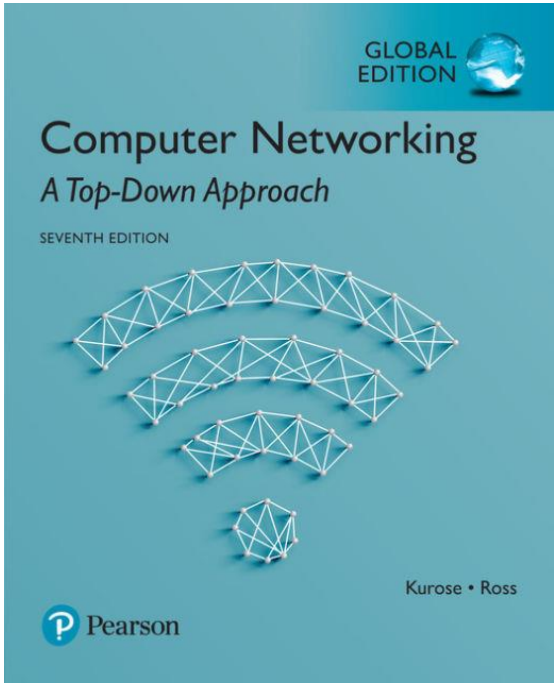
Transport Layer



Computer Networking: A Top Down Approach
7th edition
Jim Kurose, Keith Ross
Pearson, 2017

Intended Learning Outcomes

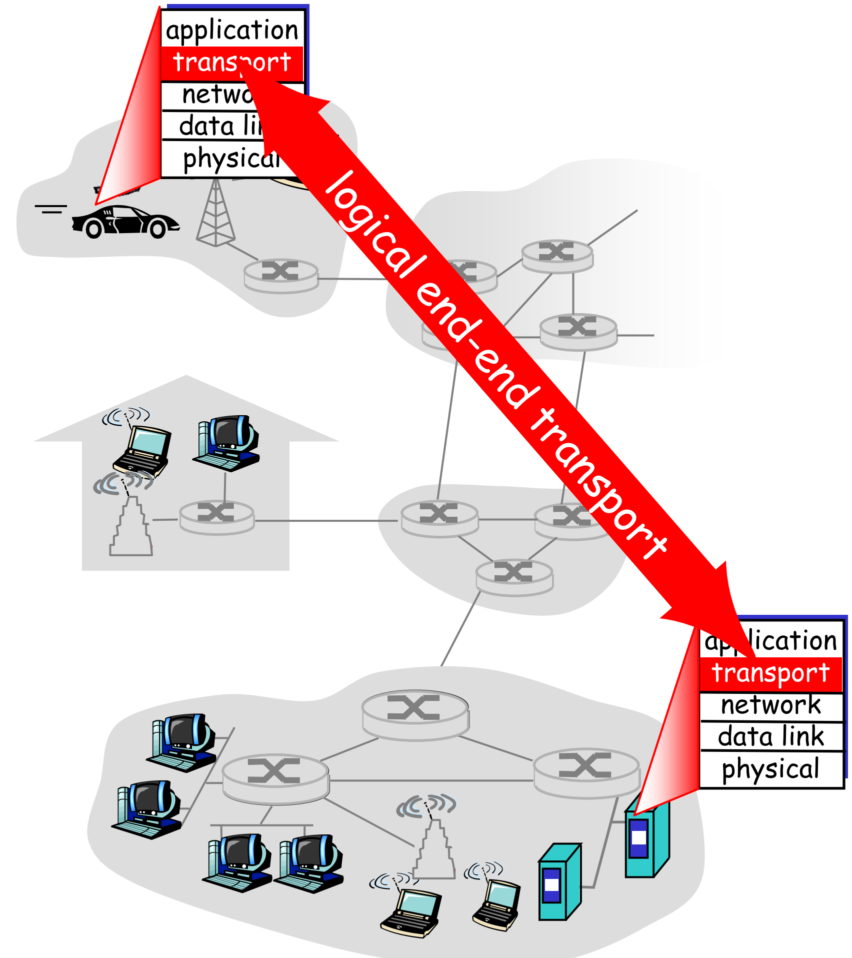
- understand principles behind transport layer services:
  - reliable data transfer
  - flow control
  - congestion control
- understand transport layer protocols in the Internet:
  - TCP: connection-oriented transport, connection management
  - TCP flow control
  - TCP congestion control

Transport Layer

- Transport-layer services
- Connectionless transport: UDP
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- 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



Transport Layer

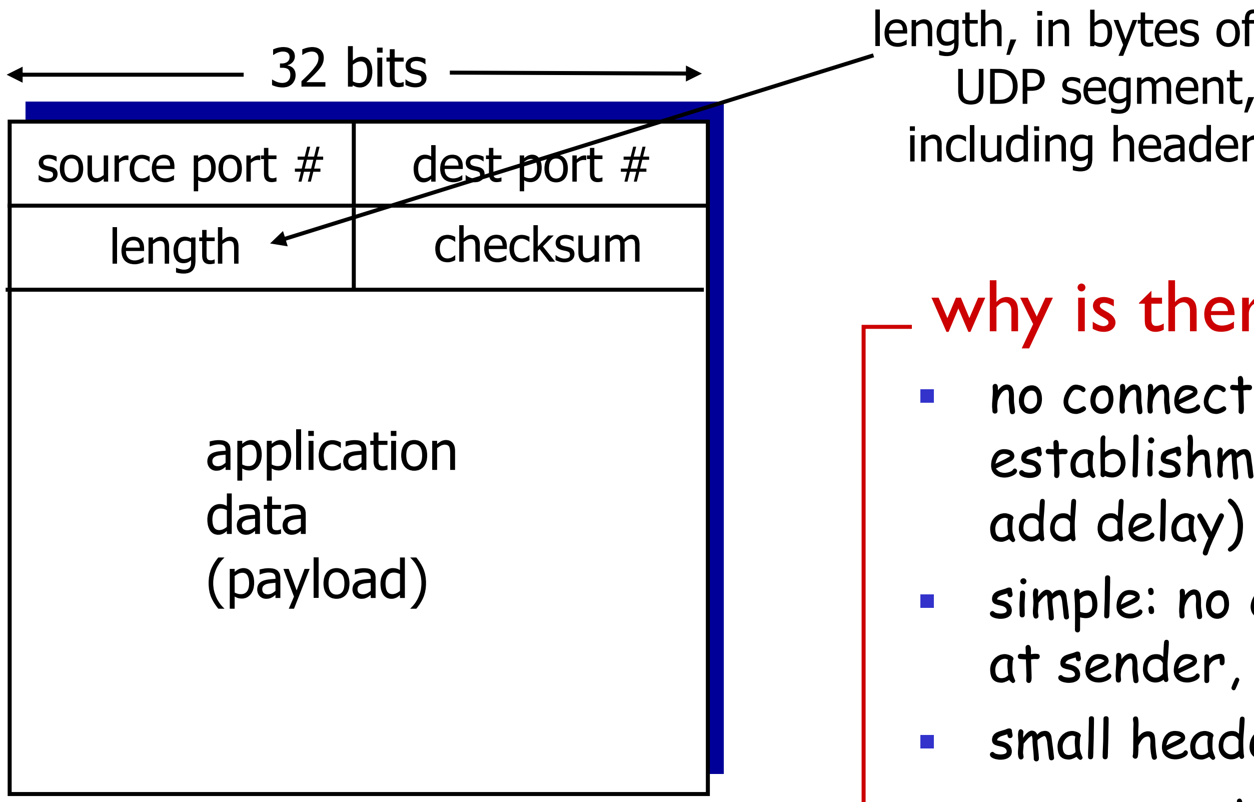
- Transport-layer services
- Connectionless transport: UDP
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

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



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 send out data as fast as desired

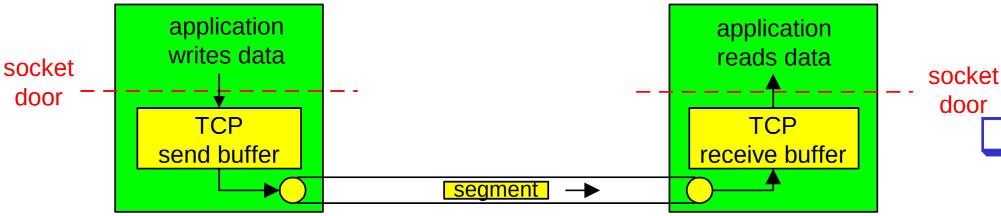
Transport Layer

- Transport-layer services
- Connectionless transport: UDP
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

TCP: Review

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
- *send & receive buffers*

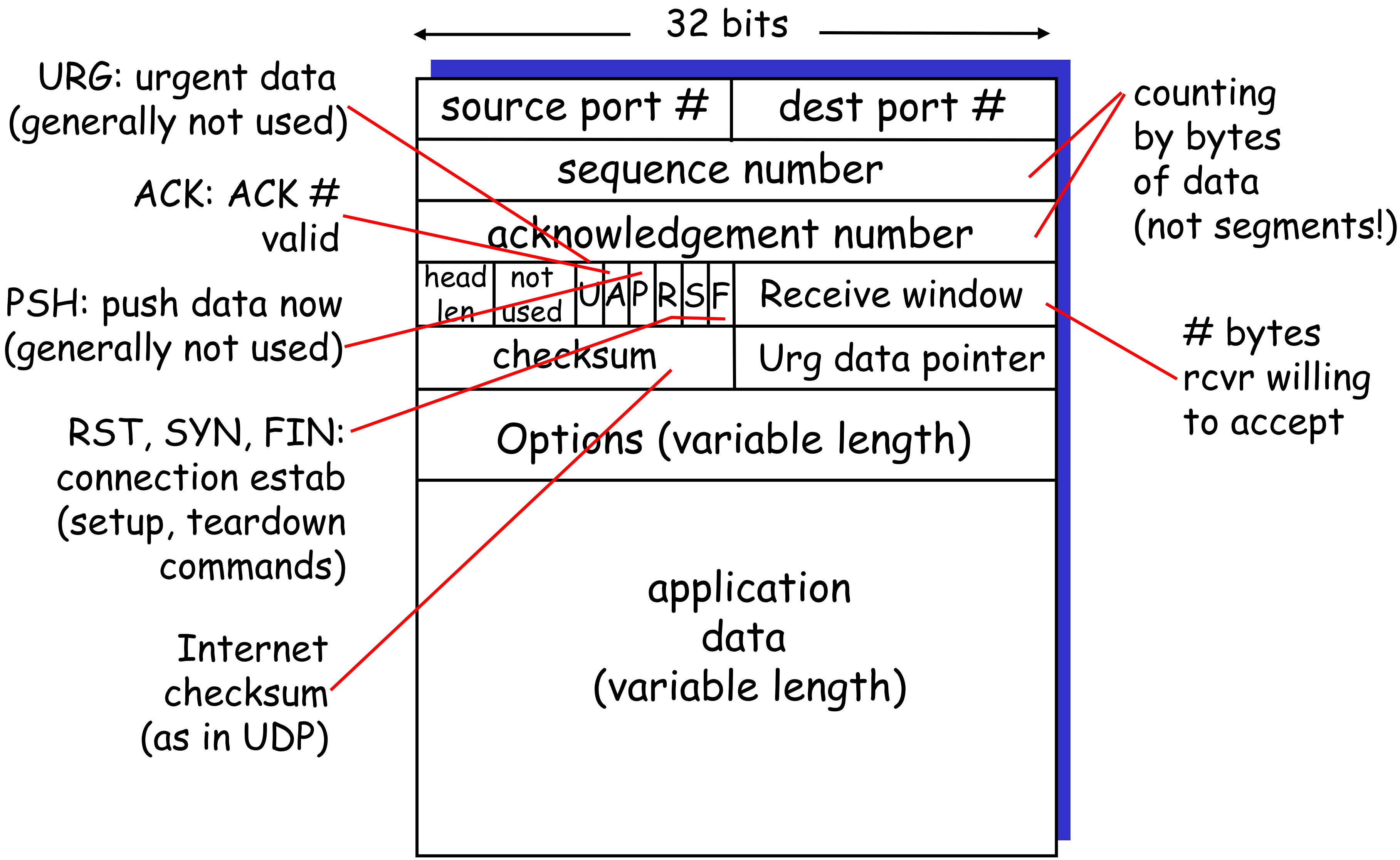


- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender & receiver states before data exchange
- flow controlled:
  - sender will not overload receiver
- congestion controlled:
  - sender will not overload network

Transport Layer

- Transport-layer services
- Connectionless transport: UDP
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

TCP segment structure



Transport Layer

- Transport-layer services
- Connectionless transport: UDP
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

TCP seq. numbers, ACKs

sequence number:

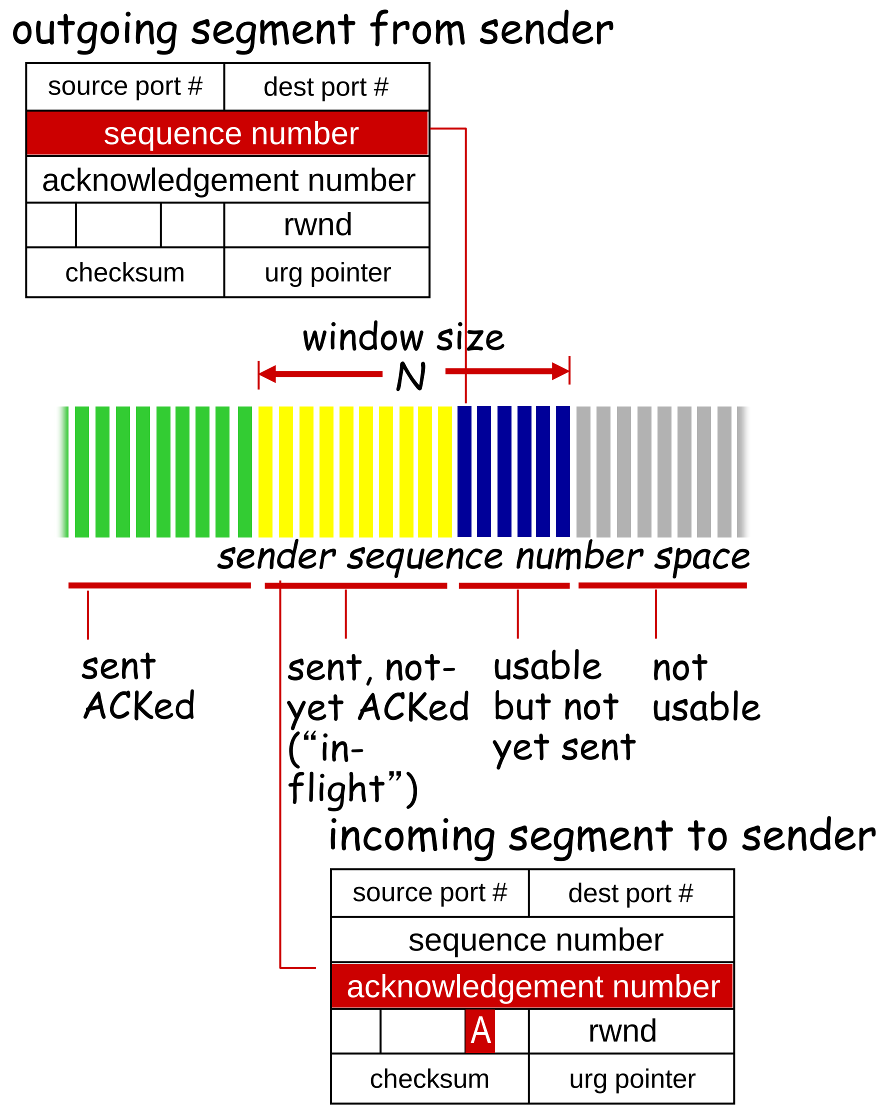
- byte stream "number" of first byte in segment's data

acknowledgement number:

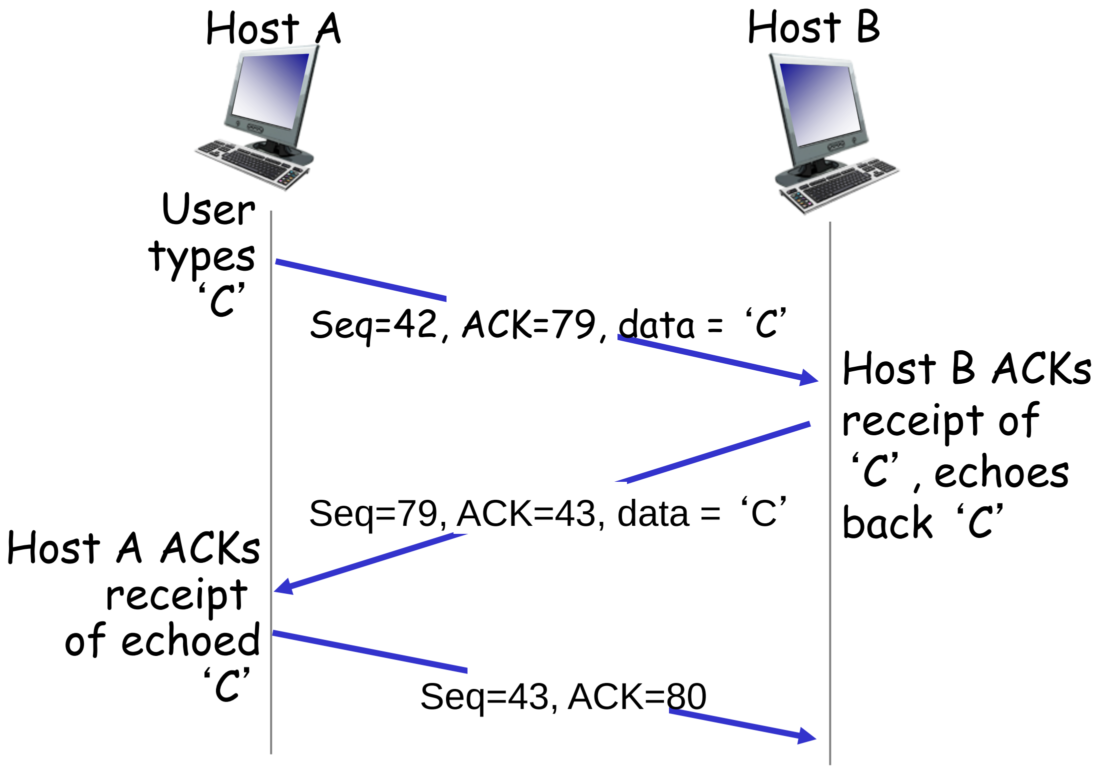
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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



TCP seq. numbers, ACK numbers



simple telnet scenario

TCP reliable data transfer (rdt)

- 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)

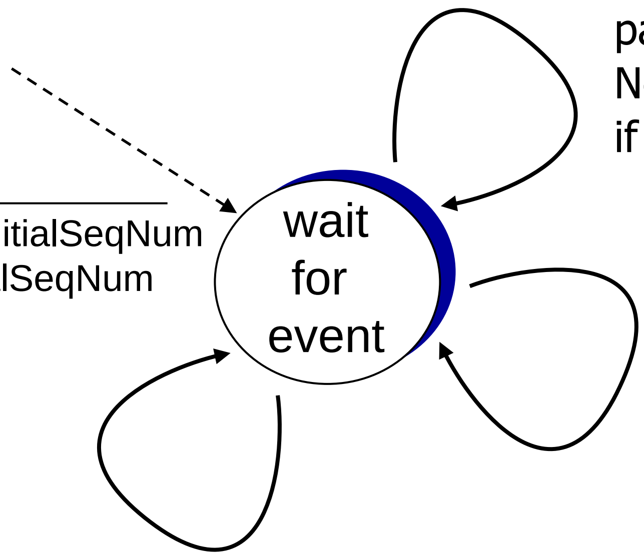
data received from application above

---

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., "send")
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer

Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum



timeout

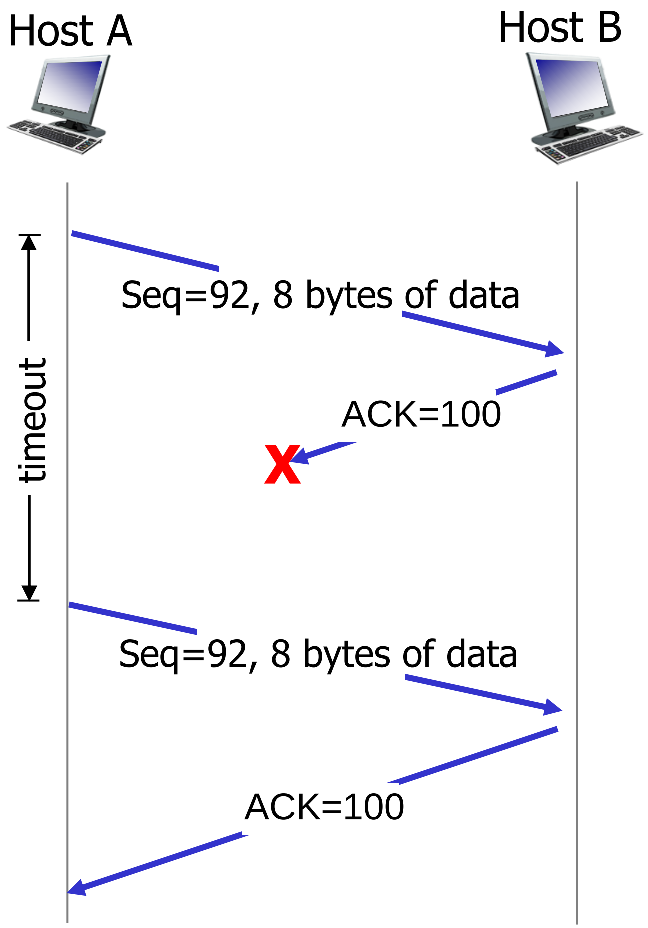
---

retransmit not-yet-acked segment
with smallest seq. #
start timer

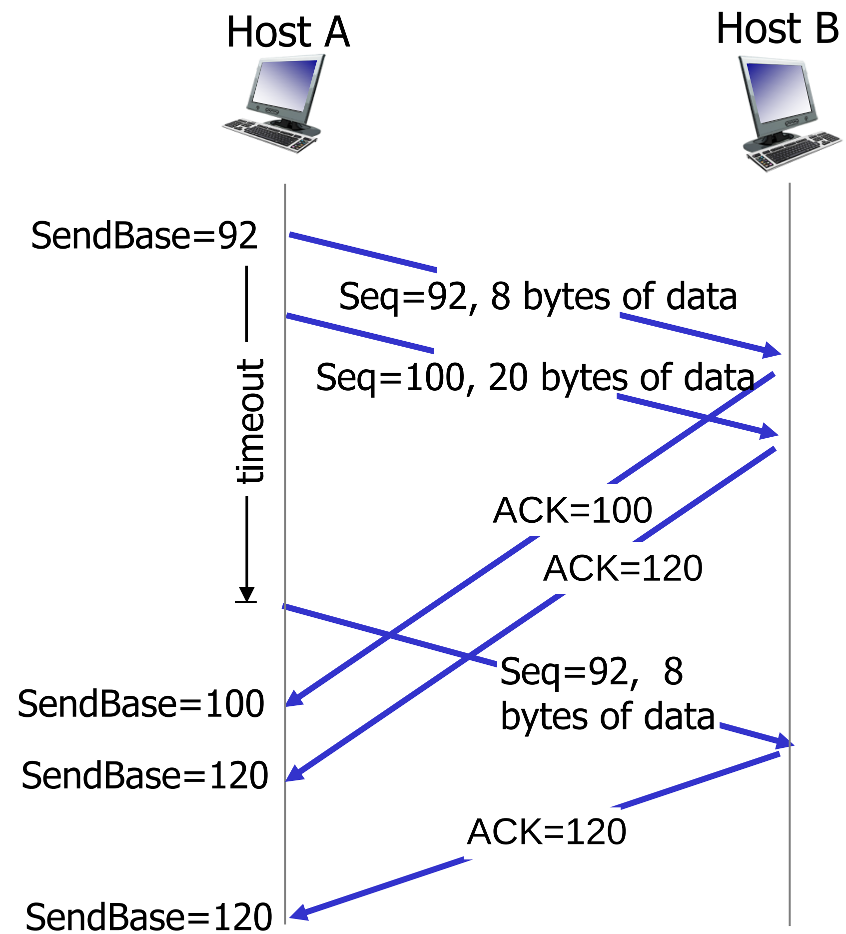
ACK received, with ACK field value y

```
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
       start timer
     else stop timer
 }
```

TCP: retransmission scenarios

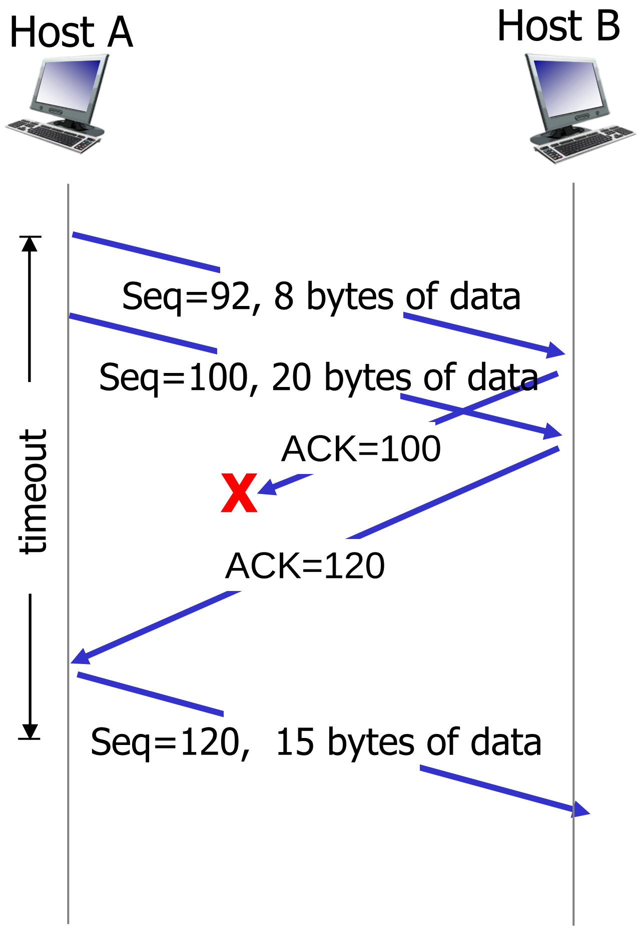


lost ACK scenario



premature timeout

TCP: retransmission scenarios: Poll 6



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

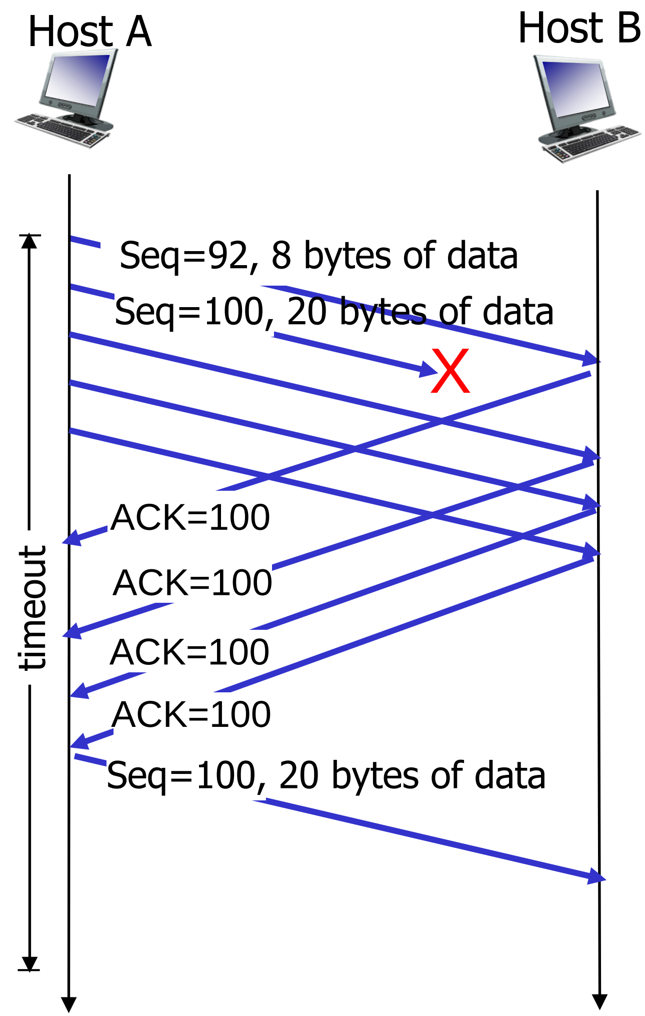
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

if sender receives 4 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

Round trip time (RTT) vs 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

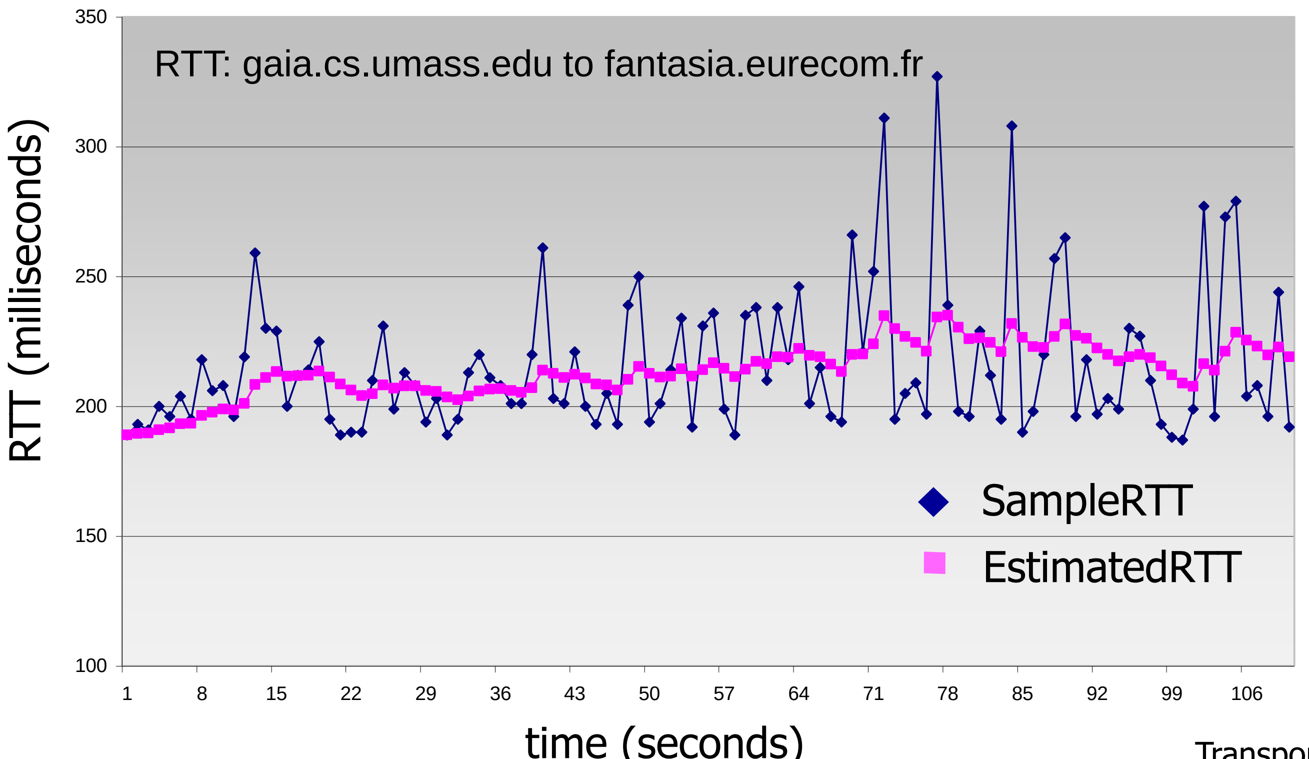
Q: how to estimate RTT?

- **`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`**

Round trip time (RTT) vs timeout

$$\mathtt{EstimatedRTT}_{i+1} = (1-\alpha)*\mathtt{EstimatedRTT}_i + \alpha*\mathtt{SampleRTT}_i$$

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



Round trip time (RTT) vs timeout

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

$$\mathbf{DevRTT}_{i+1} = (1-\beta)*\mathbf{DevRTT}_i + \beta*|\mathbf{SampleRTT}_i - \mathbf{EstimatedRTT}_i|$$

(typically, $\beta = 0.25$)

$$\mathbf{TimeoutInterval}_i = \underbrace{\mathbf{EstimatedRTT}_i}_{\text{estimated RTT}} + \underbrace{4*\mathbf{DevRTT}_i}_{\text{"safety margin"}}$$

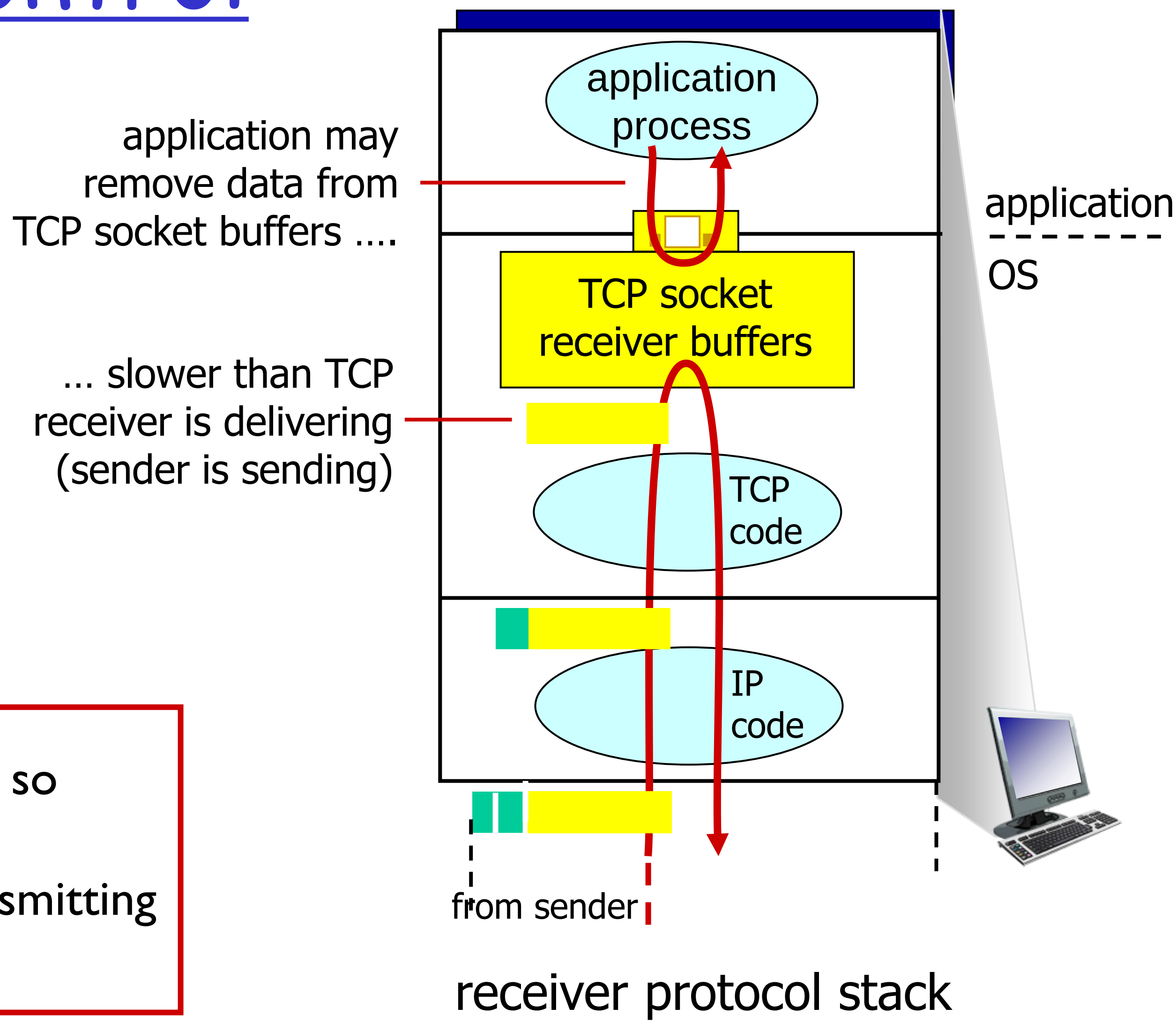
Summary for TCP reliable data transfer

- How to make reliable data?
  - Sequence number, retransmission timer, cumulative ACK
- How to shorten retransmission delay?
  - "Fast Retransmit": lost segments detection via duplicate ACKs
- How to set time-out value?
  - Exponential weighted moving average

Transport Layer

- Transport-layer services
- Connectionless transport: UDP
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

TCP flow control

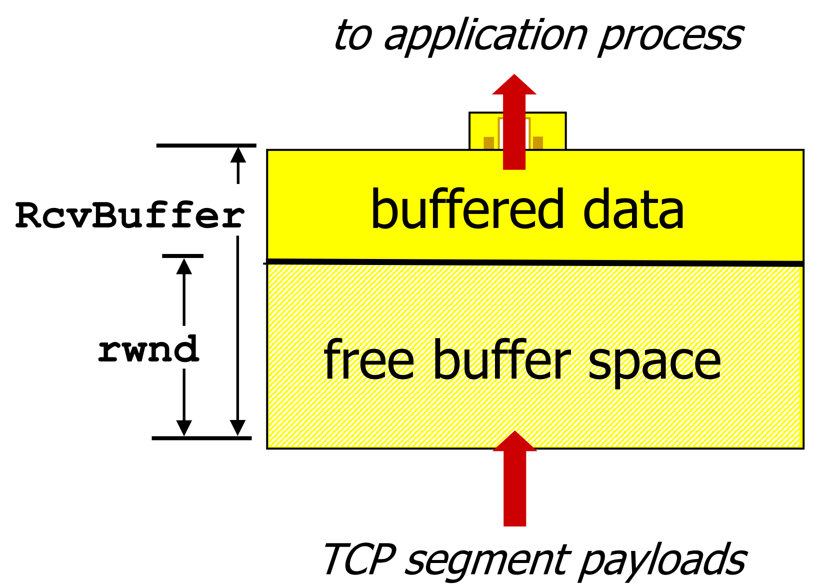


flow control

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

TCP flow control (cont'ed)

- 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 receiver buffer will not overflow



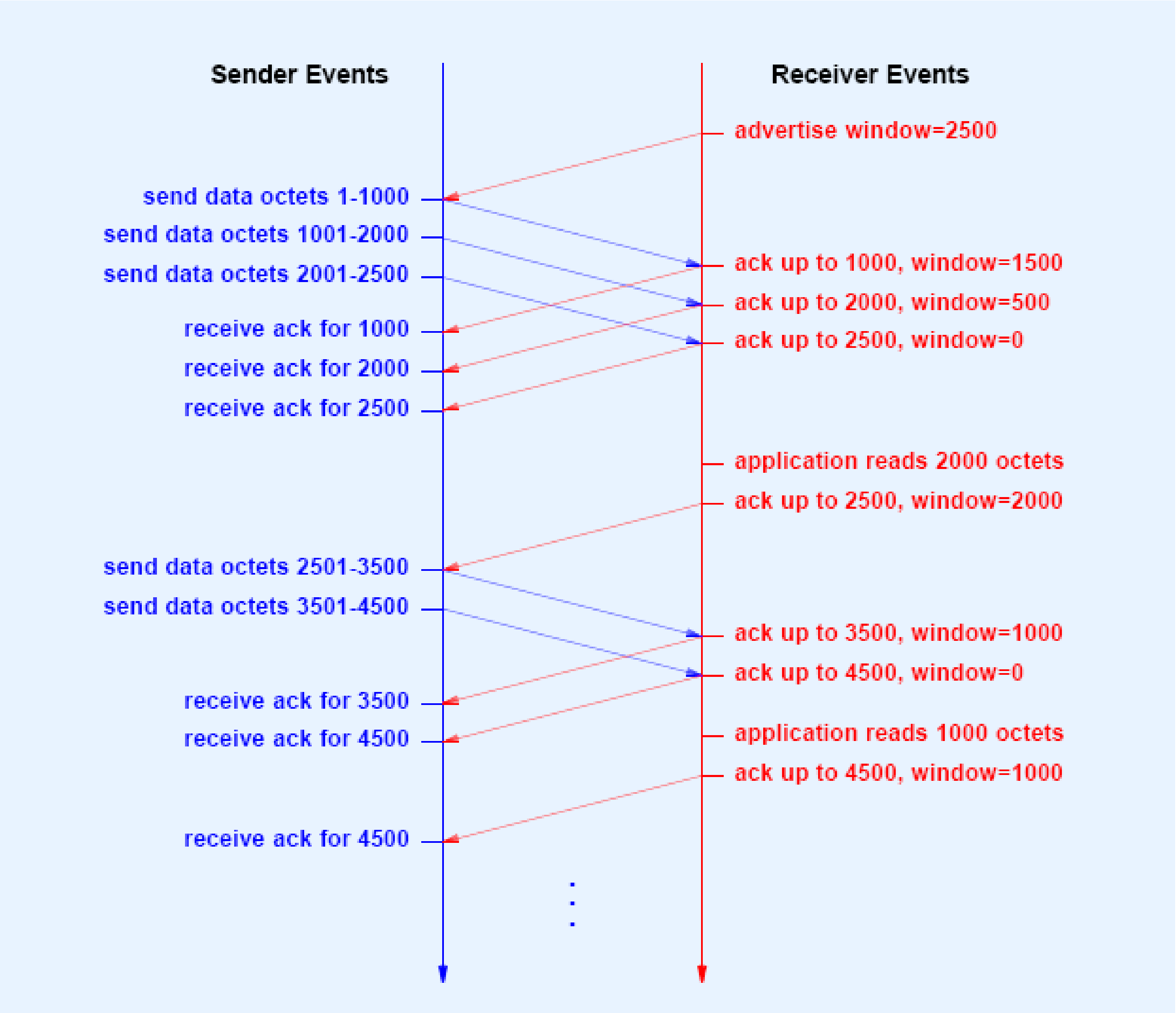
receiver-side buffering

Window Advertisement

- Each ACK carries new window information:
  - Acknowledgement number (AN)
  - Window size (W)

- ACK contains $AN = i$, $W = j$:
  - Bytes through $SN = i - 1$ acknowledged
    - Cumulative ACK
    - Byte i has not been received (It is the next byte expected)
  - Permission is granted to send $W = j$ more bytes
    - i.e. bytes i through $i + j - 1$

Illustration: Window Advertisement



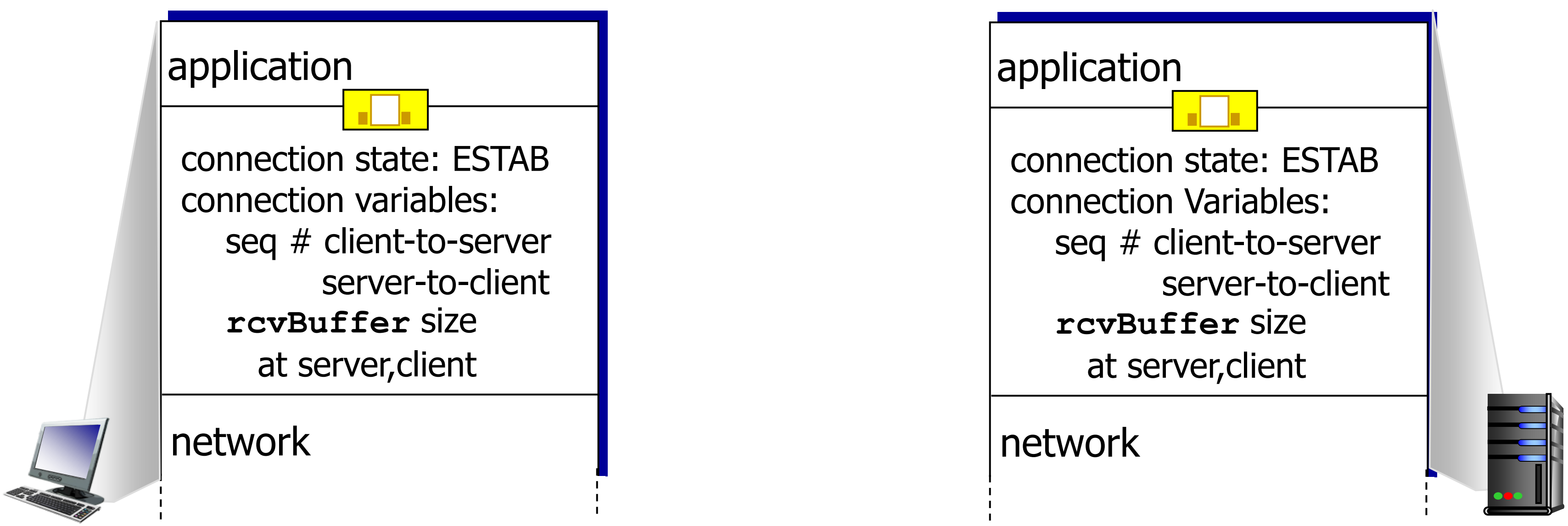
Transport Layer

- Transport-layer services
- Connectionless transport: UDP
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- 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

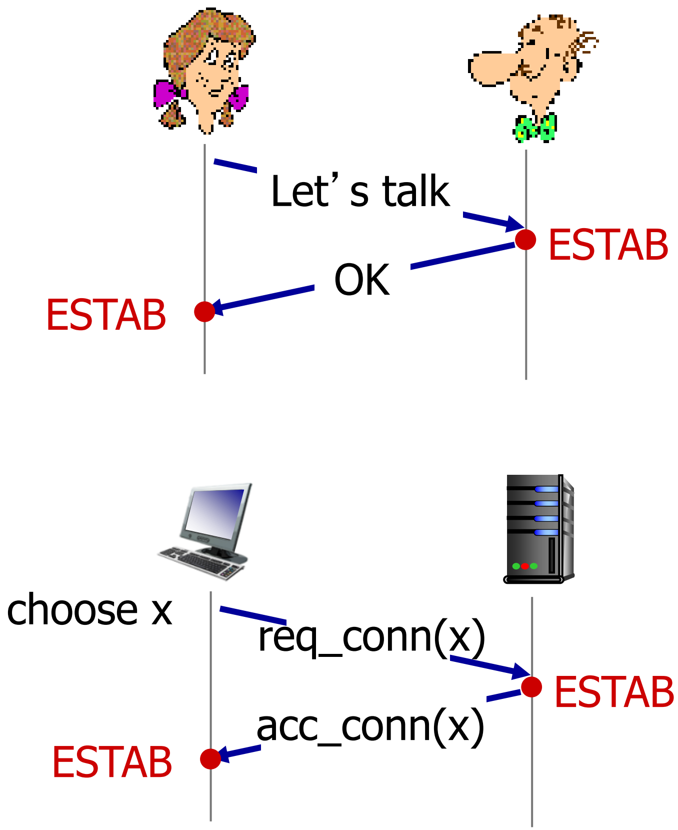



```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

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

Agreeing to establish a connection

2-way handshake:



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

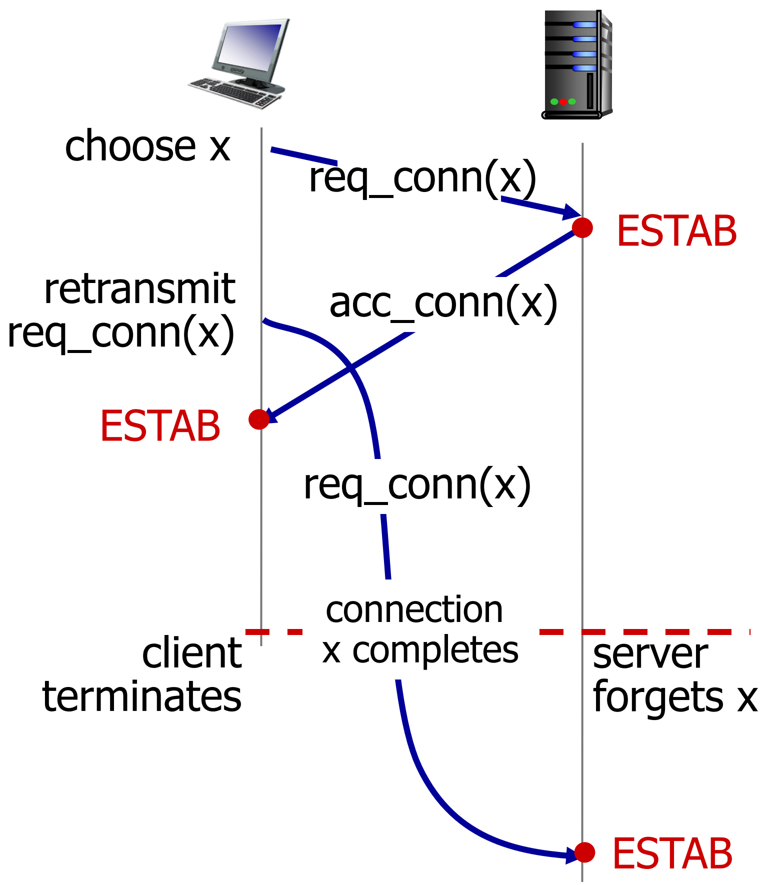
- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can't "see" other side

Problems with Two-way Handshake

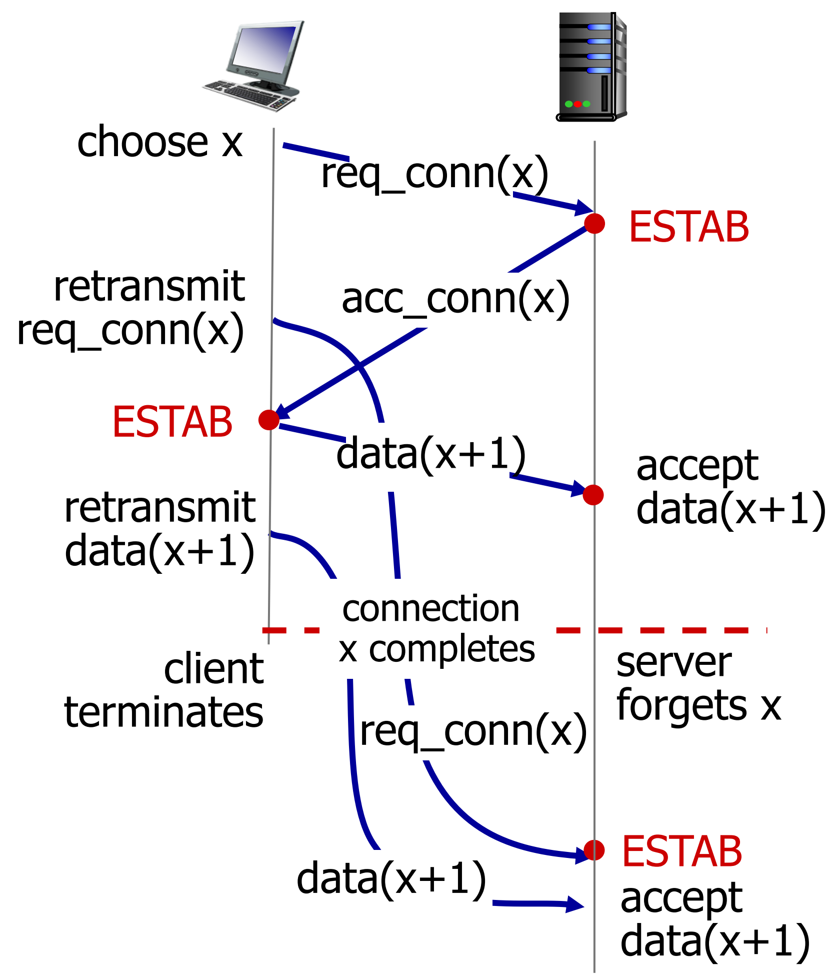
- In an *unreliable* network (e.g. the Internet), lost or delayed segments can cause problems in connection establishment, data transfer and connection termination

Agreeing to establish a connection

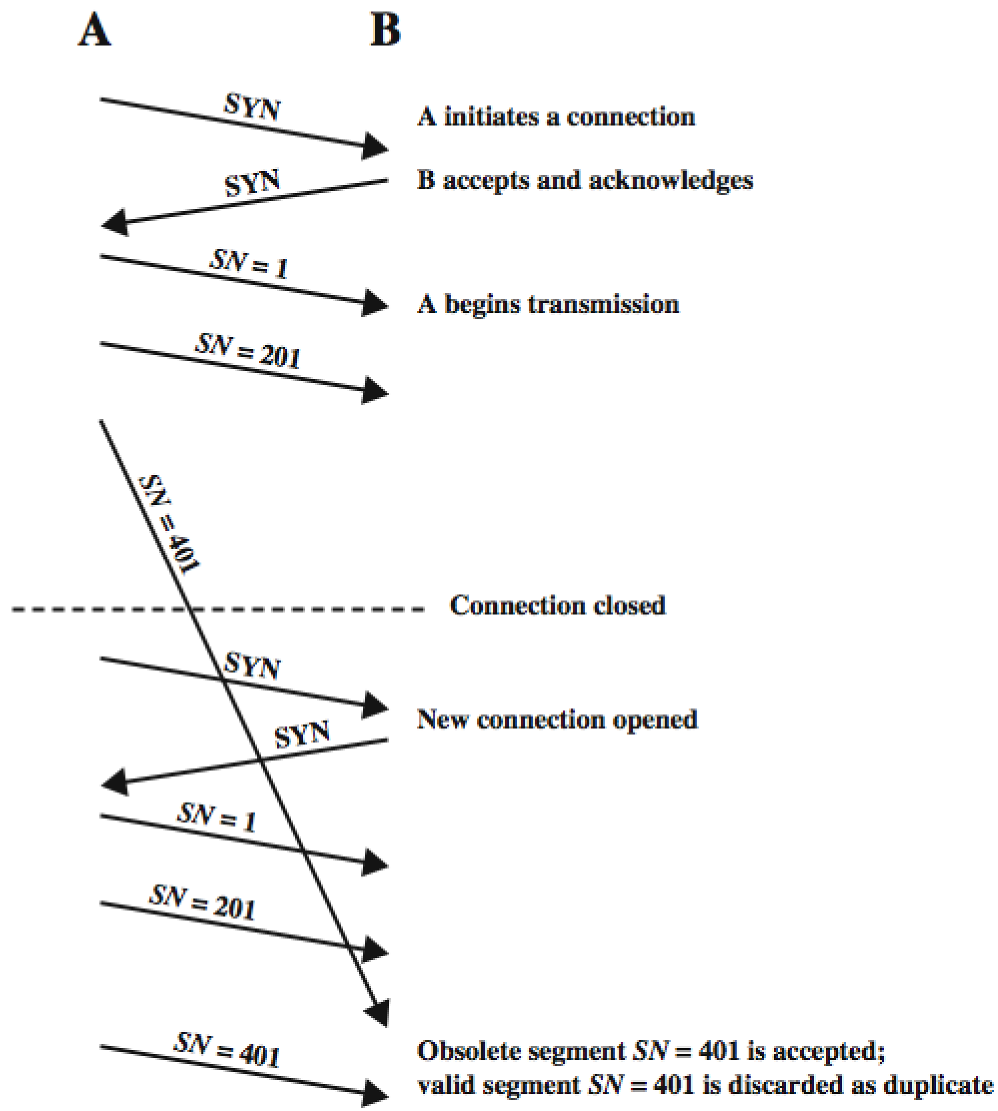
2-way handshake failure scenarios:



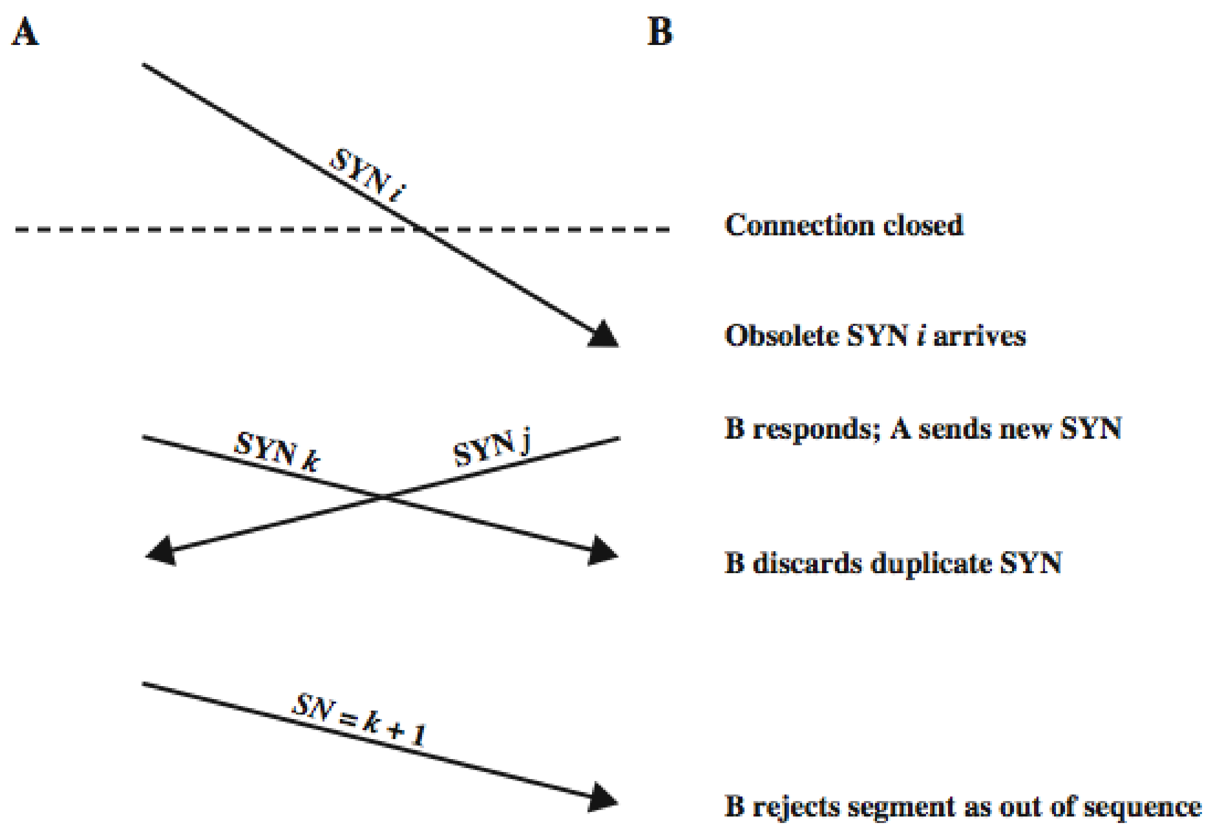
half open connection!
(no client!)



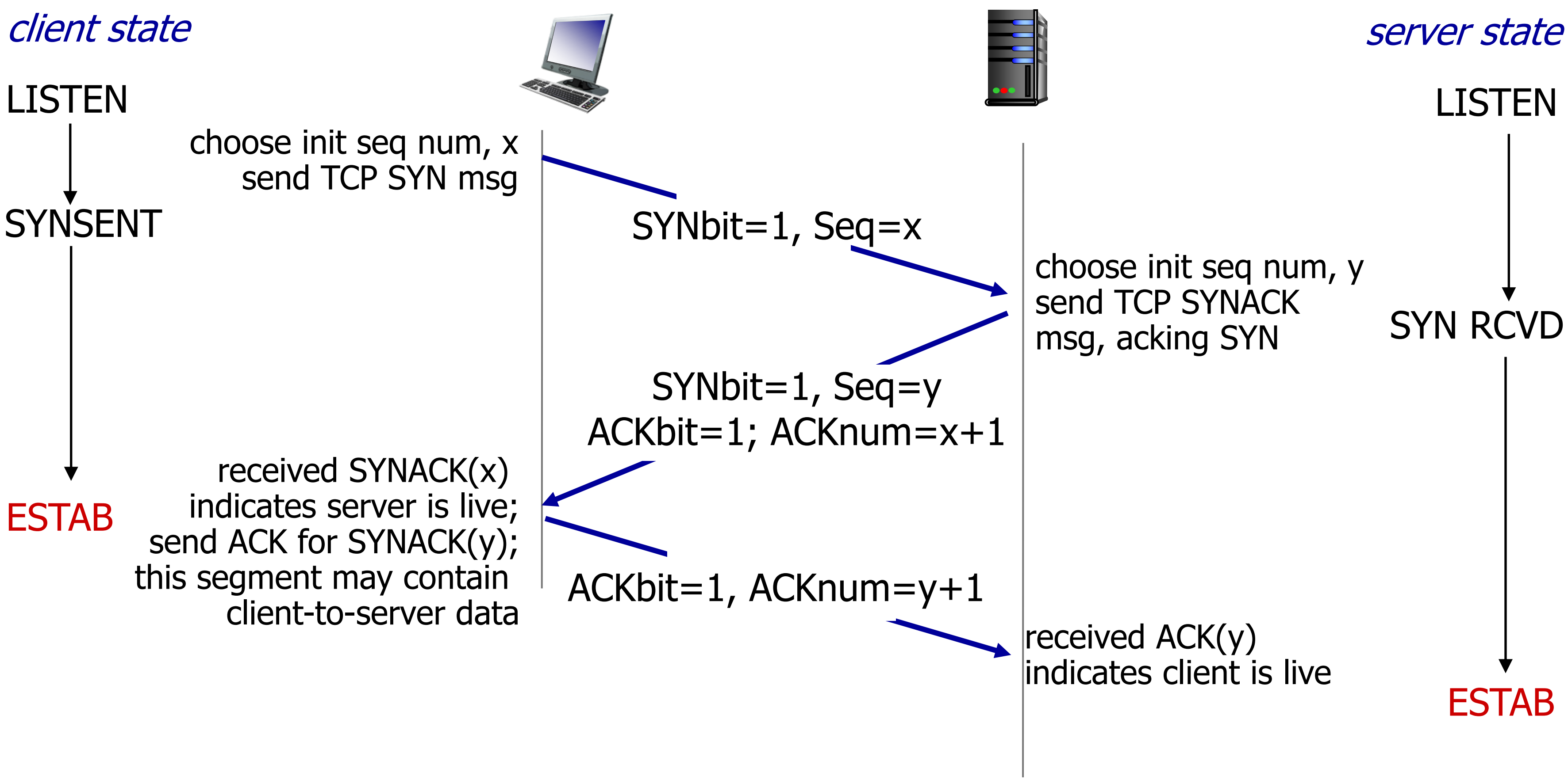
Two Way Handshake: Obsolete Data Segment



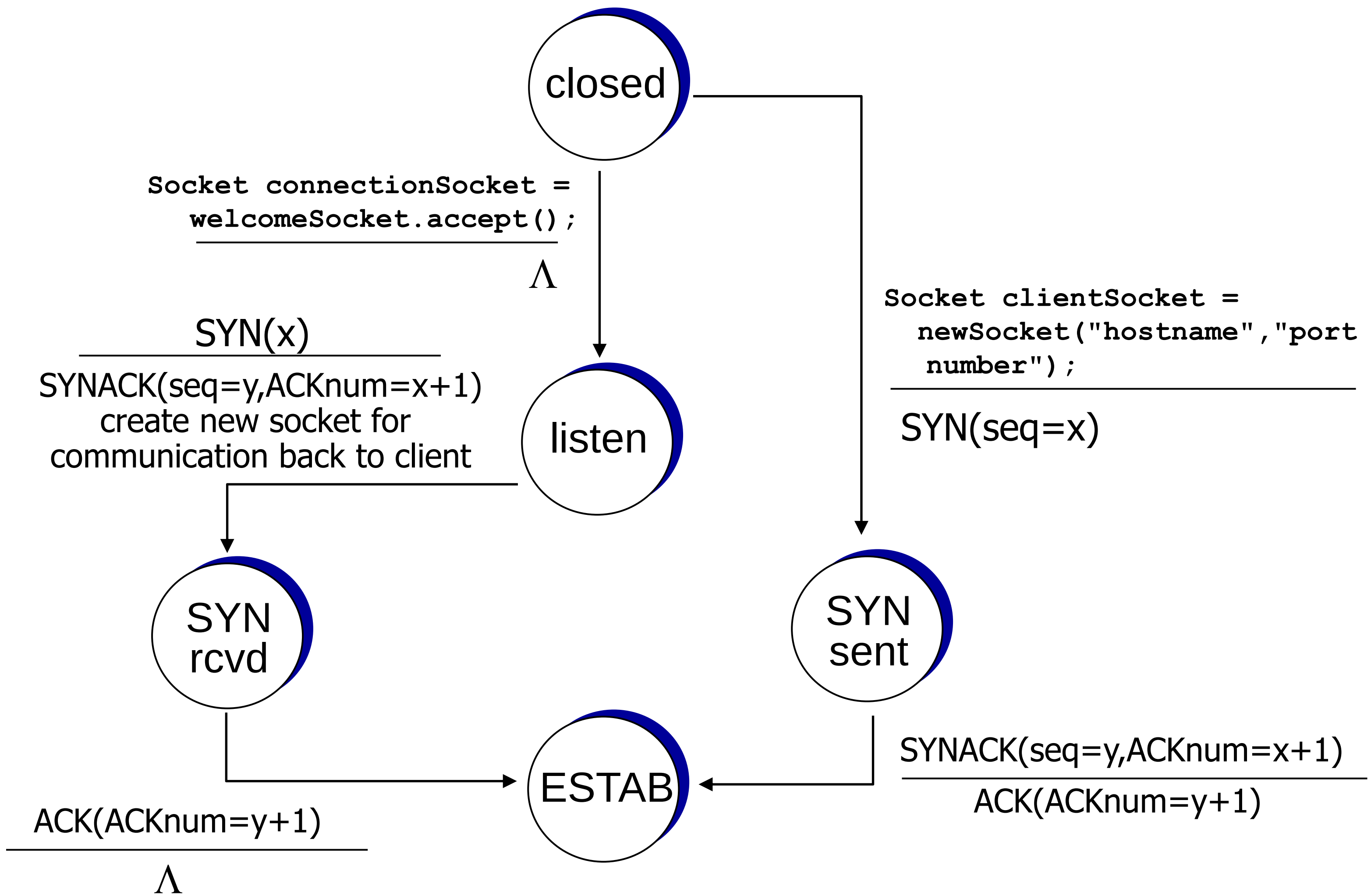
Two Way Handshake: Obsolete SYN Segment



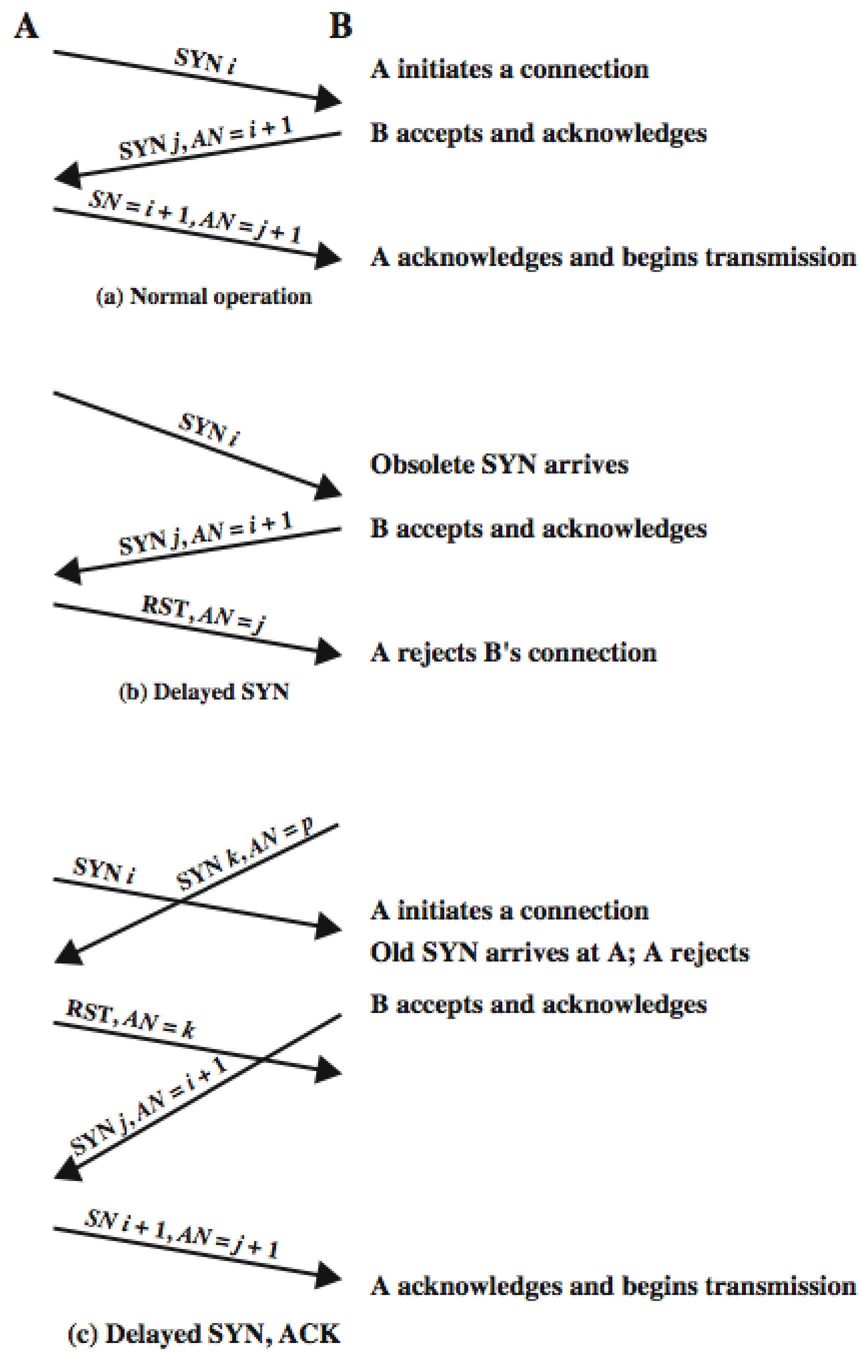
TCP 3-way handshake



TCP 3-way handshake: FSM



Three Way Handshake: Examples



Closing a connection

Question:

Do we have a perfect solution for synchronizing the disconnection on both end systems if data can be lost in the network?

See: the two-army problem or
the Romeo and Juliet problem

Closing a connection (cont'd)

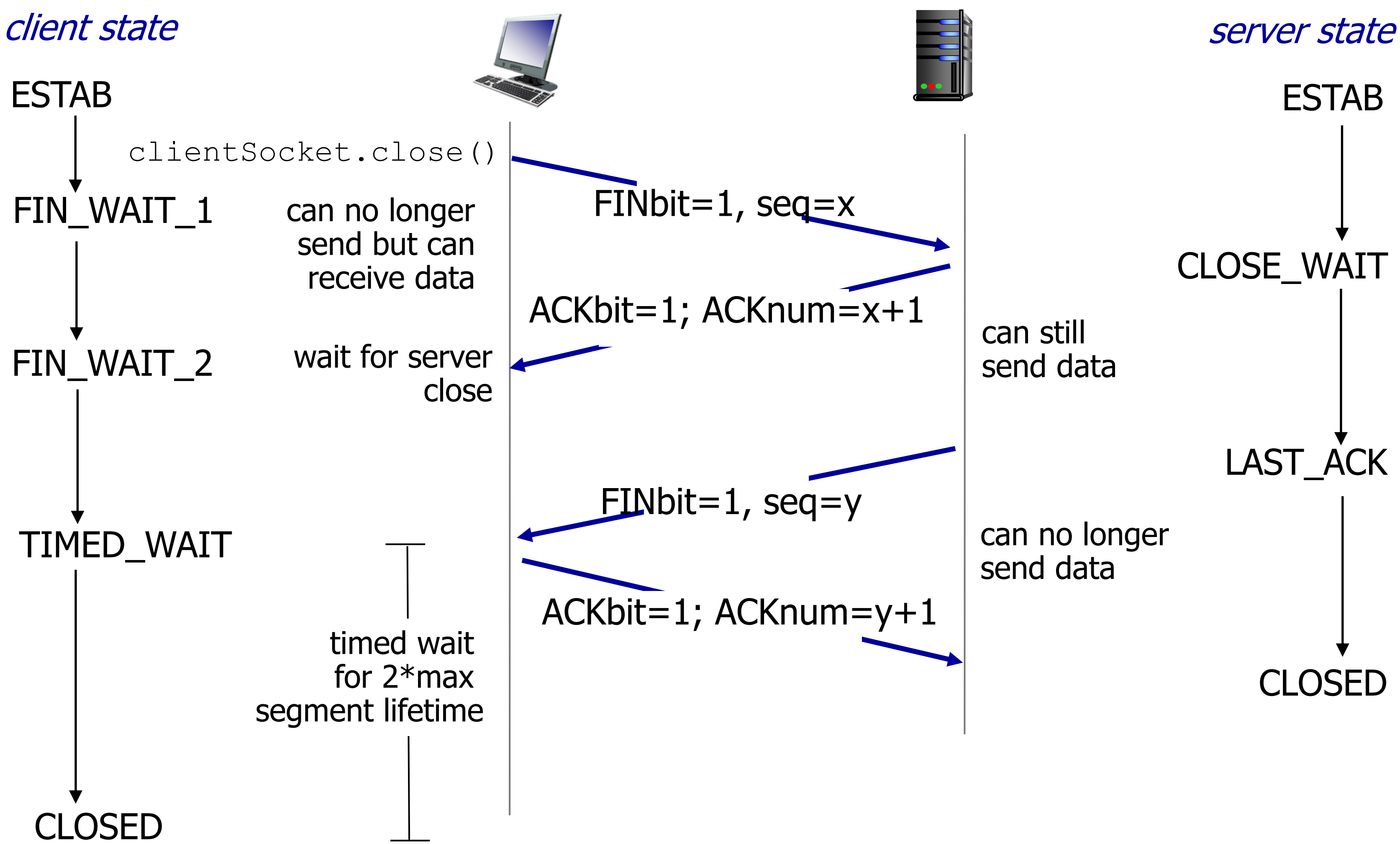
Answer:

No, but "three-way handshake" is an acceptable solution.

TCP: closing a Connection

- For better understanding, think of a TCP connection as a pair of simplex connections.
  - "Simplex" means uni-directional data flow
  - Note: A TCP connection is full duplex (i.e., bi-directional.)
- Each simplex connection is released independently using these two steps:
  - Send a TCP segment with the FIN bit set to one.
  - When the FIN is acknowledged, that direction is shut down.
- Timers are used for graceful disconnection to avoid the two-army problem.
  - Not a perfect solution, i.e. graceful disconnection cannot be guaranteed
  - In fact, there is no perfect solution at all!

TCP: closing a connection



Summary for connection management

- Problem: In an *unreliable* network (e.g. the Internet), lost or delayed segments can cause problems in connection establishment, data transfer and connection termination.
- Acceptable Solution: three way handshake
- Three way handshake is much better than two way handshake.
- Timers are used for graceful disconnection to avoid the two-army problem.

Transport Layer

- Transport-layer services
- Connectionless transport: UDP
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

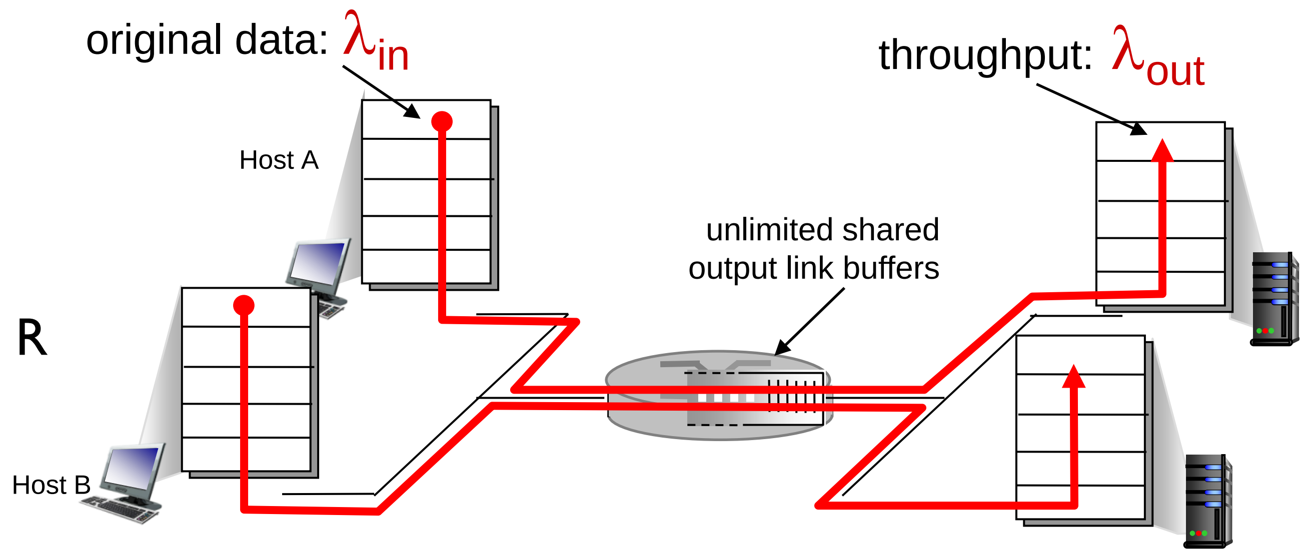
Principles of Congestion Control

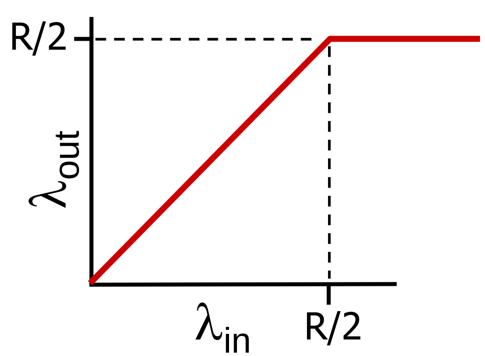
Congestion:

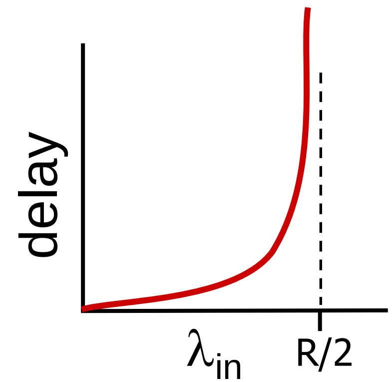
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- Signs indicating congestion:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

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







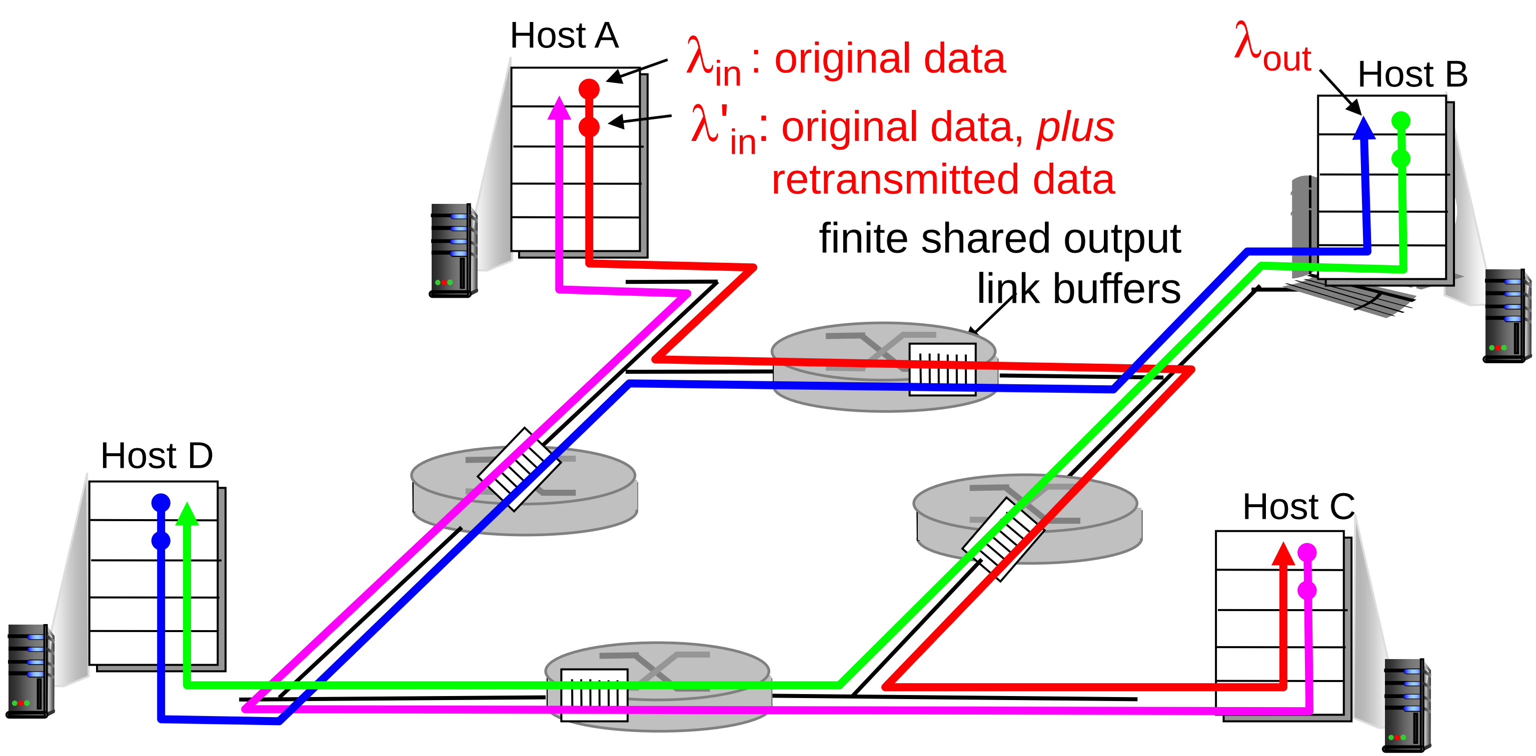
- maximum per-connection throughput: R/2
- large delays as arrival rate, λ_{in}, approaches capacity

Causes/costs of congestion: scenario 2

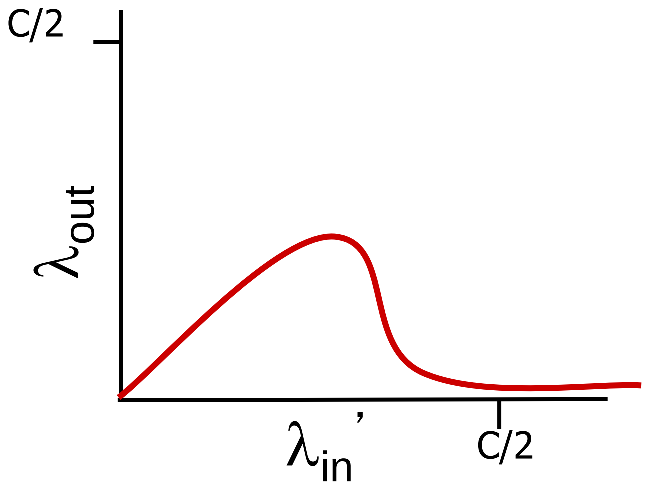
- four senders/receivers
- multihop paths
- timeout/retransmission

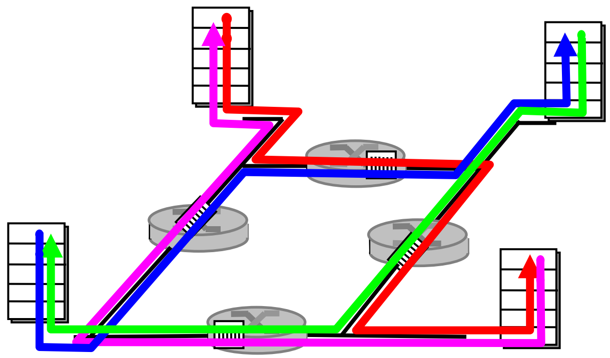
Q: what happens as λ_{in} and λ_{in}' increase ?

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



Causes/costs of congestion: scenario 2





another "cost" of congestion:

- when a 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
- approach taken by TCP

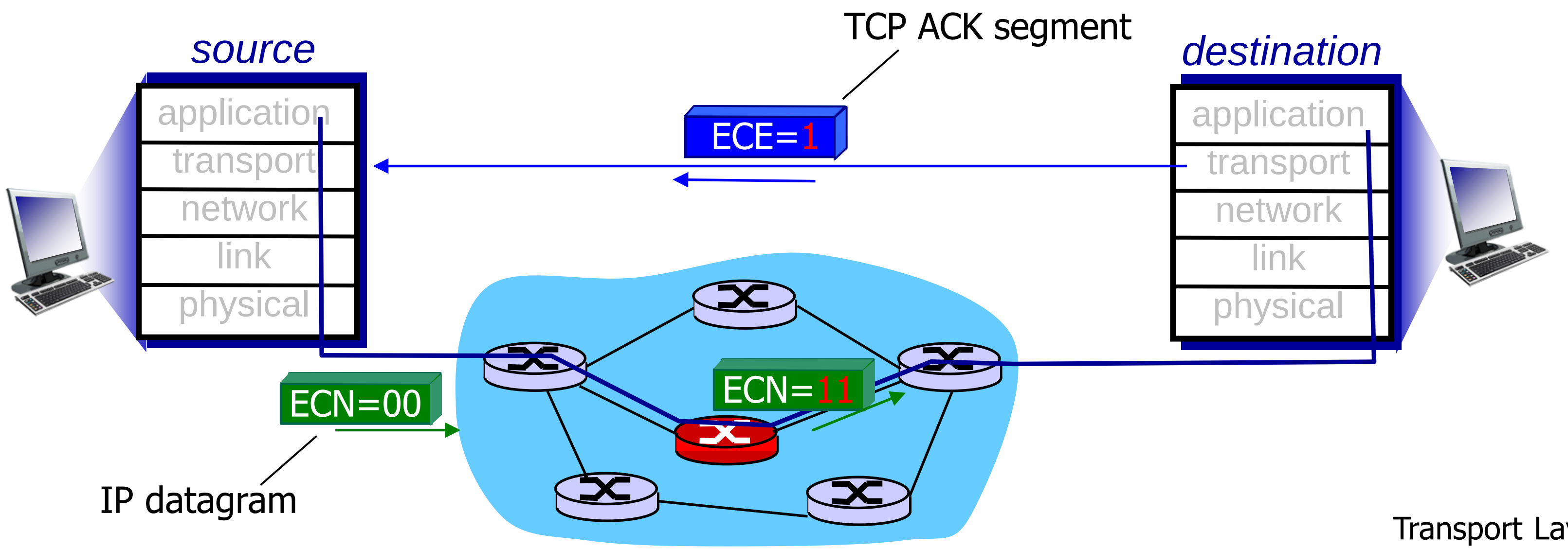
network-assisted congestion control:

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

Explicit Congestion Notification (ECN)

network-assisted congestion control:

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



Transport Layer

- Transport-layer services
- Connectionless transport: UDP
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - connection management
- Principles of congestion control
- TCP congestion control

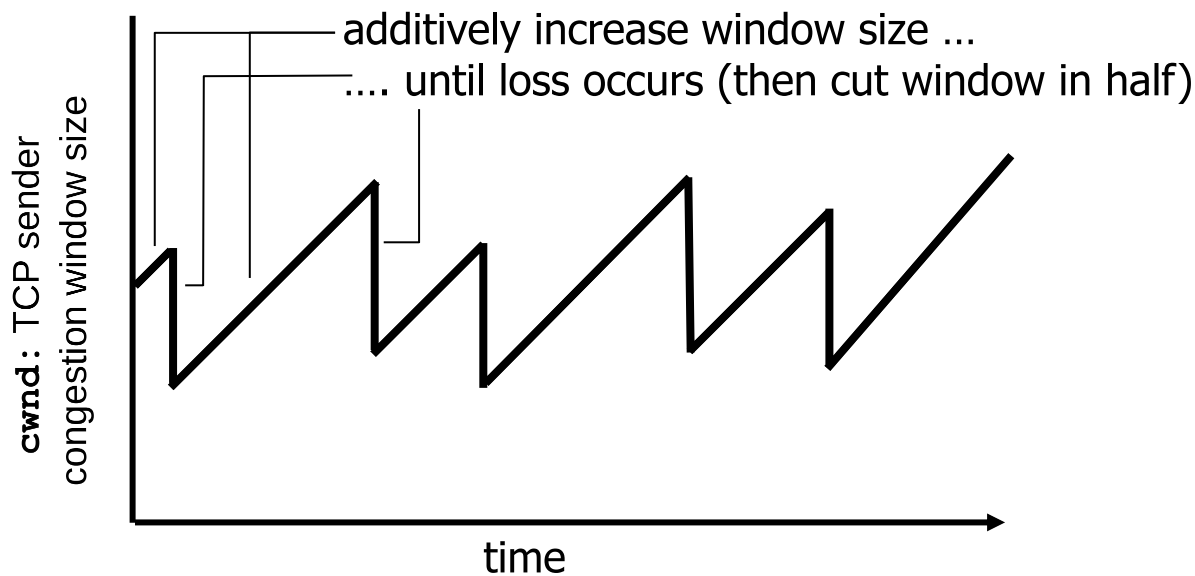
TCP congestion control:

- *goal:* TCP sender should transmit as fast as possible, but without congesting network
  - Q: how to find rate *just* below congestion level?
- decentralized: each TCP sender sets its own rate, based on *implicit* feedback:
  - *ACK:* segment received (a good thing!), network not congested, so increase sending rate
  - *lost segment:* assume loss due to congested network, so decrease sending rate

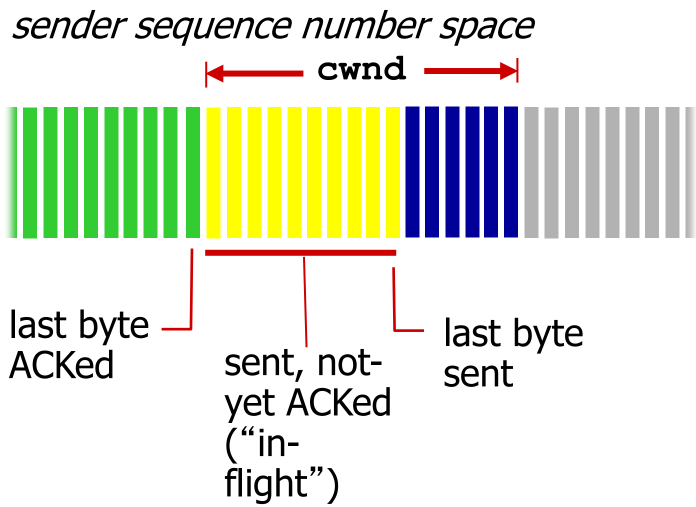
TCP congestion control: additive increase multiplicative decrease

- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **`cwnd`** by 1 Maximum Segment Size (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:

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

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

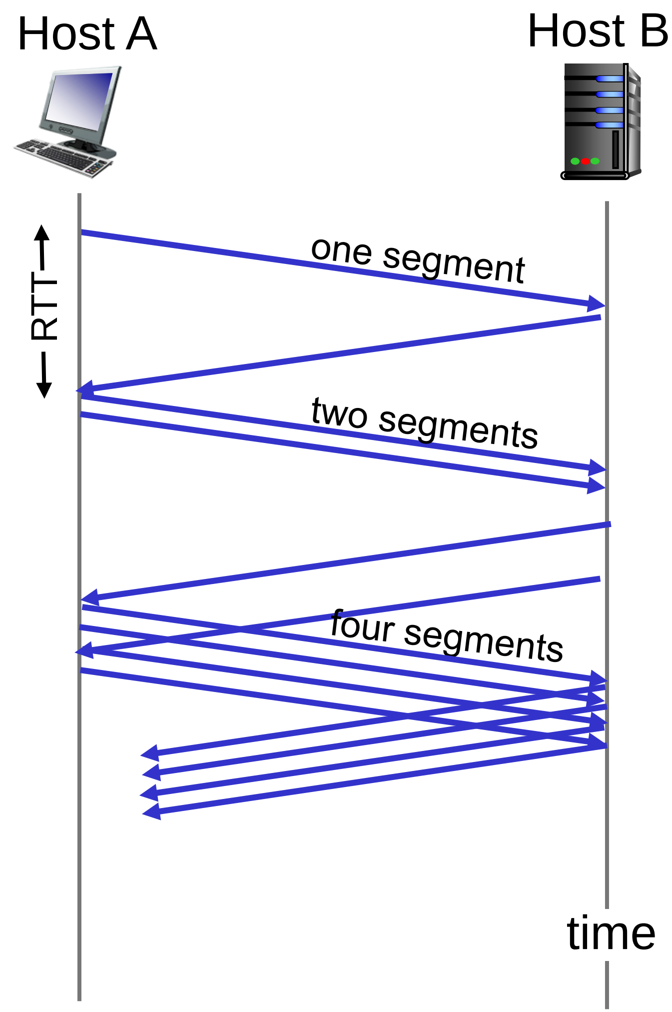
TCP sending rate:

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

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

TCP Slow Start

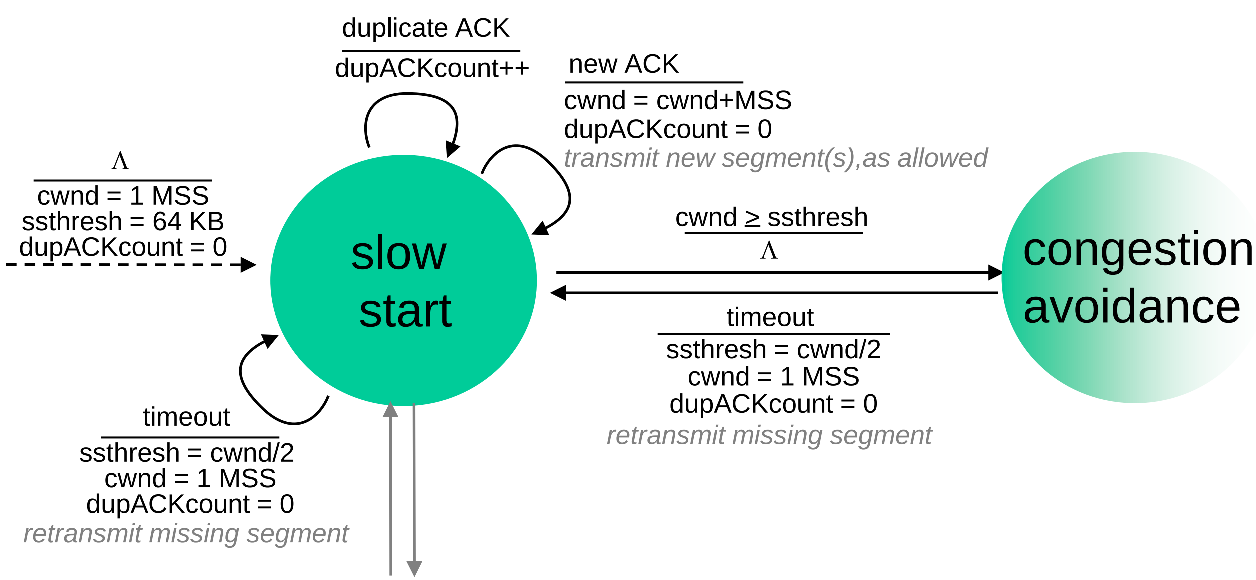
- when connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- *summary:* initial rate is slow but ramps up exponentially fast



Transitioning into/out of slowstart

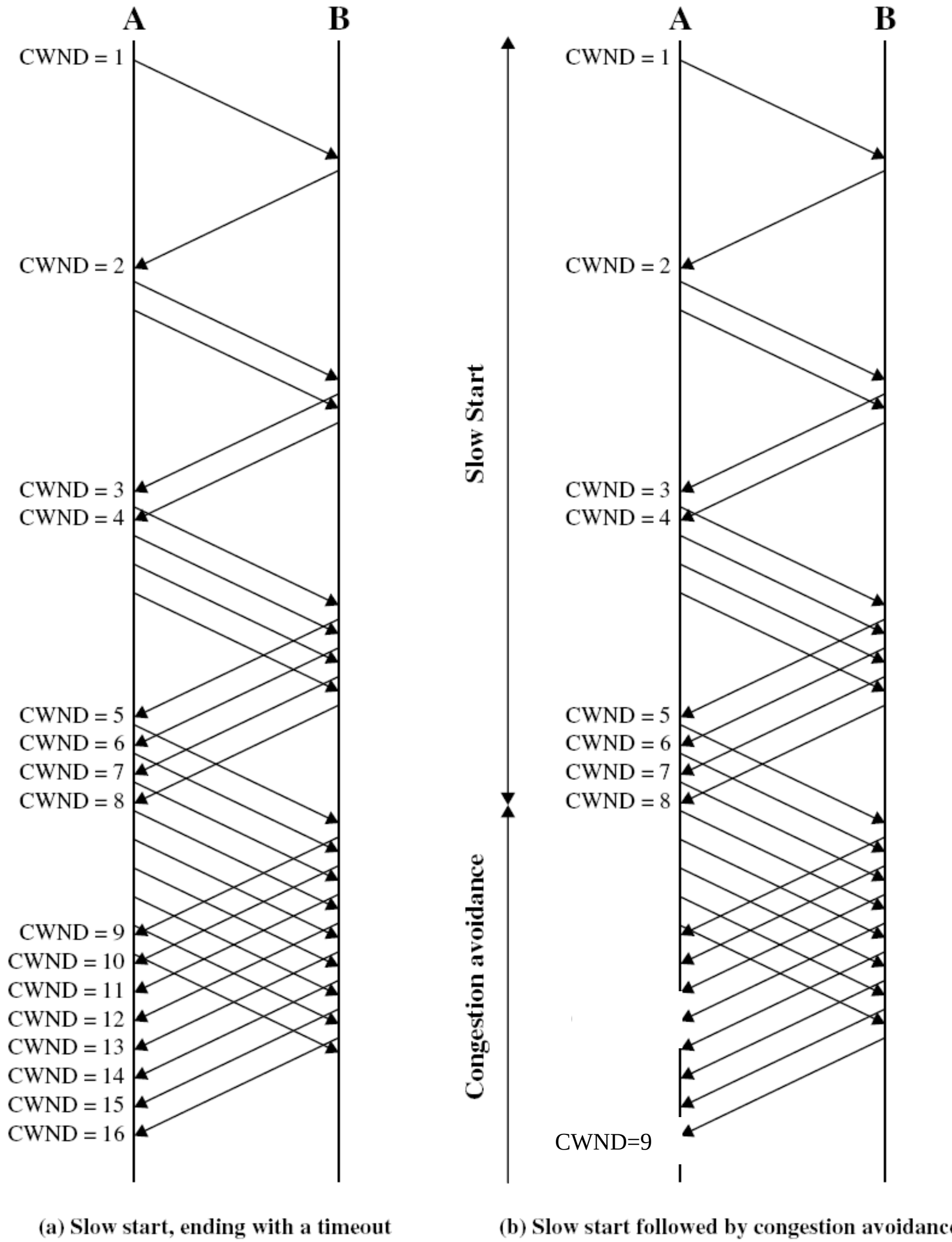
ssthresh: **cwnd** threshold maintained by TCP

- on loss event: set **ssthresh** to **cwnd/2**
  - remember (half of) TCP rate when congestion last occurred
- when **cwnd >= ssthresh**: transition from slowstart to congestion avoidance phase



TCP: congestion avoidance

- when `cwnd >= ssthresh`, `cwnd` grows linearly
  - increase `cwnd` by 1 MSS per RTT
  - approach possible congestion slower than in slowstart


A
B
CWND = 1
CWND = 2
CWND = 3
CWND = 4
CWND = 5
CWND = 6
CWND = 7
CWND = 8
CWND = 9
CWND = 10
CWND = 11
CWND = 12
CWND = 13
CWND = 14
CWND = 15
CWND = 16
(a) Slow start, ending with a timeout
Slow Start
Congestion avoidance
A
B
CWND = 1
CWND = 2
CWND = 3
CWND = 4
CWND = 5
CWND = 6
CWND = 7
CWND = 8
CWND=9
(b) Slow start followed by congestion avoidance

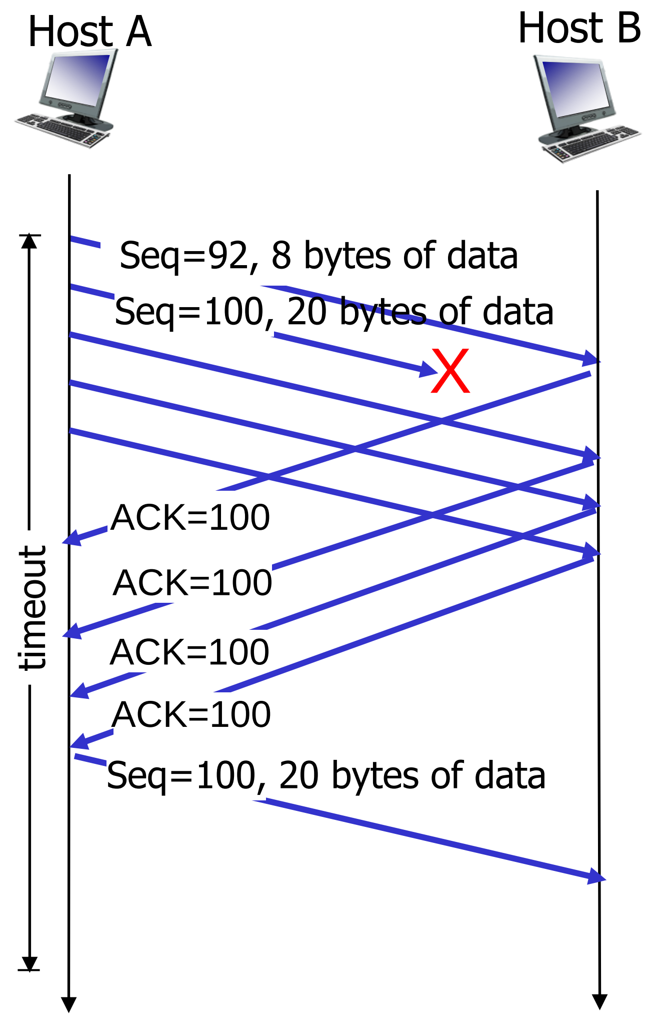
TCP Congestion Control: segment loss event

- Loss indicated by timeout:
  - cut **cwnd** to 1 MSS
  - **cwnd** then grows exponentially (as in slow start) to threshold, then grows linearly
- Loss indicated by 3 duplicate ACKs: TCP RENO
  - **cwnd** is cut in half and **cwnd** then grows linearly: less aggressively than on timeout (Fast Recovery)
  - Note that TCP Tahoe always set **cwnd** to 1 (timeout or 3 duplicate acks)

Philosophy:

- 3 dup ACKs indicates network capable of delivering some segments (recall fast retransmit)
- timeout indicates a "more alarming/serious" congestion scenario

TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

Why Fast Recovery is used?

- When TCP retransmits a segment using Fast Retransmit, a segment was assumed lost
- Some congestion avoidance measures are appropriate at this point
- Slow Start may be unnecessarily conservative since multiple acks indicate segments are getting through (meaning congestion not so serious)
- Fast Recovery: retransmit lost segment, cut **`CongWin`** in half, and proceed with linear increase of **`CongWin`** (avoiding "slow" start-up)

TCP Congestion Control: ACK received

ACK received: increase **CWND**

- slowstart phase:
  - increase exponentially fast (despite name) at connection start or following timeout
- congestion avoidance:
  - increase linearly

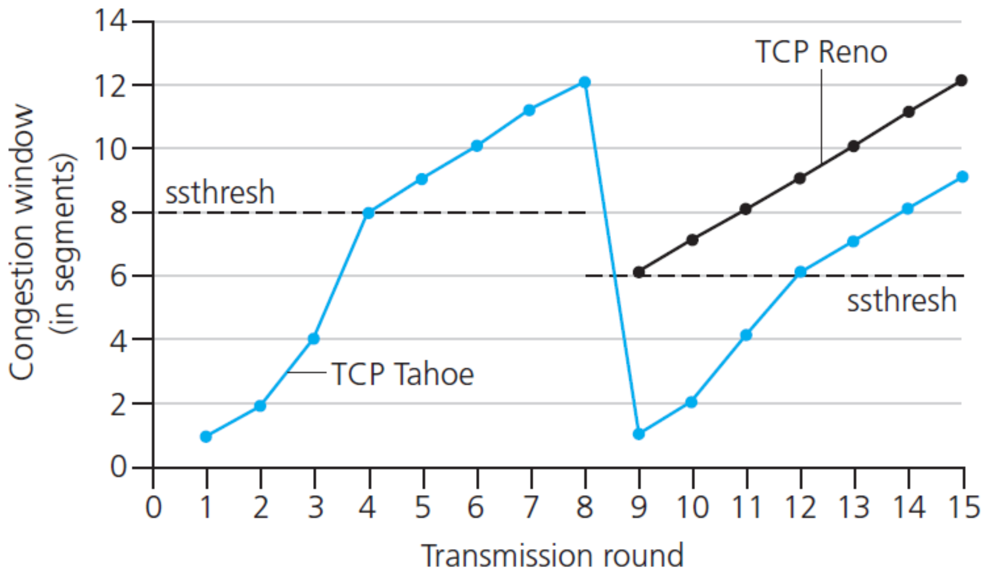
Variants of TCP Congestion Control Schemes

- TCP Tahoe: Slow Start + Congestion Avoidance.
- TCP Reno: TCP Tahoe + Fast Retransmit + Fast Recovery.

TCP: from slow start to congestion avoidance

Q: when should the exponential increase switch to linear?

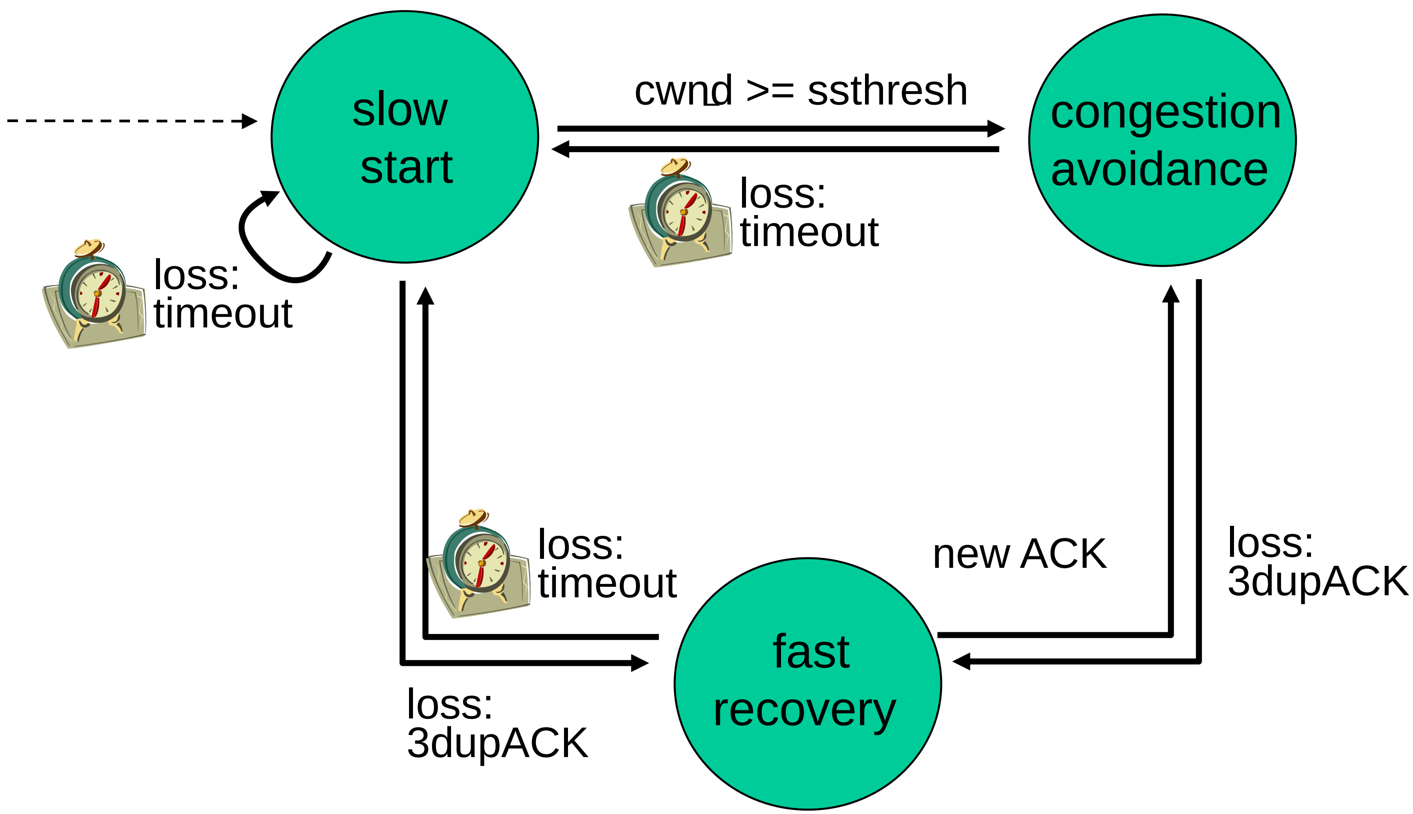
A: when **cwnd** gets to 1/2 of its value before last timeout.



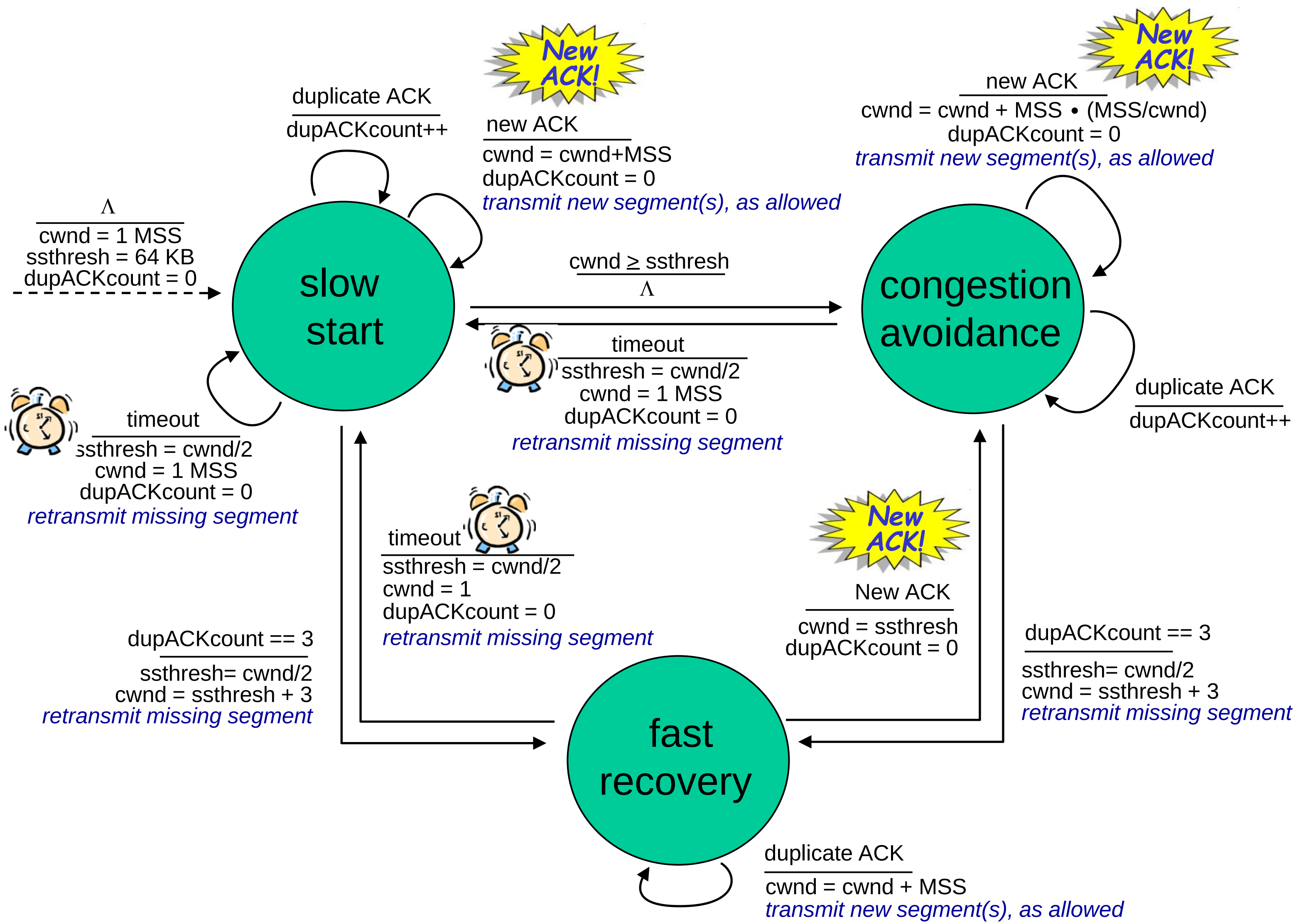
Implementation:

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

TCP Reno congestion control FSM: overview



TCP Reno congestion control FSM: details



Summary: TCP Congestion Control

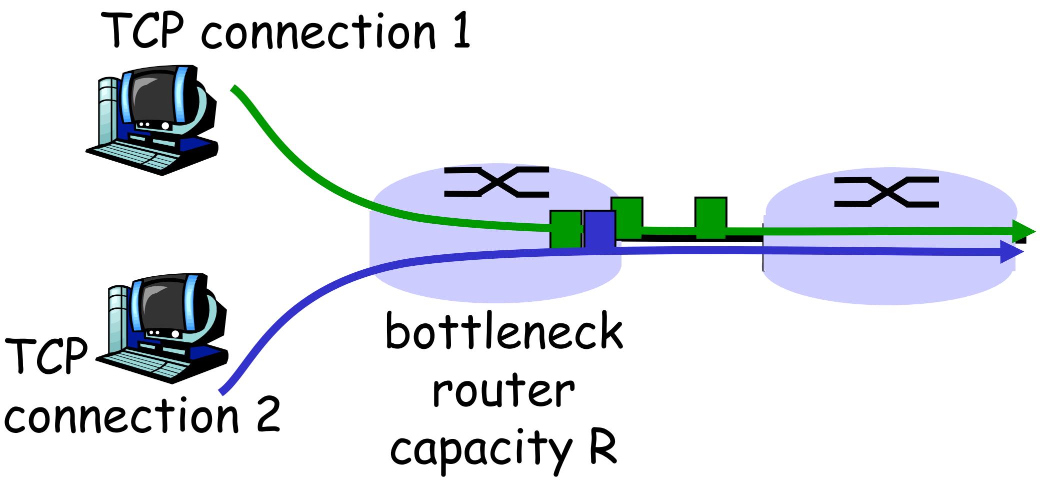
- when `cwnd < ssthresh`, sender is in slow-start phase, window grows exponentially.
- when `cwnd >= ssthresh`, sender is in congestion-avoidance phase, window grows linearly.
- when triple duplicate ACK occurs, `ssthresh` set to `cwnd/2`, `cwnd` set to `ssthresh` and go to congestion-avoidance phase.
- when timeout occurs, `ssthresh` set to `cwnd/2`, `cwnd` set to 1 MSS and go to slow-start phase.

TCP sender congestion control

State	Event	TCP Sender Action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	CongWin = CongWin + 1 MSS, If (CongWin >= Threshold) set state to “Congestion Avoidance”	Resulting in a doubling of CongWin every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	CongWin = CongWin + 1 MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
SS or CA	Loss event detected by triple duplicate ACK	Threshold = CongWin/2, CongWin = Threshold, Set state to “Congestion Avoidance”	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
SS or CA	Timeout	Threshold = CongWin/2, CongWin = 1 MSS, Set state to “Slow Start”	Enter slow start
SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

TCP Fairness

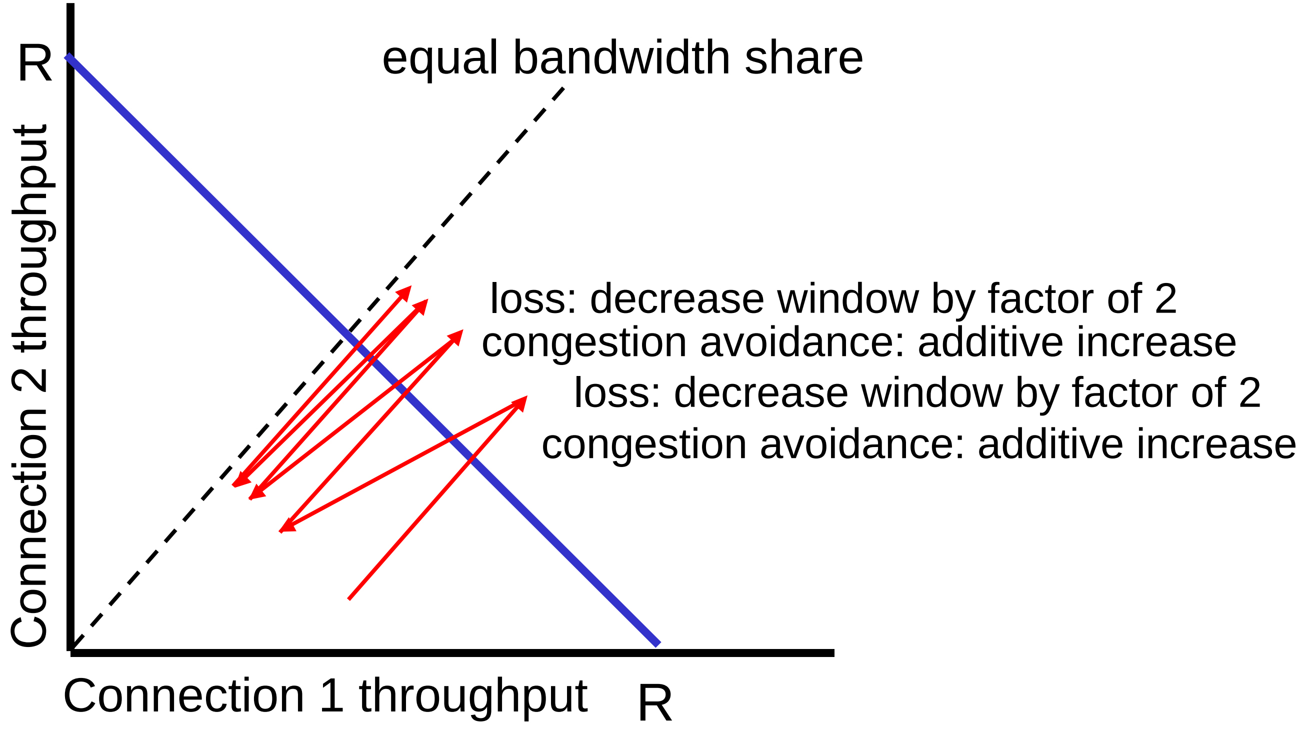
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



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 slowed down by congestion control
- instead use UDP:
  - pump audio/video at constant rate, tolerate packet loss

Fairness and parallel TCP connections

- nothing prevents app from opening parallel connections between 2 hosts.
- web browsers do this
- For example: link of rate R supporting 9 connections;
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 9 TCPs, gets rate R/2 !

Summary

- Transport-layer services
- Connectionless transport: UDP
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

Q & A