CS 305: Computer Networks

Fall 2023

Lecture 8: Transport Layer

Ming Tang

Department of Computer Science and Engineering
Southern University of Science and Technology (SUSTech)

TCP Reliable Data Transfer

- Segment structure
- Round-trip time estimation
- Reliable data transfer
- Flow control
- Control management

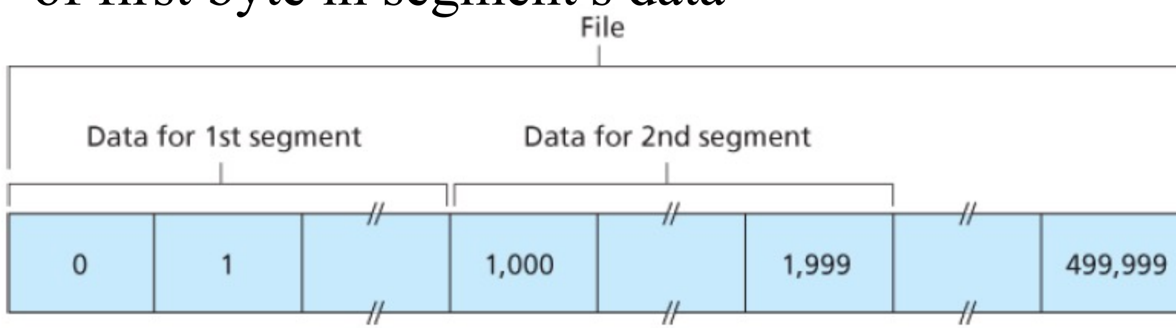
TCP seq. numbers, ACKs

TCP views data as an unstructured, but ordered, stream of bytes.

- Sequence numbers are over the stream of transmitted bytes and *not* over the series of transmitted segments

sequence numbers:

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



acknowledgements:

- seq # of next byte expected from other side
  - E.g., receiver has received bytes numbered 0 through 535 and 900 through 1000; then, acknowledgement number is 536.
- cumulative ACK

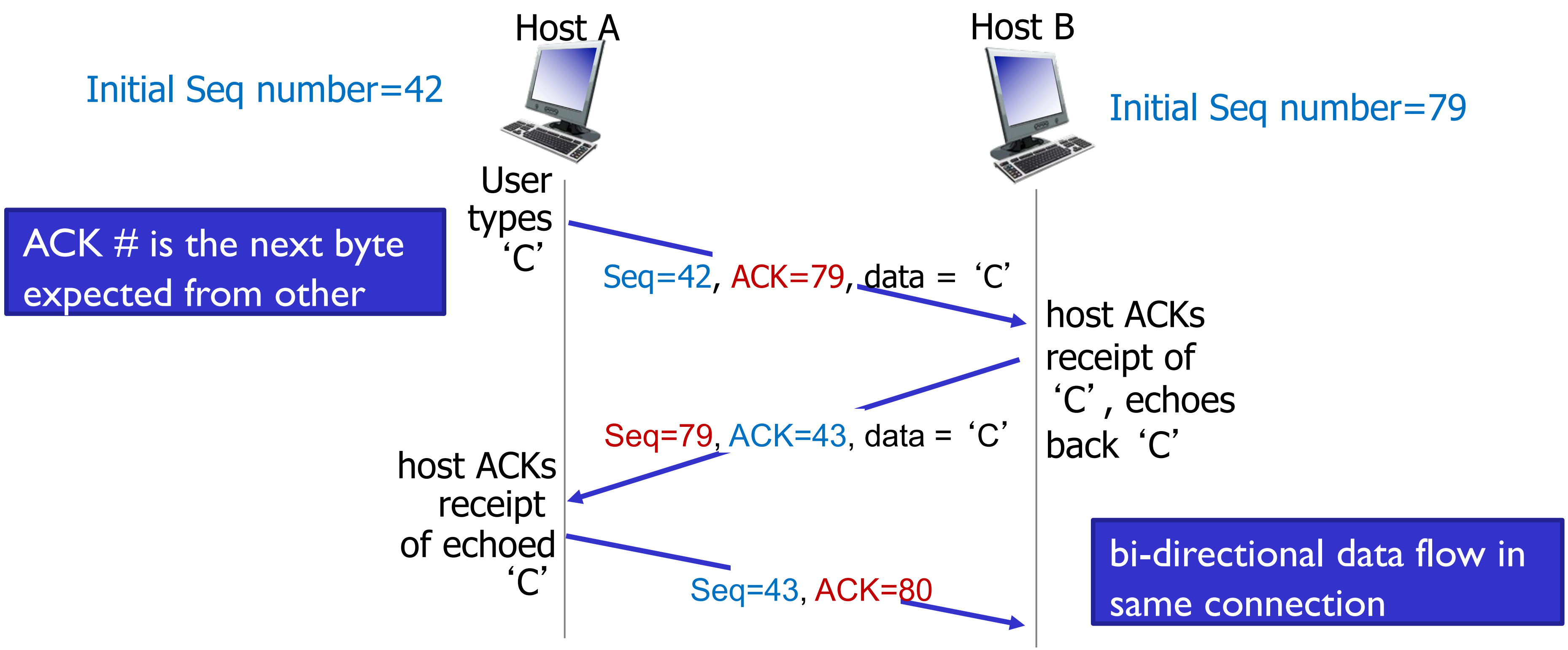
Q: how receiver handles out-of-order segments

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

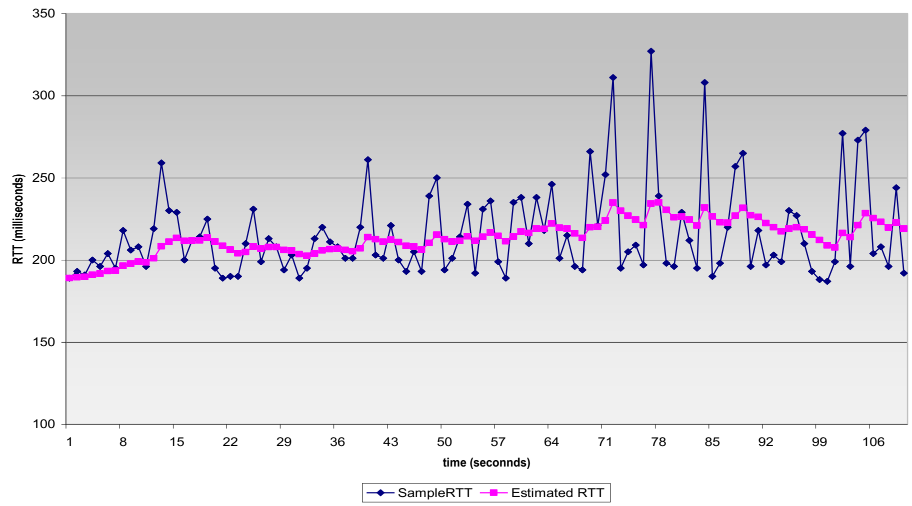
Initial sequence number is randomly chosen

Telnet Case Study

- full duplex data:
  - bi-directional data flow in same connection



TCP round trip time, timeout



TCP timeout interval: `EstimatedRTT` plus "safety margin"

large variation in `EstimatedRTT` -> larger safety margin

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

estimated RTT

"safety margin"

TCP Reliable Data Transfer

- Segment structure
- Round-trip time estimation
- Reliable data transfer
- Flow control
- Control management

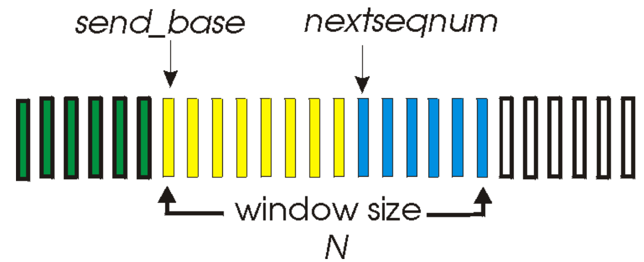
TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
  - pipelined segments: window size, SendBase
  - 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

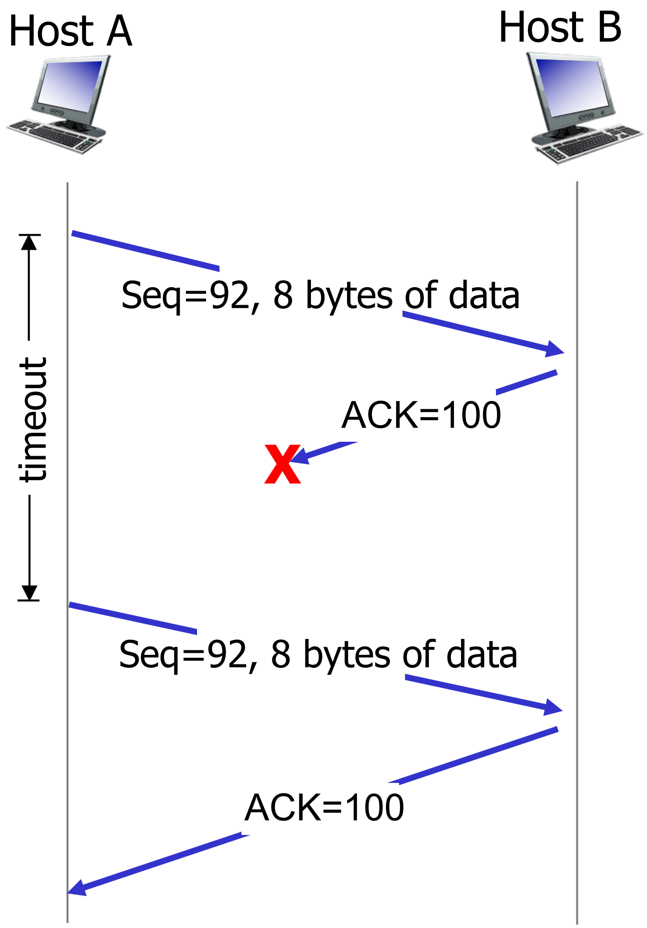
ack rcvd:

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

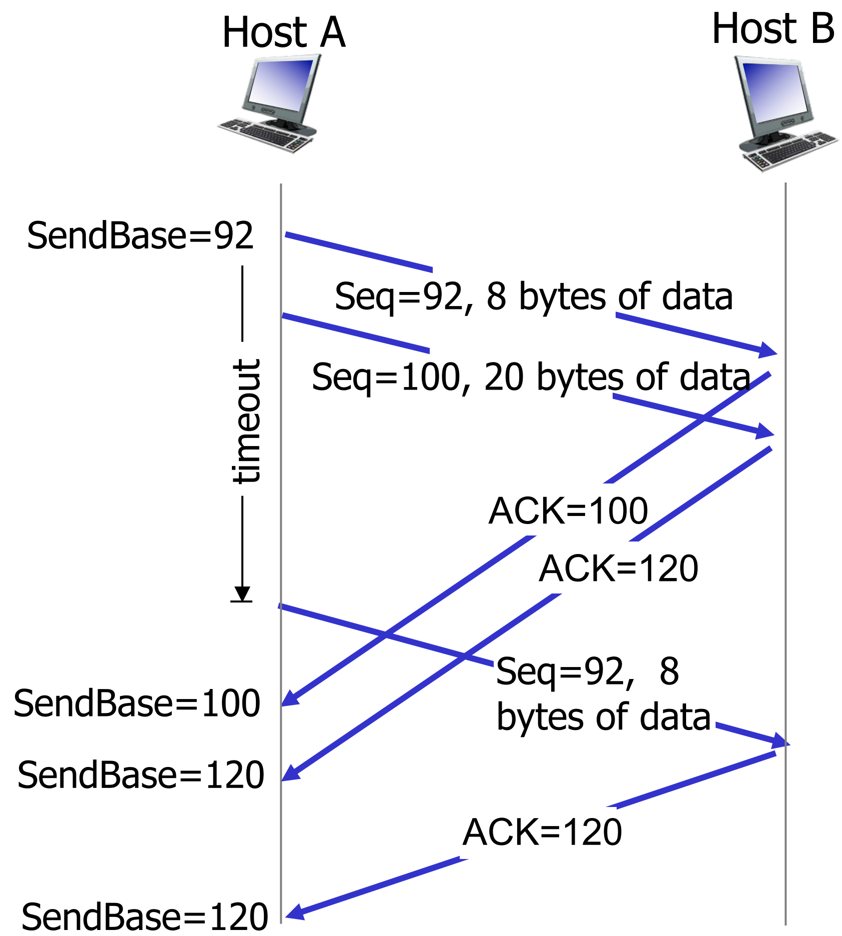
TCP sender events:

```
NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

loop (forever) {
    switch(event)

        event: data received from application above
            create TCP segment with sequence number NextSeqNum
            if (timer currently not running)
                start timer
            pass segment to IP
            NextSeqNum=NextSeqNum+length(data)
            break;

        event: timer timeout
            retransmit not-yet-acknowledged segment with
                smallest sequence number
            start timer
            break;

        event: ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase=y
                if (there are currently any not-yet-acknowledged segments)
                    start timer
                }
            break;

    } /* end of loop forever */
```

TCP: retransmission scenarios

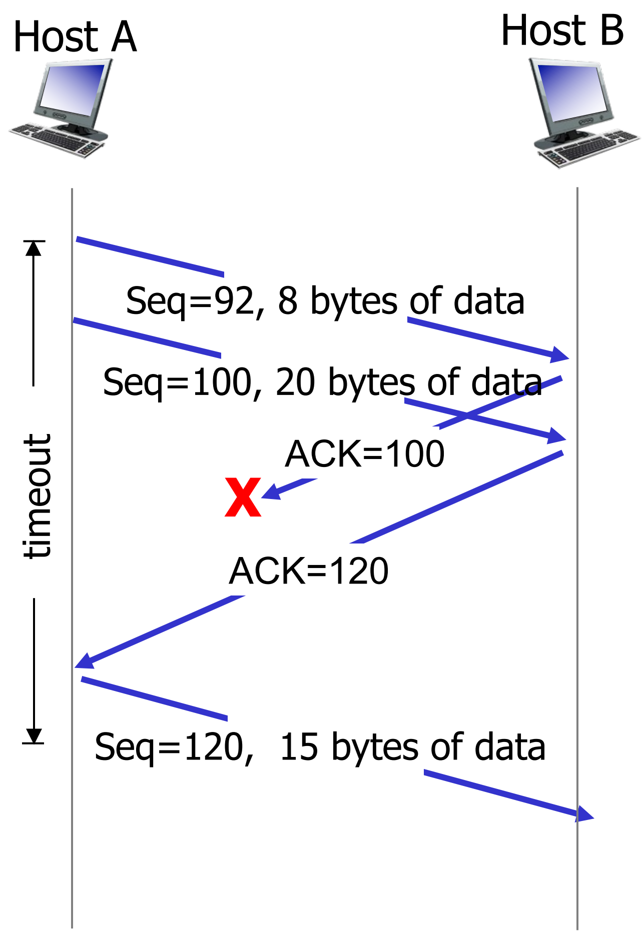


lost ACK scenario



premature timeout

TCP: retransmission scenarios



cumulative ACK

TCP receiver [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 duplicate ACKs for same data

("triple duplicate ACKs"), resend unacked segment with smallest seq #

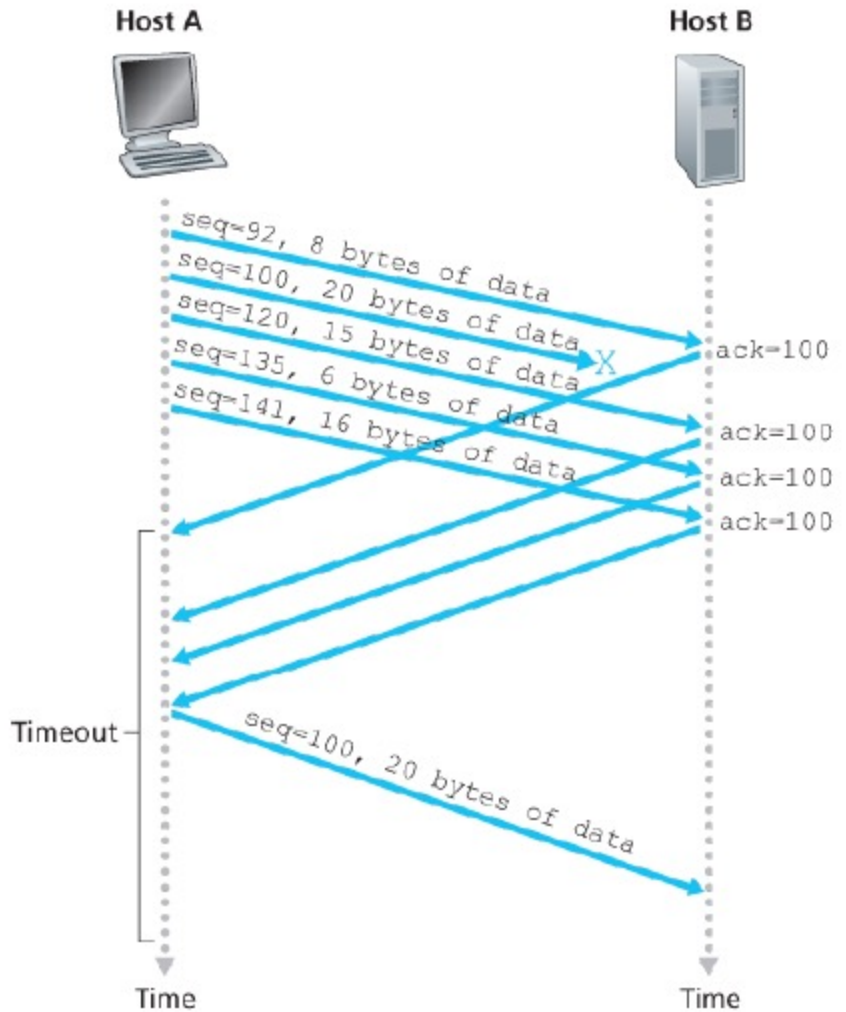
- likely that unacked segment lost, so don't wait for timeout

TCP fast retransmit

```
NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

loop (forever) {
    switch(event)

        event: data received from applicatio
            create TCP segment with sequence
            if (timer currently not running)
                start timer
            pass segment to IP
            NextSeqNum=NextSeqNum+length(data)
            break;

        event: timer timeout
            retransmit not-yet-acknowledged segment
                smallest sequence number
            start timer
            break;

        event: ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase=y
                if (there are currently any not-yet-acknowledged segments)
                    start timer
                }
            break;

} /* end of loop forever */
```

```
event: ACK received, with ACK field value of y
            if (y > SendBase) {
            SendBase=y
            if (there are currently any not yet
                        acknowledged segments)
               start timer
            }
            else {/* a duplicate ACK for already ACKed
                  segment */
               increment number of duplicate ACKs
                  received for y
               if (number of duplicate ACKS received
                  for y==3)
                  /* TCP fast retransmit */
                  resend segment with sequence number y
               }
           break;
```

TCP fast retransmit

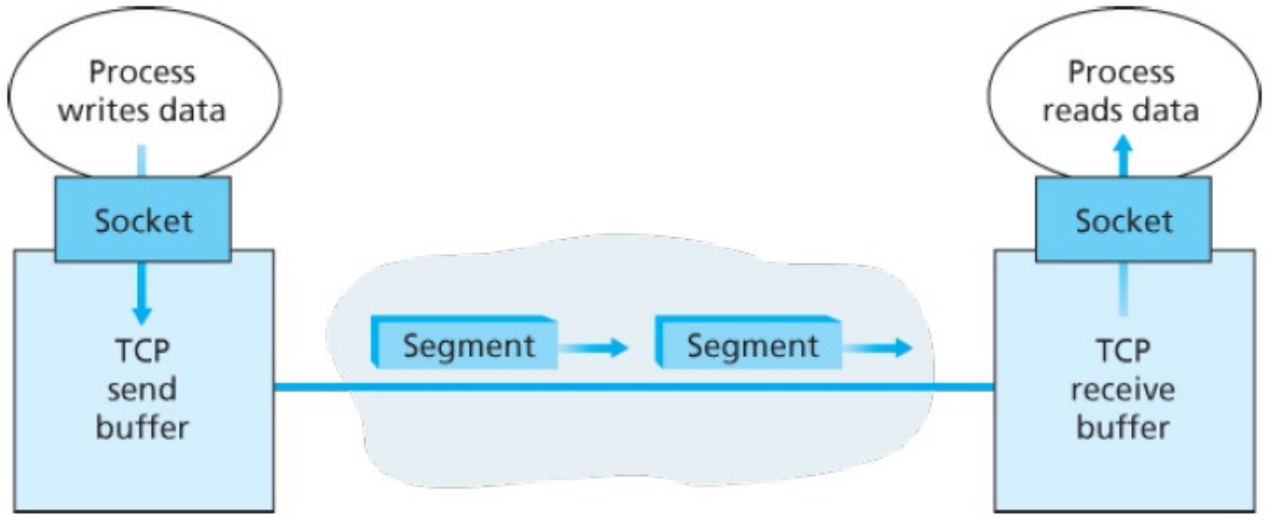


fast retransmit after sender receipt of triple duplicate ACK

TCP Reliable Data Transfer

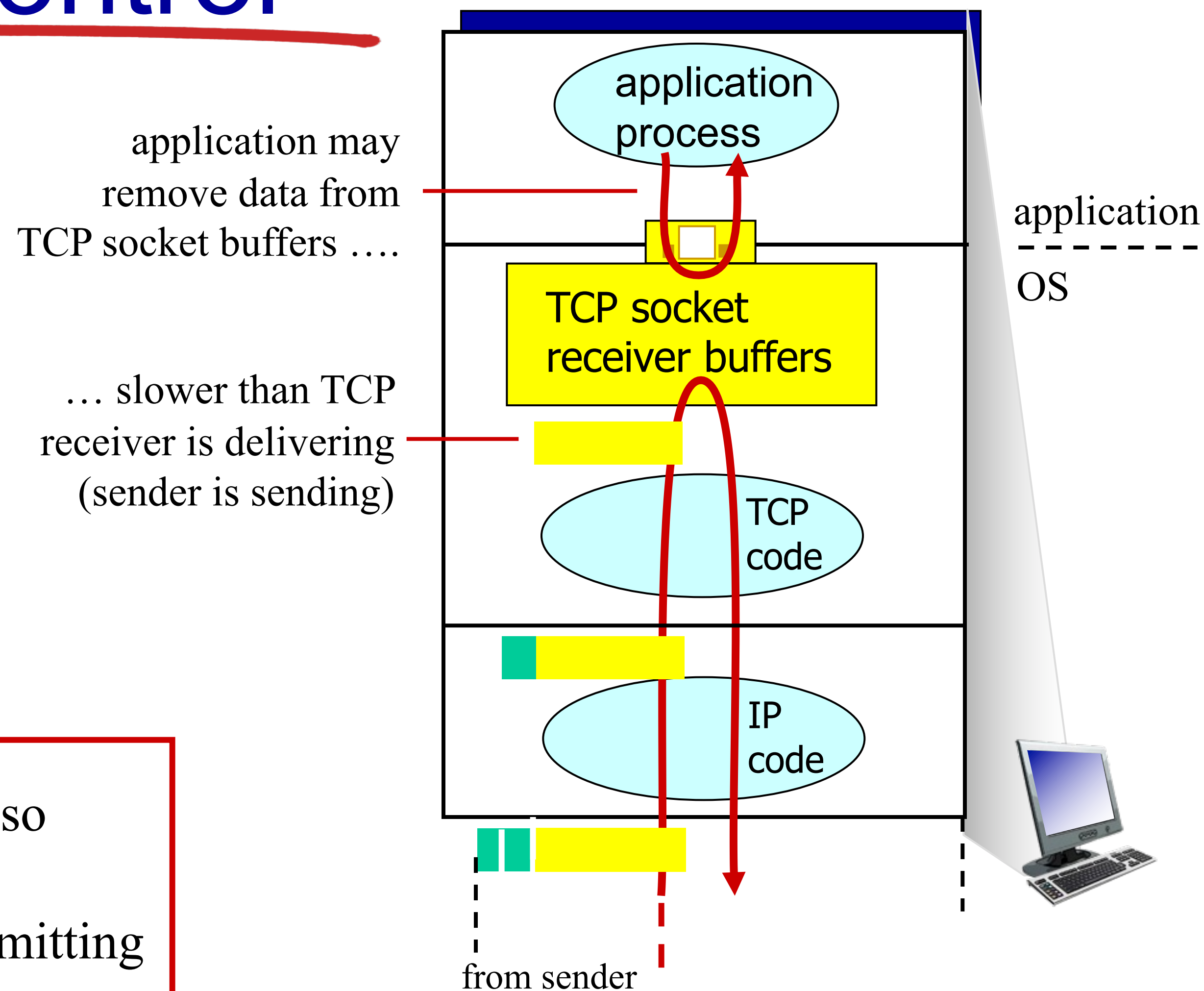
- Segment structure
- Round-trip time estimation
- Reliable data transfer
- Flow control
- Control management

TCP: Overview



- TCP connection
- TCP grab chunks of data from the sender buffer
- TCP receives a segment at the other end, place it in receiver buffer
- application reads the stream from the receive buffer

TCP flow control



receiver protocol stack

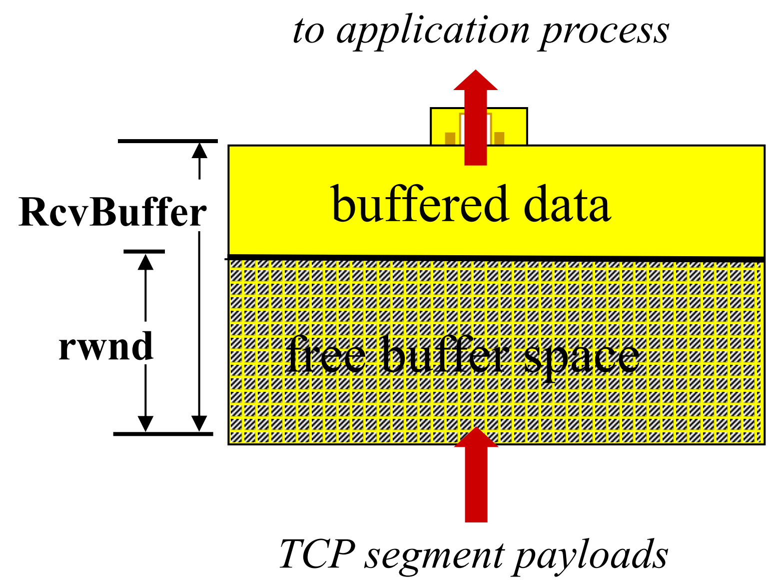
flow control

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

TCP flow control

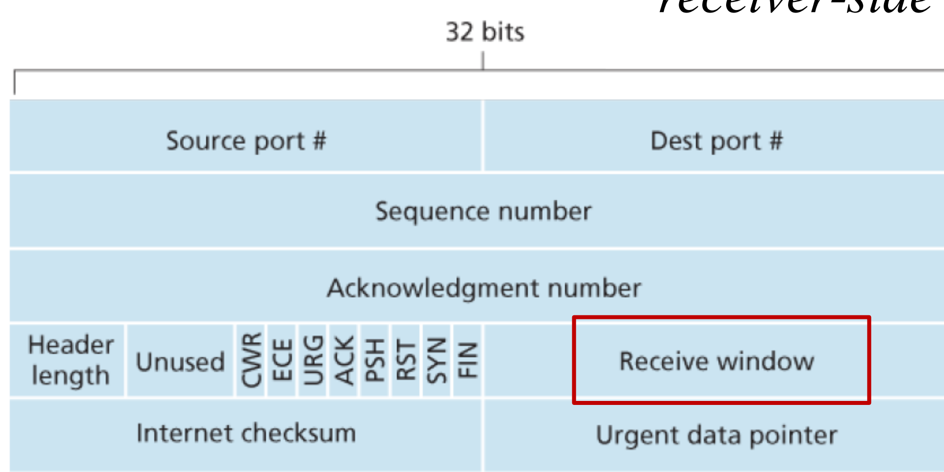
Receiver "advertises" free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments

- **RcvBuffer** size set via socket options (typical default is 4096 bytes)
- many operating systems autoadjust **RcvBuffer**



rwnd=RcvBuffer−[LastByteRcvd−LastByteRead]

receiver-side buffering

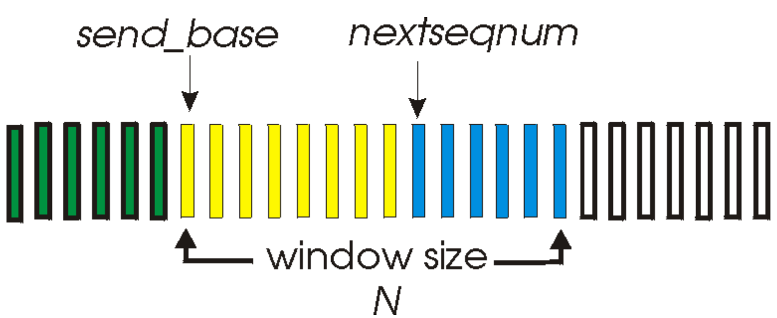


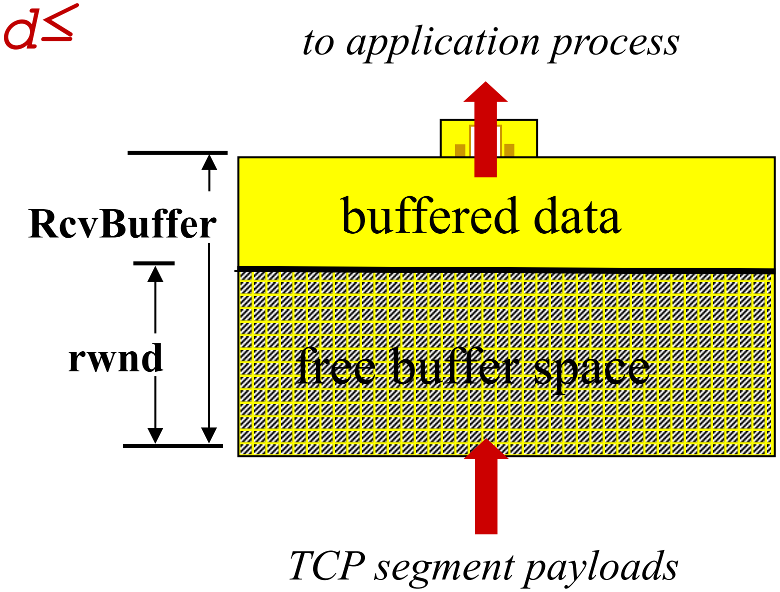
TCP flow control

Sender limits amount of unacked ("in-flight") data to receiver's **rwnd** value

LastByteSent-LastByteAcked $\leq$ *rwnd*

Guarantees receive buffer will not overflow





receiver-side buffering

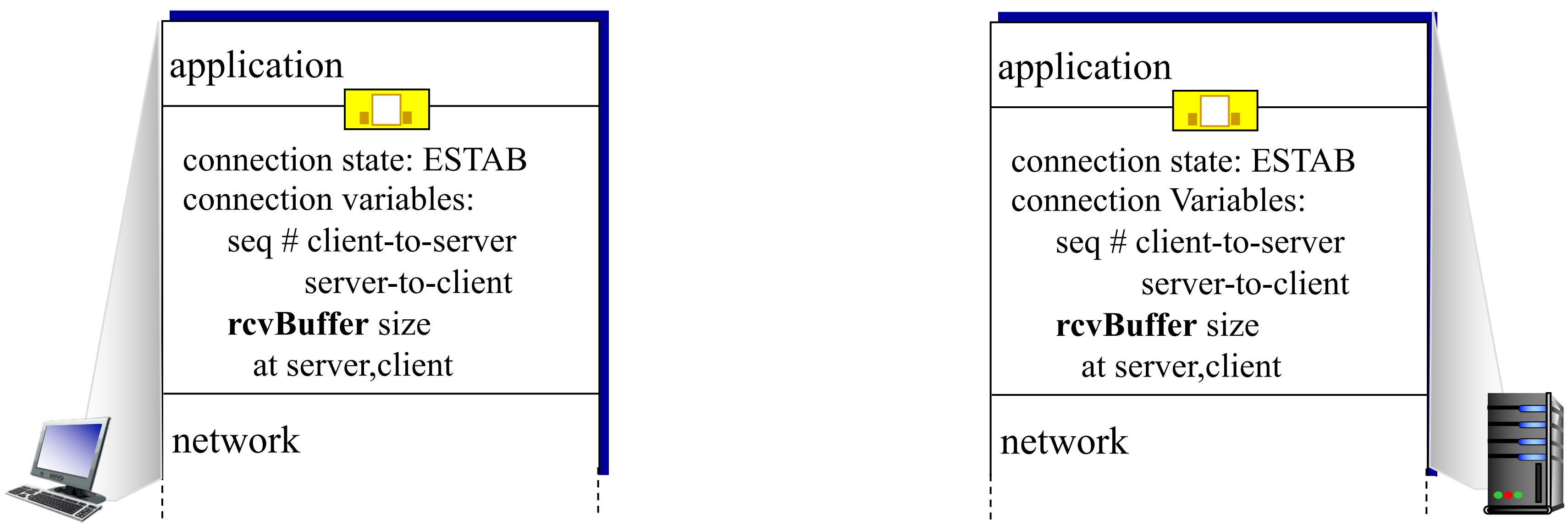
TCP Reliable Data Transfer

- Segment structure
- Round-trip time estimation
- Reliable data transfer
- Flow control
- Control management

Connection Management

before exchanging data, sender/receiver "handshake":

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

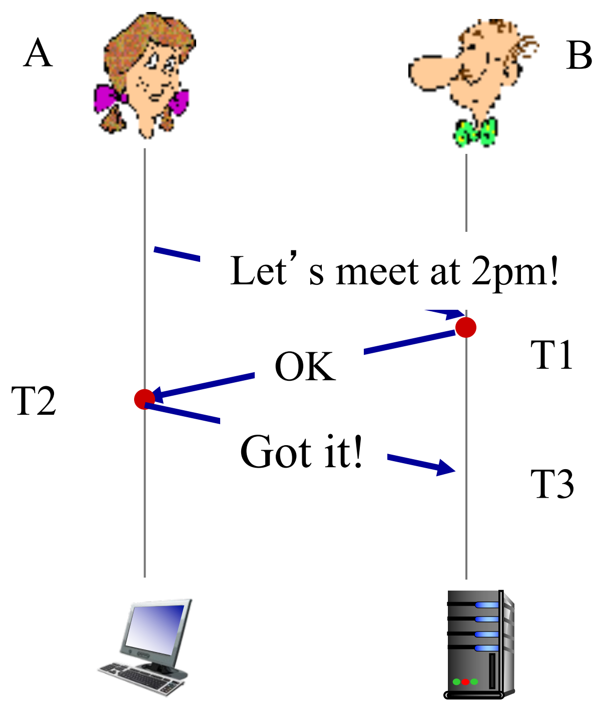



```
clientSocket = socket(AF_INET, SOCK_STREAM);
clientSocket.connect((hostname,port number));
```

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

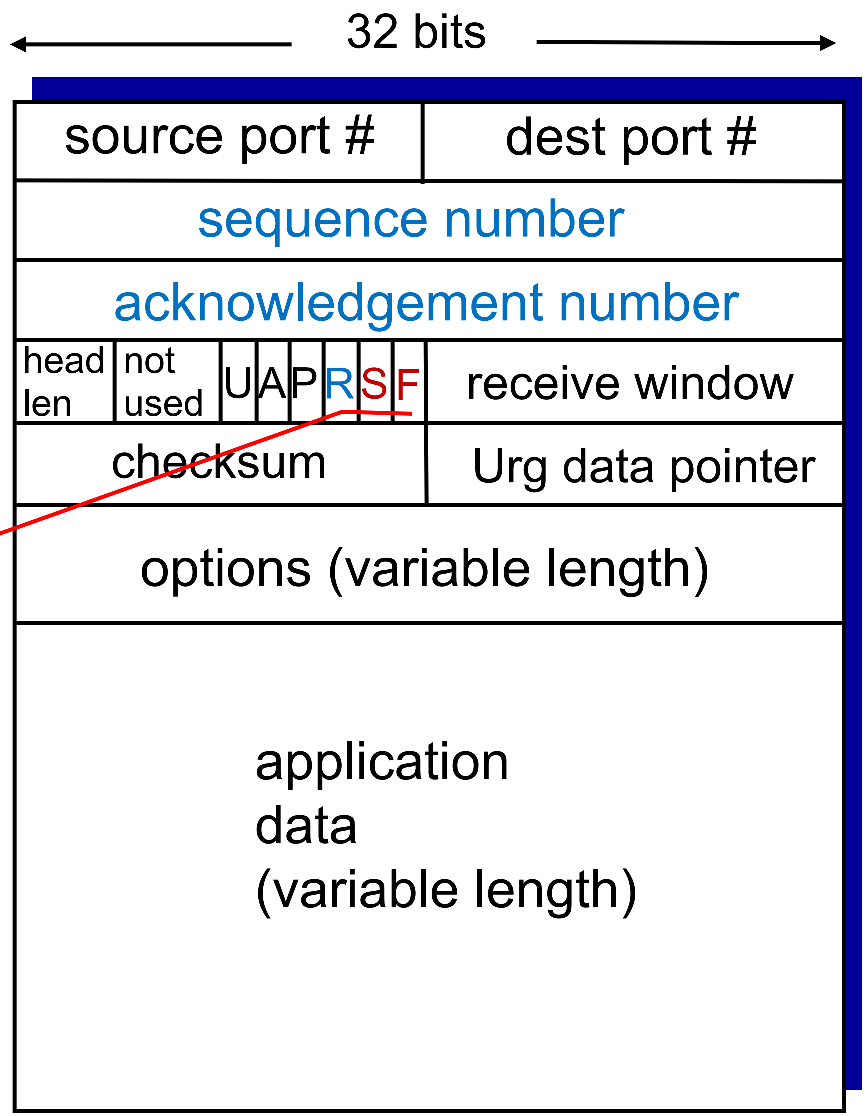
Agreeing to establish a connection

3-way handshake:



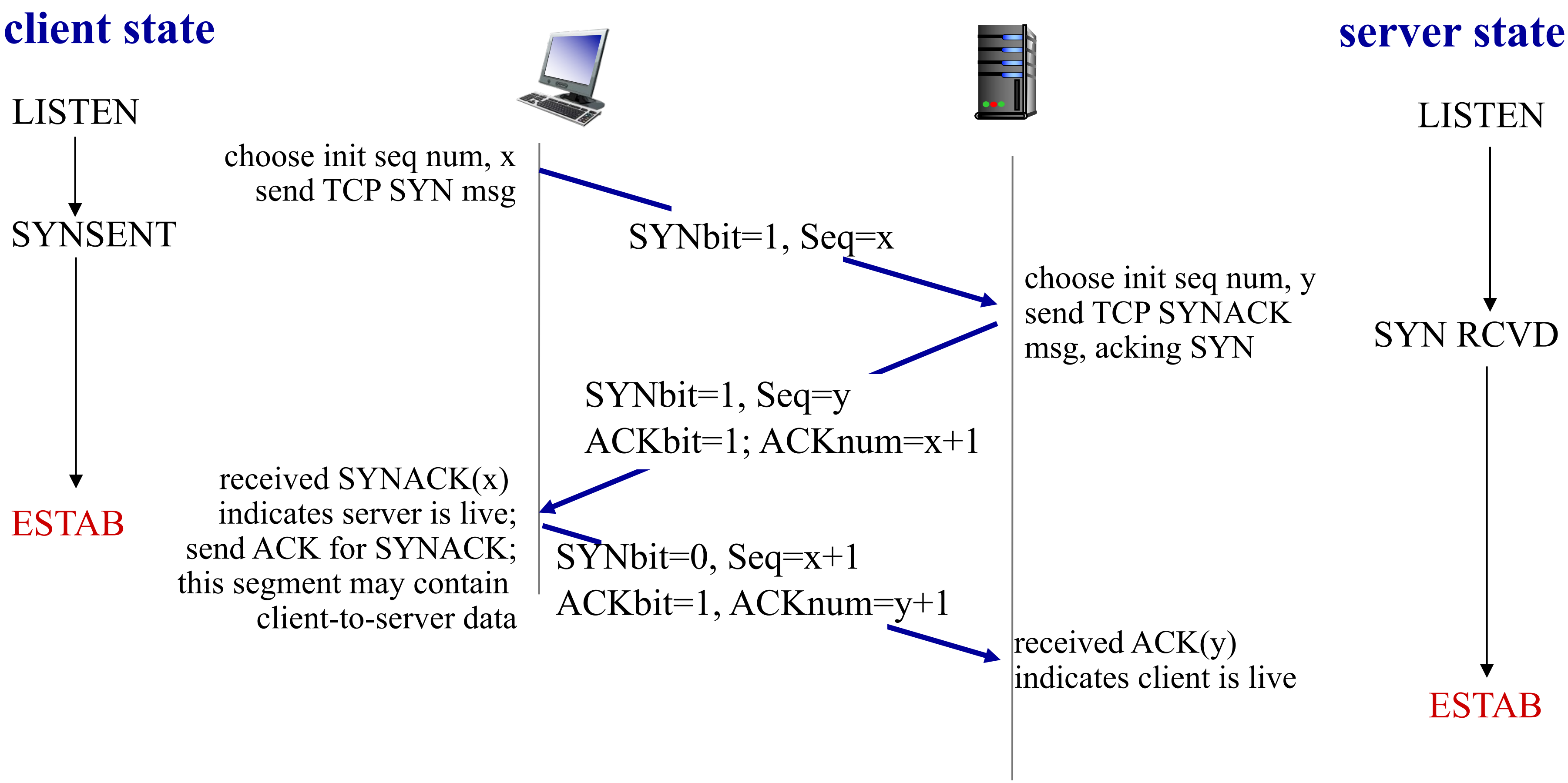
- T1 : B knows A's transmitter and B's receiver is OK
- T2: A knows A's transceiver and B's transceiver is OK, B has no more information than T1
- T3: Both A and B know their transceiver are OK, they can start the communication!

TCP segment structure



RST, SYN, FIN:
connection estab
(setup and teardown)

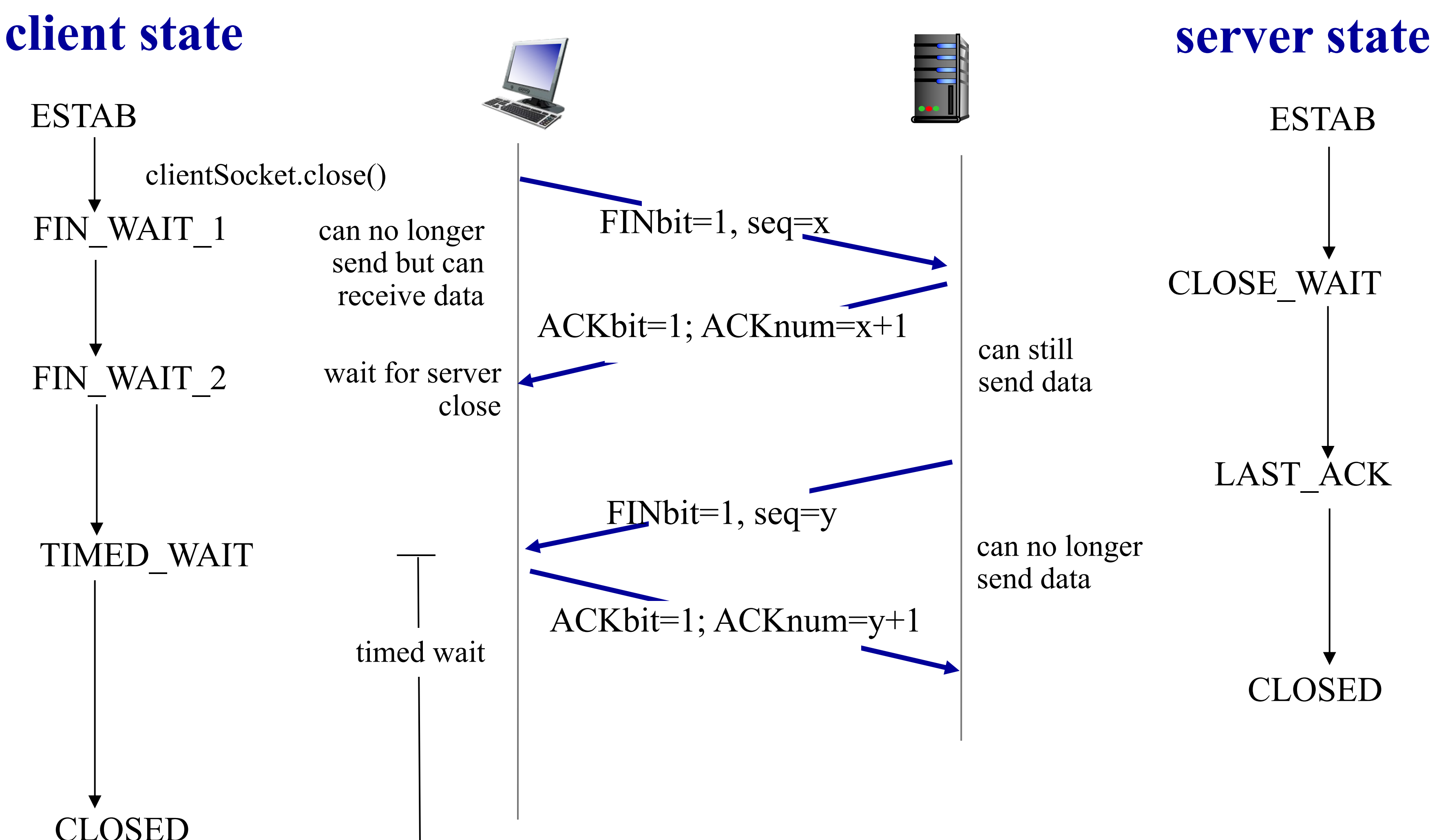
TCP 3-way handshake



Once these three steps have been completed, the client and server hosts can send segments containing data to each other.

- In each of these future segments, SYNbit=0

TCP: closing a connection

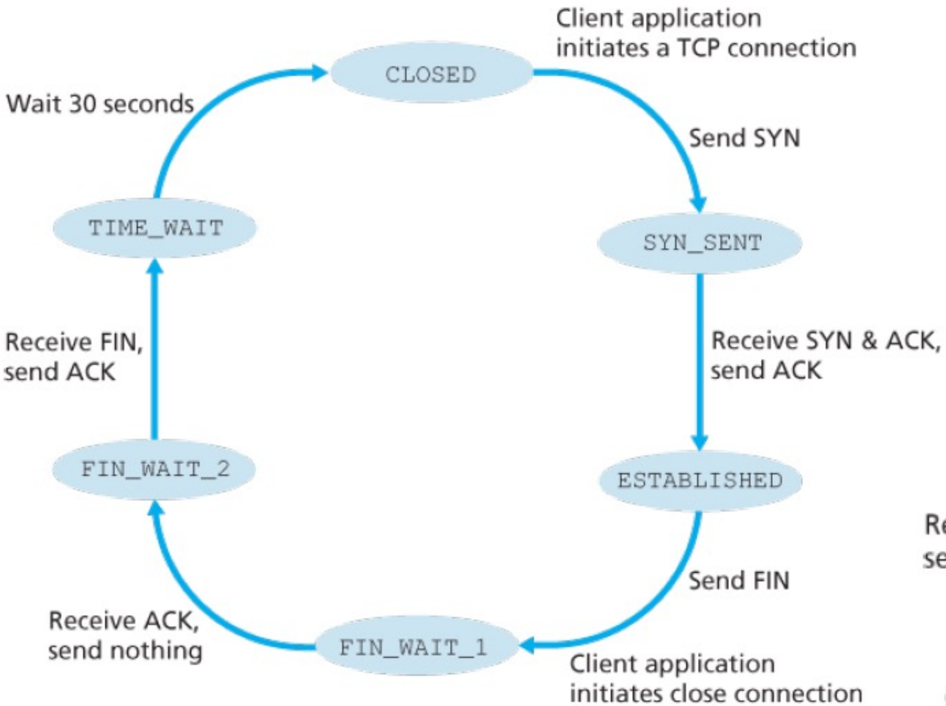


The TIME_WAIT state lets the TCP client resend the final acknowledgment in case the ACK is lost.

TCP: closing a connection

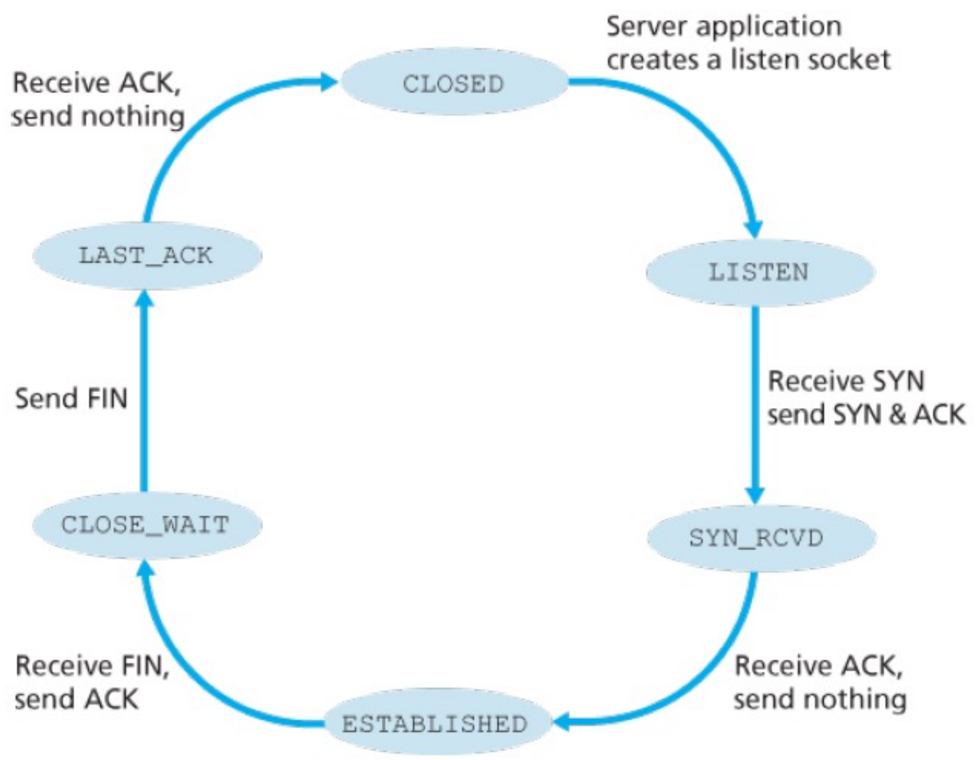
- Four-way handshaking
  - Either of the two processes participating in a TCP connection can end the connection.
- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
- Why FIN and ACK can not be sent in one msg as SYNACK in connection establishment?
  - The other side may still have packets need to be sent. It can not send FIN until the transmission is finished.

TCP States



Client Side

Server Side



Reset Segment

When a host receives a TCP segment whose port numbers or source IP address do not match with any of the ongoing sockets.

- Then the host will send a special reset segment to the source. RST flag bit is set to 1.
- "I don't have a socket for that segment. Please do not resend the segment."

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

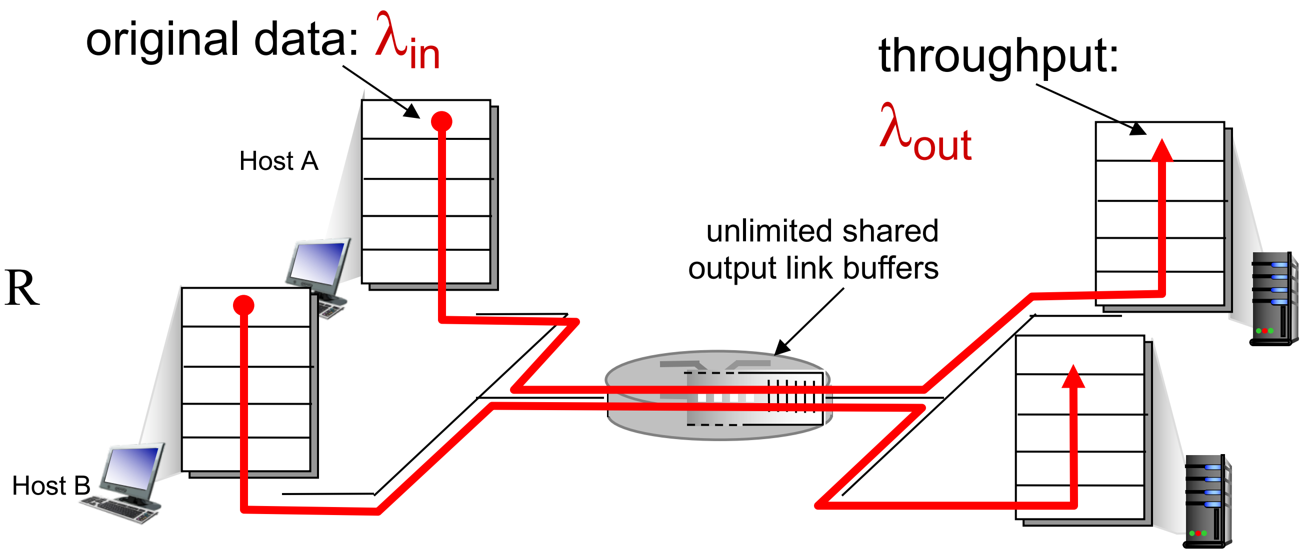
Principles of congestion control

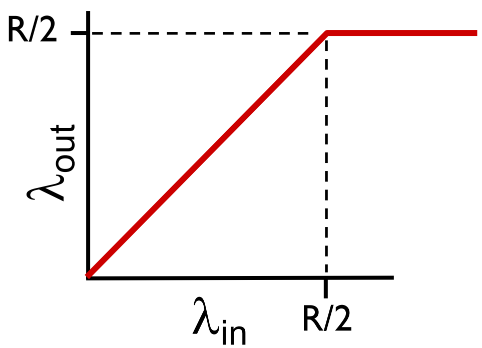
Congestion:

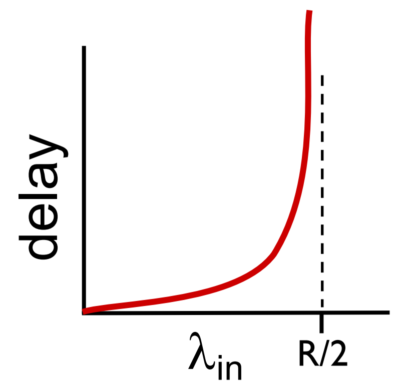
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

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





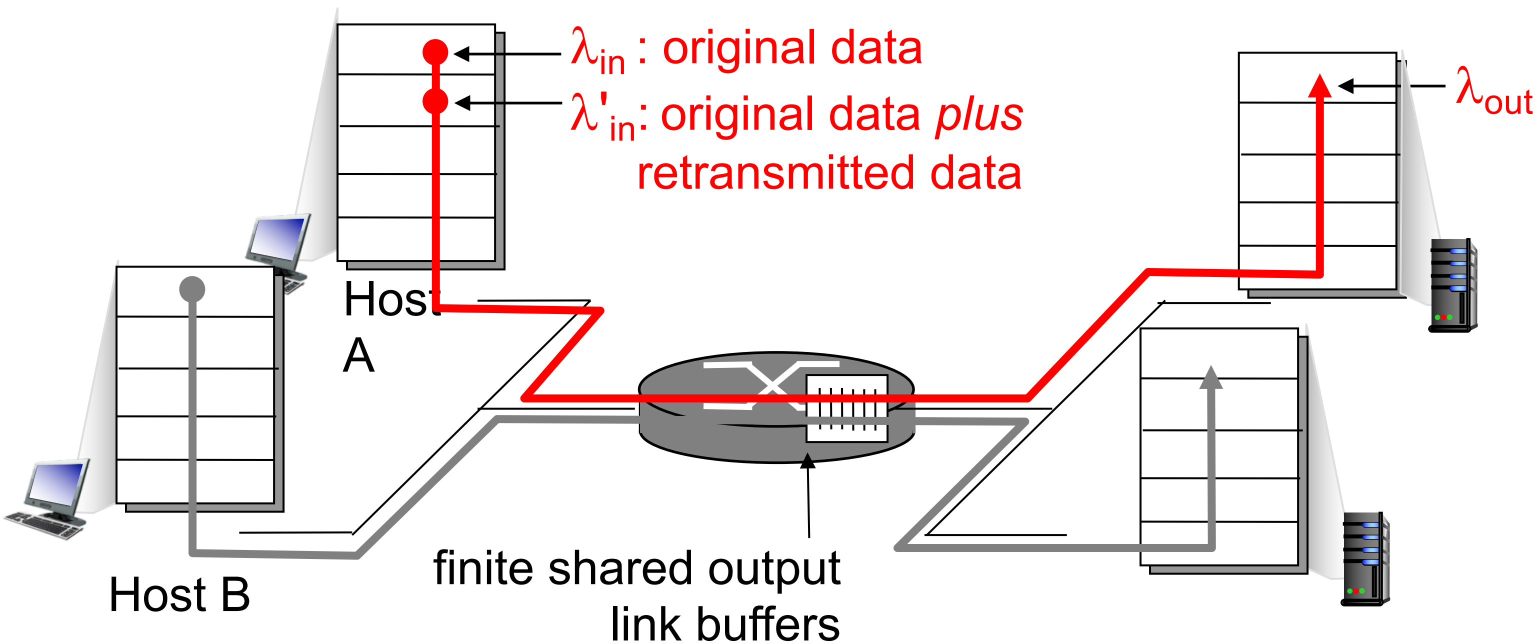


large queuing delays are experienced as the packet-arrival rate nears the link capacity.

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

Causes/costs of congestion: scenario 2

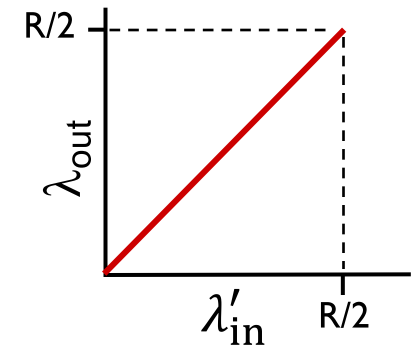
- one router, *finite* buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

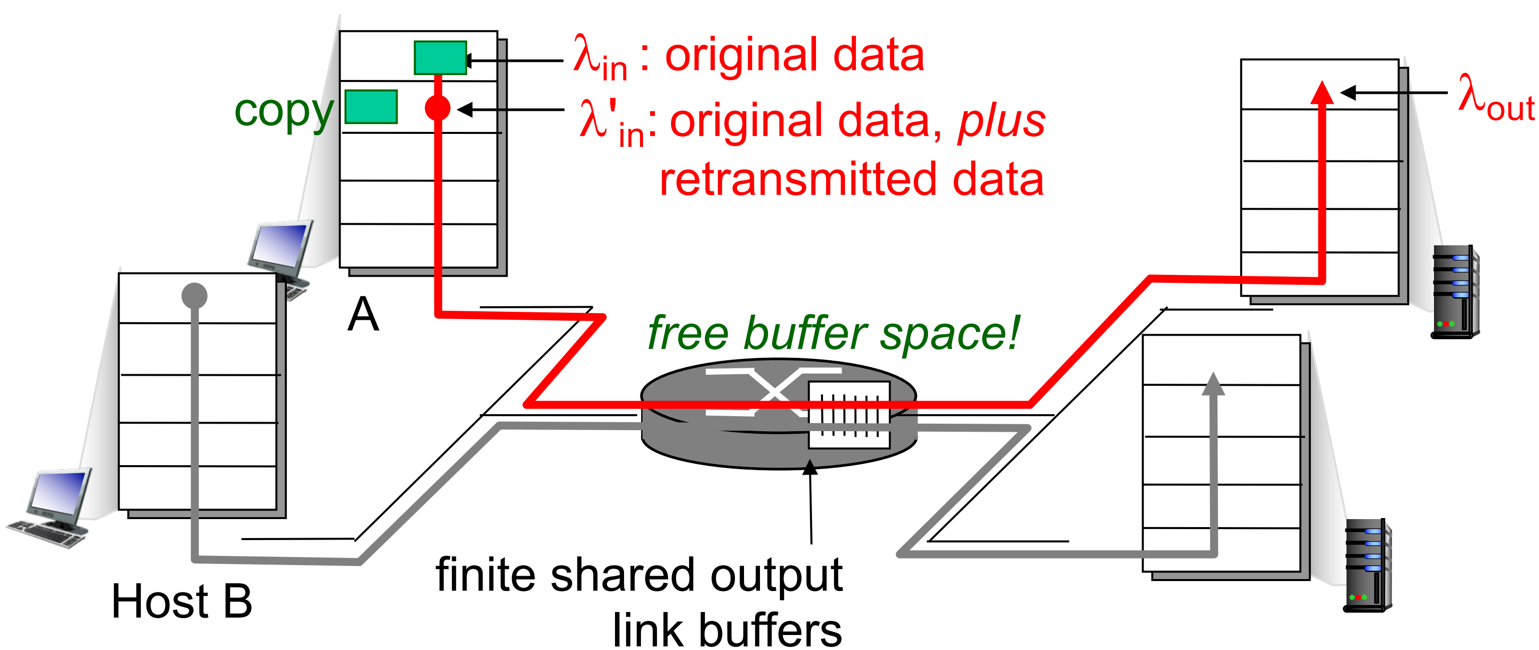


Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

- Sender sends only when router buffers available
- No loss occurs : $\lambda'_{in} = \lambda_{in}$

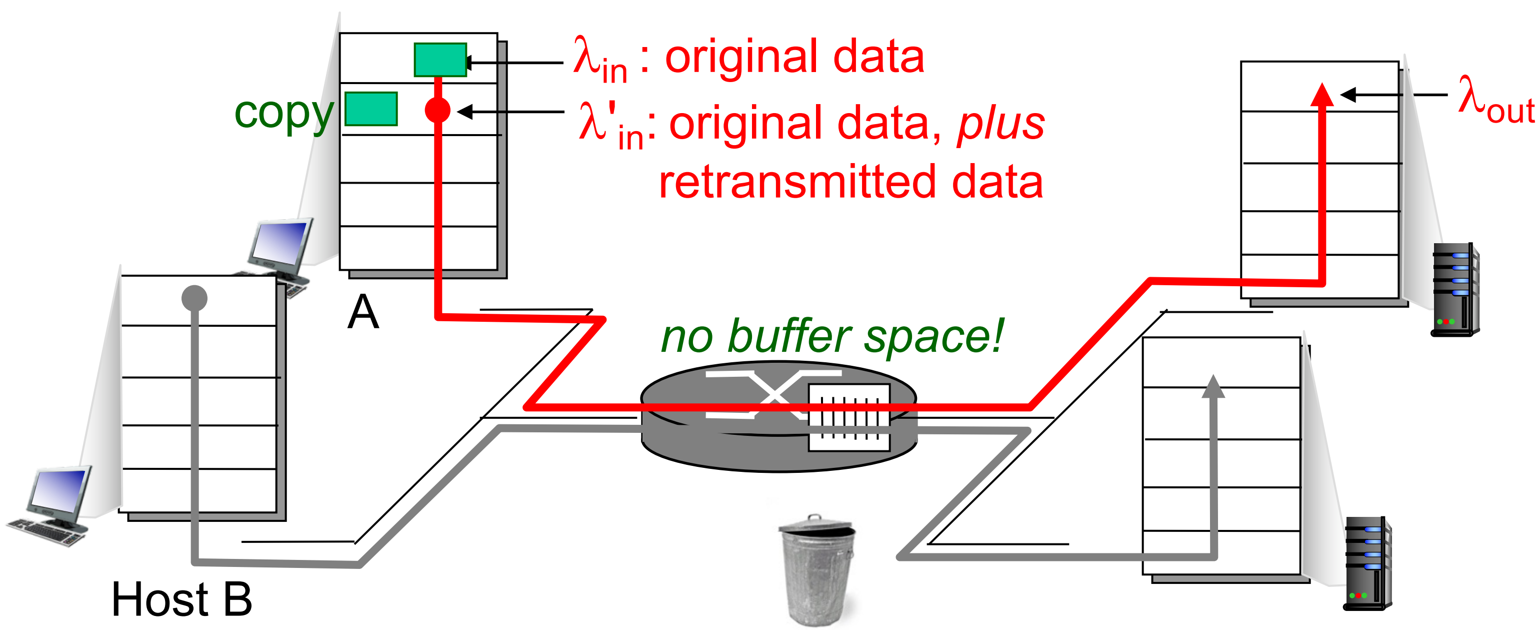




Causes/costs of congestion: scenario 2

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

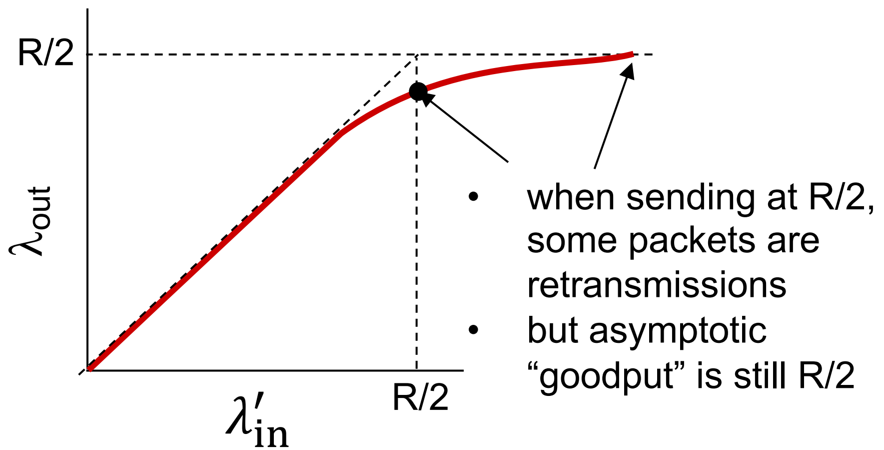
- sender only resends if packet *known* to be lost
- $\lambda'_{\text{in}} \geq \lambda_{\text{in}}$



Causes/costs of congestion: scenario 2

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

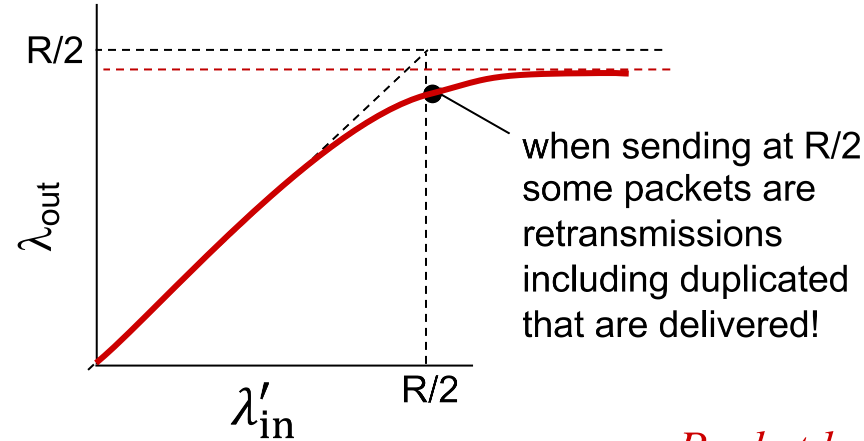
- sender only resends if packet *known* to be lost
- $\lambda'_{in} \geq \lambda_{in}$



Causes/costs of congestion: scenario 2

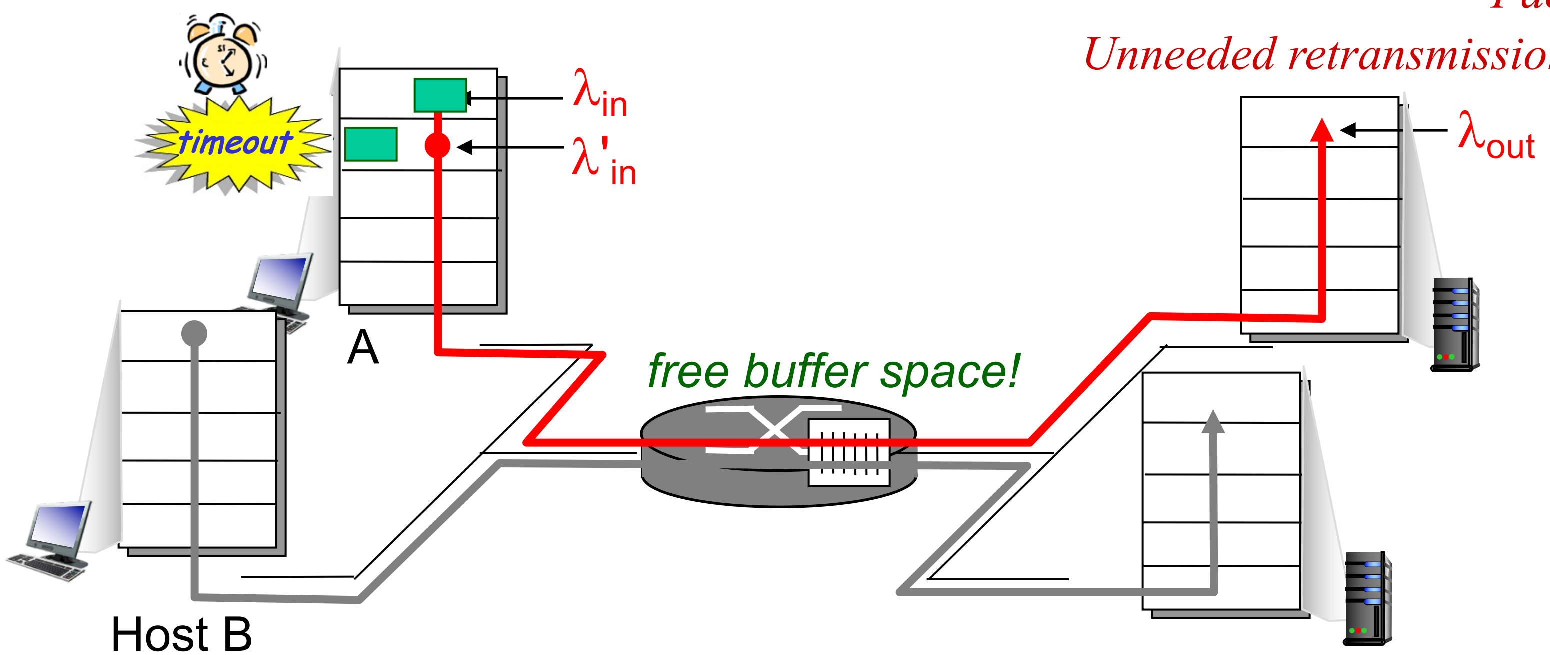
Realistic: duplicates

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



Packet loss

Unneeded retransmission: waste

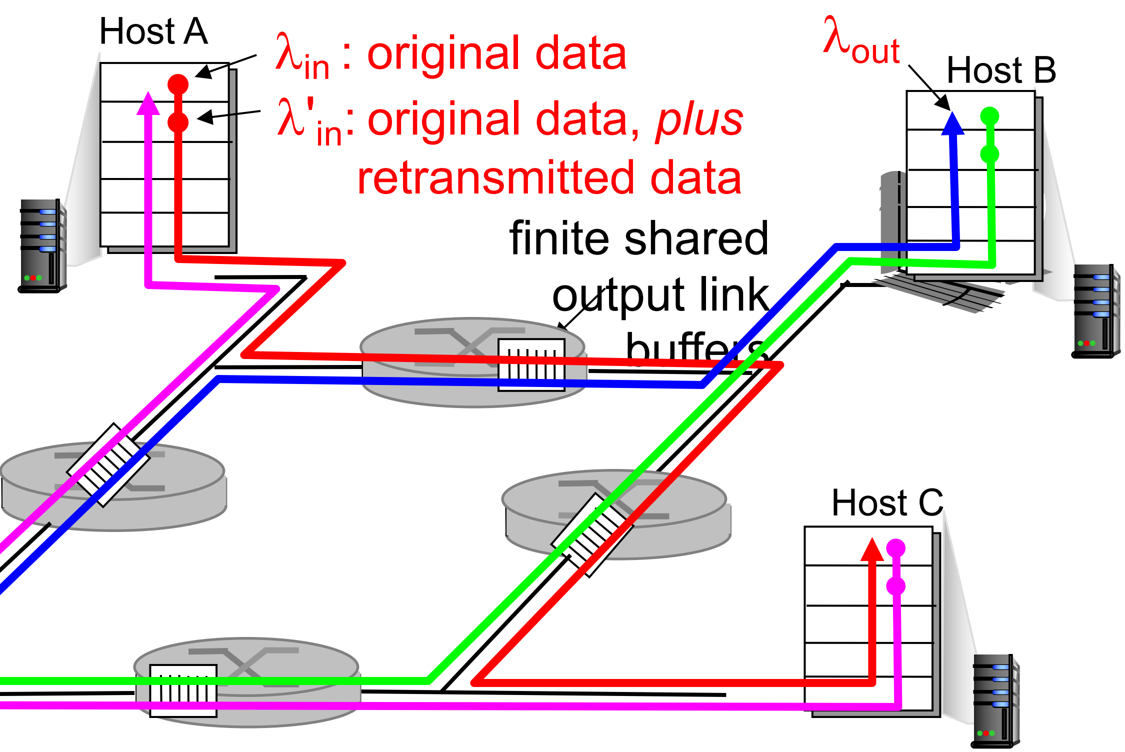


Causes/costs of congestion: scenario 3

For small values of λ_{in}:

- buffer overflows are rare
- the throughput λ_{out} approximately equals the offered load $\lambda'_{\text{in}} = \lambda_{\text{in}}$.

- four senders
- multihop paths
- timeout/retransmit

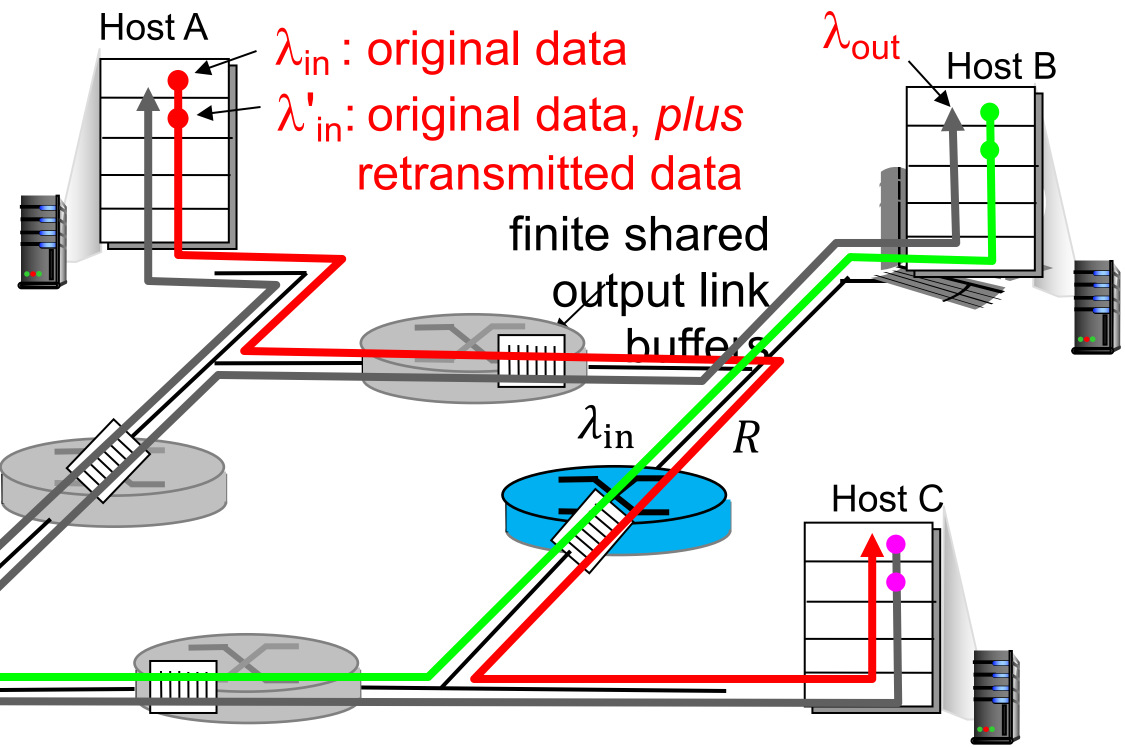


Causes/costs of congestion: scenario 3

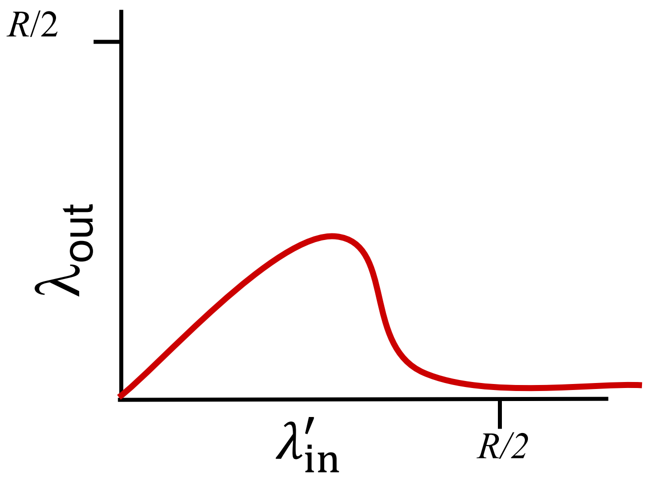
Q: what happens as λ_{in} increases ?

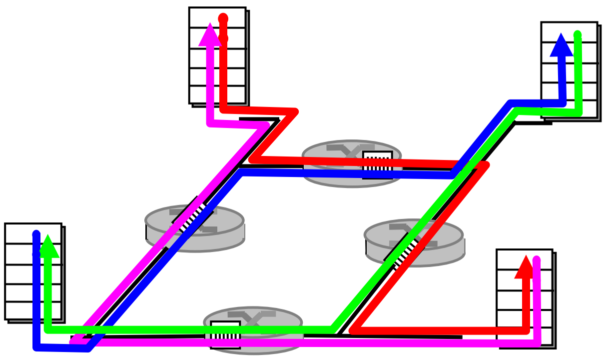
A: as green λ_{in} increases, all arriving red pkts at upper queue are dropped, red throughput goes 0

- four senders
- multihop paths
- timeout/retransmit



Causes/costs of congestion: scenario 3





another "cost" of congestion:

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

Cause and Cost of Congestion

Cause

- Shared link; limited link capacity
- Sending at a high rate

Cost of Congestion

- Delay
- Packet lost and retransmission
- Unneeded retransmission: waste
- "upstream" transmission capacity was wasted

Approaches to Congestion Control

End-to-end congestion control:

- TCP segment loss or round-trip segment delay
- TCP decreases its window size accordingly

Network-assisted congestion control:

- routers provide feedback to the sender and/or receiver
- a single bit indicating congestion at a link; the maximum host sending rate the router can support

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

- TCP use end-to-end congestion control
- have each sender limit the rate at which it sends traffic into its connection as a function of perceived network congestion

Questions for achieving congestion control:

Q1: How does a TCP sender limit the rate at which it sends traffic into its connection?

Q2: How does a TCP sender perceive that there is congestion on the path between itself and the destination?

Q3: What algorithm should the sender use to change its send rate as a function of perceived end-to-end congestion?

TCP congestion control: additive increase multiplicative decrease

Questions for achieving congestion control:

Q1: How does a TCP sender limit the rate at which it sends traffic into its connection?

Congestion window: **cwnd**

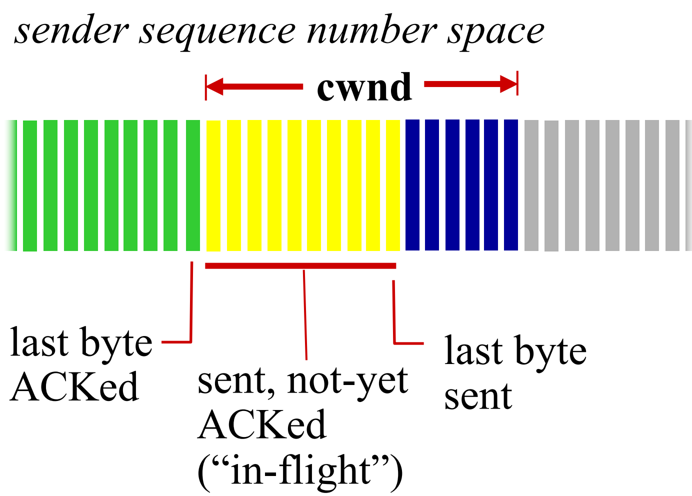
LastByteSent – LastByteAcked ≤ min{cwnd, rwnd}

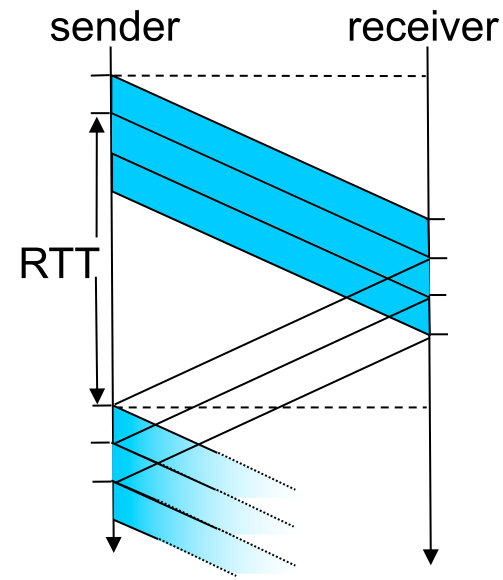
Q2: How does a TCP sender perceive that there is congestion on the path between itself and the destination?

Timeout; three duplicate ACKs

Q3: What algorithm should the sender use to change its send rate as a function of perceived end-to-end congestion?

Congestion window





- sender limits transmission:

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

- **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 congestion control: additive increase multiplicative decrease

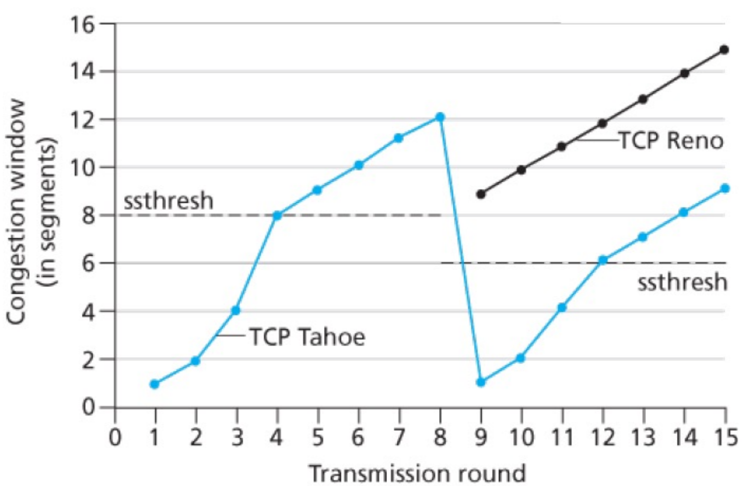
Q3: What algorithm should the sender use to change its send rate as a function of perceived end-to-end congestion?

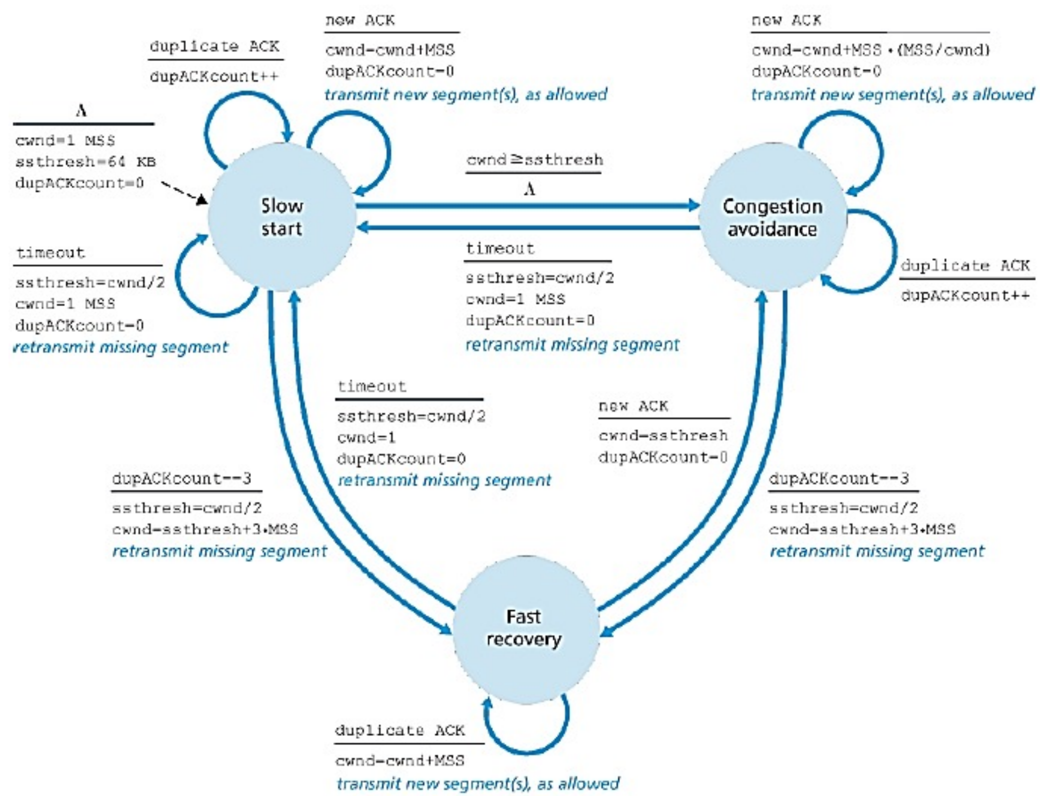
- A lost segment → congestion → decrease rate
- An acknowledged segment → the network is fine → increase rate
- Bandwidth probing: network condition may change

TCP Congestion Control: details

The congestion control algorithm has three major components:

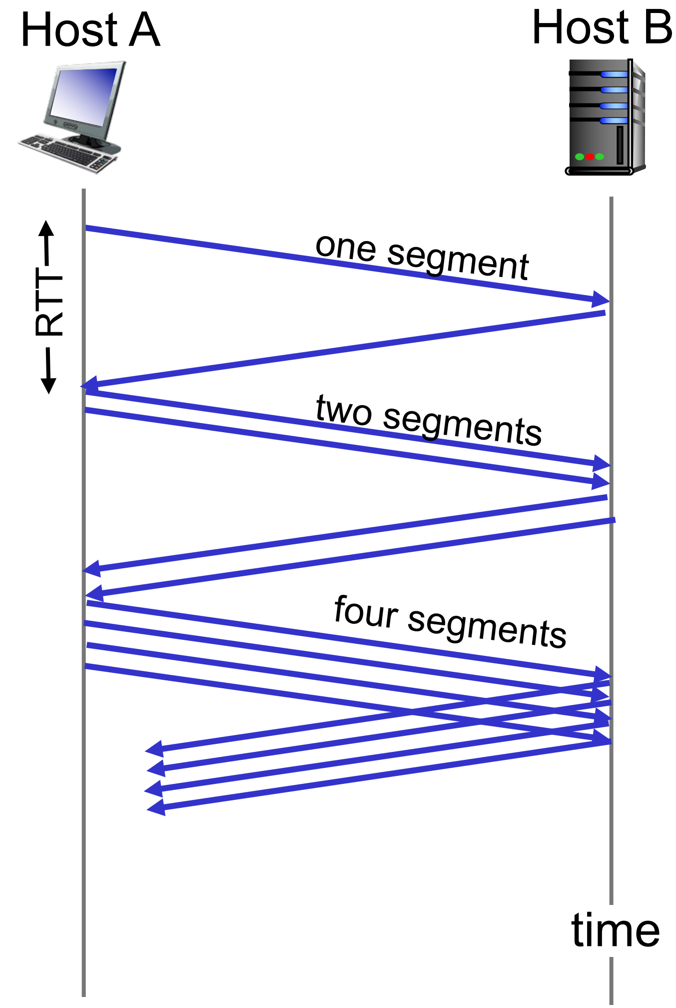
- Slow start : exponentially increase
- Congestion avoidance: linearly increase
- Fast recovery:



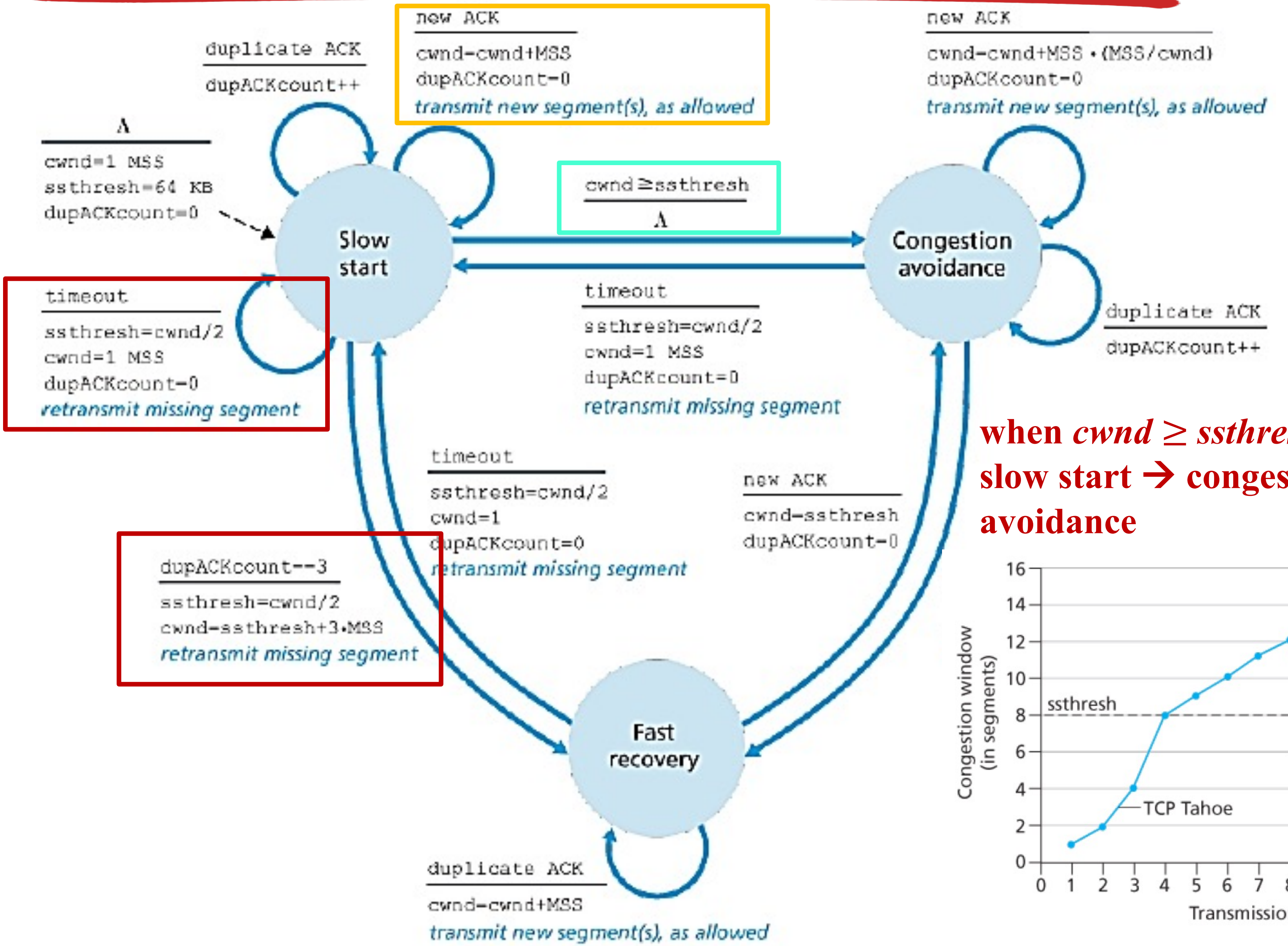


TCP Slow Start

- when connection begins or timeout occurs, increase rate exponentially until packet lost:
  - 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 Congestion Control: FSM



when *cwnd* ≥ *ssthresh*: slow start → congestion avoidance

TCP: detecting, reacting to loss

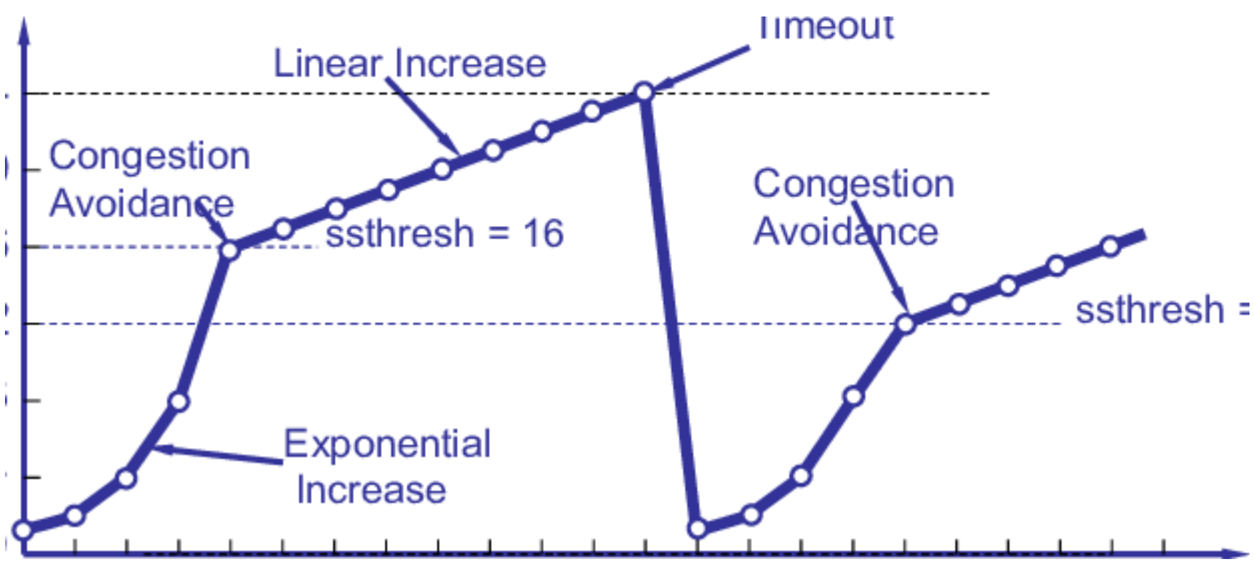
- loss indicated by timeout:
  - **cwnd** set to 1 MSS; **ssthresh** = **cwnd**/2
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
- loss indicated by 3 duplicate ACKs:
  - TCP Tahoe always sets **cwnd** to 1 (timeout or 3 duplicate acks) → Slow start
    - **cwnd** set to 1 MSS; **ssthresh** = **cwnd**/2
  - TCP RENO
    - dup ACKs indicate network capable of delivering some segments → Fast Recovery
    - ssthresh = cwnd / 2; cwnd = ssthresh + 3MSS

TCP: switching from slow start to Congestion Avoidance

Q: when should the exponential increase switch to linear?

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

Thus, when timeout occurs, ssthresh = cwnd/2

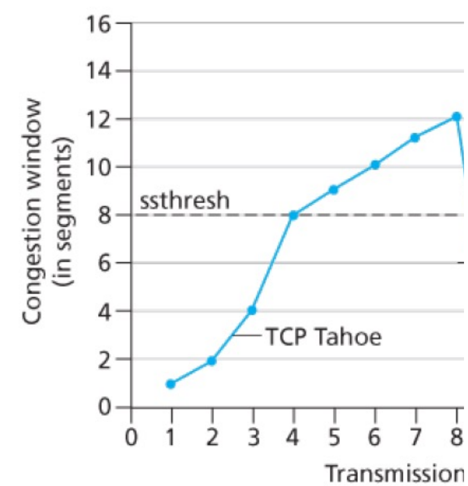


TCP Congestion Avoidance

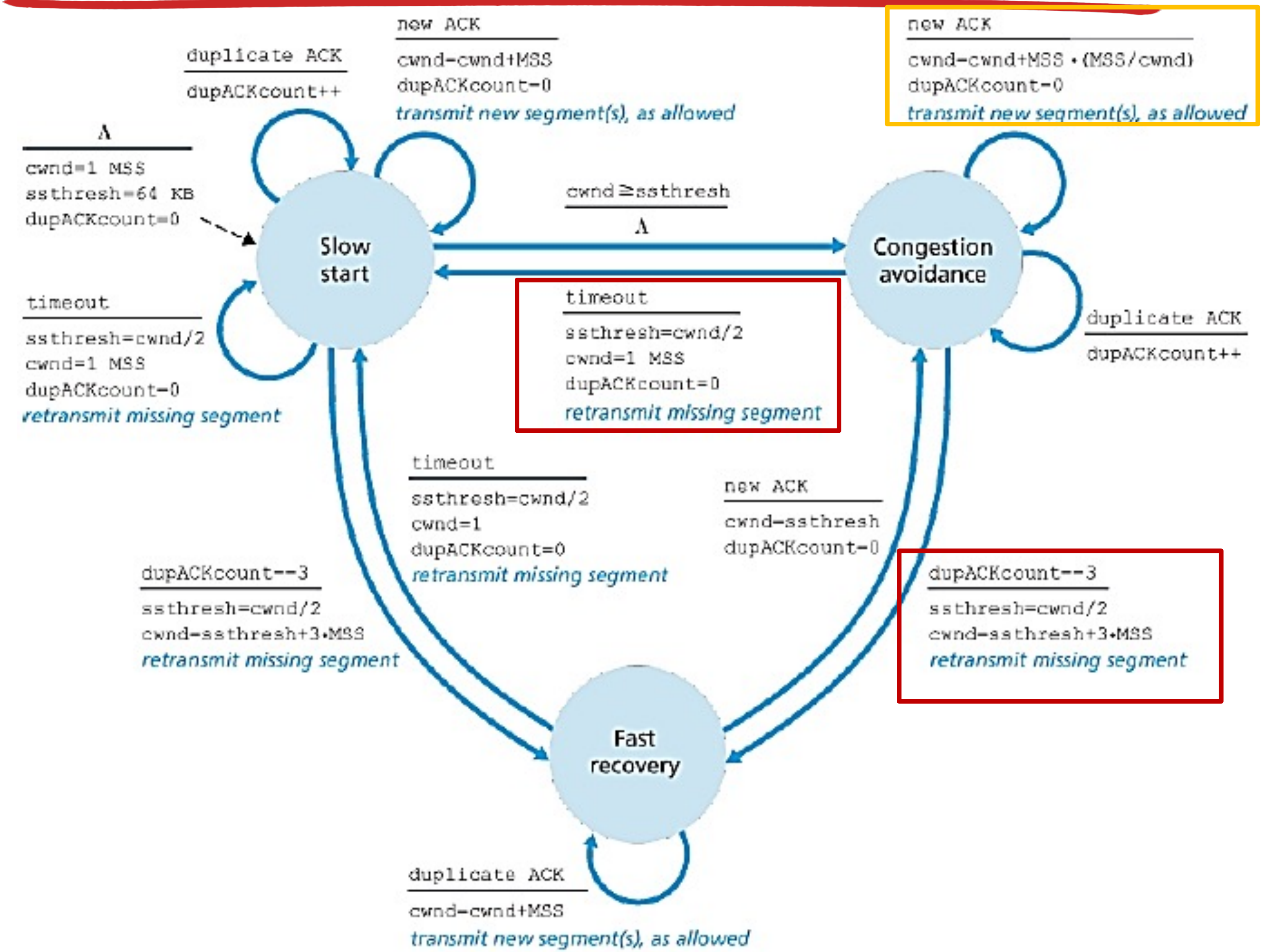
Trigger: $cwnd \geq ssthresh$

Increases *cwnd* linearly: by one MSS every RTT

- Increase *cwnd* by (MSS/*cwnd*)MSS bytes whenever a new acknowledgment arrives.
- E.g., if MSS is 1,460 bytes and *cwnd* is 14,600 bytes, then 10 segments are being sent within an RTT.
  - Each arriving ACK (assuming one ACK per segment) increases the congestion window size by 1/10 MSS,



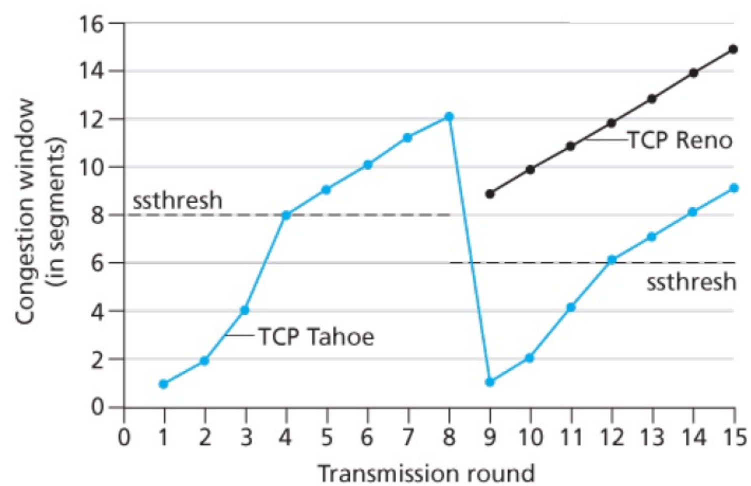
TCP Congestion Control: FSM



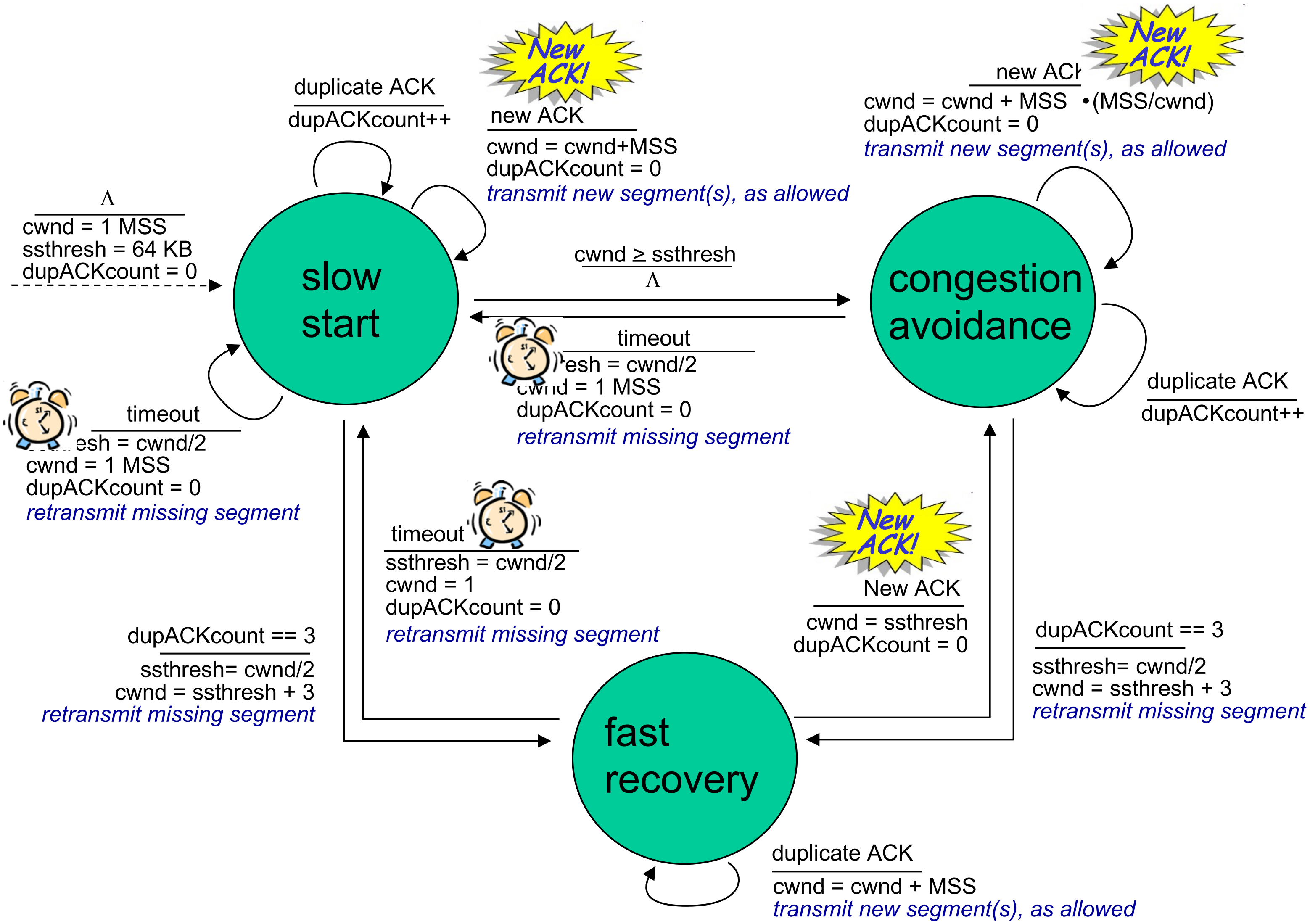
TCP Fast Recovery

- Trigger（RENO）: triple duplicate ACKs
- ***ssthresh = cwnd / 2; cwnd = ssthresh + 3MSS***

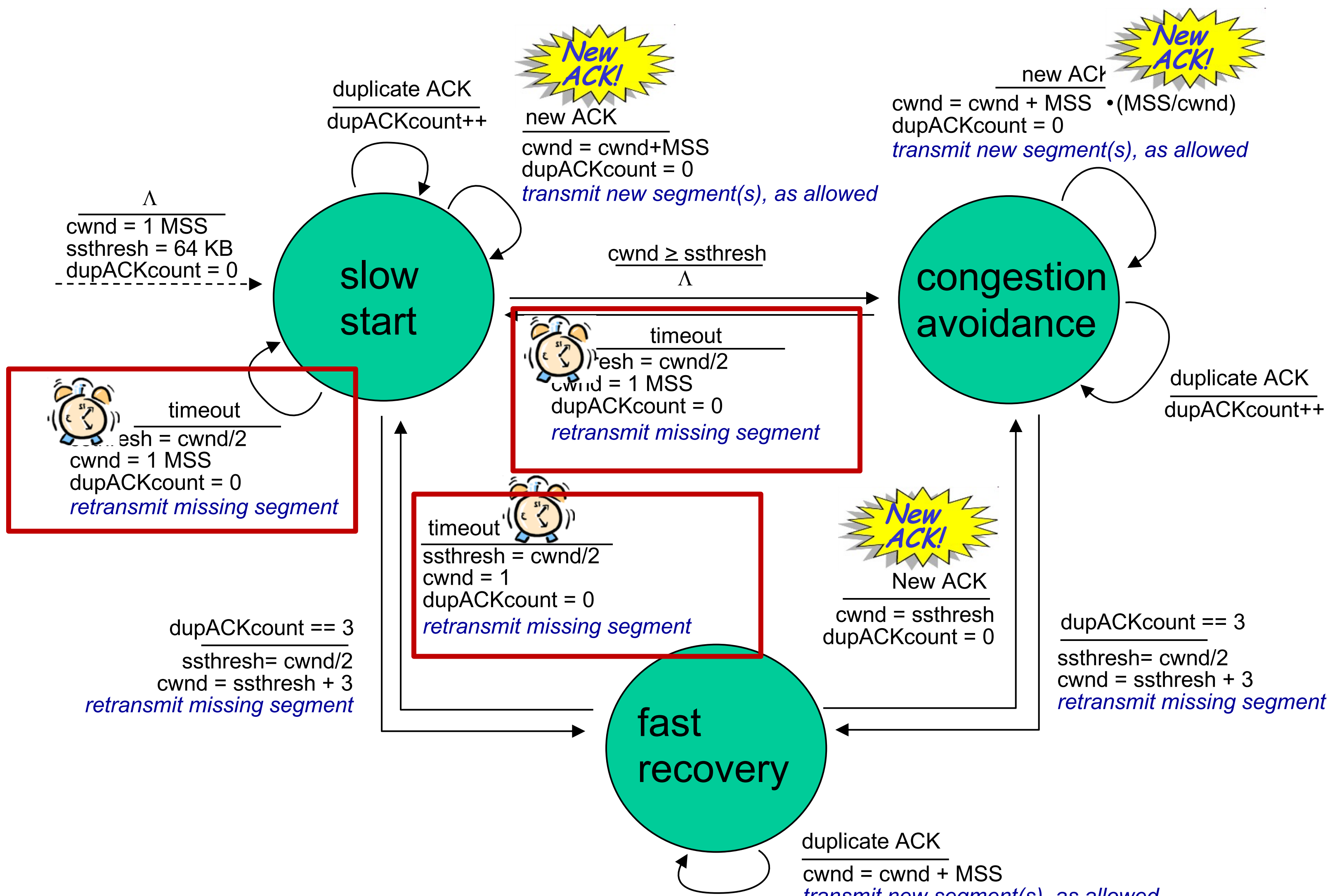
- The value of *cwnd* is increased by 1 MSS for every duplicate ACK received for the missing segment that caused TCP to enter the fast-recovery state
- when an ACK arrives for the missing segment, enter congestion avoidance



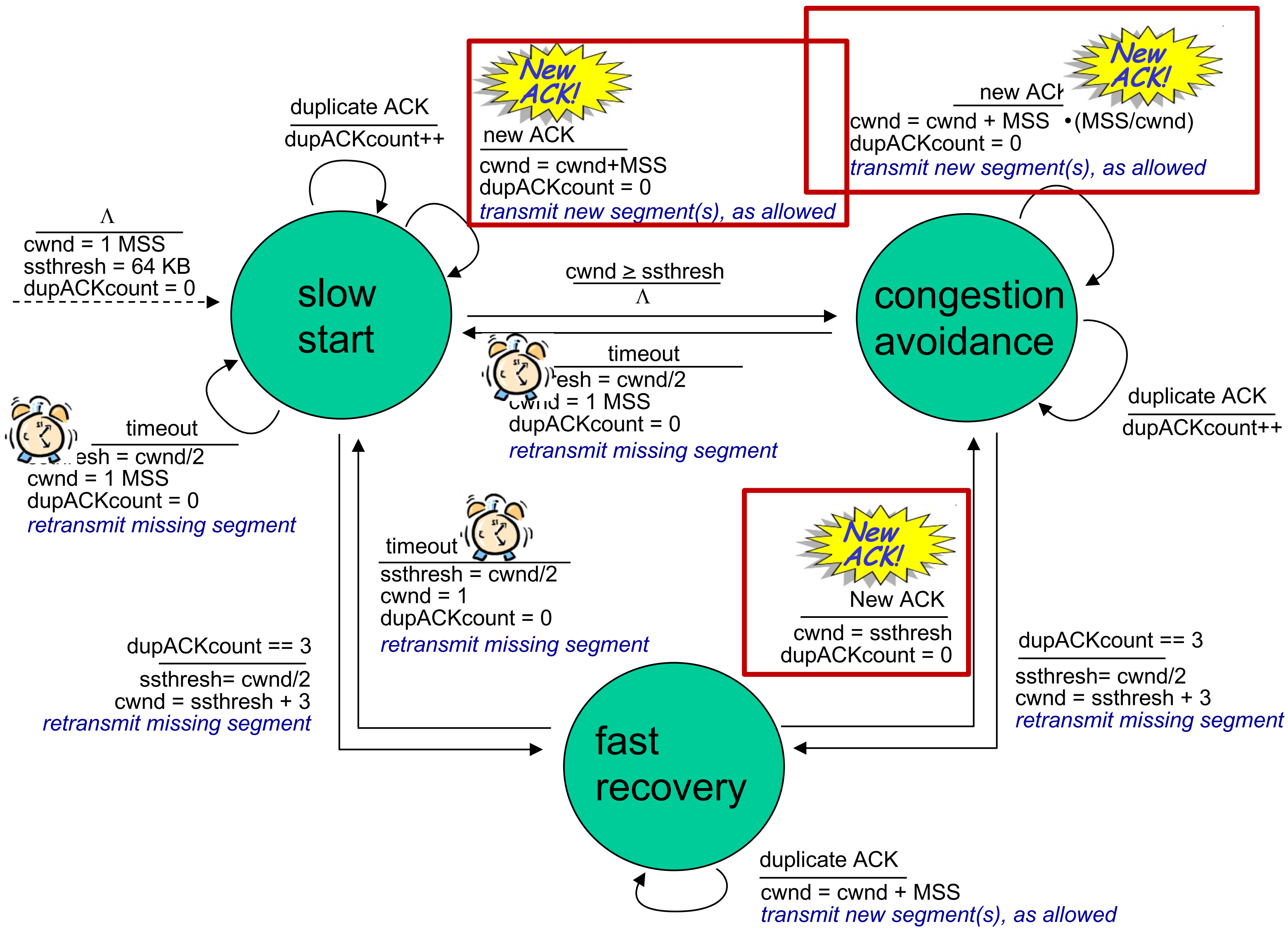
Summary: TCP Congestion Control



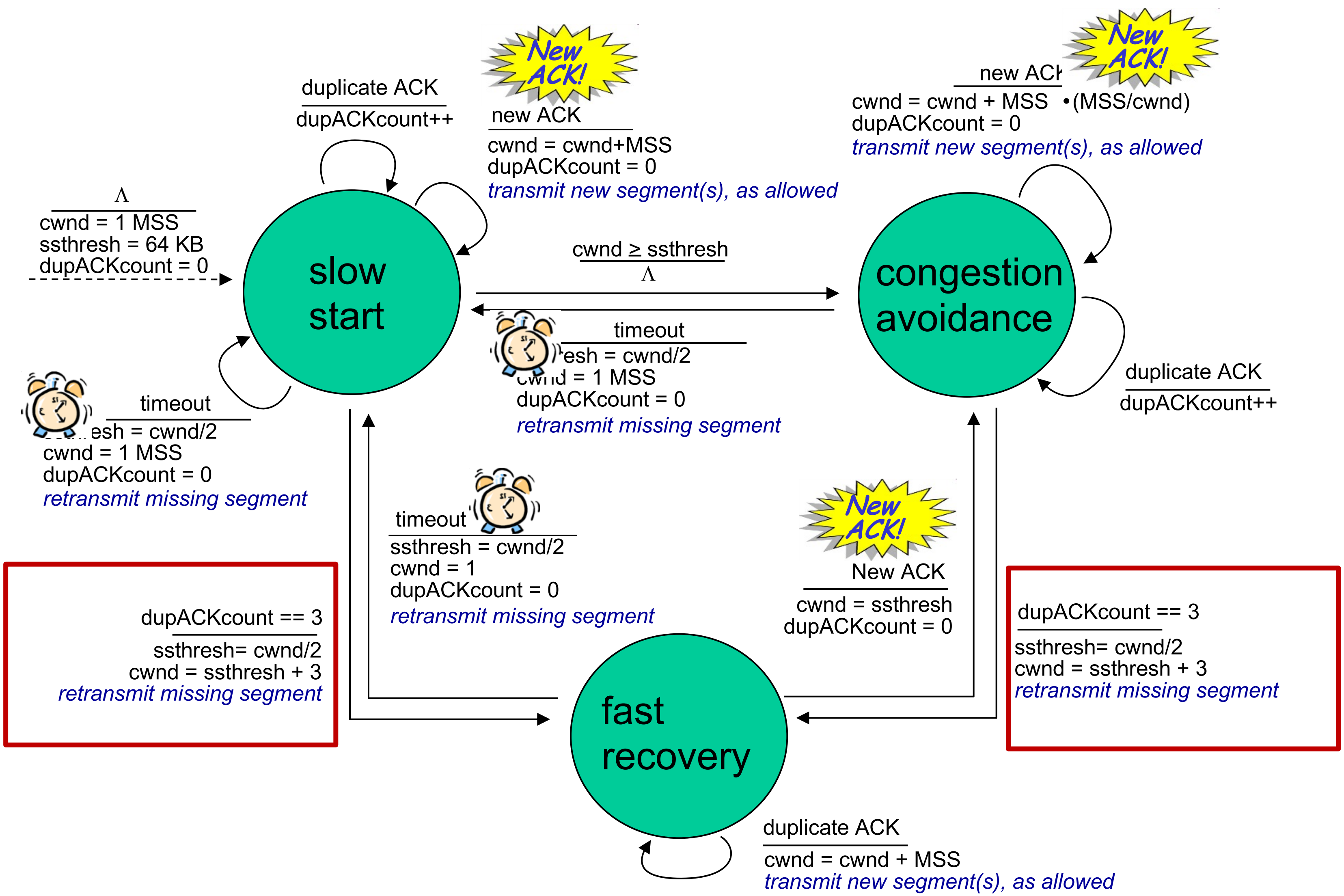
Summary: TCP Congestion Control



Summary: TCP Congestion Control



Summary: TCP Congestion Control

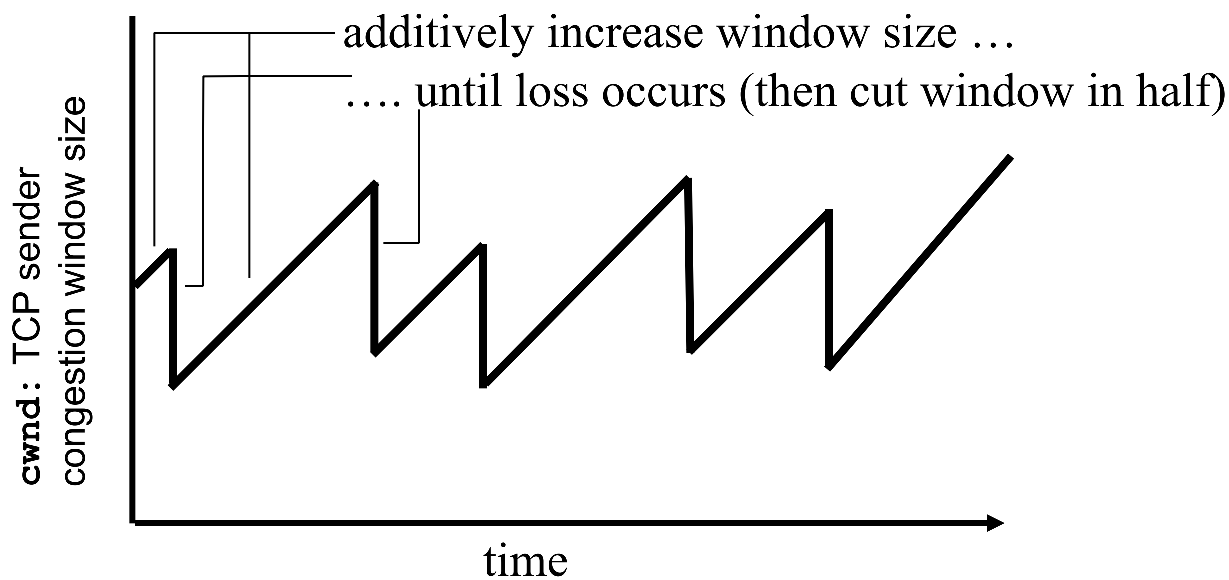


TCP congestion control: additive increase multiplicative decrease

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

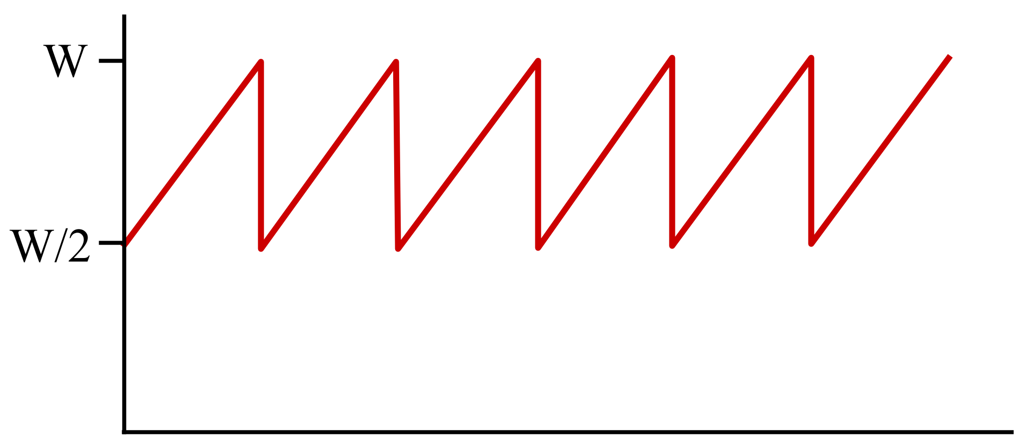
AIMD: probing for bandwidth



TCP throughput

- Avg. TCP throughput 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

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



TCP Futures: TCP over "long, fat pipes"

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

- 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]:

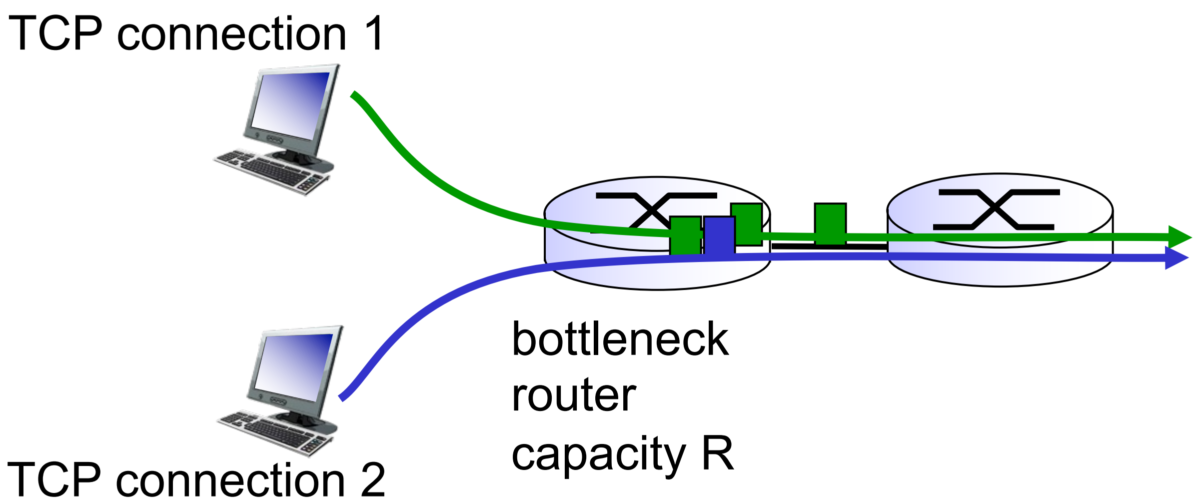
$$\text{TCP throughput} = \frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

➜ 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

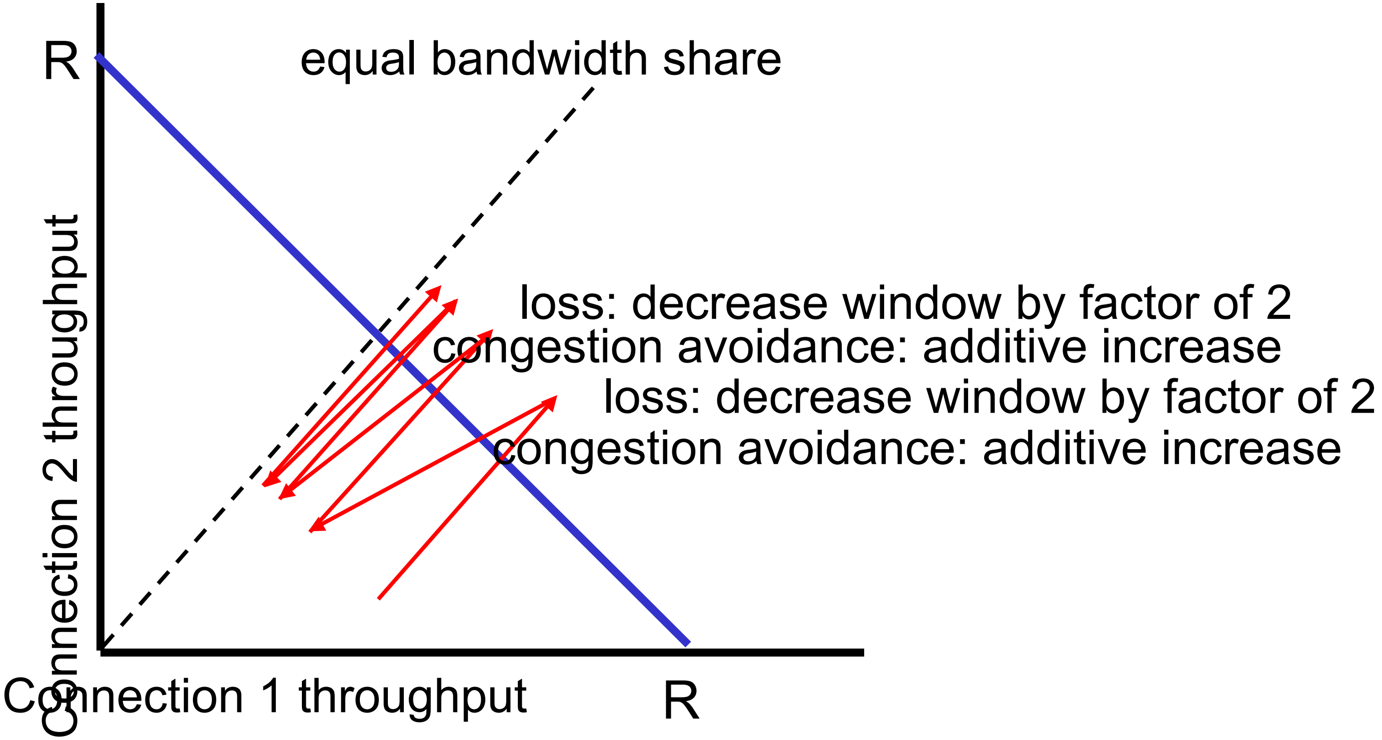
Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*



Why is TCP fair?

two competing sessions:

- additive increase gives slope of 1, as throughout 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
- use UDP:
  - send audio/video at constant rate, tolerate packet loss
- UDP sources to crowd out TCP traffic

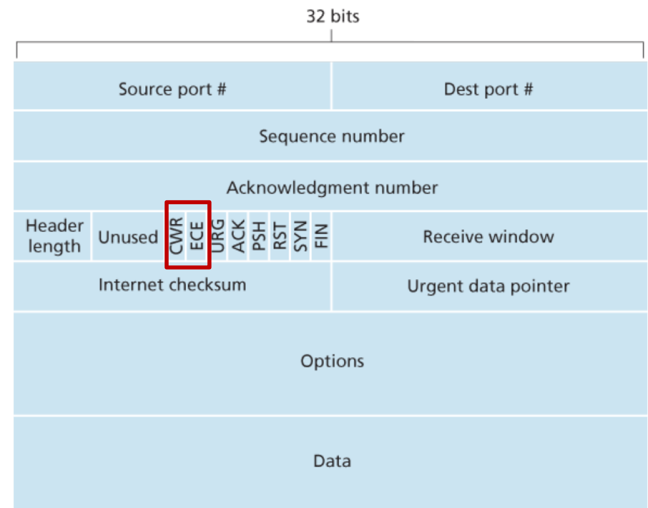
Fairness, parallel TCP connections

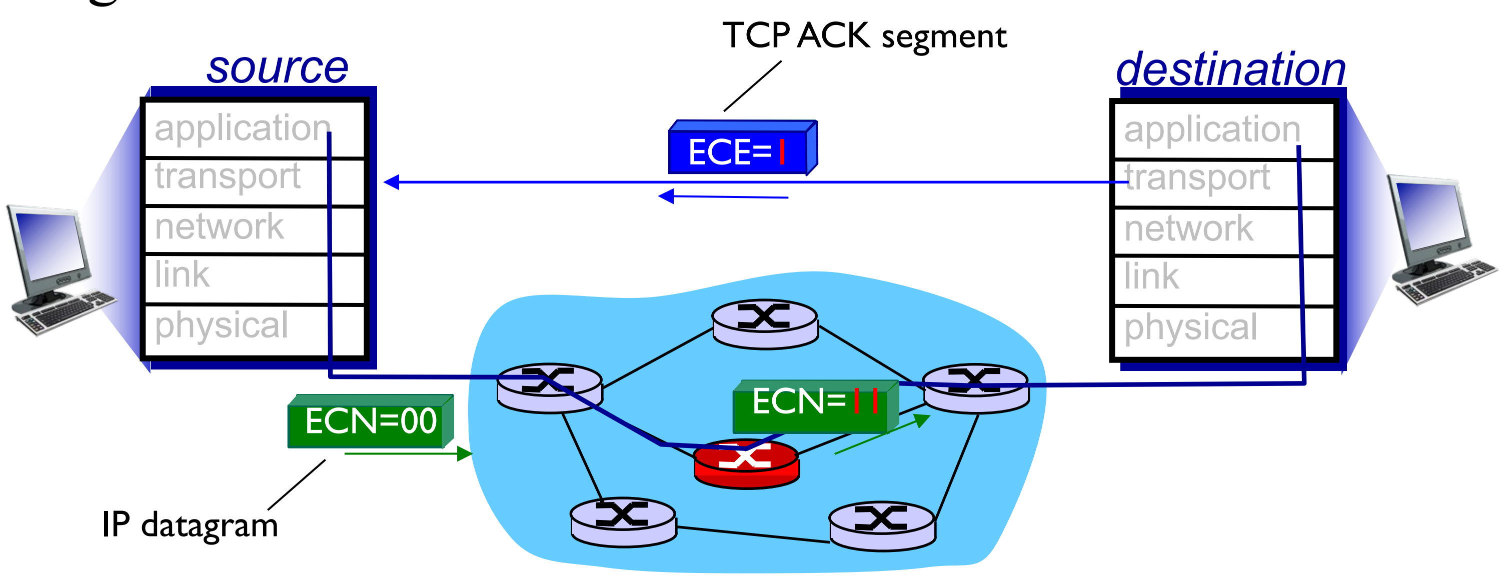
- application can open multiple parallel connections between two hosts
- web browsers do this
- e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets more than R/2

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 sets ECE bit on receiver-to-sender ACK segment to notify sender of congestion





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"