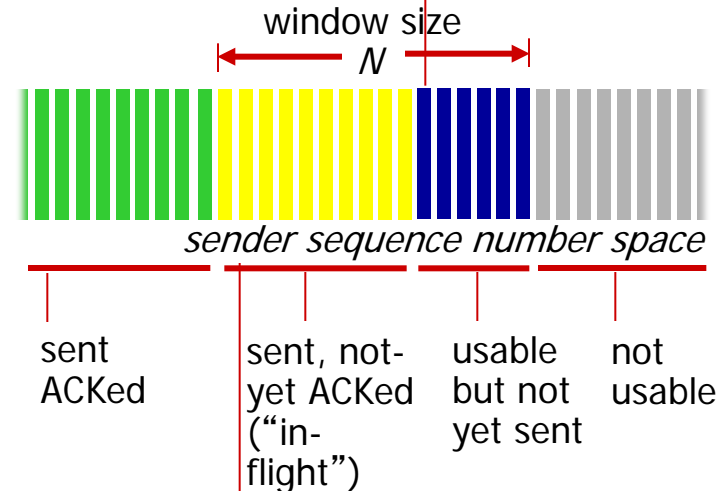
- seq # of next byte expected from other side
- cumulative ACK

**Q:** how receiver handles out-of-order segments

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

outgoing segment from sender

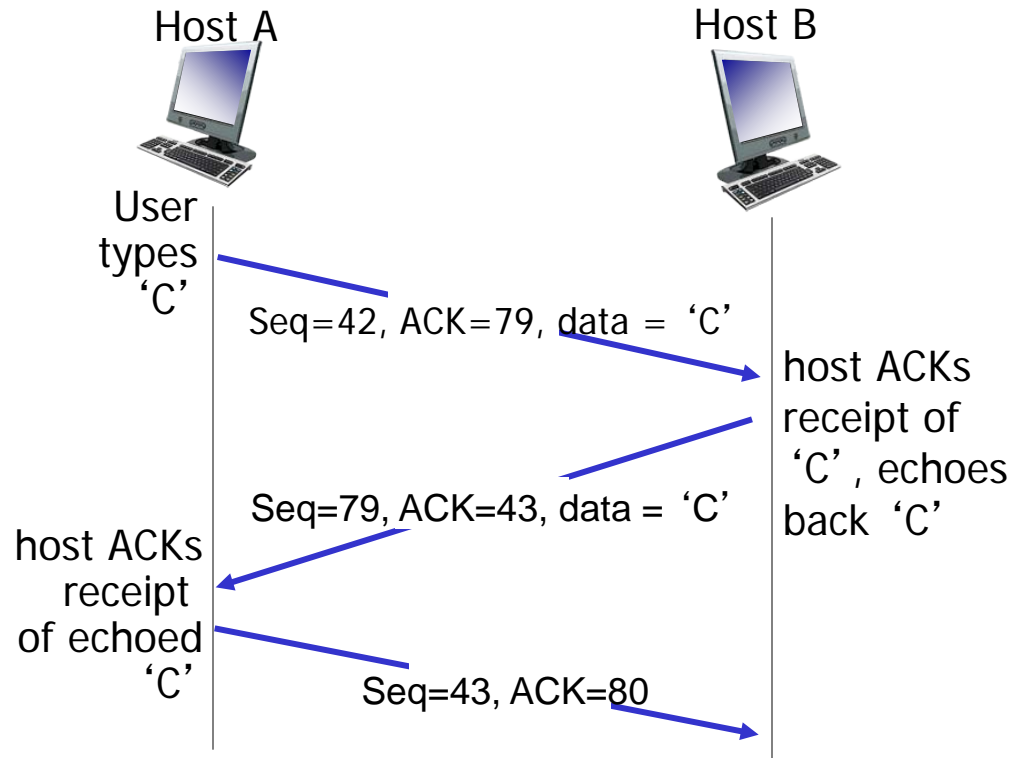
source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer



incoming segment to sender

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

# 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

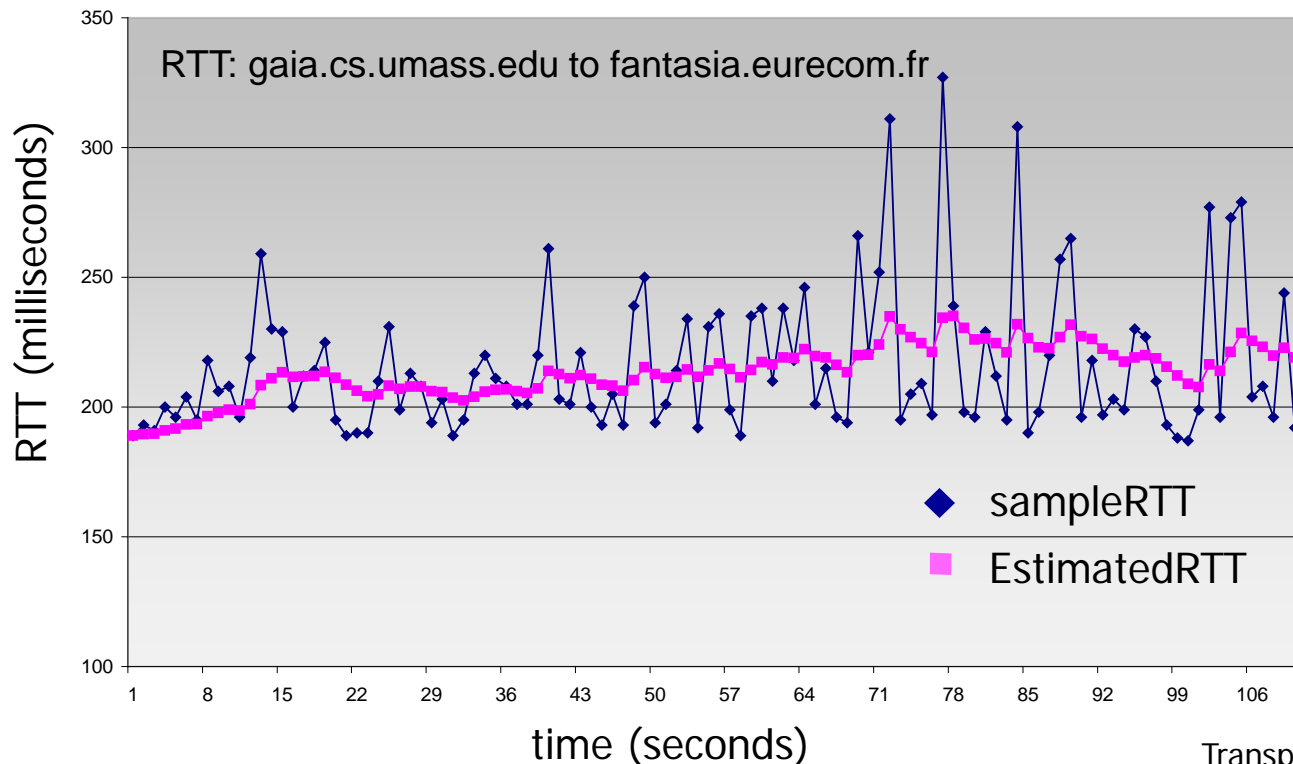
Q: how to estimate RTT?

- ❖ **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

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

- ❖ 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`:

$$\begin{aligned}\text{DevRTT} = & (1-\beta)*\text{DevRTT} + \\ & \beta*|\text{SampleRTT}-\text{EstimatedRTT}| \\ & (\text{typically, } \beta = 0.25)\end{aligned}$$

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



↑  
estimated RTT

↑  
“safety margin”

# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

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

3.6 principles of congestion control

3.7 TCP congestion control

# TCP reliable data transfer

- ❖ TCP creates rdt service on top of IP's unreliable service

- pipelined segments
- cumulative acks
- 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 #
- ❖ 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`

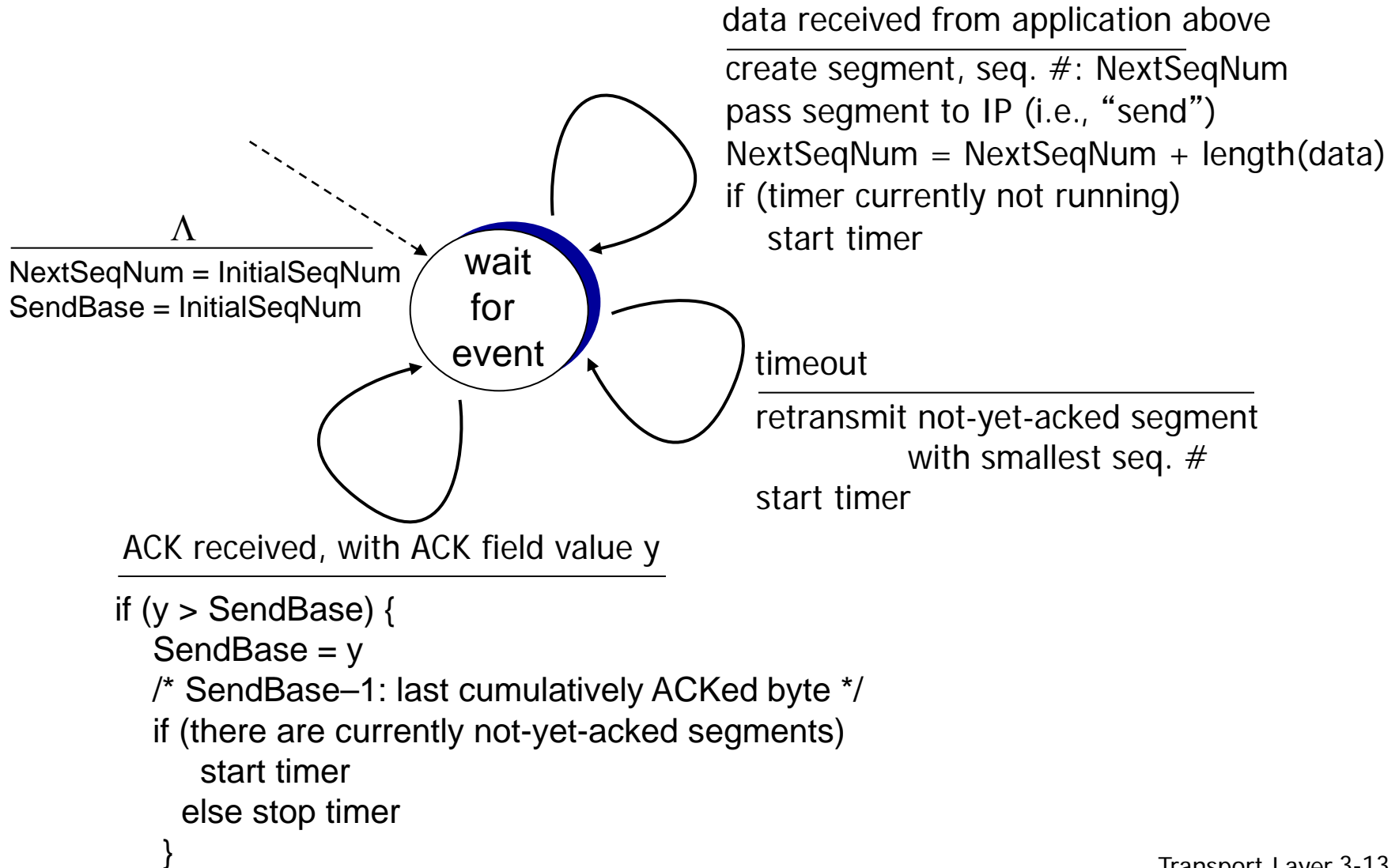
## *timeout:*

- ❖ retransmit segment that caused timeout
- ❖ restart timer

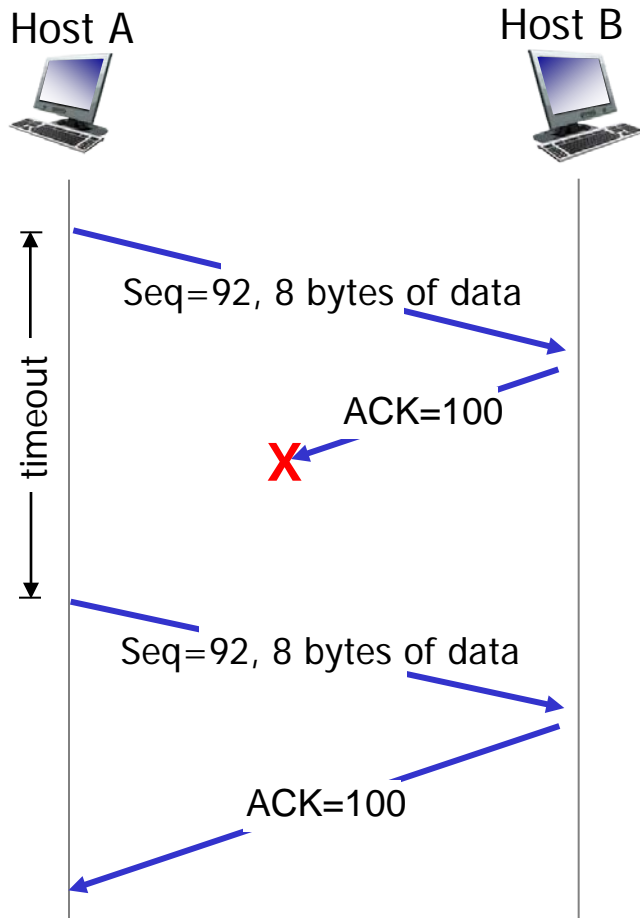
## *ack rcvd:*

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

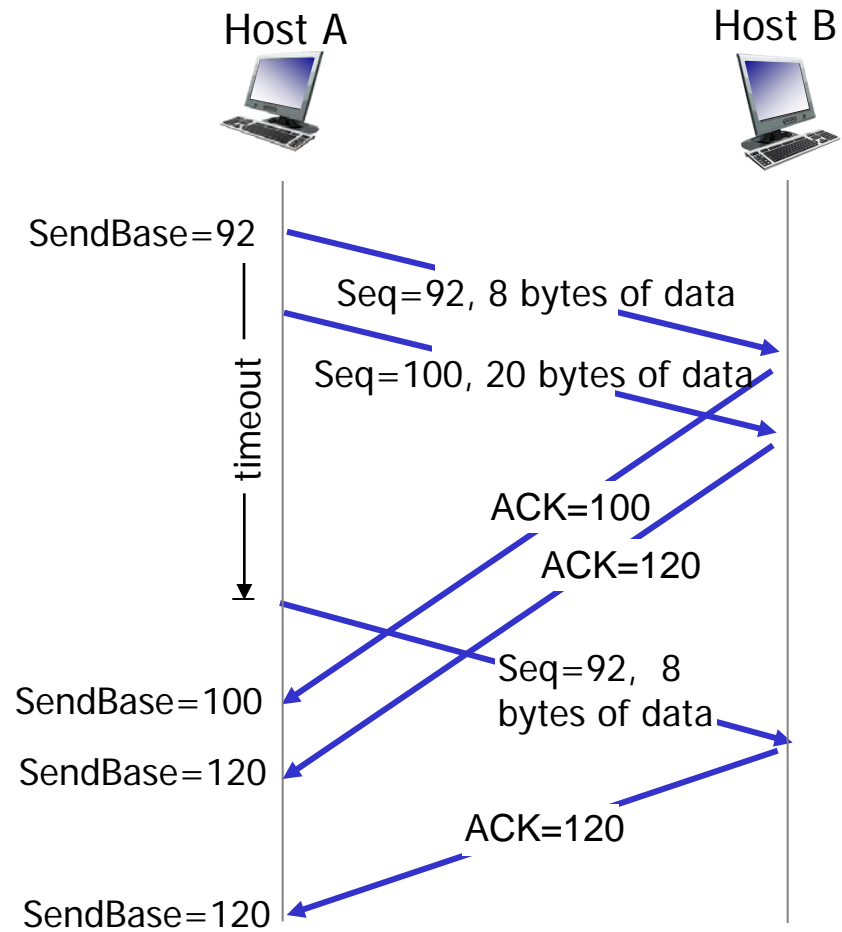
# TCP sender (simplified)



# TCP: retransmission scenarios

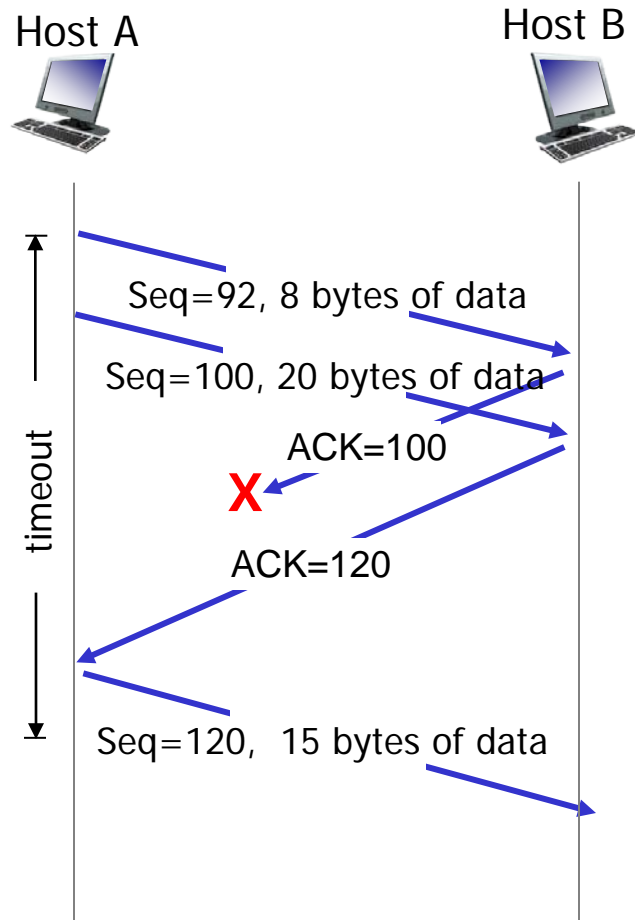


lost ACK scenario



premature timeout

# TCP: retransmission scenarios



cumulative ACK

# TCP ACK 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 <i>duplicate ACK</i> , 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

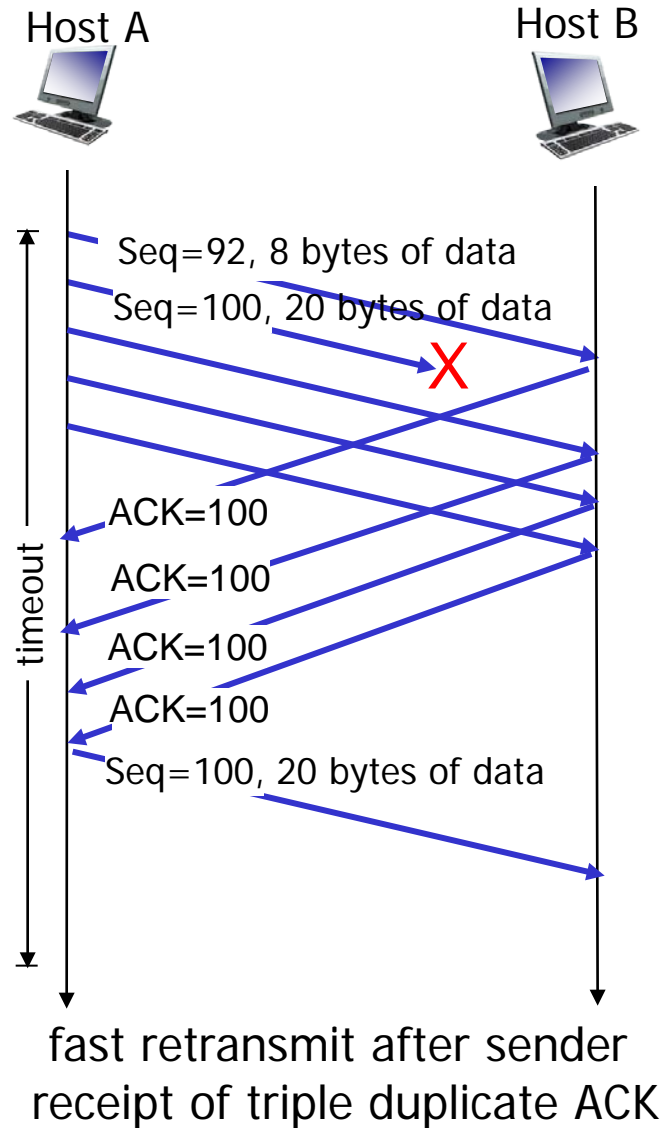
- ❖ time-out period often relatively long:
  - long delay before resending lost packet
- ❖ detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

## *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



# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

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

3.6 principles of congestion control

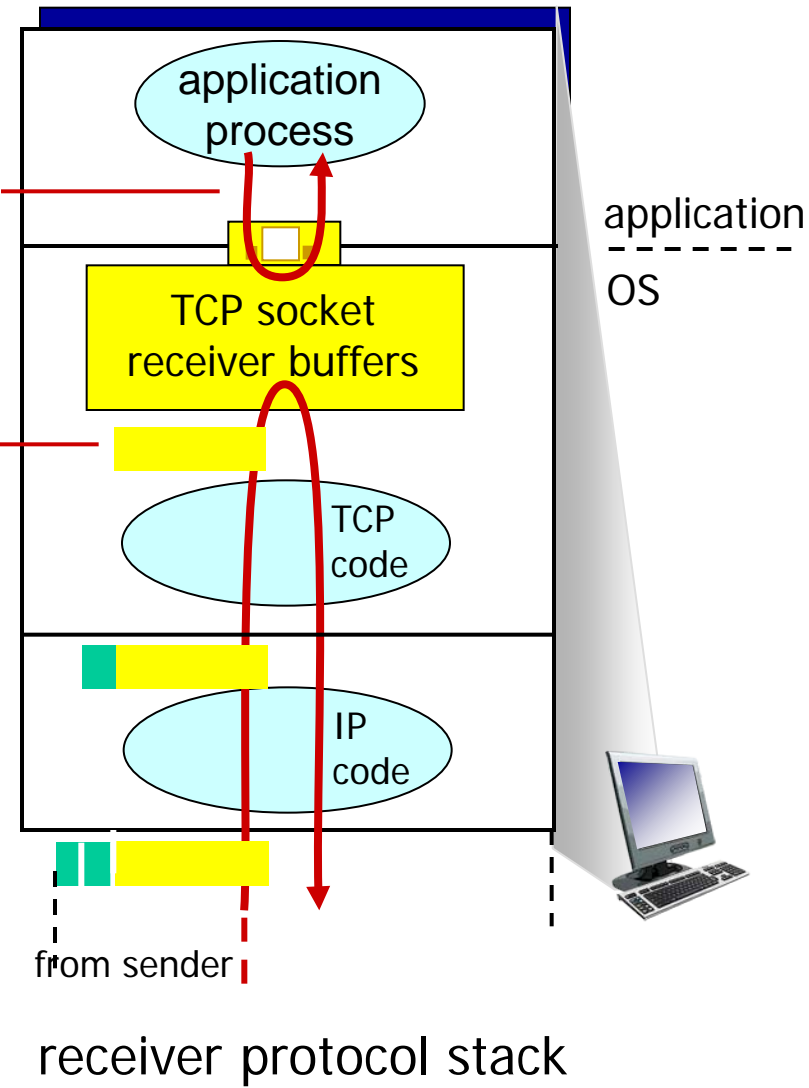
3.7 TCP congestion control

# TCP flow control

application may  
remove data from  
TCP socket buffers ....

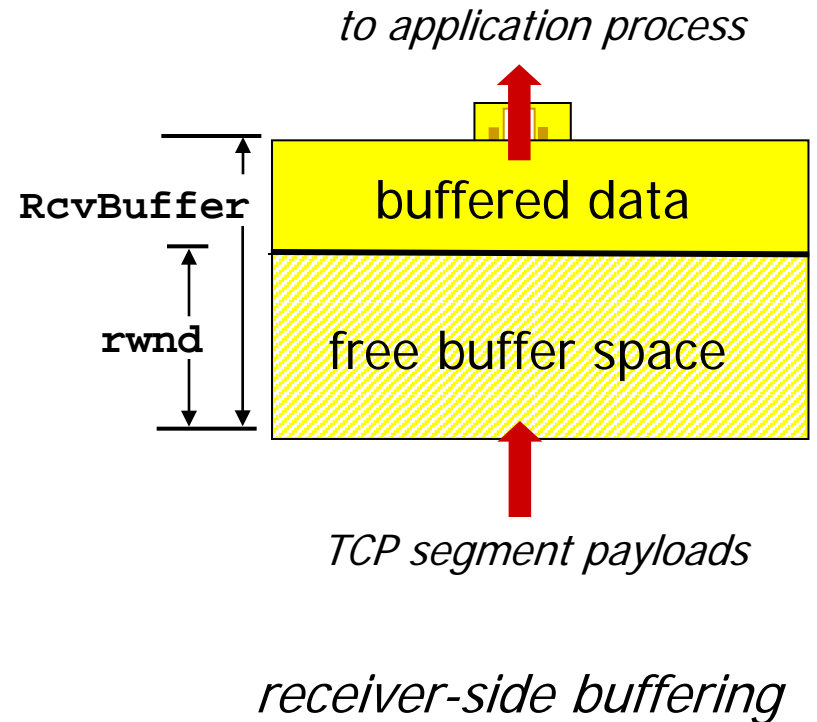
... slower than TCP  
receiver is delivering  
(sender is sending)

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



# TCP flow control

- ❖ receiver “advertises” free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- ❖ sender limits amount of unacked (“in-flight”) data to receiver’s **rwnd** value
- ❖ guarantees receive buffer will not overflow



# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

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

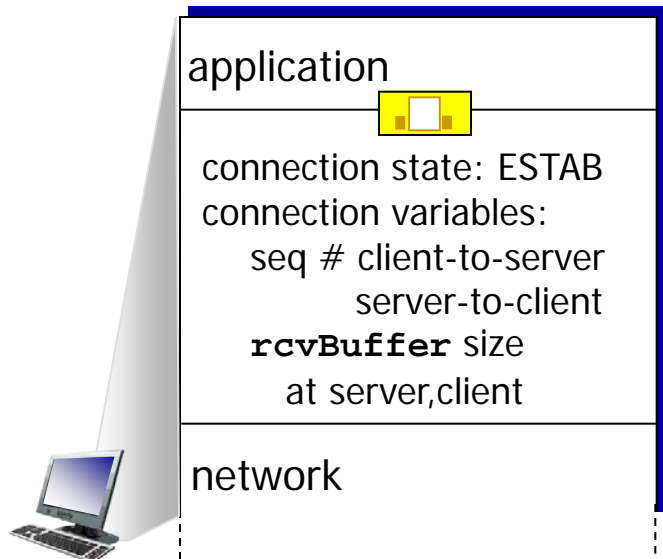
3.6 principles of congestion control

3.7 TCP congestion control

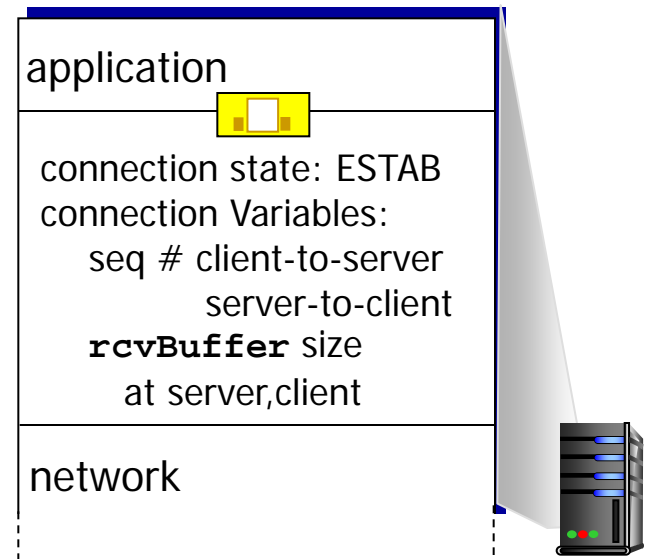
# Connection Management

before exchanging data, sender/receiver “handshake”:

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



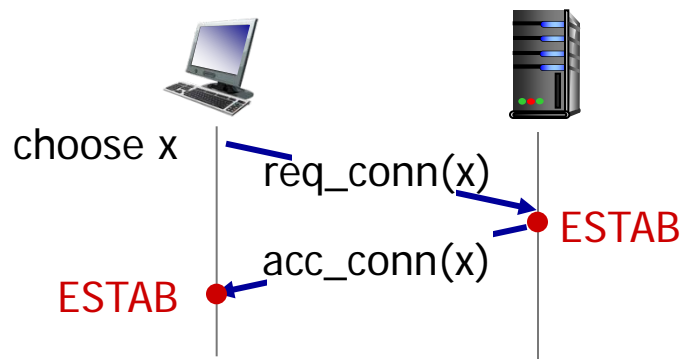
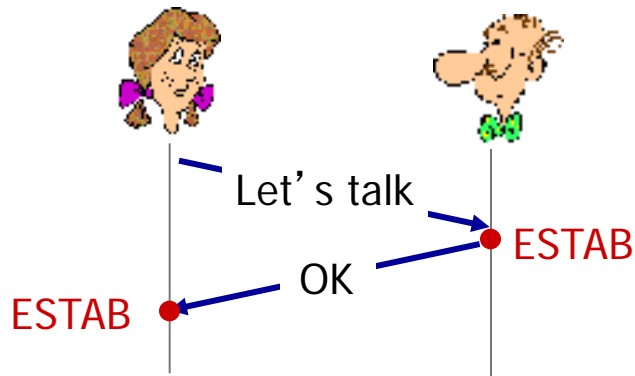
```
Socket clientSocket =  
    newSocket("hostname", "port  
    number");
```



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

# Agreeing to establish a connection

2-way handshake:



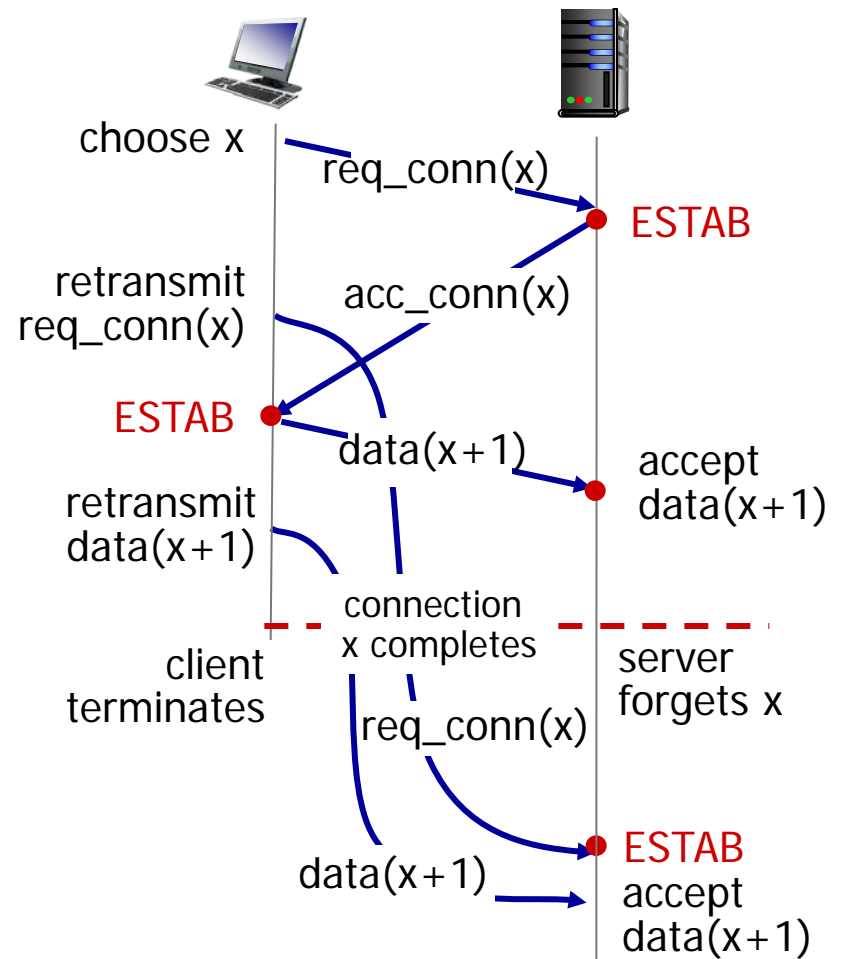
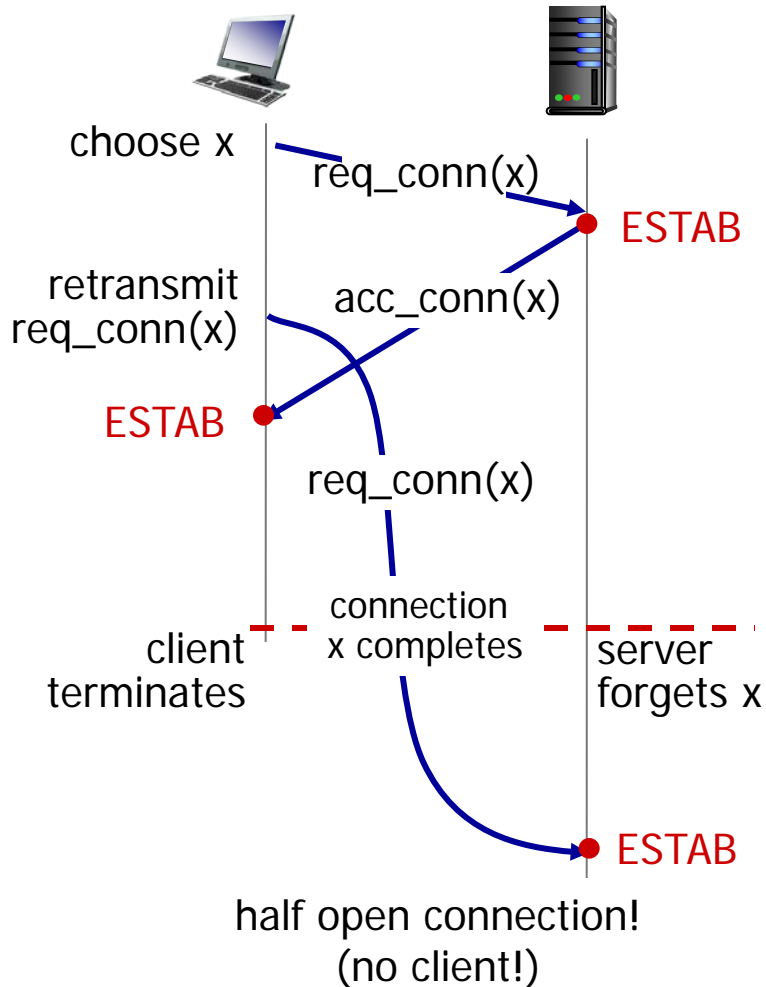
Q: will 2-way handshake always work in network?

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

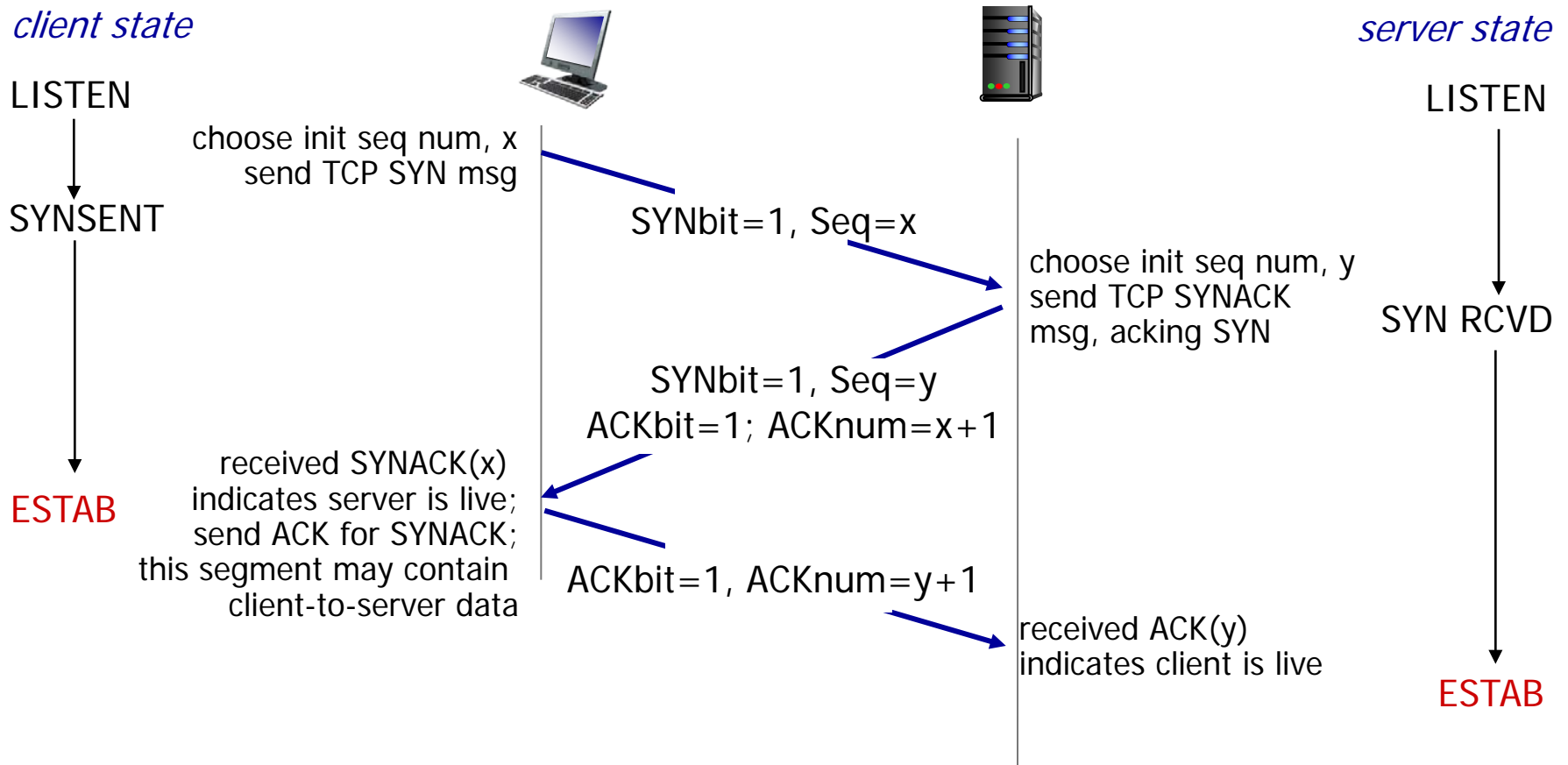


# Agreeing to establish a connection

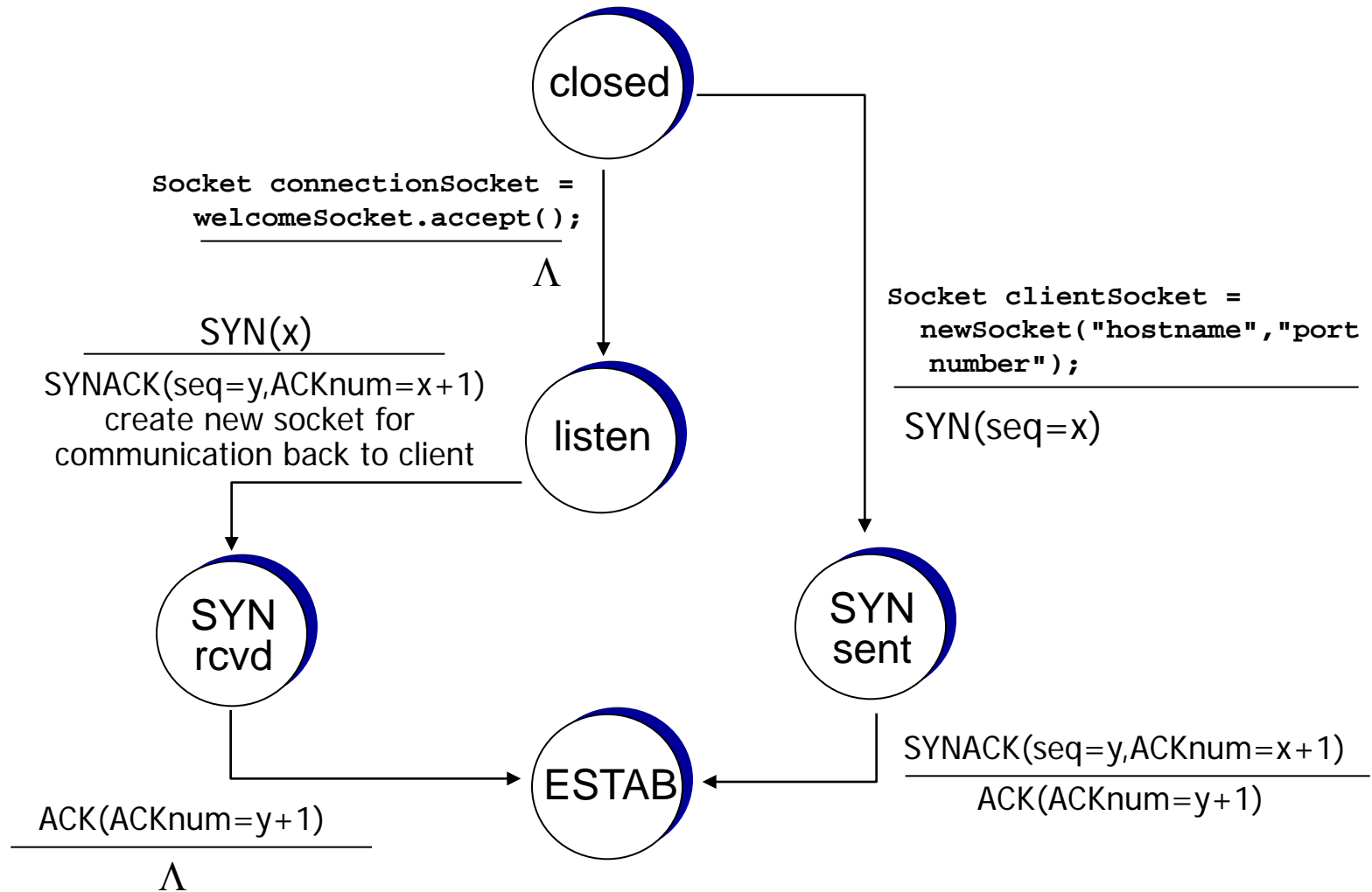
2-way handshake failure scenarios:



# 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

*client state*

ESTAB

`clientSocket.close()`

FIN\_WAIT\_1

can no longer  
send but can  
receive data

FIN\_WAIT\_2

wait for server  
close

TIMED\_WAIT

timed wait  
for  $2 * \text{max}$   
segment lifetime

CLOSED



FINbit=1, seq=x

ACKbit=1; ACKnum=x+1

FINbit=1, seq=y

ACKbit=1; ACKnum=y+1

can still  
send data

can no longer  
send data

*server state*

ESTAB

CLOSE\_WAIT

LAST\_ACK

CLOSED

# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

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

3.6 principles of congestion control

3.7 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!

# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

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

3.6 principles of congestion control

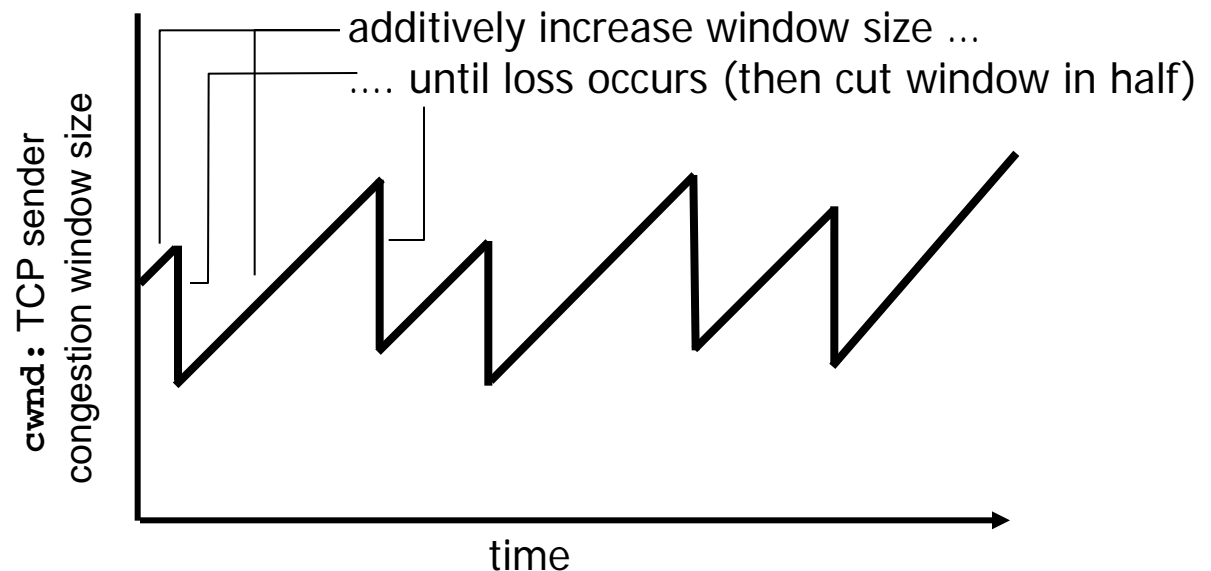
3.7 TCP congestion control



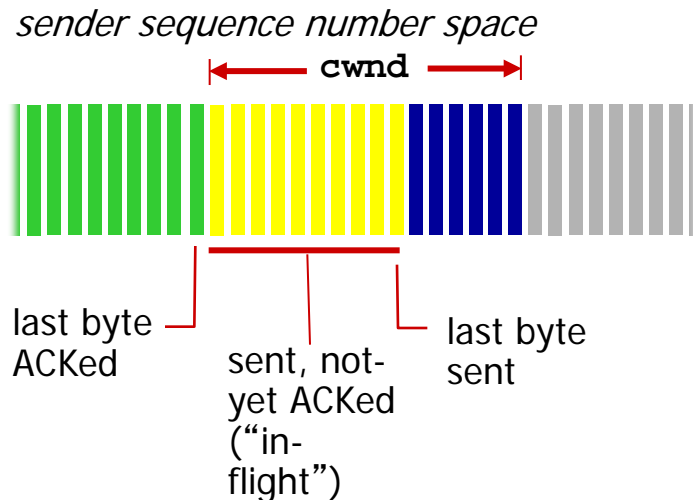
# TCP congestion control: additive increase multiplicative decrease

- ❖ *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: details



- ❖ sender limits transmission:

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

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

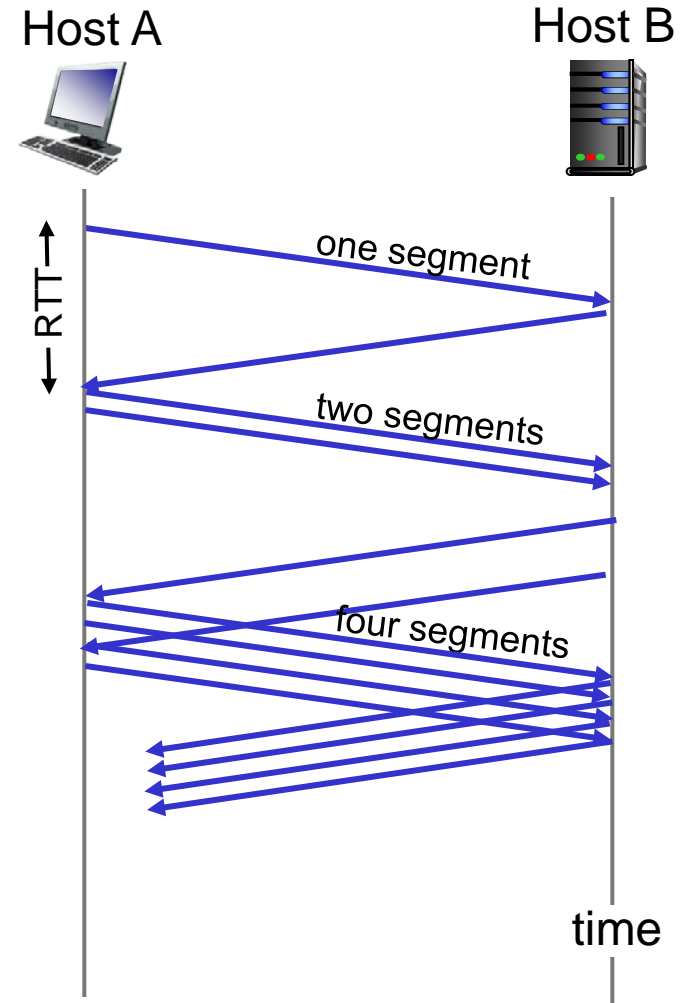
*TCP sending rate:*

- ❖ *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` for every ACK received
- ❖ 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)

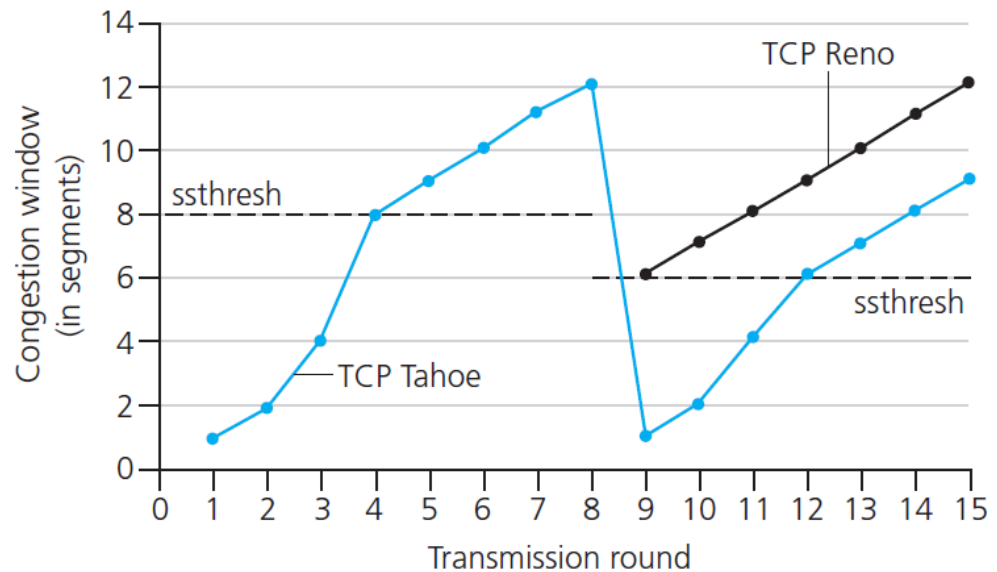
# TCP: switching from slow start to CA

**Q:** when should the exponential increase switch to linear?

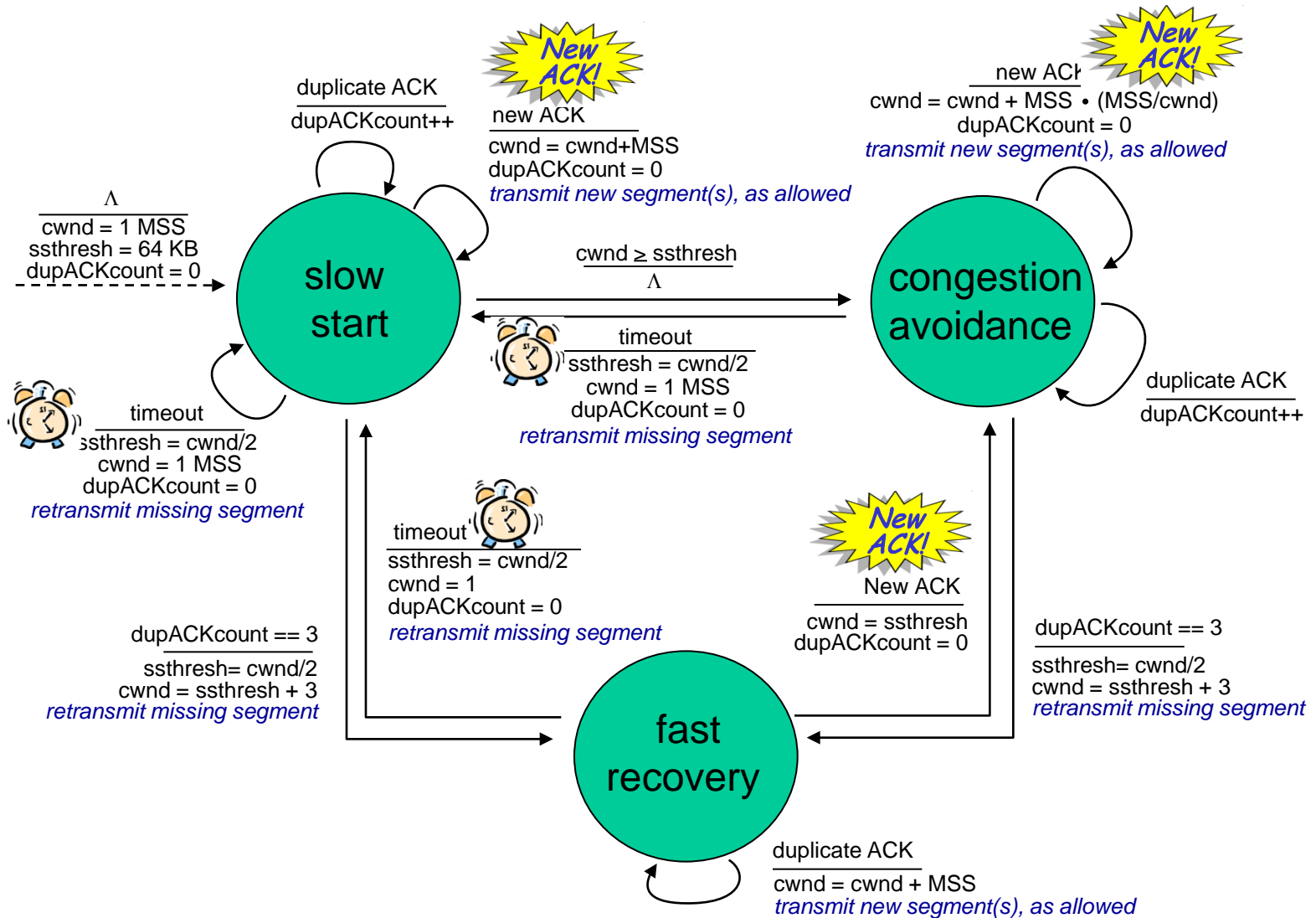
**A:** when **cwnd** gets to 1/2 of its value before timeout.

## Implementation:

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



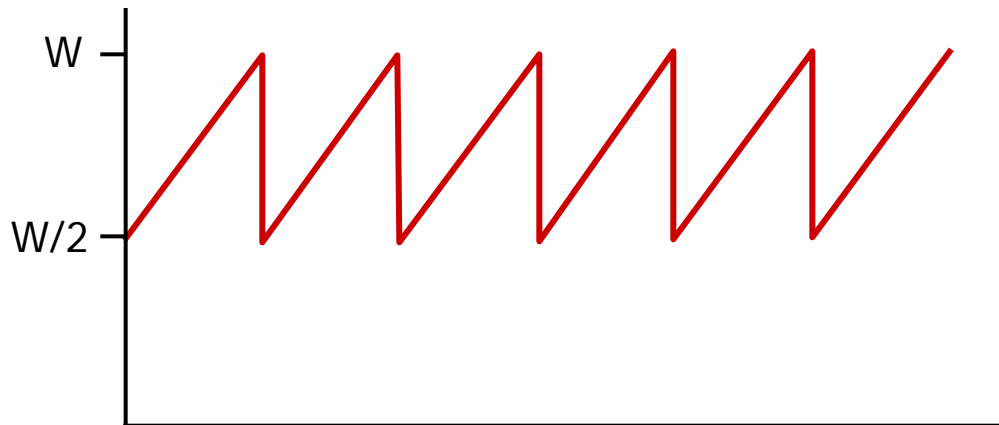
# Summary: TCP Congestion Control



# TCP throughput

- ❖ 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  $\frac{3}{4} W$
  - avg. thruput is  $\frac{3}{4}W$  per RTT

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



# TCP Futures: TCP over “long, fat pipes”

- ❖ example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- ❖ requires  $W = 83,333$  in-flight segments
- ❖ throughput in terms of segment loss probability,  $L$  [Mathis 1997]:

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

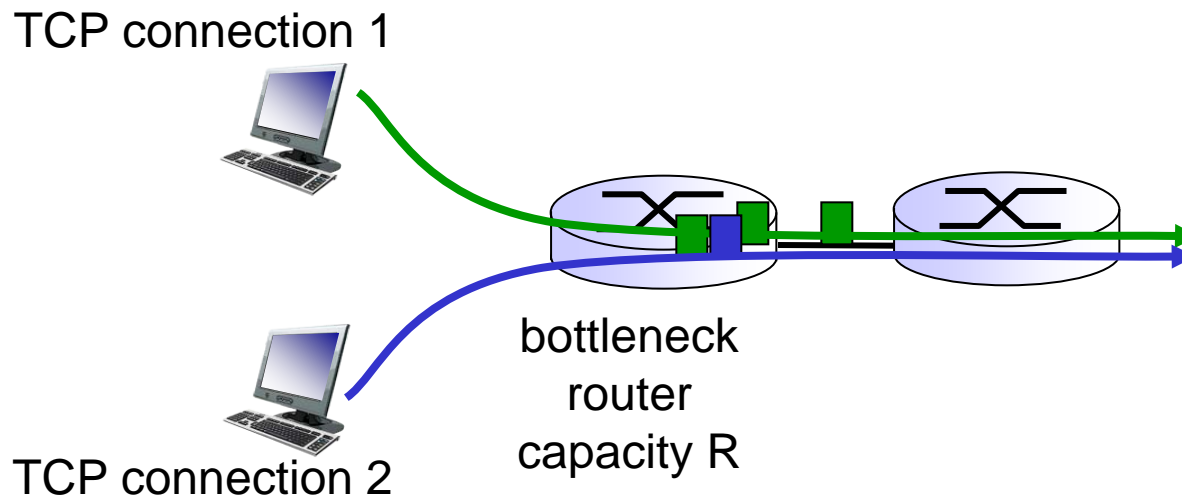
→ to achieve 10 Gbps throughput, need a loss rate of  $L = 2 \cdot 10^{-10}$  – *a very small loss rate!*

- ❖ new versions of TCP for high-speed



# TCP Fairness

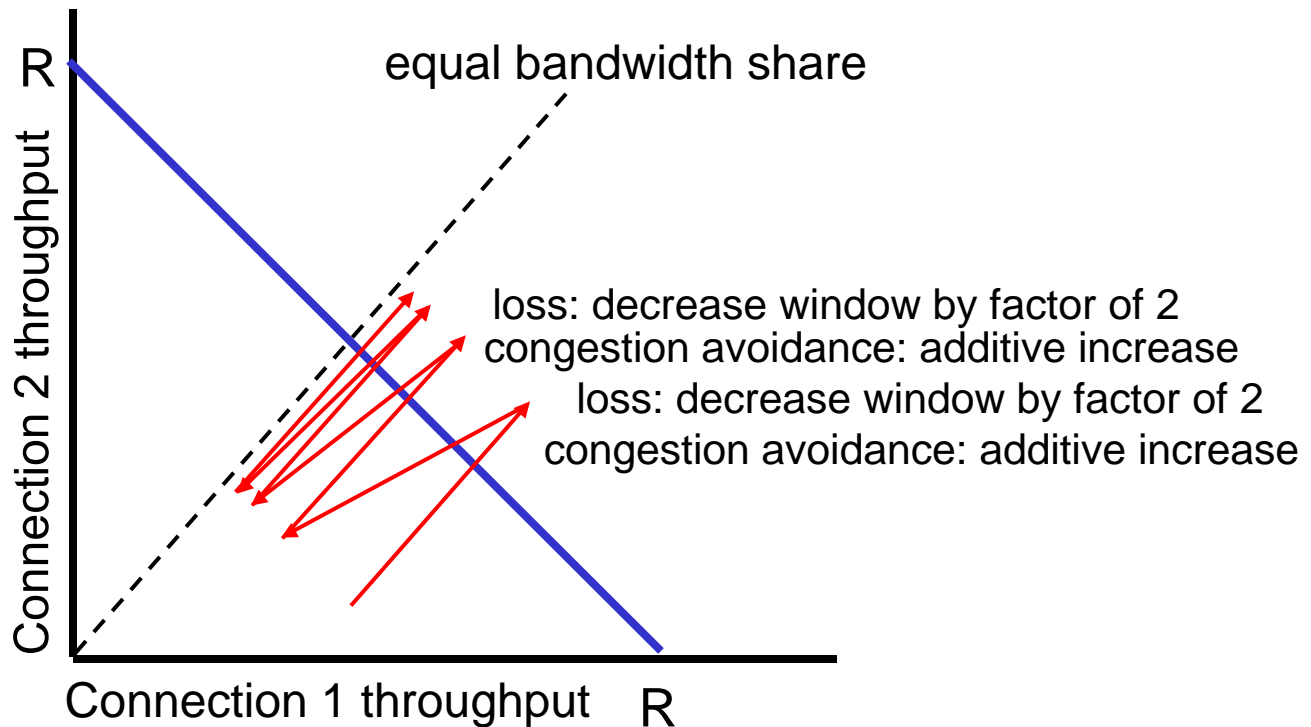
*fairness goal:* if  $K$  TCP sessions share same bottleneck link of bandwidth  $R$ , each should have average rate of  $R/K$



# Why is TCP fair?

two competing sessions:

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



# Fairness (more)

## *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

## *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  $R/2$

# Chapter 3: summary

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

## next:

- ❖ leaving the network “edge” (application, transport layers)
- ❖ into the network “core”