

Animations

- GO-BACK-N
- SELECTIVE REPEAT

Review: Building Reliable Data Transfer Protocols

- * Protocols based on retransmission (ACK and NAK) are known as ARQ (Automatic Repeat Request) protocols.
- * Receivers must be able to determine when bit errors have occurred (error detection). How? Checksums (add'l data) must be added to each packet.
- * Protocols in which the sender will not send new data until it is sure that the receiver has correctly received the current packet are known as stop-and-wait protocols.
- * Sequence numbers are used to deal with duplicate packets that may be introduced into the sender-to-receiver channel.
- * Timers are used to ensure deadlocks do not occur (e.g., sender waiting indefinitely for something that has been lost). At least for $RTT + \text{processing} + \text{delay time}$.
- * To improve the performance of stop-and-wait protocols, pipelining is often used. That is, we allow the sender to send multiple packets without waiting for acknowledgements.
⇒ need to increase the range of sequence numbers and ensure sufficient buffer space is available.

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) inits sender, receiver state before data exchange
- **flow controlled:**
 - sender will not overwhelm receiver

TCP seq. numbers, ACKs

sequence numbers:

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

acknowledgements:

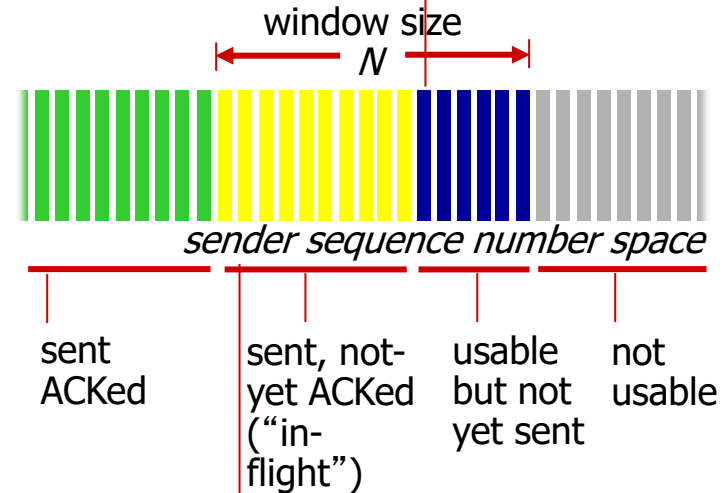
- seq # of next byte expected from other side
- cumulative ACK

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
	rwnd
checksum	urg pointer

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

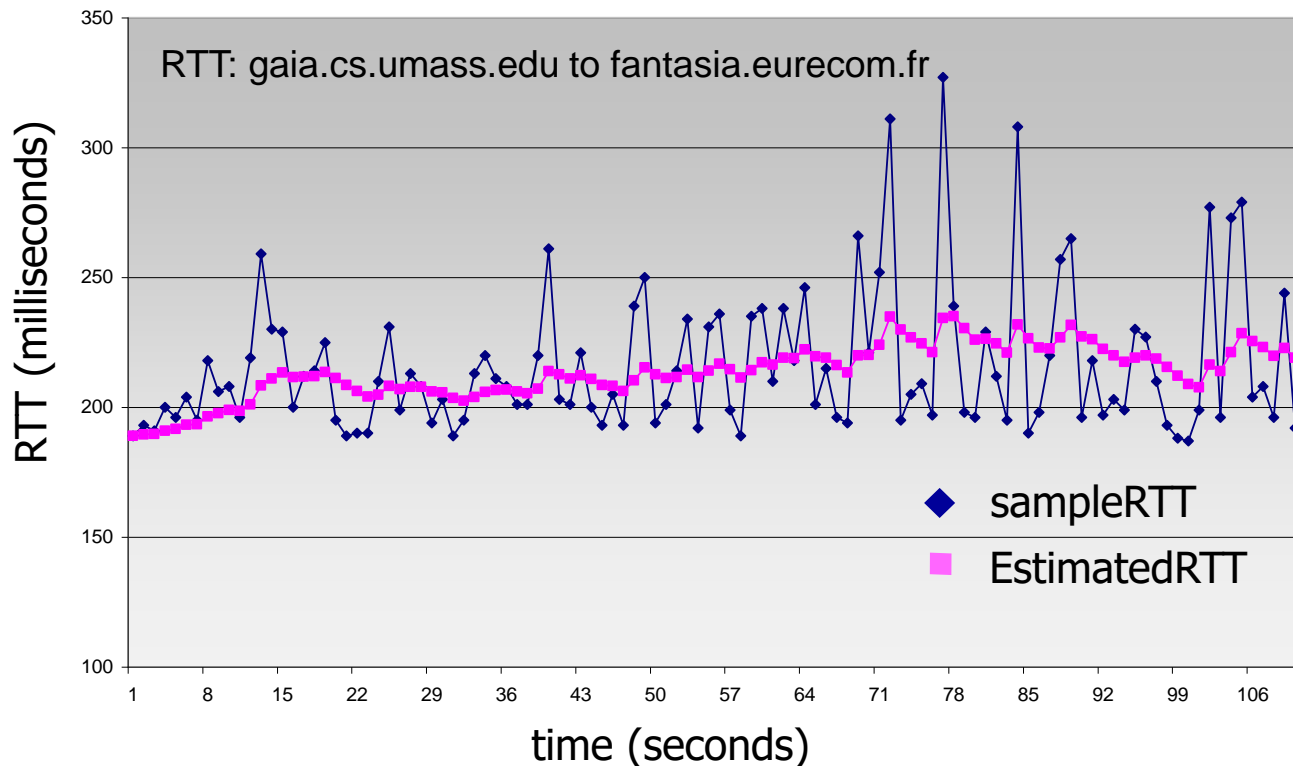
Q: how to estimate RTT?

- **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, $\beta = 0.25$)

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



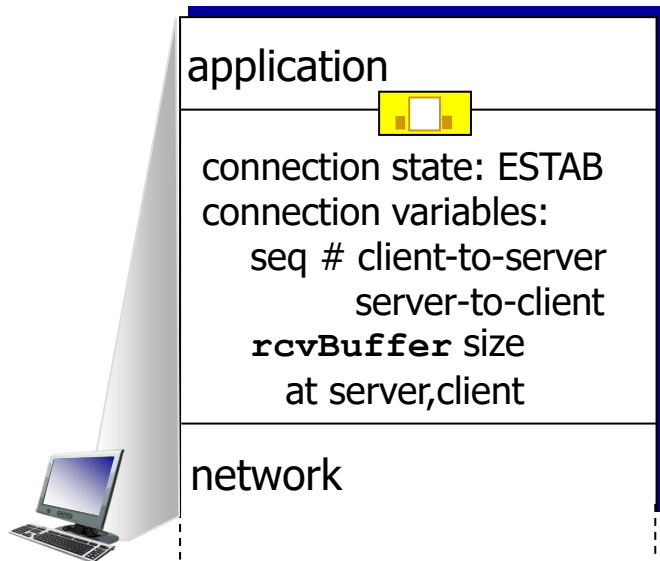
↑
estimated RTT

↑
“safety margin”

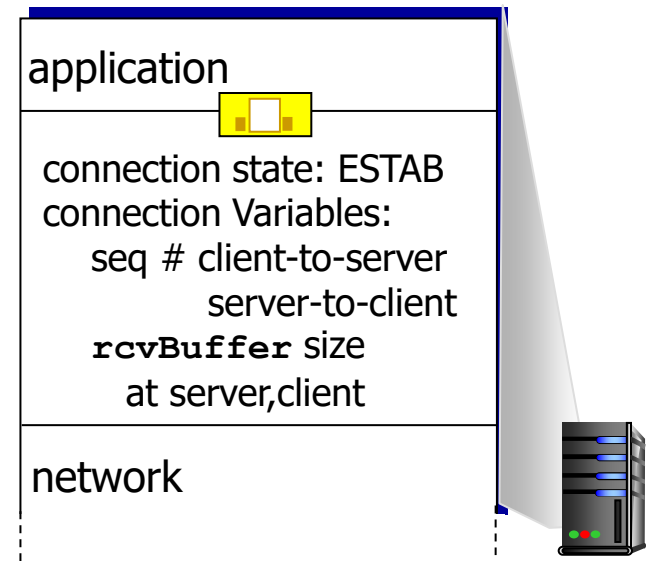
Connection Management

before exchanging data, sender/receiver “handshake”:

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



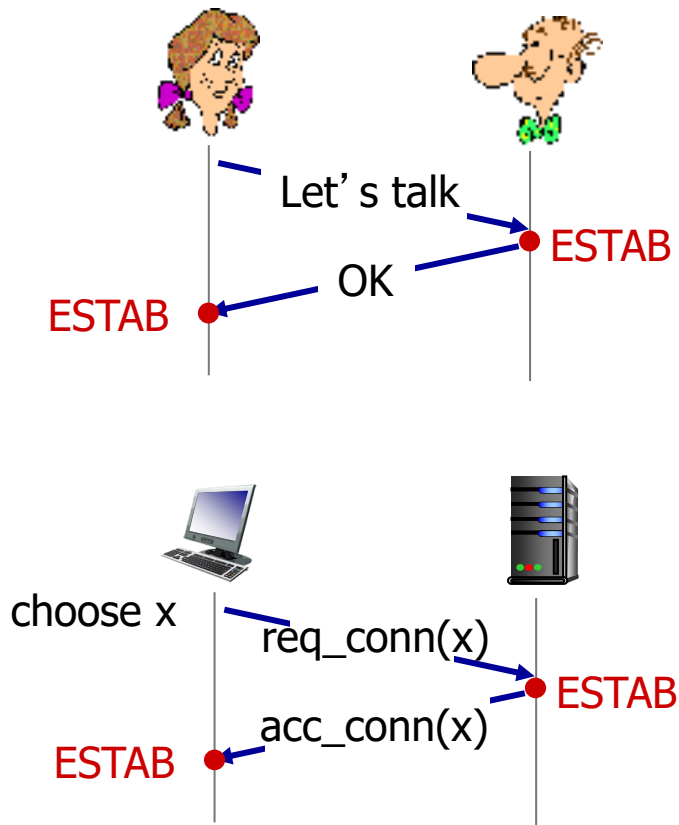
```
Socket clientSocket =  
    newSocket("hostname", "port  
    number");
```



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


Agreeing to establish a connection

2-way handshake:

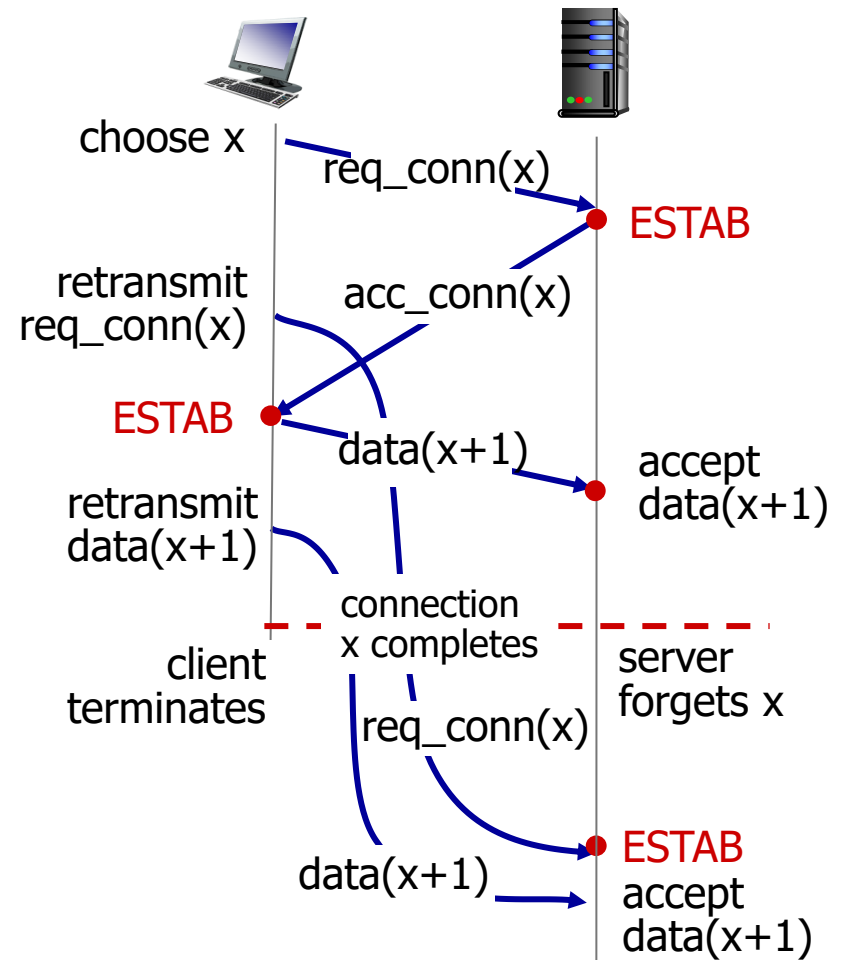
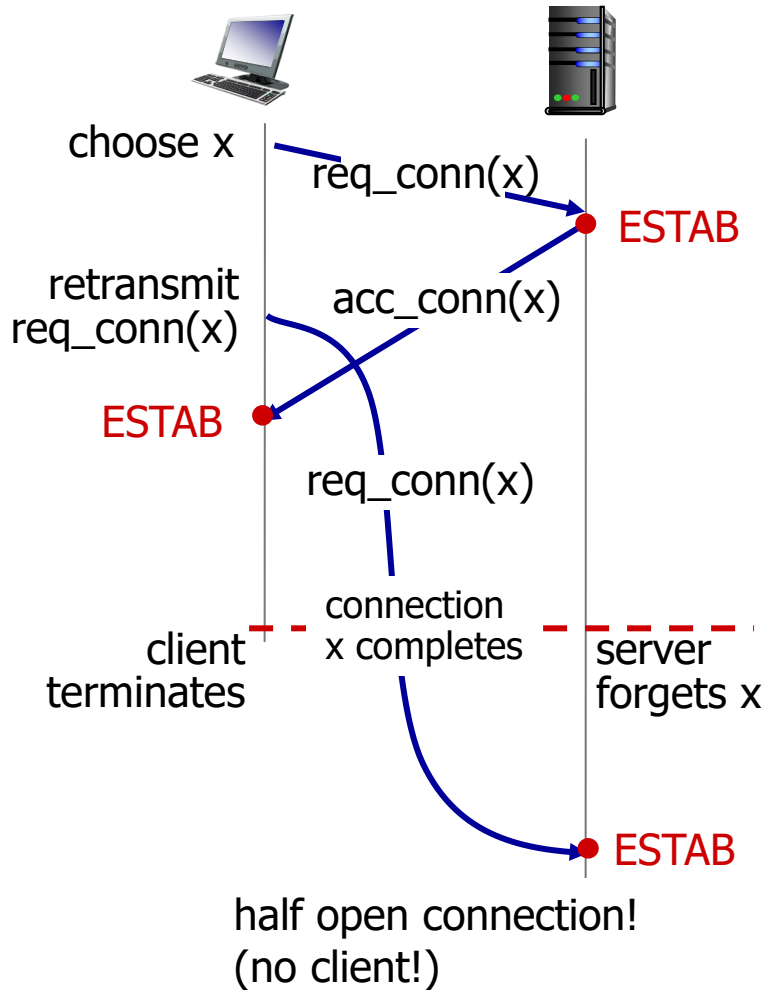


Q: will 2-way handshake always work in network?

- 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

client state

LISTEN

SYNSENT

ESTAB

choose init seq num, x
send TCP SYN msg

received SYNACK(x)
indicates server is live;
send ACK for SYNACK;
this segment may contain
client-to-server data



SYNbit=1, Seq=x

SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1

ACKbit=1, ACKnum=y+1

server state

LISTEN

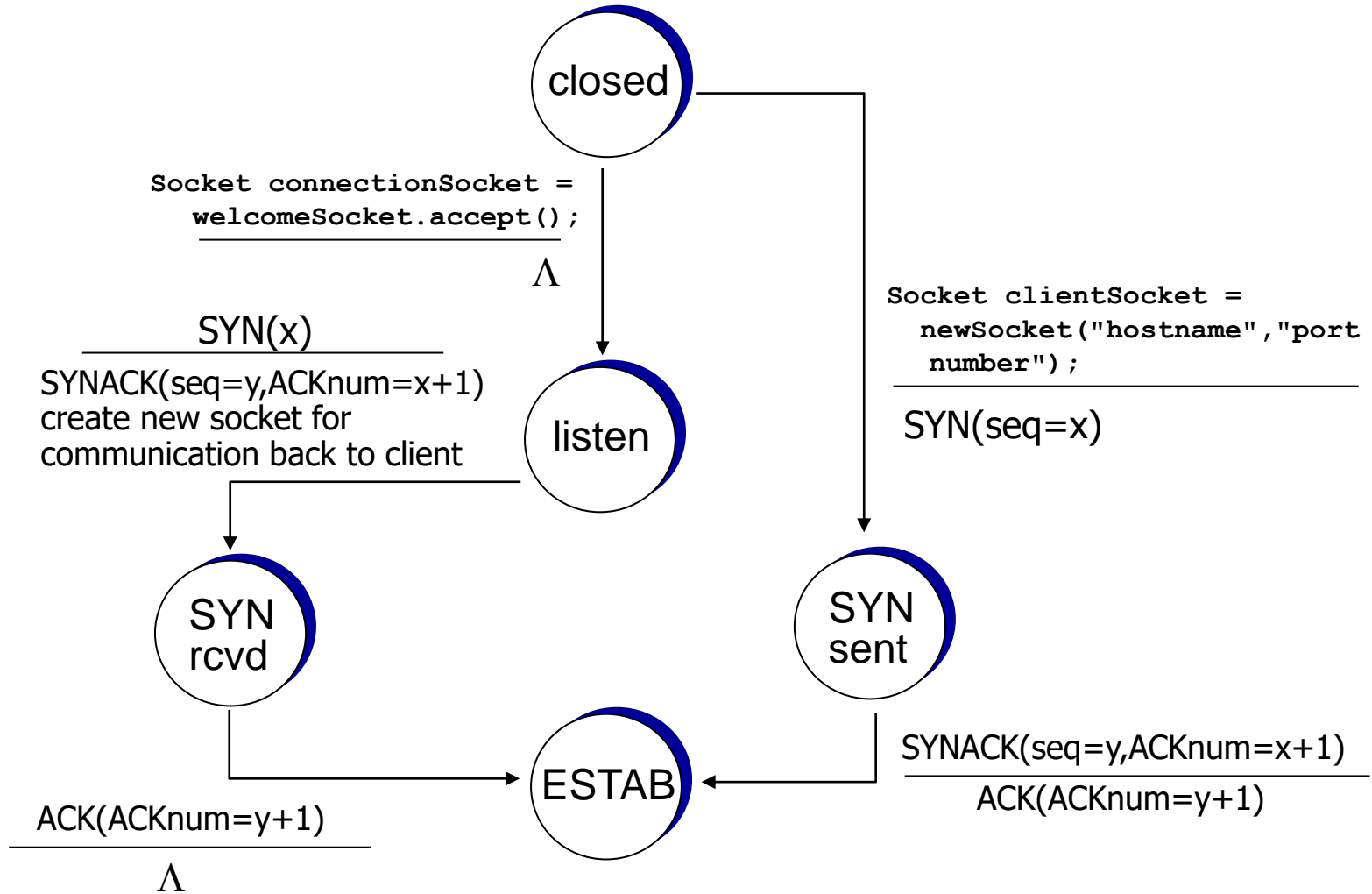
SYN RCVD

ESTAB

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

received ACK(y)
indicates client is live

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

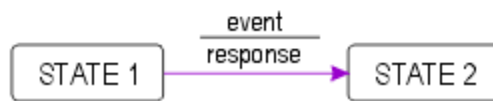
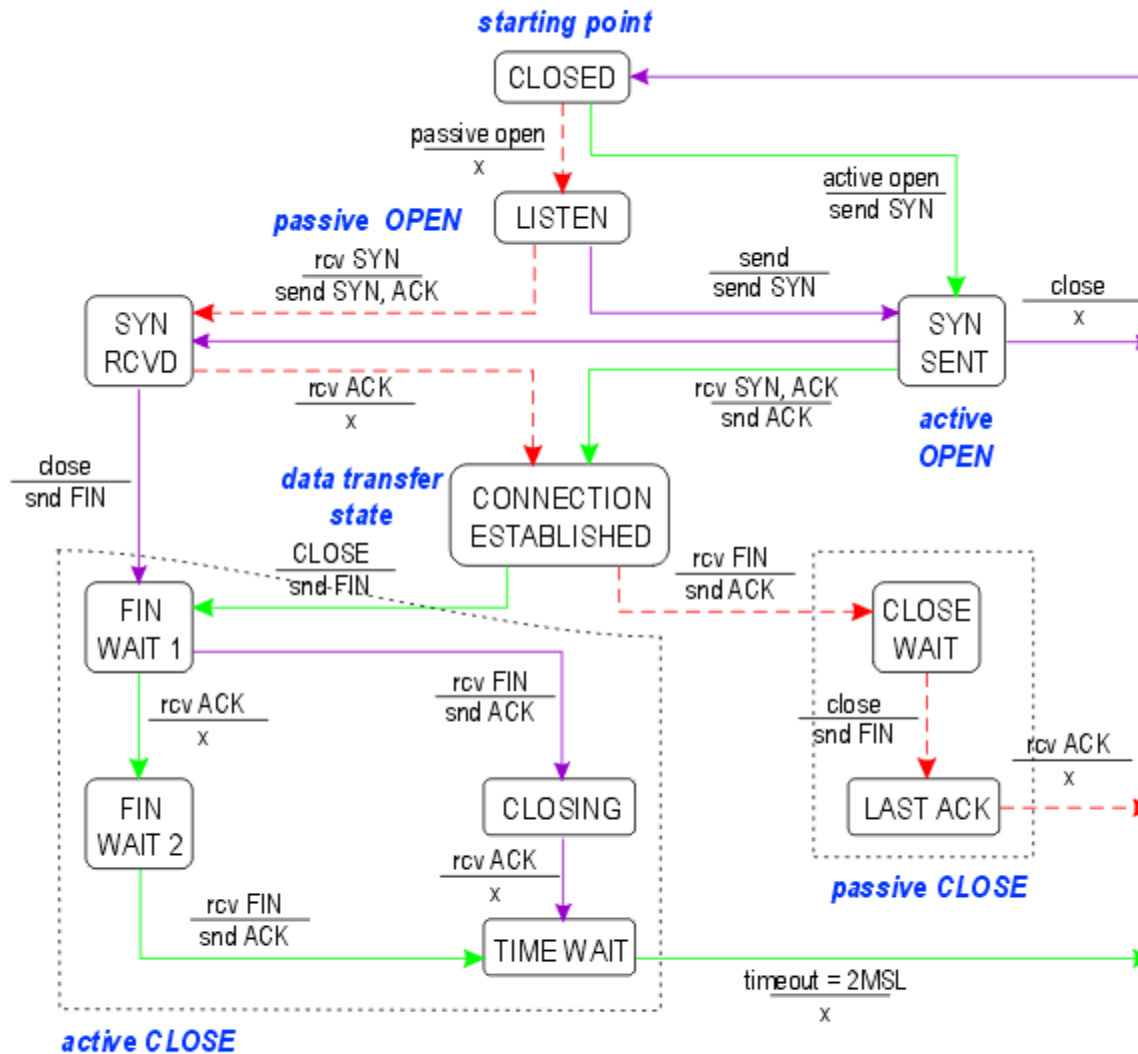
ESTAB

CLOSE_WAIT

LAST_ACK

CLOSED

TCP finite state machine



--- Typical server transactions

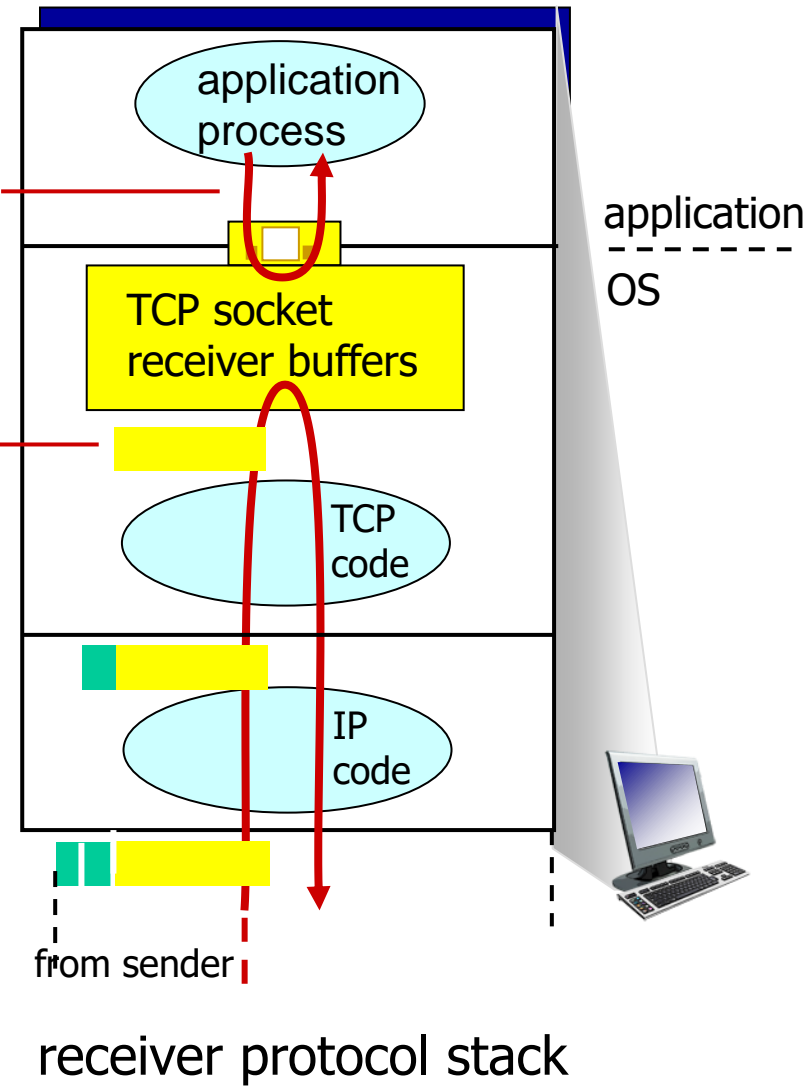
— Typical client transactions

MSL = Maximum Segment Lifetime (120 sec)

TCP flow control

application may remove data from TCP socket buffers ...

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

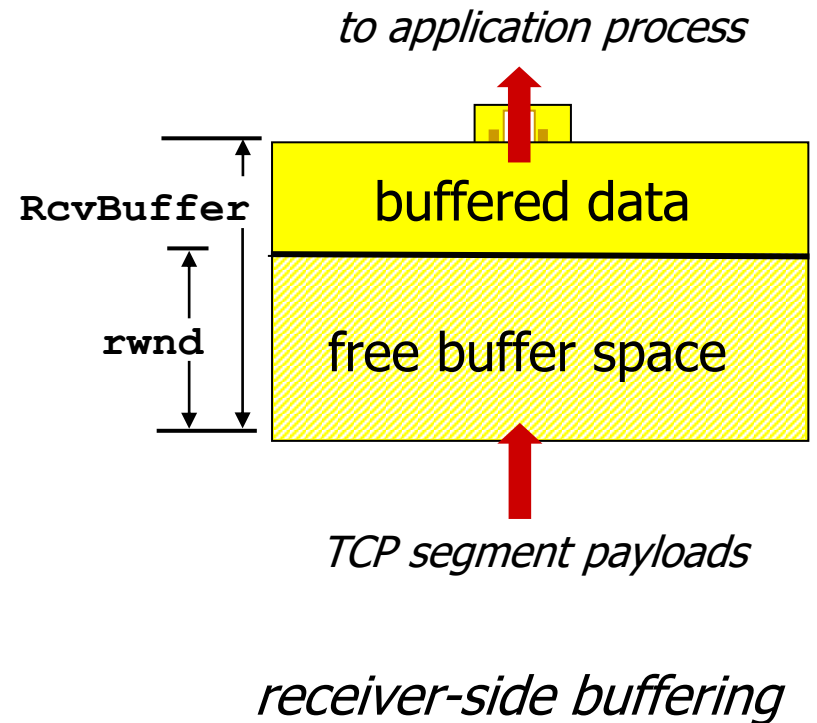


flow control

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



TCP Congestion control

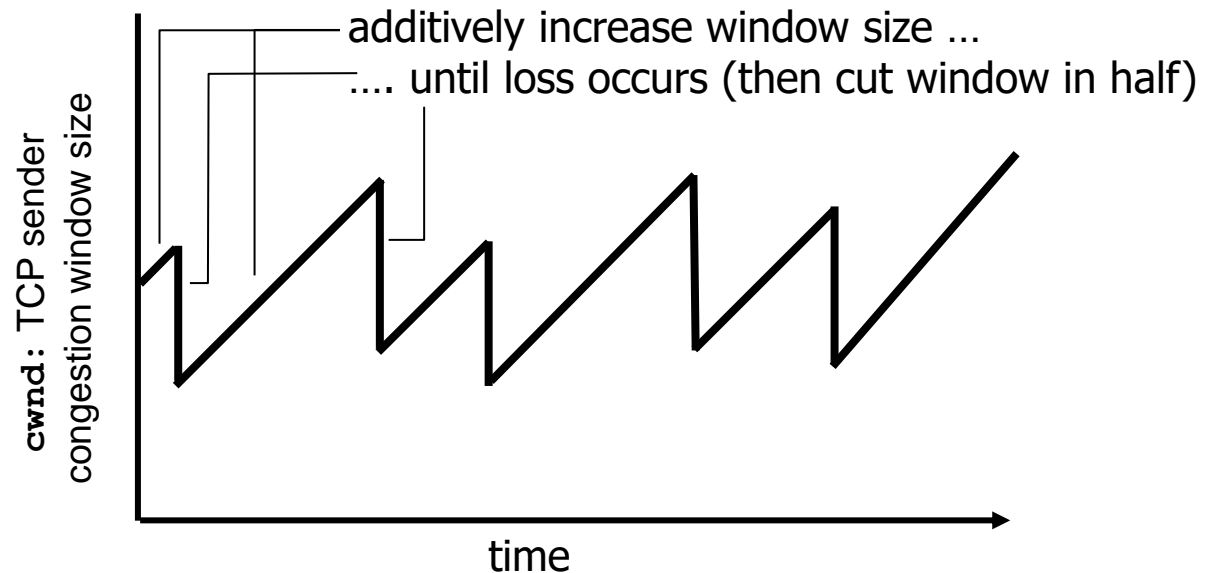
congestion:

- 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!

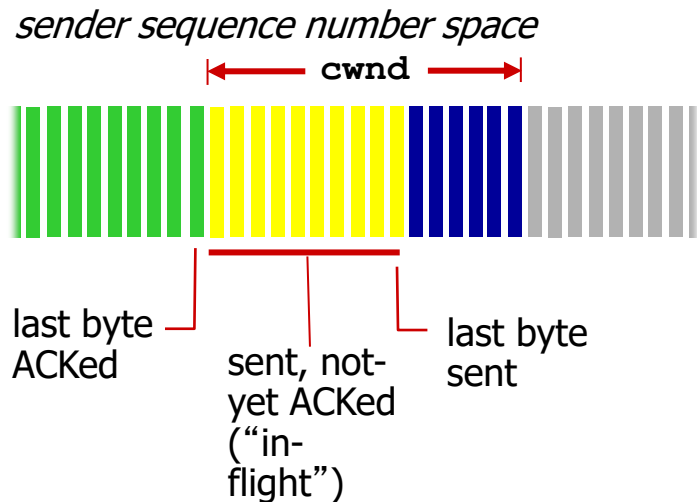
TCP congestion control: additive increase multiplicative decrease

- *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:

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

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

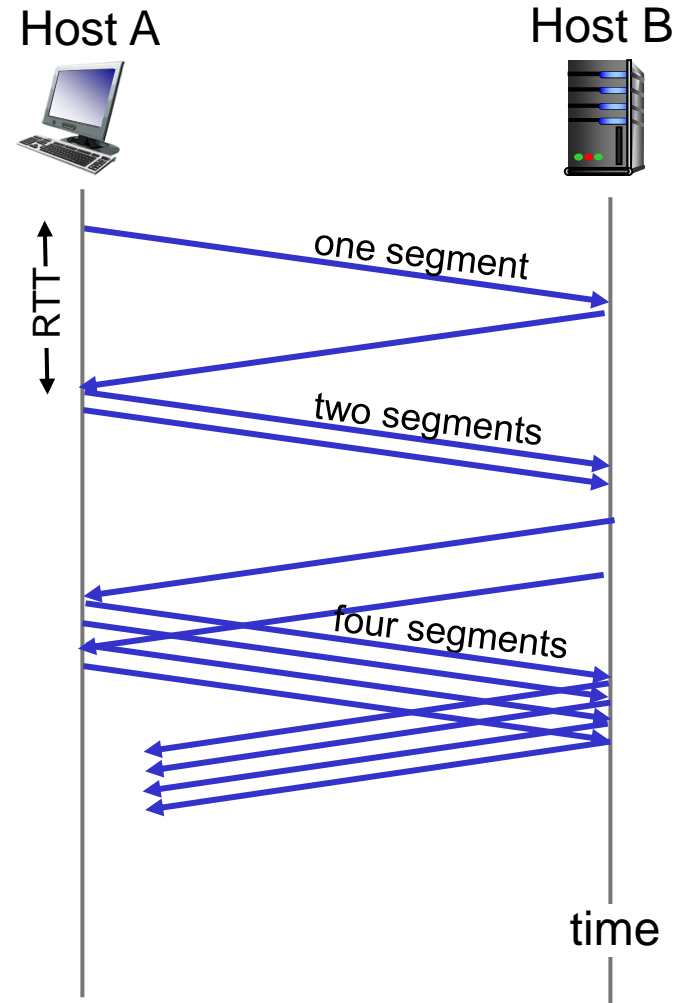
TCP sending rate:

- *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)

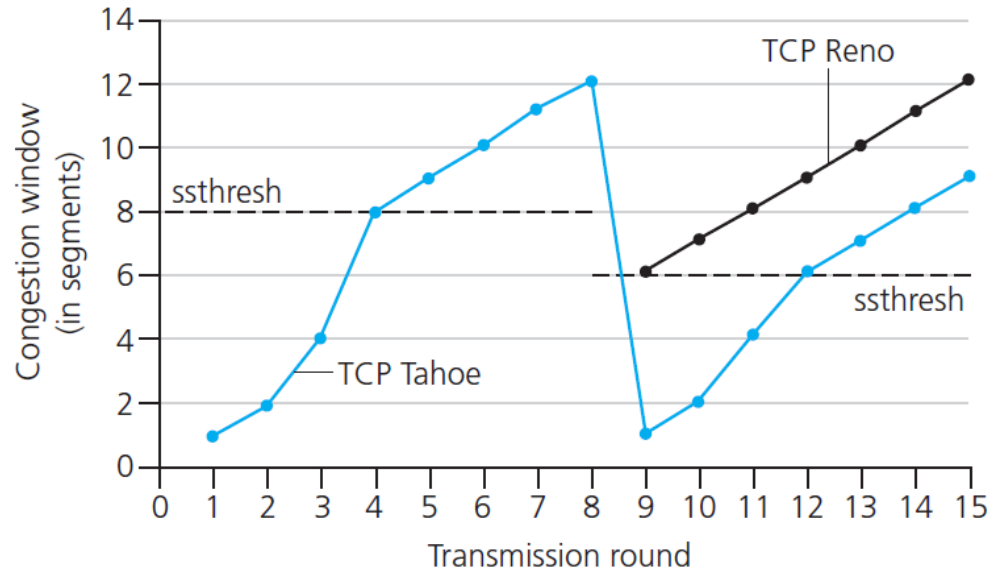
TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

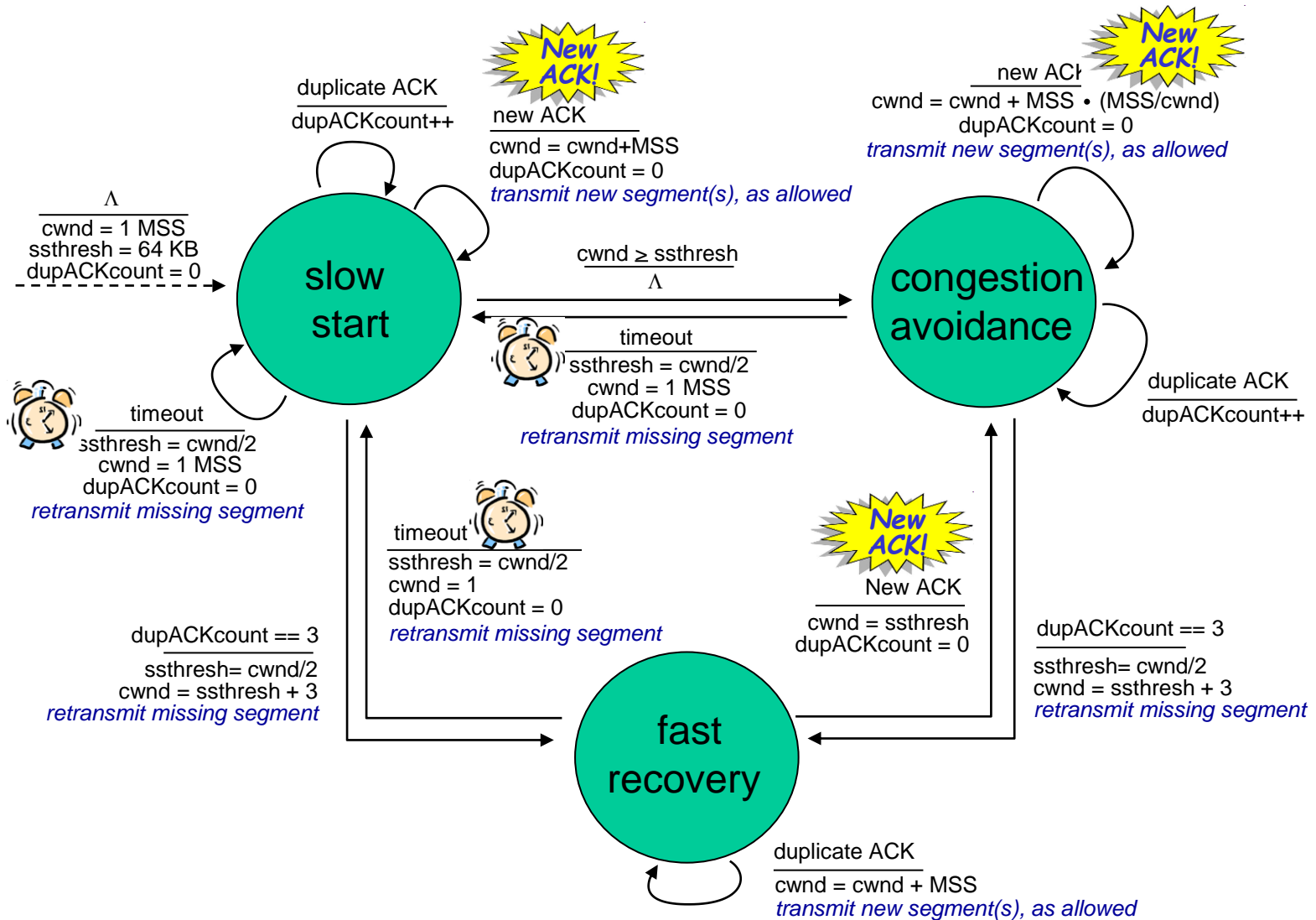
A: when **cwnd** gets to 1/2 of its value before timeout.

Implementation:

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



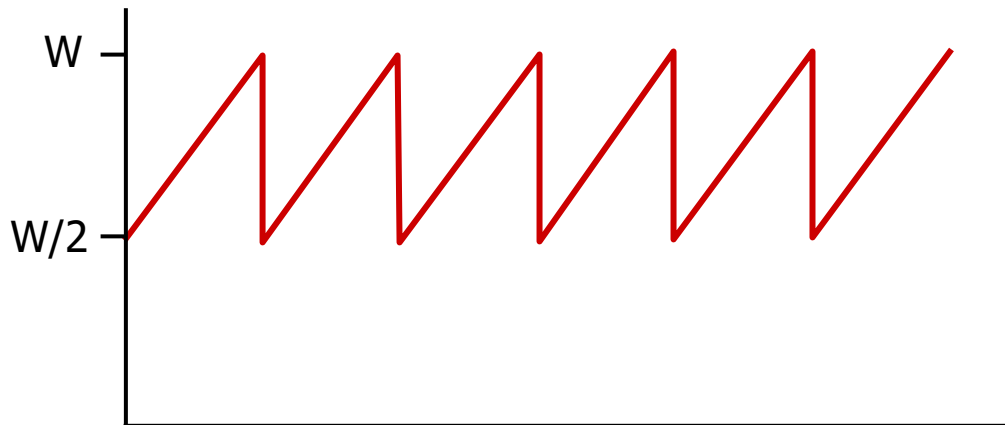
Summary: TCP Congestion Control



TCP throughput

- avg. TCP thrupt 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. thrupt is $\frac{3}{4}W$ per RTT

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



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]:

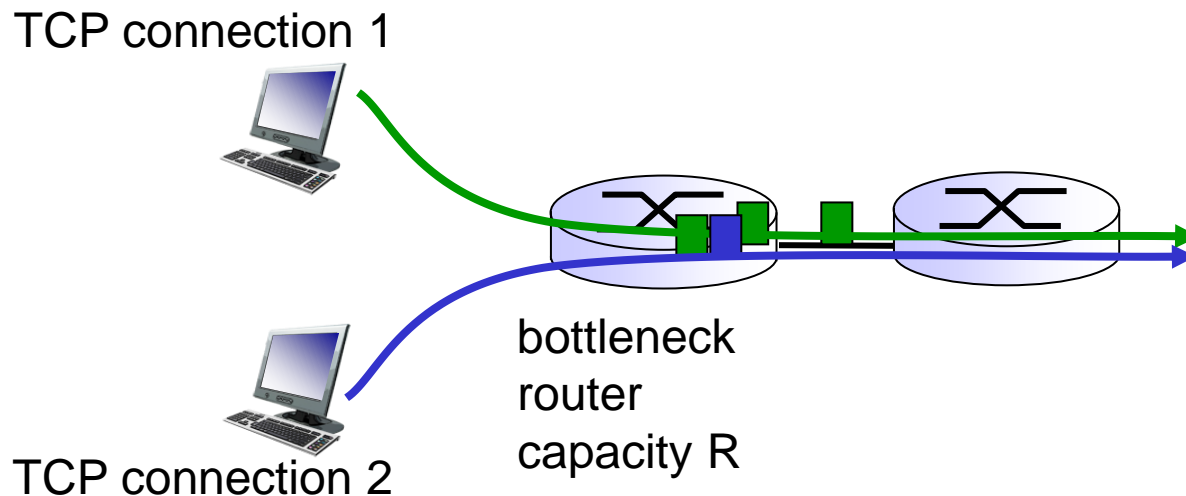
$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT} \sqrt{L}}$$

→ 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

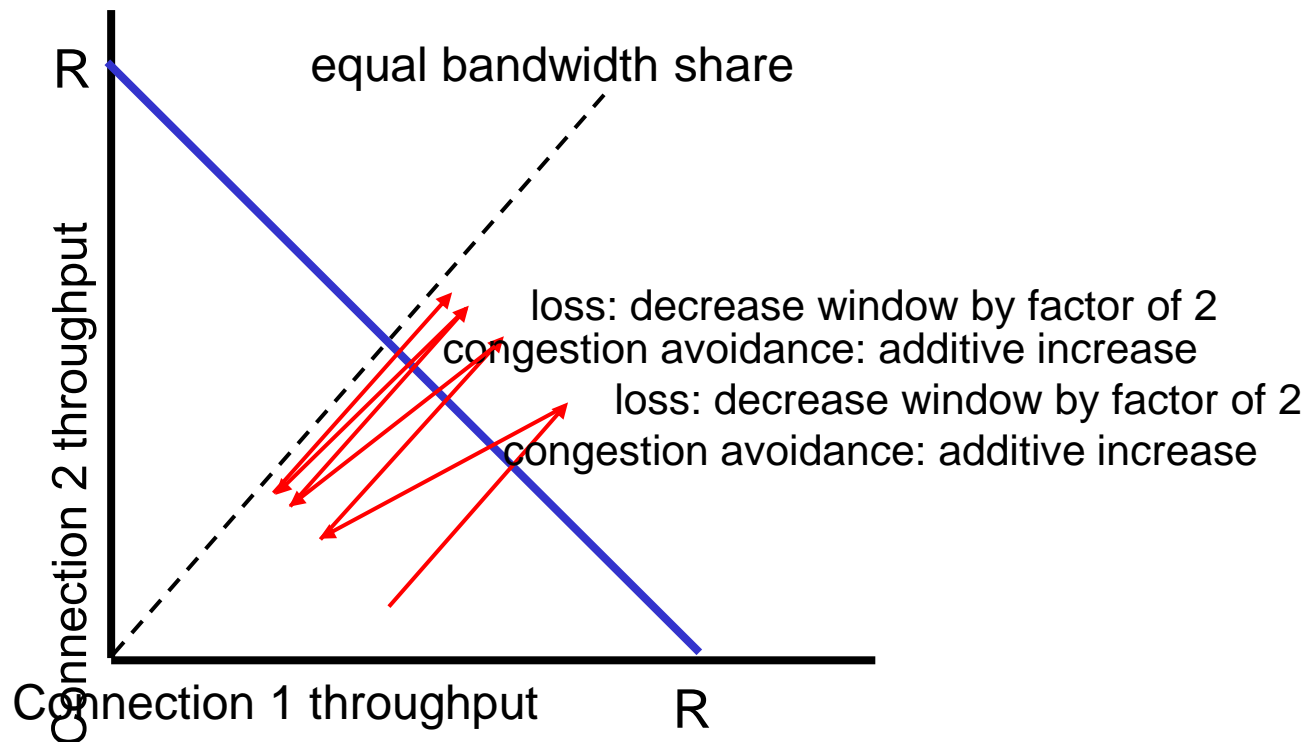
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R , each should have average rate of R/K



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 $R/2$

Explicit Congestion Notification (ECN)

network-assisted congestion control:

- two bits in IP header (ToS field) marked *by network router* to indicate congestion
- congestion indication carried to receiving host
- receiver (seeing congestion indication in IP datagram)) sets ECE bit on receiver-to-sender ACK segment to notify sender of congestion

