# COMP 3331/9331: Computer Networks and Applications

1. TCP Reliability
2. TCP Flow Control
3. TCP Connection management
4. TCP Congestion Control

## Week 5

## Transport Layer (Continued)

Reading Guide: Chapter 3, Sections: 3.5 – 3.7

# Transport Layer Outline

# Recall: Components of a solution for reliable transport
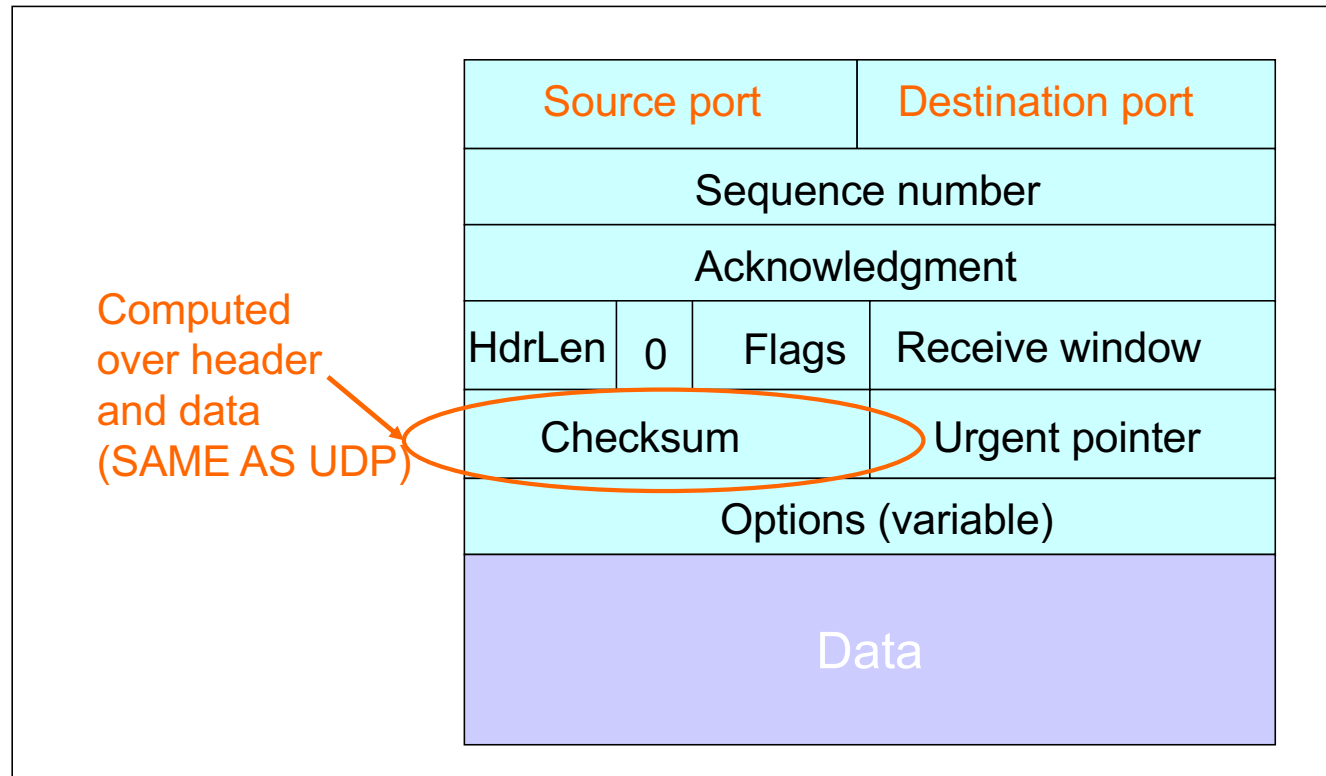
- ❖ Checksums (for error detection)
- ❖ Timers (for loss detection)
- ❖ Acknowledgments
  - ▪ Cumulative
  - ▪ Selective
- ❖ Sequence numbers (duplicates, windows)
- ❖ Sliding Windows (for efficiency)
  - ▪ Go-Back-N (GBN)
  - ▪ Selective Repeat (SR)

# What does TCP do?

Many of our previous ideas, but some key differences

❖ Checksum

# TCP Header

| Source port | Destination port |
|---|---|
| Sequence number | |
| Acknowledgment | |
| HdrLen | 0 | Flags | Receive window |
| Checksum | Urgent pointer |
| Options (variable) | |
| Data | |

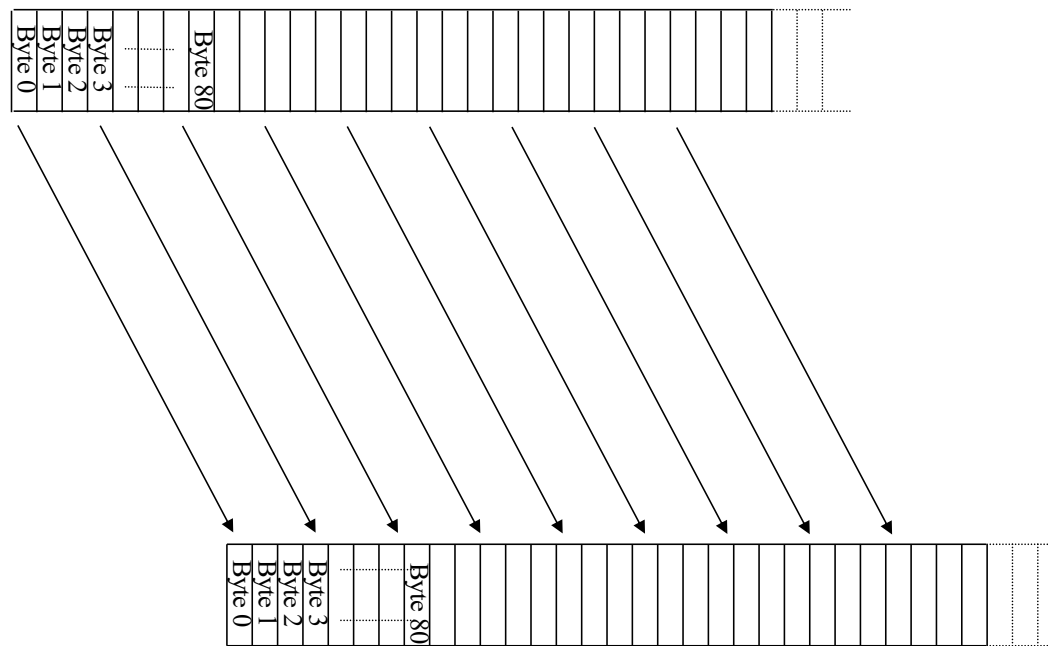Computed over header and data (SAME AS UDP)

# What does TCP do?

Many of our previous ideas, but some key differences

- ❖ Checksum
- ❖ **Sequence numbers are byte offsets**
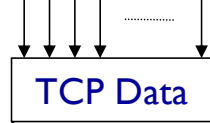
# TCP "Stream of Bytes" Service ..

Application @ Host A

Application @ Host B

# .. Provided Using TCP "Segments"
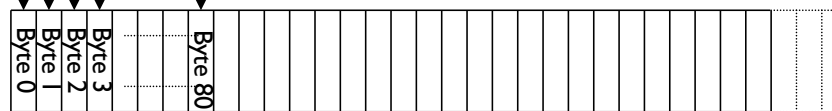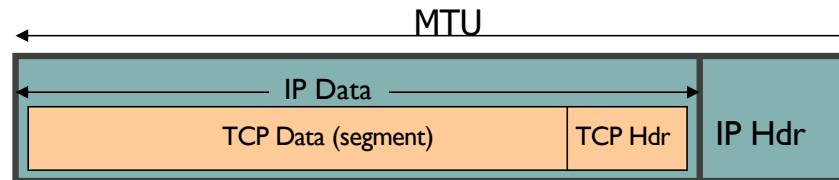
Host A

Byte 0 | Byte 1 | Byte 2 | Byte 3 | ... | Byte 80

TCP Data
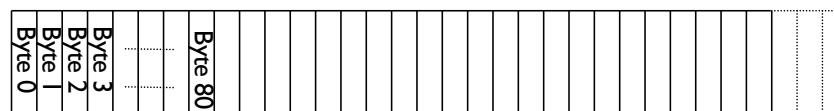
*Segment* sent when:
1. Segment full (Max Segment Size),
2. Not full, but instructed by the Application e.g., 1 byte in Telnet

TCP Data

Host B

Byte 0 | Byte 1 | Byte 2 | Byte 3 | ... | Byte 80

# TCP Maximum Segment Size

MTU

| IP Data | | |
|---|---|---|
| TCP Data (segment) | TCP Hdr | IP Hdr |

- ❖ IP packet
  - ▪ No bigger than Maximum Transmission Unit (MTU) of link layer
  - ▪ E.g., up to 1500 bytes with Ethernet
- ❖ TCP packet
  - ▪ IP packet with a TCP header and data inside
  - ▪ TCP header $\geq$ 20 bytes long
- ❖ TCP **segment**
  - ▪ No more than Maximum Segment Size (MSS) bytes
  - ▪ E.g., up to 1460 consecutive bytes from the stream
  - ▪ MSS = MTU – 20 (minimum IP header ) – 20 ( minimum TCP header )

# Sequence Numbers

ISN (initial sequence number)

k bytes

Host A

Sequence number
= 1$^{st}$ byte in segment =
ISN + k
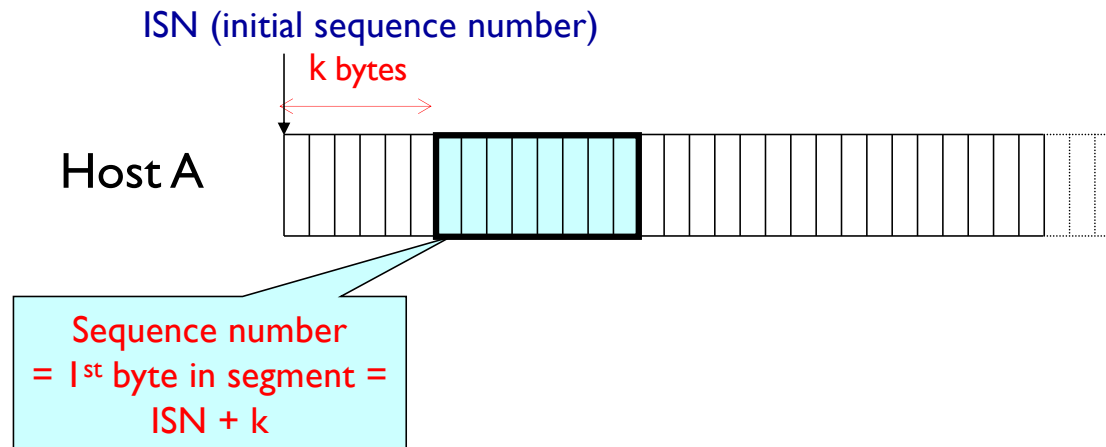
## Sequence numbers:

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

# Sequence & Ack Numbers

ISN (initial sequence number)

k

Host A

Sequence number
= 1st byte in segment =
ISN + k

TCP Data | TCP HDR

ACK sequence number
= next expected byte
= seqno + length(data)

TCP Data | TCP HDR

Host B

11
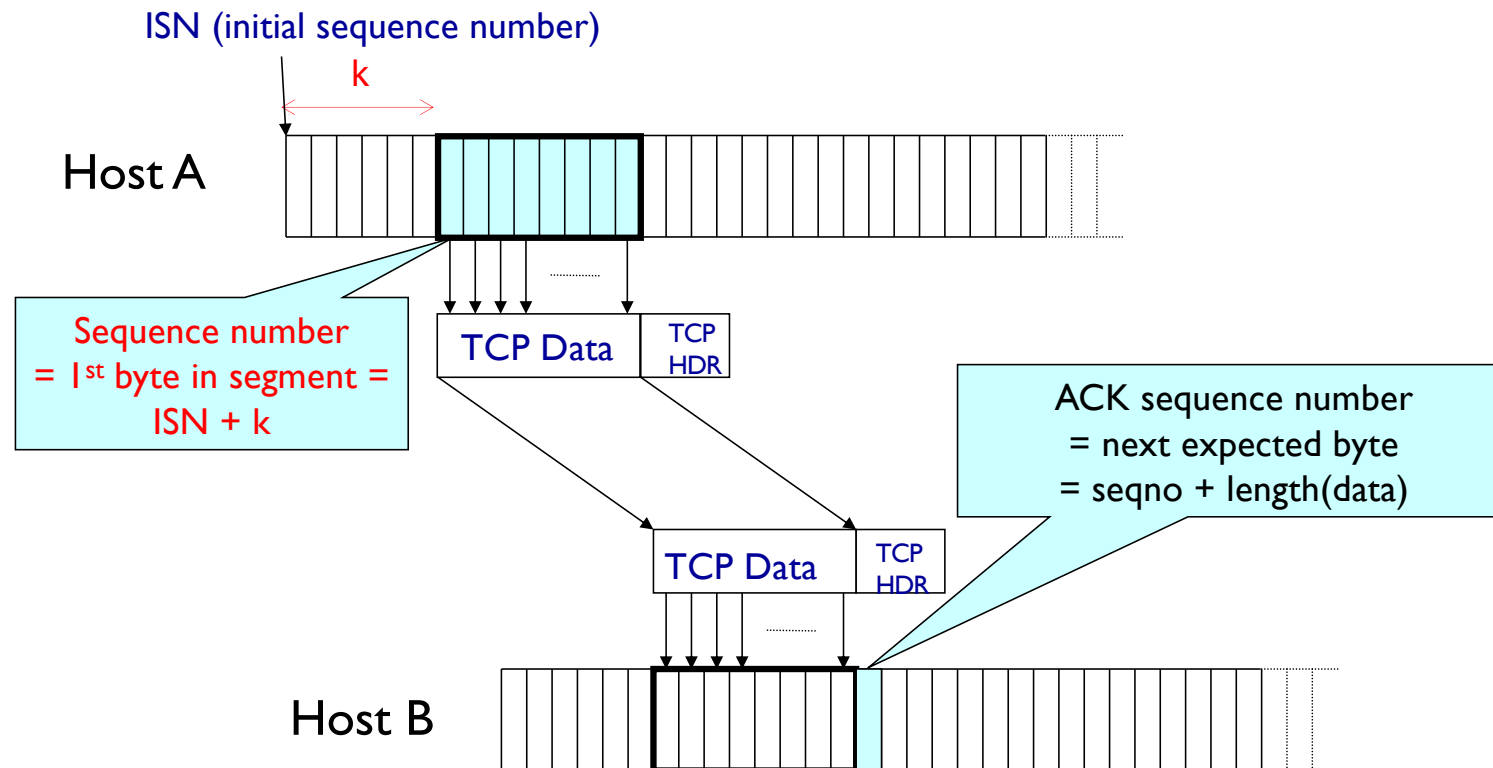
# TCP Header

Acknowledgment gives seqno just beyond highest seqno received **in order**
( "*What Byte is Next* ")

| Source port | | Destination port | |
|---|---|---|---|
| Sequence number | | | |
| Acknowledgment | | | |
| HdrLen | 0 | Flags | Receive window |
| Checksum | | Urgent pointer | |
| Options (variable) | | | |
| Data | | | |
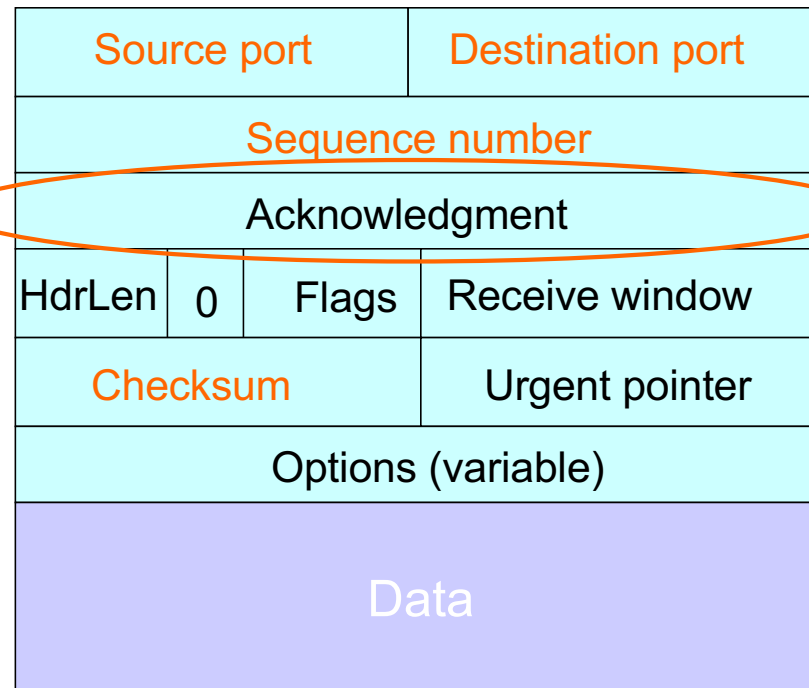
# Why choose random ISN?

❖ Avoids ambiguity with back-to-back connections between same end-points
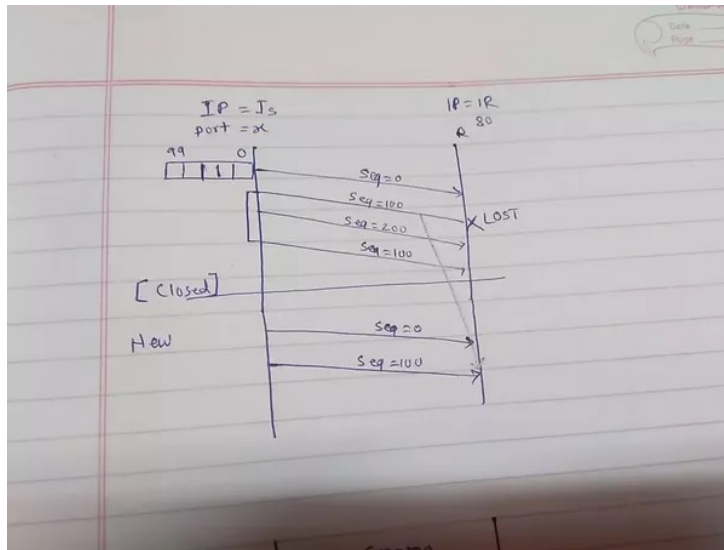


(a) When ISN=0

(b) When ISN is random

❖ Potential security issue if the ISN is known

# What does TCP do?

## Most of our previous tricks, but a few differences

❖ Checksum

❖ Sequence numbers are byte offsets

❖ Receiver sends cumulative acknowledgements (like GBN)

# ACKing and Sequence Numbers
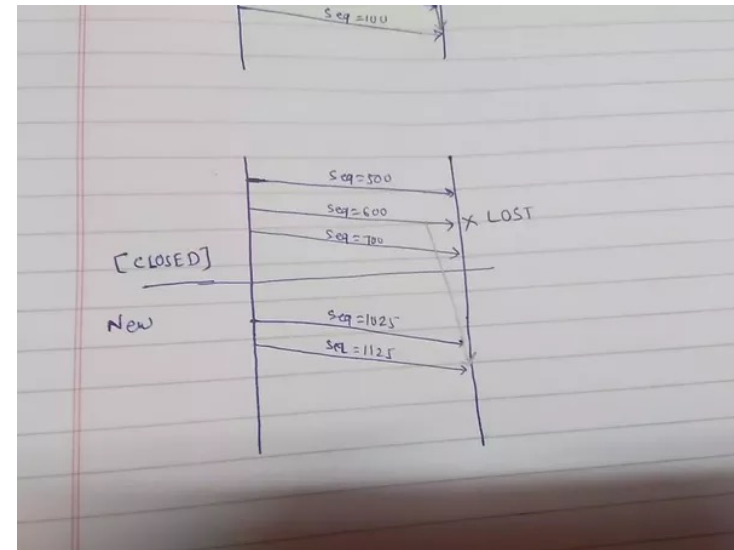
❖ Sender sends packet
  ▪ Data starts with sequence number X
  ▪ Packet contains B bytes [X, X+1, X+2, ....X+B-1]

❖ Upon receipt of packet, receiver sends an ACK
  ▪ If all data prior to X already received:
    • ACK acknowledges X+B (because that is next expected byte)

  ▪ If highest in-order byte received is Y such that (Y+1) < X
    • ACK acknowledges Y+1
    • Even if this has been ACKed before

# An Example



Host A

Host B

ISN=100

Seq=100, Data=50 — Seq 100 to Seq 149 → ACK=150, Received 100-149

Seq=150, Data=50 → ACK=200, Received 150-199

Seq=200, Data=50 → ACK=250, Received 200-249

Seq=250, Data=50 → ACK=300, Received 250-299

Seq=???, Data=50
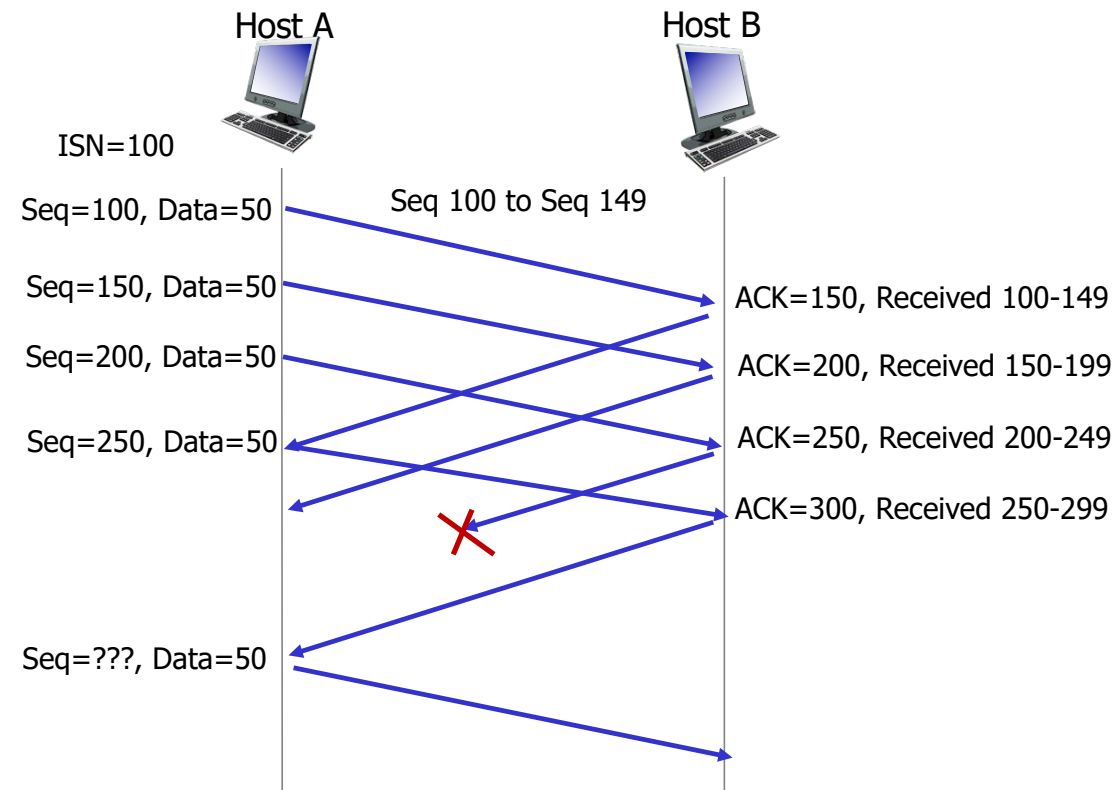
Seq = 300 (new segment)
Since TCP uses cumulative ACKs, the receipt of ACK 300 before a timeout (for seg with sequence number 200) implies the receiver has received all 4 segments sent above

# Another Example



Host A

Host B

ISN=100

Seq=100, Data=50

Seq=150, Data=50

ACK=150, Received 100-149

Seq=200, Data=50

Seq=250, Data=50

ACK=???, Received 200-249

ACK=???, Received 250-299

Both ACKs will be 150 as the receiver has received everything up to 149 in correct sequence. The two out of order segments are buffered.
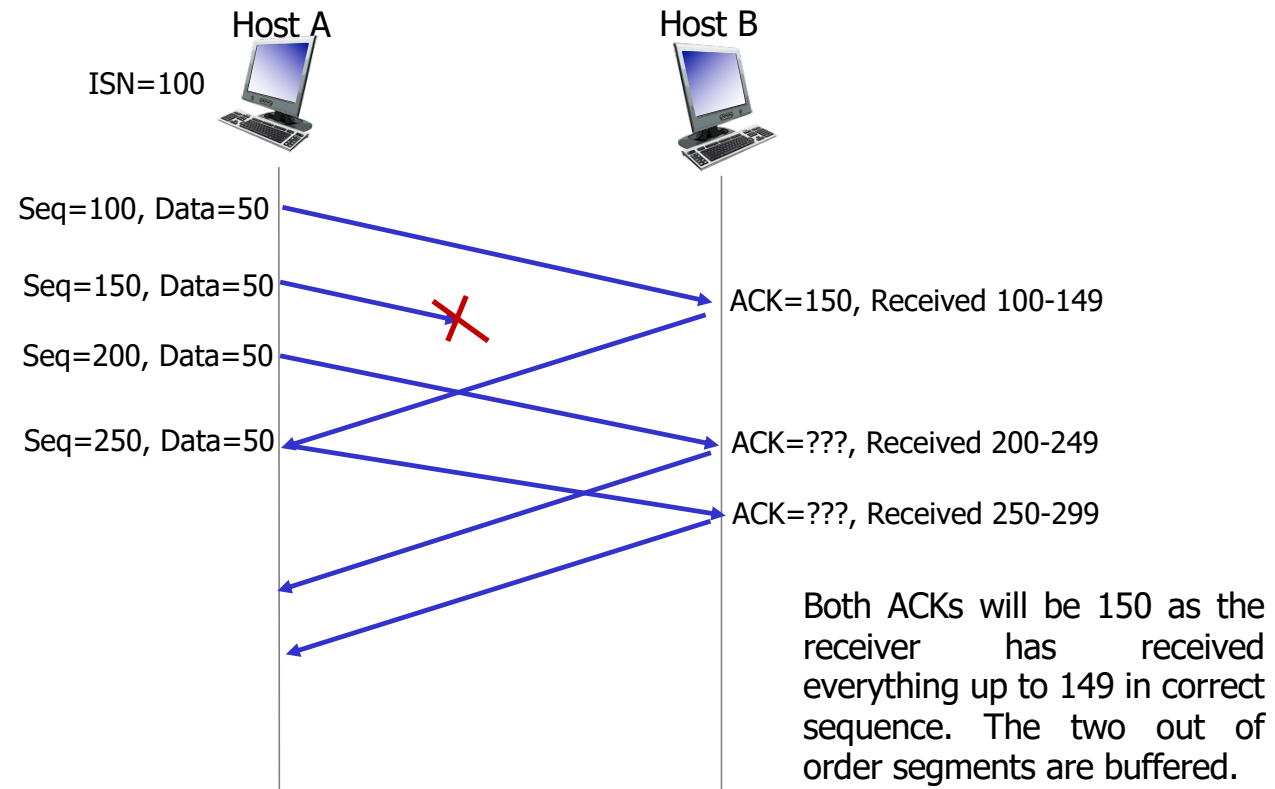
# Piggybacking

❖ So far, we've assumed distinct "sender" and "receiver" roles

❖ Usually both sides of a connection (i.e. the application processes) send some data
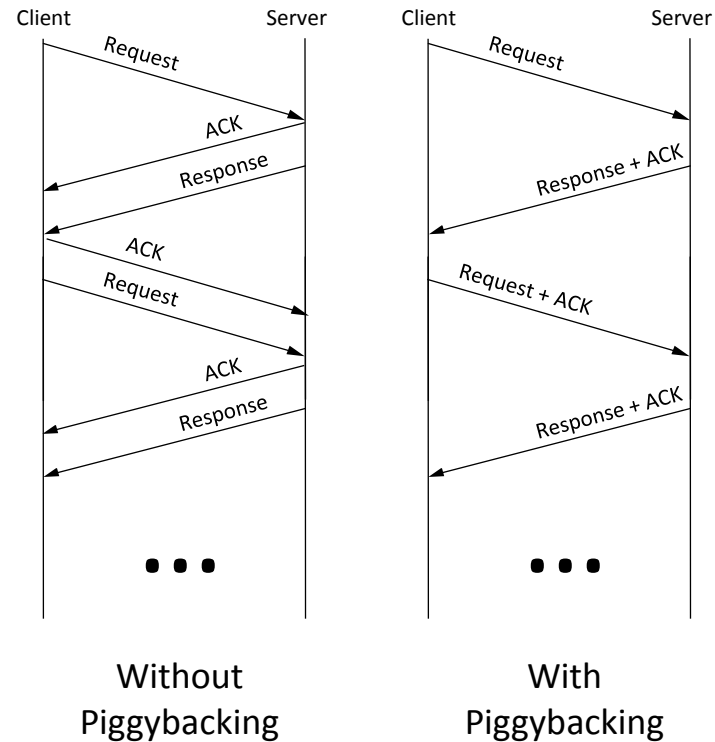


Without Piggybacking      With Piggybacking

# Example



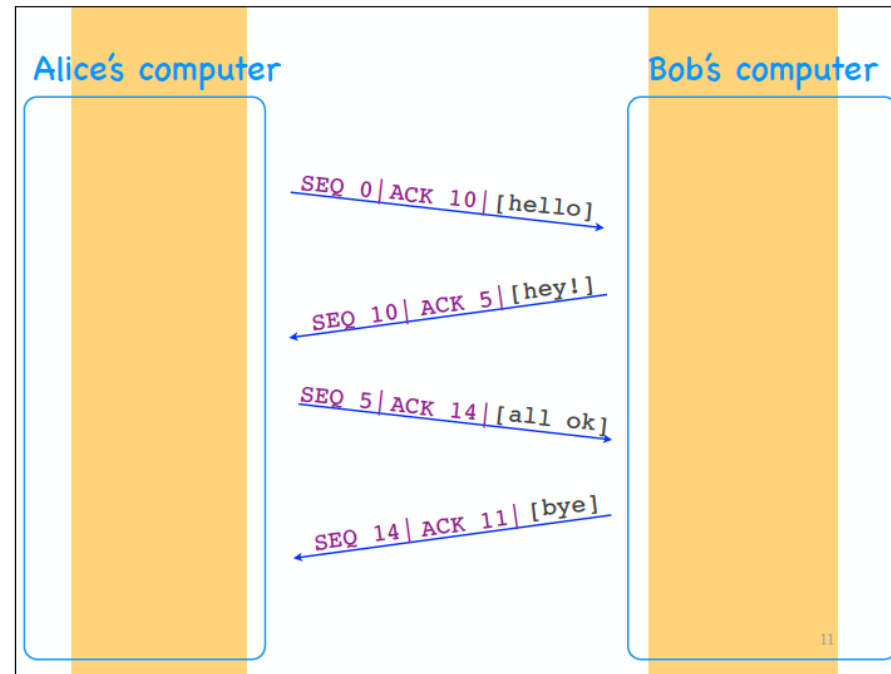Note: Connection establishment not shown. Alice's end point selects the initial sequence number as 0 while Bob's end point selects the initial sequence number as 10

# Another Example
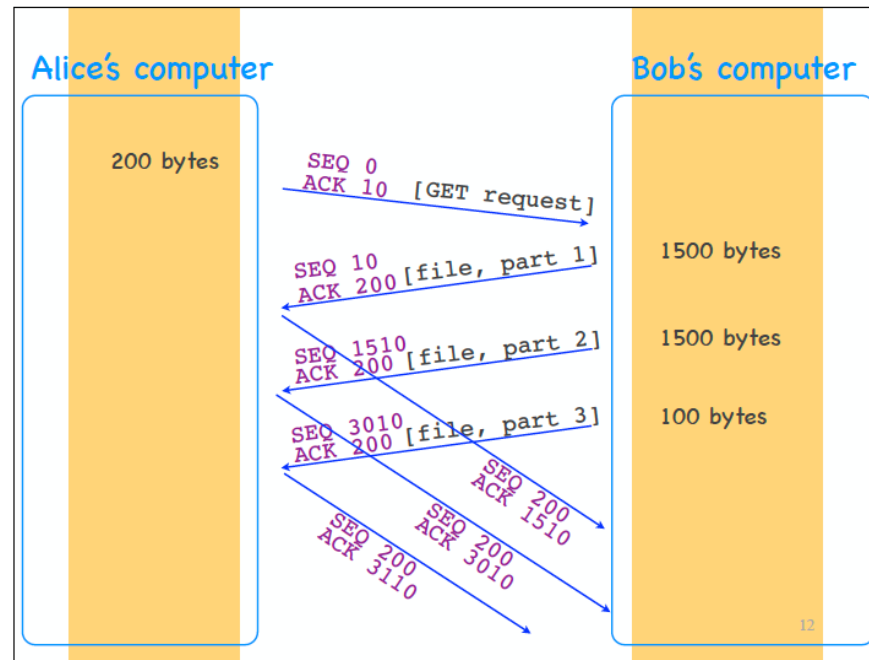


Note: Connection establishment not shown. Alice's end point selects the initial sequence number as 0 while Bob's end point selects the initial sequence number as 10

HTTP response split into 3 segments (MSS = 1500 bytes)

# Quiz

Seq = 101, 2 KBytes of data

ACK = ?
Seq = 1024, 1 KByte of data
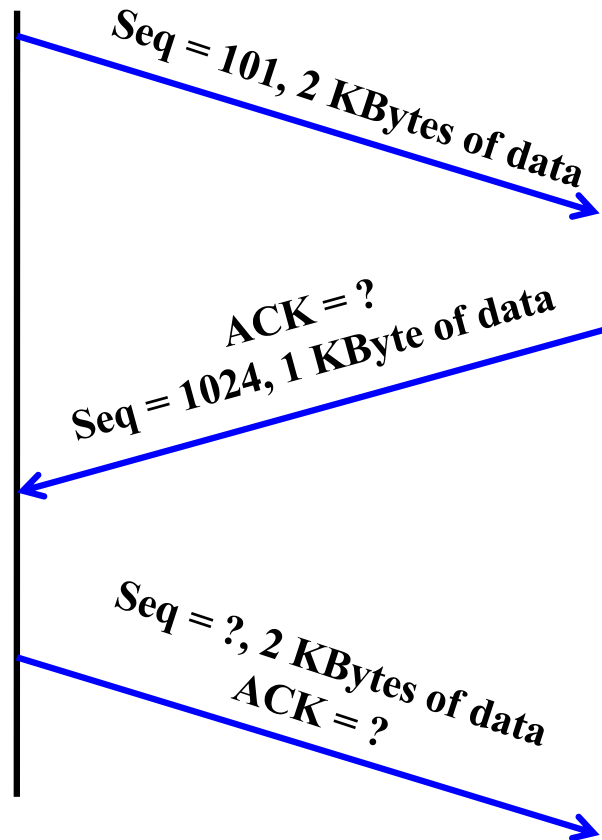
ACK =101 + 2048 = 2149

Seq = ?, 2 KBytes of data
ACK = ?

Seq = 2149

ACK =1024 + 1024 = 2048

# What does TCP do?

Most of our previous tricks, but a few differences

- ❖ Checksum
- ❖ Sequence numbers are byte offsets
- ❖ Receiver sends cumulative acknowledgements (like GBN)
- ❖ Receivers can buffer out-of-sequence packets (like SR)

# Loss with cumulative ACKs

❖ Sender sends packets with 100 bytes and sequence numbers:
   ▪ 100, 200, 300, 400, 500, 600, 700, 800, 900, …

❖ Assume the fifth packet (seq. no. 500) is lost, but no others

❖ 6th packet onwards are buffered

❖ Stream of ACKs will be:
   ▪ 200, 300, 400, 500, 500, 500, 500,…

# What does TCP do?

## Most of our previous tricks, but a few differences

- ❖ Checksum
- ❖ Sequence numbers are byte offsets
- ❖ Receiver sends cumulative acknowledgements (like GBN)
- ❖ Receivers do not drop out-of-sequence packets (like SR)
- ❖ Sender maintains a single retransmission timer (like GBN) and retransmits on timeout (*how much?*)

# TCP round trip time, timeout
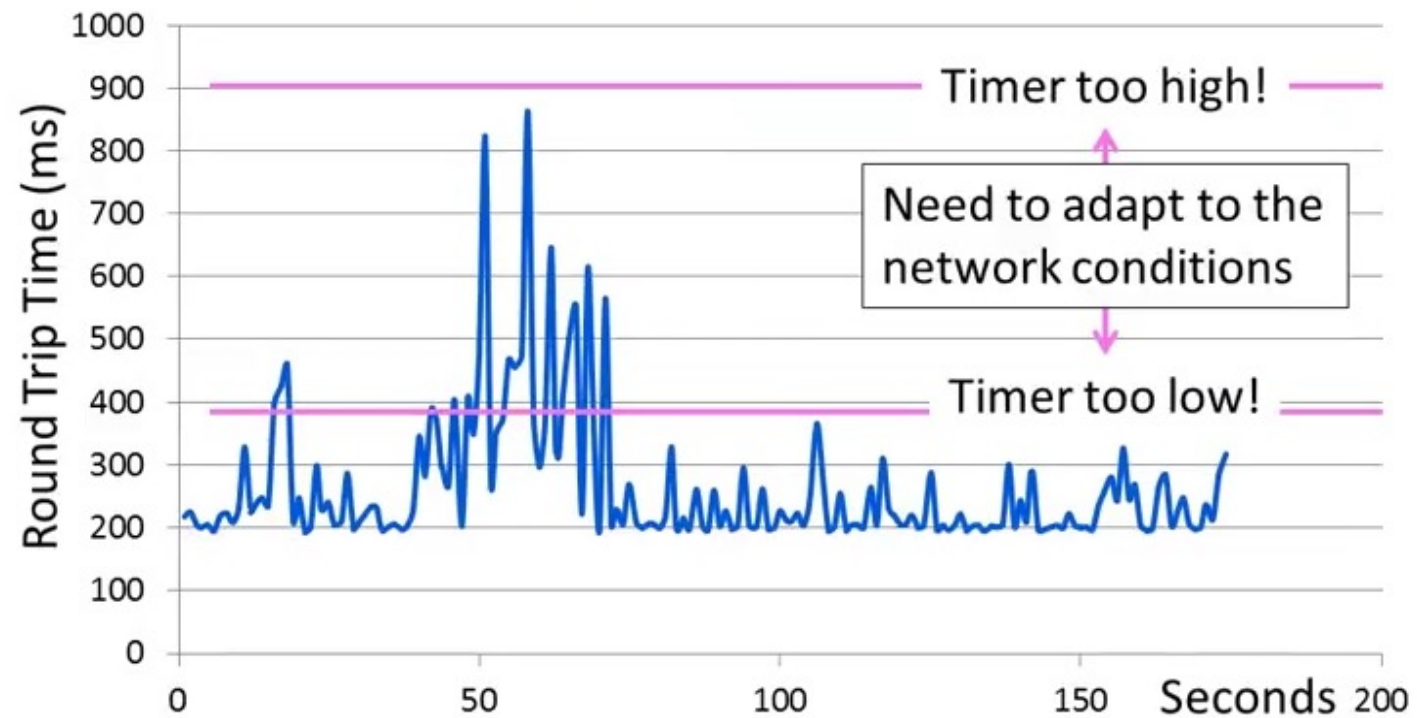
# TCP round trip time, timeout

*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 (EWMA)
- influence of past sample decreases exponentially fast
- typical value: $\alpha$ = 0.125

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr

RTT (milliseconds)

time (seconds)

◆ sampleRTT
■ EstimatedRTT
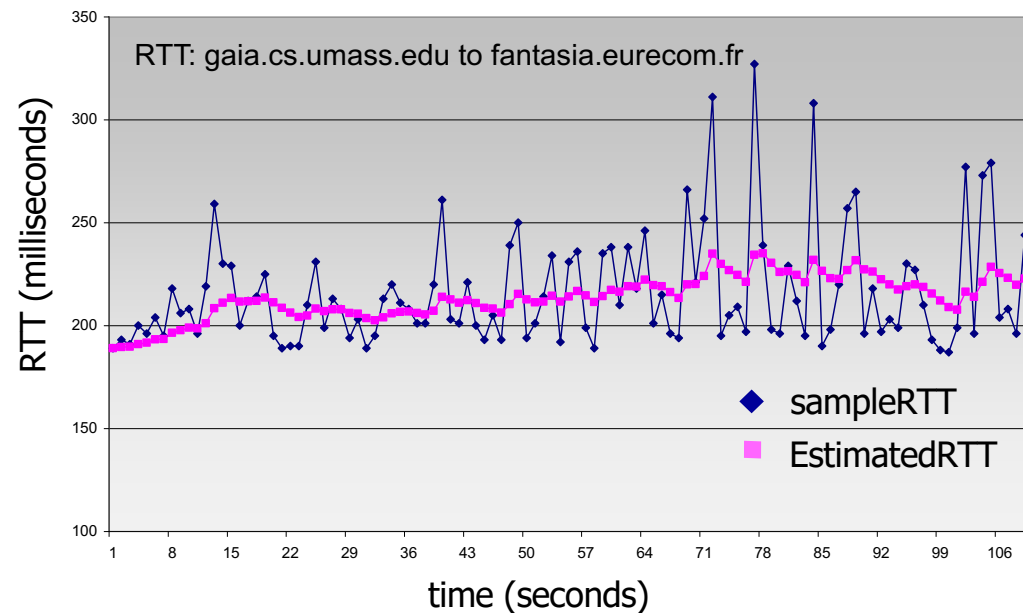
# TCP round trip time, timeout

- timeout interval: `EstimatedRTT` plus "safety margin"
  - large variation in `EstimatedRTT`:  want a larger safety margin

`TimeoutInterval = EstimatedRTT + 4*DevRTT`

estimated RTT          "safety margin"

- `DevRTT`: EWMA of `SampleRTT`  deviation from `EstimatedRTT`:

`DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|`

(typically, β  = 0.25)

Practice Problem:
http://wps.pearsoned.com/ecs_kurose_compnetw_6/216/55463/14198700.cw/index.html

# TCP round trip time, timeout
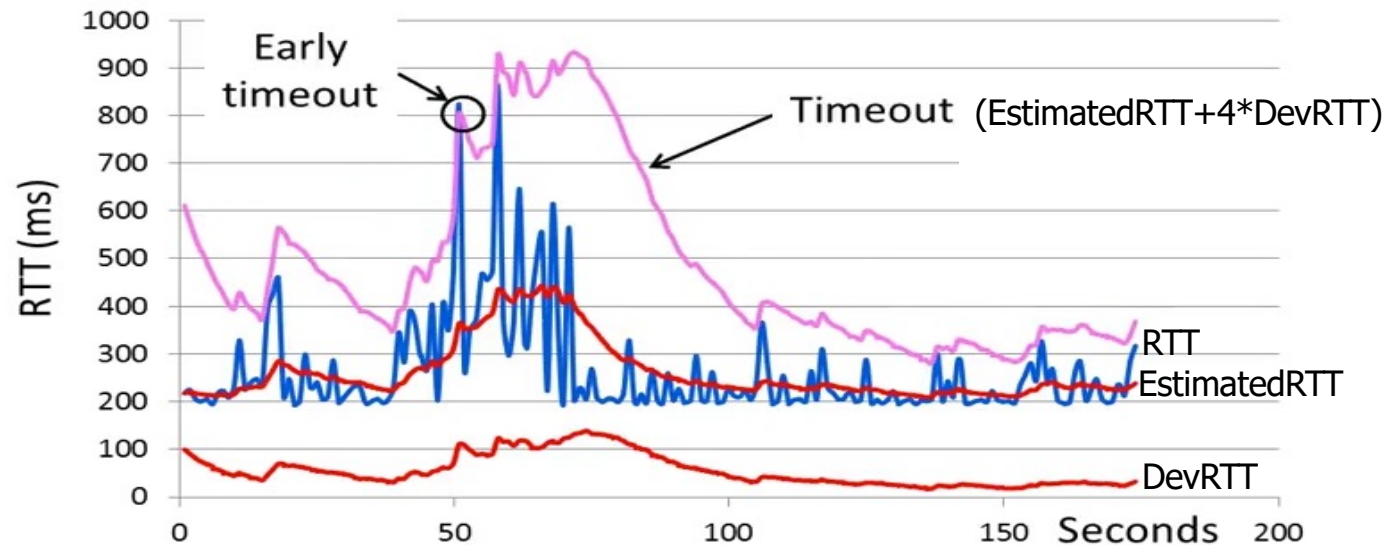


**TimeoutInterval = EstimatedRTT + 4*DevRTT**

estimated RTT

"safety margin"

Figure: Credits Prof David Wetherall UoW

# Why exclude retransmissions in RTT computation?

❖ How do we differentiate between the real ACK, and ACK of the retransmitted packet?

❖ Sender cannot differentiate between the two scenarios shown below
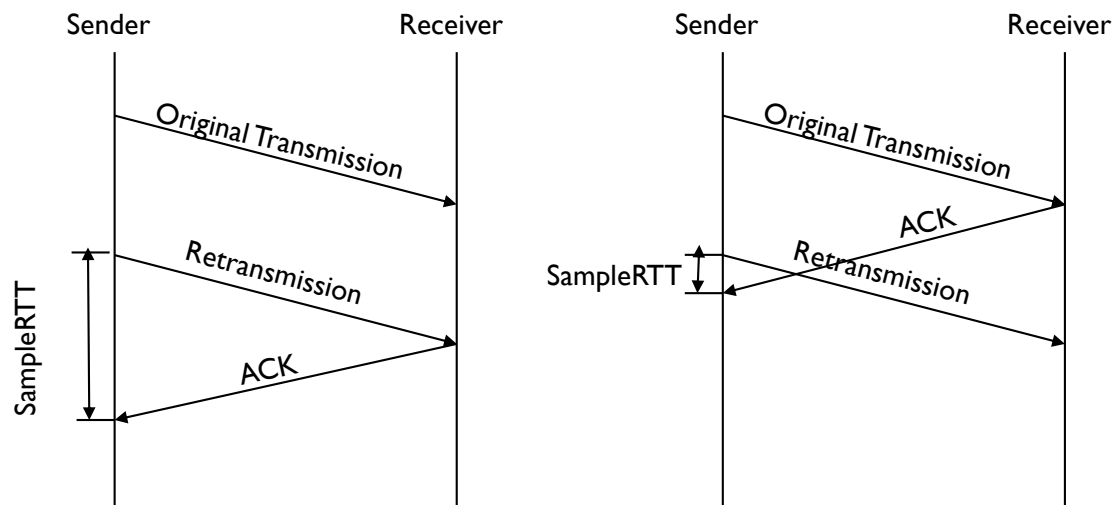
# TCP Sender (simplified)

**event: data received from application**

- 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`

*event: timeout*

- retransmit segment that caused timeout
- restart timer

*event: ACK received*

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

# 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: retransmission scenarios
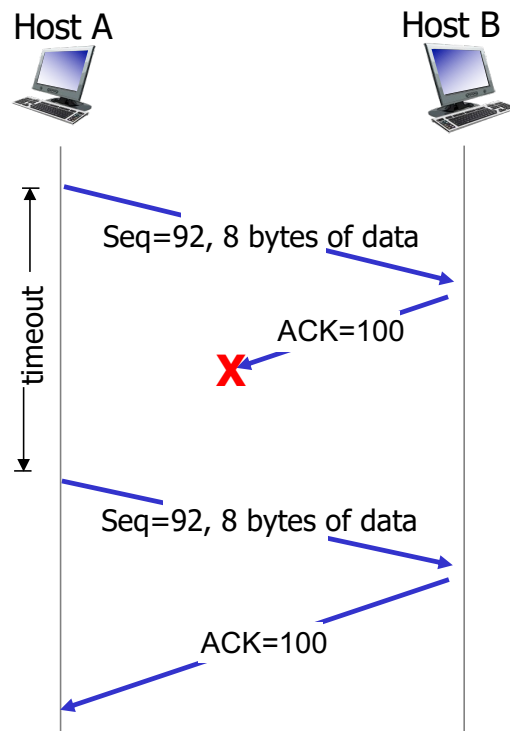


lost ACK scenario
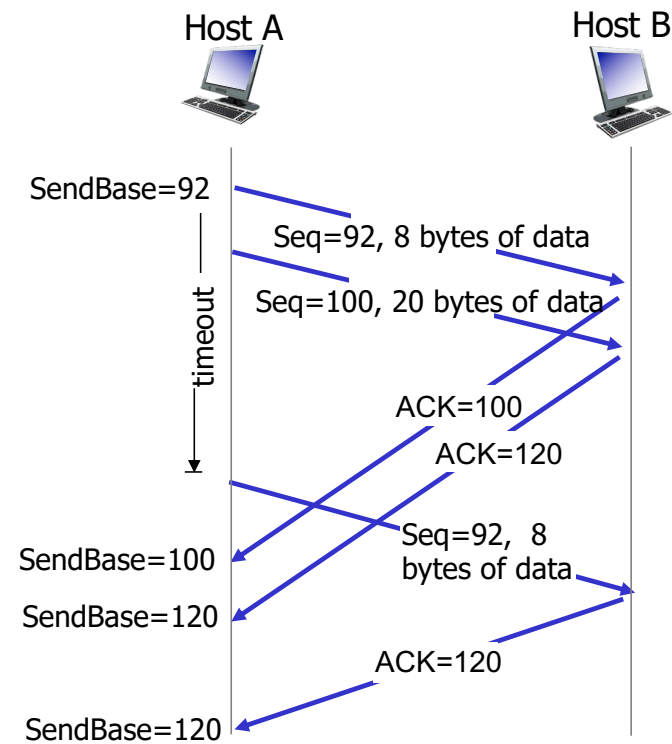
premature timeout

# TCP: retransmission scenarios
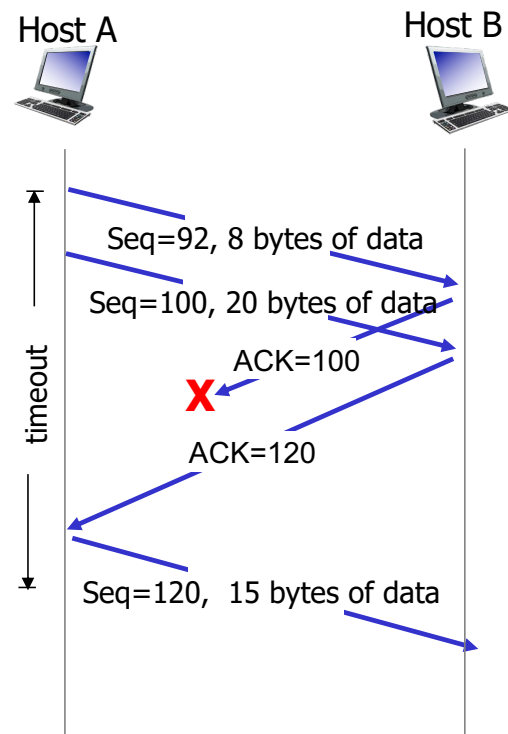


cumulative ACK
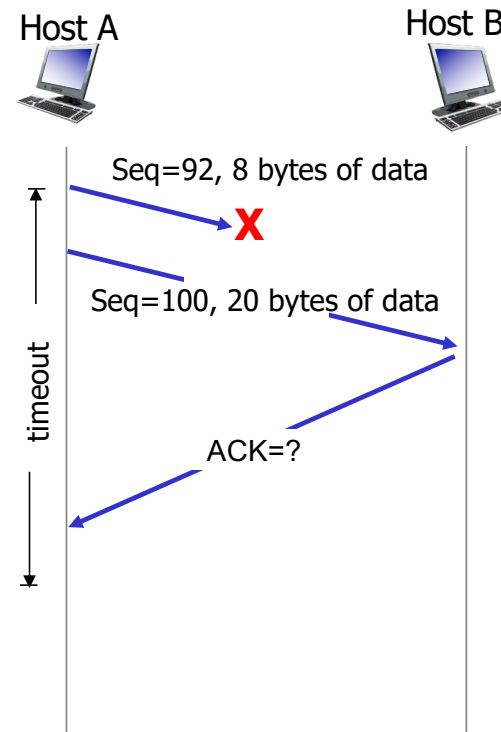
cumulative ACK

# What does TCP do?

## Most of our previous tricks, but a few differences

❖ Checksum

❖ Sequence numbers are byte offsets

❖ Receiver sends cumulative acknowledgements (like GBN)

❖ Receivers may not drop out-of-sequence packets (like SR)

❖ Sender maintains a single retransmission timer (like GBN) and retransmits on timeout

❖ Introduces fast retransmit: optimisation that uses duplicate ACKs to trigger early retransmission

# TCP fast retransmit

**TCP fast retransmit**

if sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #

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

💡 Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!
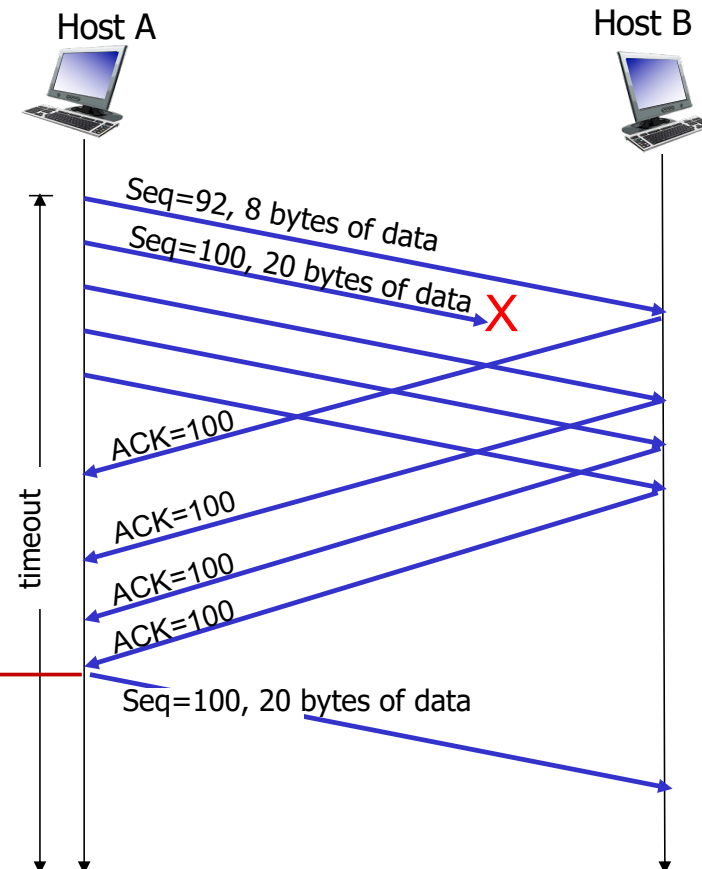
Host A

Host B

Seq=92, 8 bytes of data

Seq=100, 20 bytes of data  X

ACK=100

ACK=100

ACK=100

ACK=100

Seq=100, 20 bytes of data

timeout

# Quiz: TCP Sequence Numbers?

A TCP Sender is about to send a segment of size 100 bytes with sequence number 1234 and ack number 436. What is the highest sequence number up to (and including) which this sender has received from the receiver?

A. 1233
B. 436
C. 435
D. 1334
E. 536

# Quiz: TCP Sequence Numbers?

A TCP Sender is about to send a segment of size 100 bytes with sequence number 1234 and ack number 436. Is it possible that the receiver has received byte number 1335?

A. Yes
B. No

# Quiz: TCP Sequence Numbers?

The following statement is true about the TCP sliding window protocol for implementing reliable data transfer

A. It exclusively uses the ideas of Go-Back-N
B. It exclusively uses the ideas of Selective Repeat
C. It uses a combination of ideas of Go-Back-N and Selective-Repeat
D. It uses none of the ideas of Go-Back-N and Selective-Repeat

# Transport Layer 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

*Q:* What happens if network layer delivers data faster than application layer removes data from socket buffers?

Application removing data from TCP socket buffers

application process

TCP socket receiver buffers

TCP code

Network layer delivering IP datagram payload into TCP socket buffers

IP code

from sender

receiver protocol stack

# TCP flow control

*Q:* What happens if network layer delivers data faster than application layer removes data from socket buffers?

Application removing data from TCP socket buffers

application process

TCP socket receiver buffers

TCP code

Network layer delivering IP datagram payload into TCP socket buffers

IP code

from sender

receiver protocol stack

# TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

Application removing data from TCP socket buffers

receive window
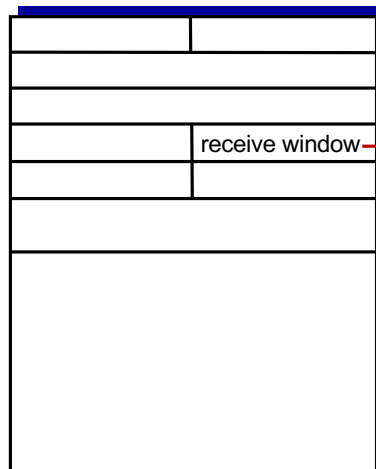
flow control: # bytes receiver willing to accept

application process
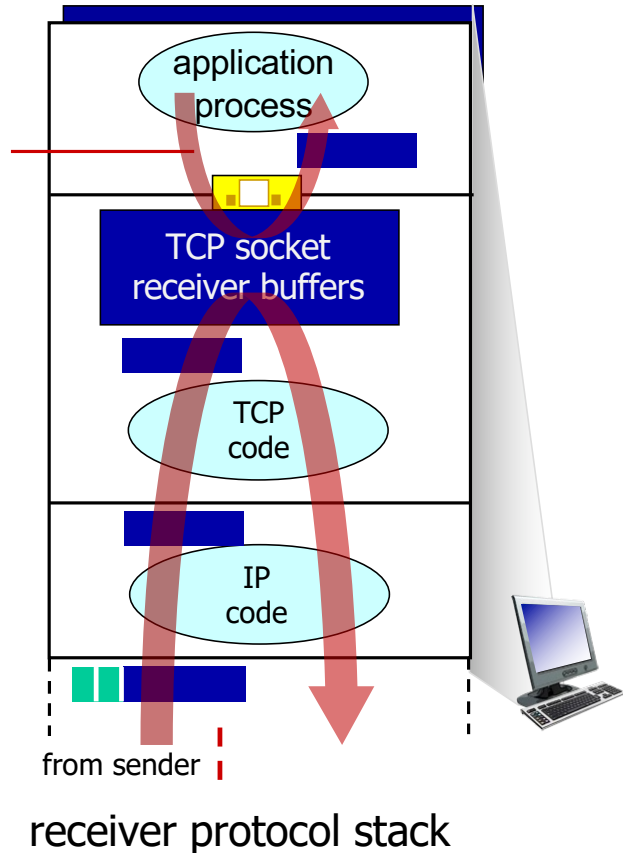
TCP socket receiver buffers

TCP code

IP code

from sender

receiver protocol stack

# TCP flow control

*Q:* What happens if network layer delivers data faster than application layer removes data from socket buffers?

**flow control**

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

Application removing data from TCP socket buffers

application process

TCP socket receiver buffers

TCP code

IP code

from sender

receiver protocol stack
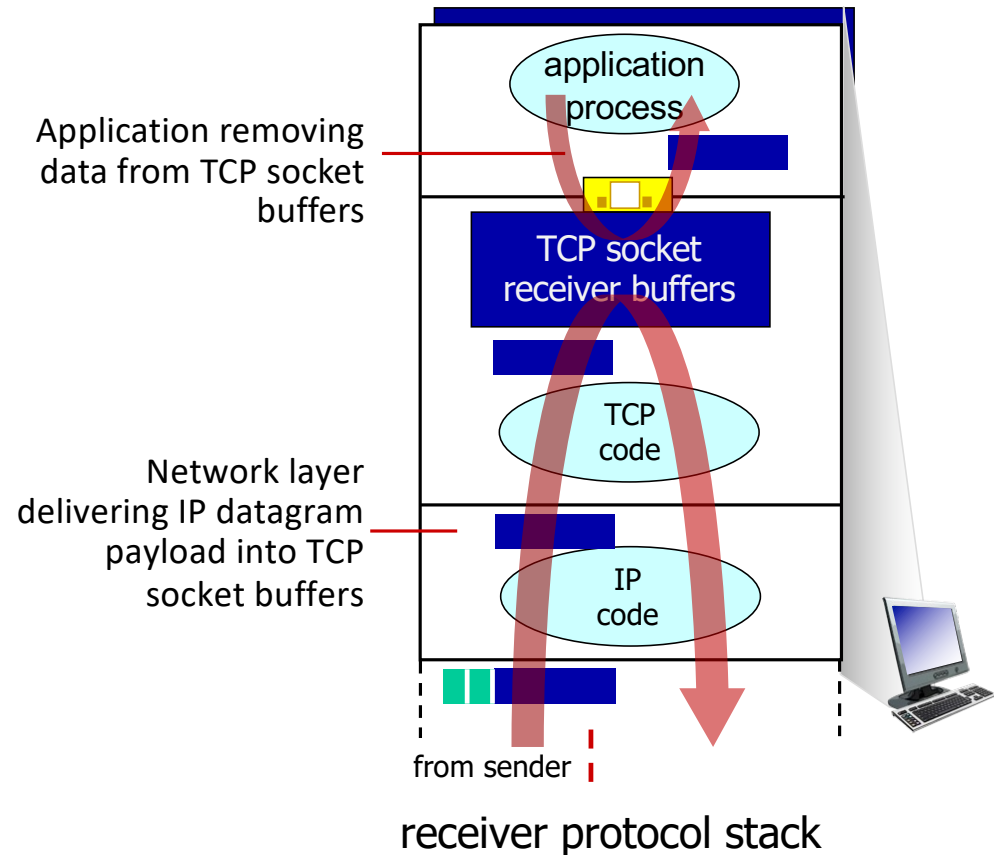
# TCP flow control

- TCP receiver "advertises" free buffer space in **rwnd** field in TCP header
  - **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 received **rwnd**
- guarantees receive buffer will not overflow

to application process

RcvBuffer

buffered data

rwnd

free buffer space

TCP segment payloads

TCP receiver-side buffering
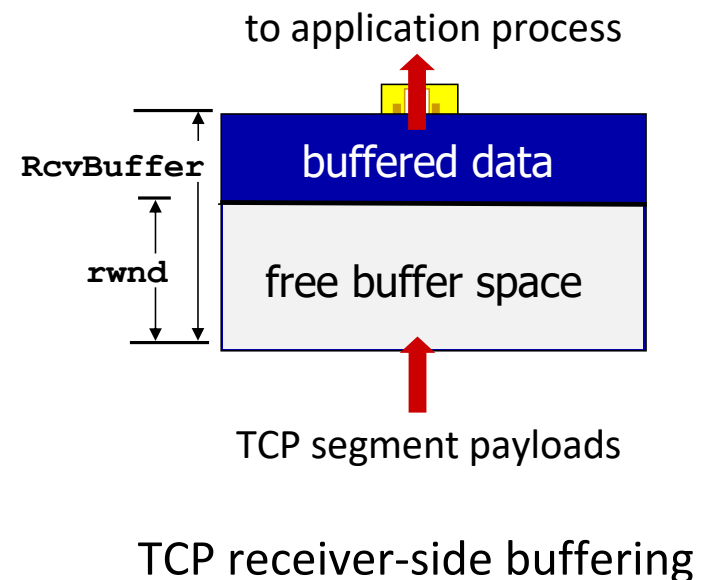
# TCP flow control

- TCP receiver "advertises" free buffer space in **rwnd** field in TCP header

  - **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 received **rwnd**

- guarantees receive buffer will not overflow

flow control: # bytes receiver willing to accept

receive window
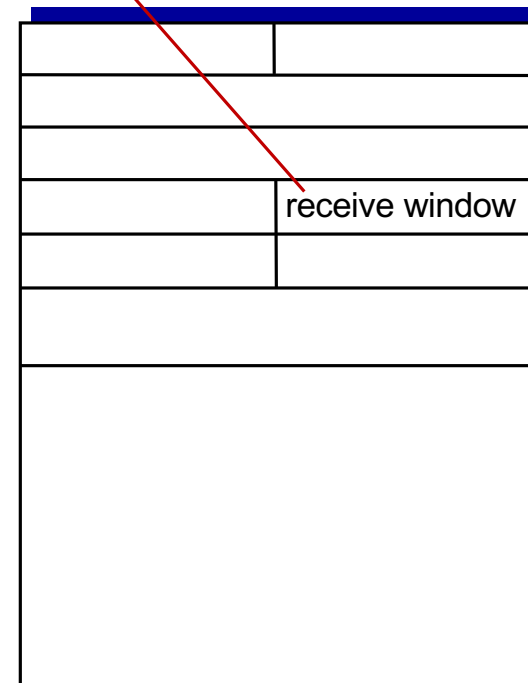
TCP segment format

# TCP flow control

❖ What if `rwnd = 0?`

- Sender would stop sending data

- Eventually the receive buffer would have space when the application process reads some bytes

- But how does the receiver advertise the new `rwnd` to the sender?

❖ Sender keeps sending TCP segments with one data byte to the receiver

❖ These segments are dropped but acknowledged by the receiver with a zero-window size

❖ Eventually when the buffer empties, non-zero window is advertised

# Transport Layer 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 connection management

before exchanging data, sender/receiver "handshake":

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters (e.g., starting seq #s)

application

connection state: ESTAB
connection variables:
    seq # client-to-server
         server-to-client
  **rcvBuffer** size
    at server,client

network

application

connection state: ESTAB
connection Variables:
    seq # client-to-server
         server-to-client
  **rcvBuffer** size
    at server,client
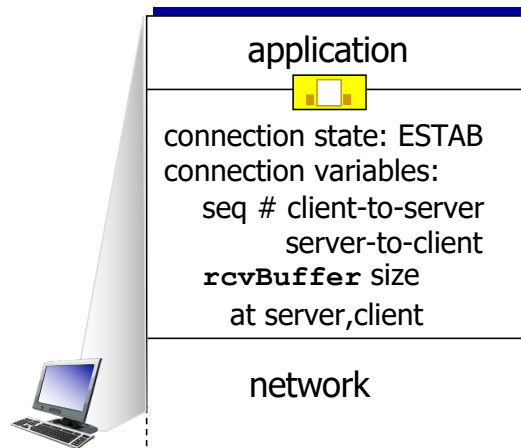
network

```
Socket clientSocket =
  newSocket("hostname","port number");
```
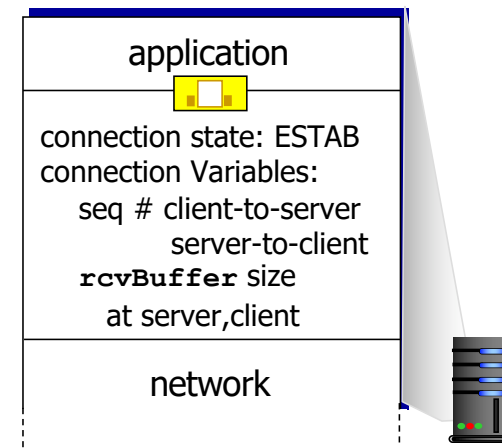
```
Socket connectionSocket =
  welcomeSocket.accept();
```

# Agreeing to establish a connection

2-way handshake:
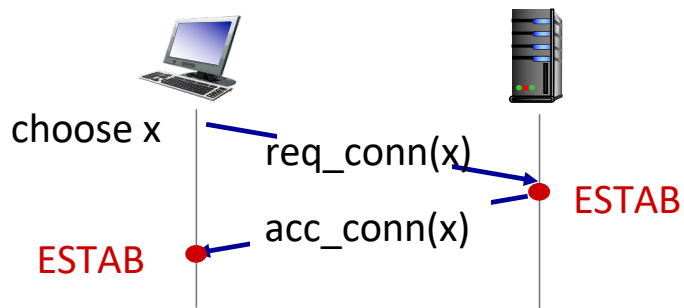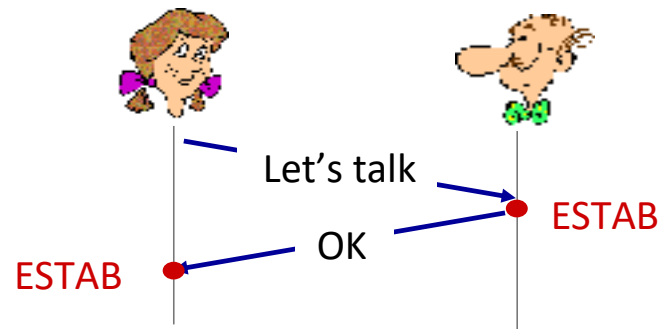


*Q:* will 2-way handshake always work in network?

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

# 2-way handshake scenarios



choose x

req_conn(x)

ESTAB

acc_conn(x)

ESTAB

data(x+1)

accept
data(x+1)

ACK(x+1)

connection
x completes
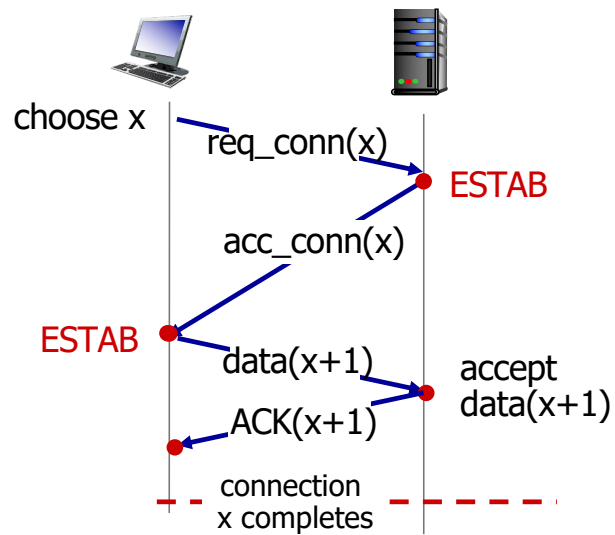
No problem!

# 2-way handshake scenarios



choose x

req_conn(x)

ESTAB

retransmit req_conn(x)

acc_conn(x)

ESTAB

req_conn(x)

connection x completes

client terminates

server forgets x

ESTAB
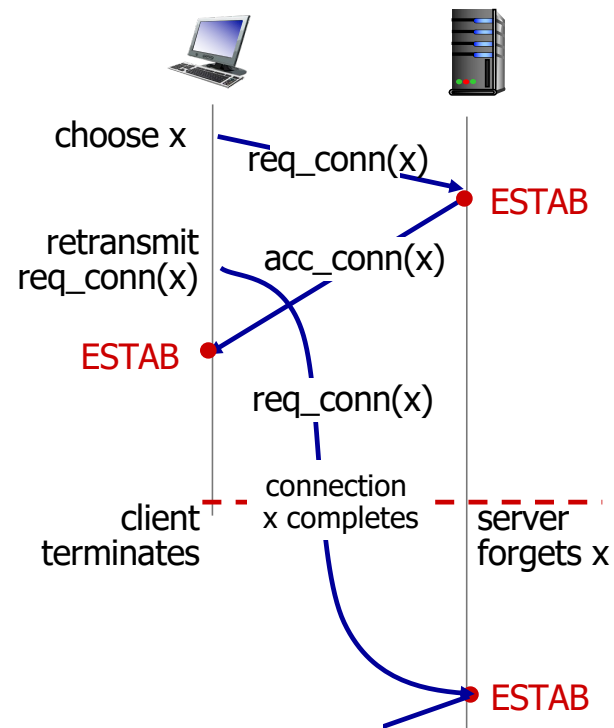
❌ Problem: half open connection! (no client)

# 2-way handshake scenarios



choose x
req_conn(x)
ESTAB

retransmit
req_conn(x)
acc_conn(x)

ESTAB
data(x+1)
accept
data(x+1)

retransmit
data(x+1)

connection
x completes

client
terminates

server
forgets x

req_conn(x)
ESTAB

data(x+1)
accept
data(x+1)

❌ Problem: dup data accepted!

# TCP 3-way handshake

SYN Consumes 1 Sequence No

## Server state

```
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
connectionSocket, addr = serverSocket.accept()
```

## Client state

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```
LISTEN
```
clientSocket.connect((serverName,serverPort))
```

choose init seq num, x
send TCP SYN msg

SYNSENT

SYNbit=1, Seq=x

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

LISTEN

SYN RCVD

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

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

ESTAB

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

received ACK(y)
indicates client is live

ESTAB

# A human 3-way handshake protocol

# TCP 3-way handshake: Partial state machine



**closed**

welcomeSocket.listen(1);
―――――――――――
Λ

SYN(x)
―――――――――――
connectionSocket, addr =
welcomeSocket.accept();

SYNACK(seq=y,ACKnum=x+1)

**listen**

clientSocket.connect((serverName,
serverPort))
―――――――――――
SYN(seq=x)

**SYN rcvd**

**SYN sent**

SYNACK(seq=y,ACKnum=x+1)
―――――――――――
ACK(ACKnum=y+1)

**ESTAB**

ACK(ACKnum=y+1)
―――――――――――
Λ
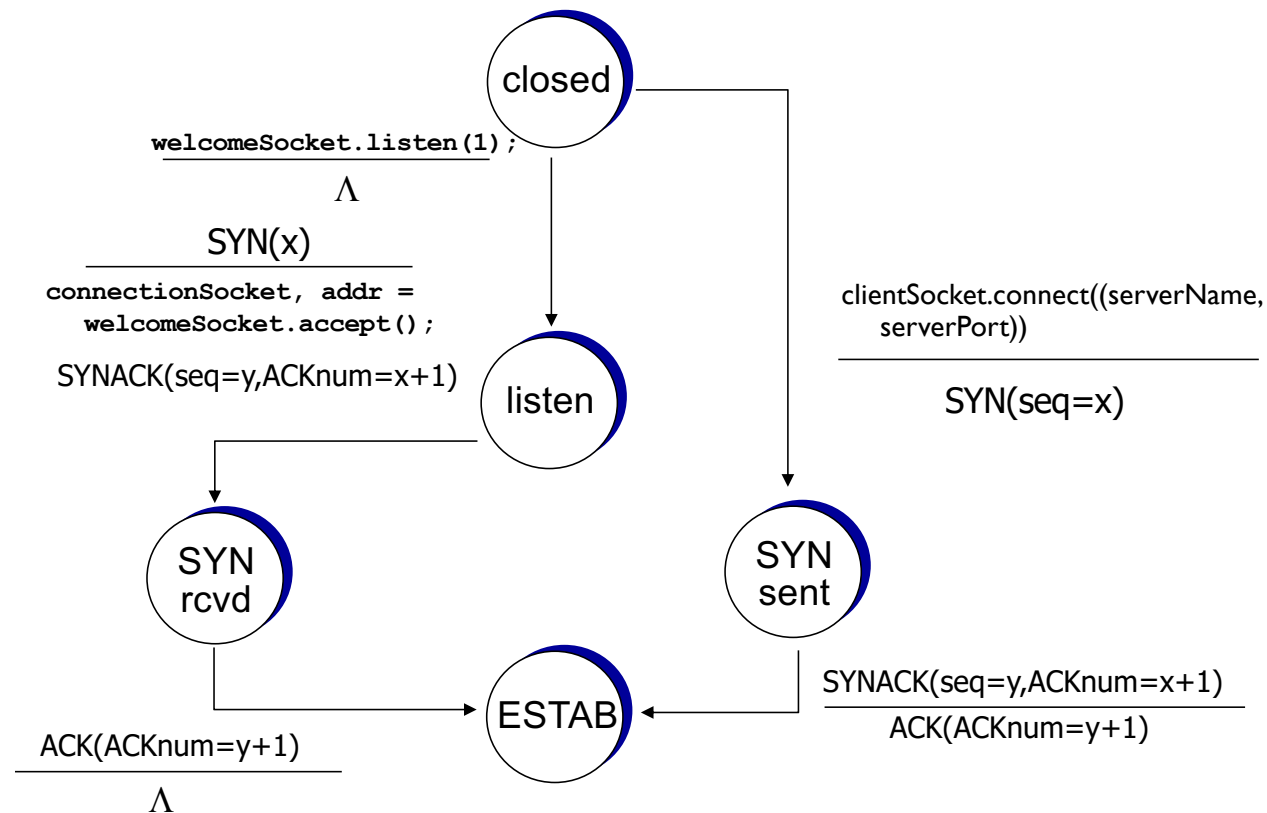
# What if the SYN Packet Gets Lost?

❖ Suppose the SYN packet gets lost
  ▪ Packet is lost inside the network, or:
  ▪ Server discards the packet (e.g., it's too busy)

❖ Eventually, no SYN-ACK arrives
  ▪ Sender sets a timer and waits for the SYN-ACK
  ▪ … and retransmits the SYN if needed

❖ How should the TCP sender set the timer?
  ▪ Sender has no idea how far away the receiver is
  ▪ Hard to guess a reasonable length of time to wait
  ▪ **SHOULD** (RFCs 1122,2988) use default of 3 second,
    RFC 6298 use default of 1 second

# 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

# Normal Termination, One at a Time

FIN Consumes 1 Sequence No

*client state*

*server state*

ESTAB

`clientSocket.close()`

FIN_WAIT_1    can no longer send but can receive data

FINbit=1, seq=x

ESTAB

CLOSE_WAIT

ACKbit=1; ACKnum=x+1

FIN_WAIT_2    wait for server close

can still send data

FINbit=1, seq=y

LAST_ACK

TIMED_WAIT

can no longer send data

ACKbit=1; ACKnum=y+1

timed wait for 2*max segment lifetime

CLOSED

CLOSED

TIMED_WAIT: Can retransmit ACK if last ACK is lost

# Normal Termination, Both Together



client state

ESTAB

`clientSocket.close()`

FIN_WAIT_1

can no longer send but can receive data

FINbit=1, seq=x

ACKbit=1; ACKnum=x+1

TIMED_WAIT

wait for server close

FINbit=1, seq=y

timed wait for 2*max segment lifetime

ACKbit=1; ACKnum=y+1

CLOSED

server state

ESTAB

CLOSE_WAIT

FIN + ACK together

LAST_ACK

can no longer send data

CLOSED

# Simultaneous Closure



client state

ESTAB

FIN_WAIT_1    `clientSocket.close()`

can no longer send but can receive data

wait for server close

CLOSING

TIMED_WAIT

CLOSED

FINbit=1, seq=x

FINbit=1, seq=y

ACKbit=1, ACKnum=x+1

ACKbit=1, ACKnum=y+1

can no longer send data

Send Ack

server state

ESTAB

FIN_WAIT_1

CLOSING

TIMED_WAIT

CLOSED

# Abrupt Termination



- ❖ A sends a RESET (**RST**) to B
  - ▪ E.g., because application process on A crashed
- ❖ That's it
  - ▪ B does not ack the **RST**
  - ▪ Thus, **RST** is not delivered reliably
  - ▪ And: any data in flight is lost
  - ▪ But: if B sends anything more, will elicit another **RST**

# TCP SYN Attack (SYN flooding)

- ❖ Miscreant creates a fake SYN packet
  - ▪ Destination is IP address of victim host (usually some server)
  - ▪ Source is some spoofed IP address
- ❖ Victim host on receiving creates a TCP connection state i.e allocates buffers, creates variables, etc and sends SYN ACK to the spoofed address (half-open connection)
- ❖ ACK never comes back
- ❖ After a timeout connection state is freed
- ❖ However for this duration the connection state is unnecessarily created
- ❖ Further miscreant sends large number of fake SYNs
  - ▪ Can easily overwhelm the victim
- ❖ Solutions:
  - ▪ Increase size of connection queue
  - ▪ Decrease timeout wait for the 3-way handshake
  - ▪ Firewalls: list of known bad source IP addresses
  - ▪ TCP SYN Cookies (explained on next slide)

# TCP SYN Cookie

- ❖ On receipt of SYN, server does not create connection state
- ❖ It creates an initial sequence number (*init_seq*) that is a hash of source & dest IP address and port number of SYN packet (secret key used for hash)
  - ▪ Replies back with SYN ACK containing *init_seq*
  - ▪ Server does not need to store this sequence number
- ❖ If original SYN is genuine, an ACK will come back
  - ▪ Same hash function run on the same header fields to get the initial sequence number (*init_seq*)
  - ▪ Checks if the ACK is equal to (*init_seq*+1)
  - ▪ Only create connection state if above is true
- ❖ If fake SYN, no harm done since no state was created

http://etherealmind.com/tcp-syn-cookies-ddos-defence/

# Quiz: TCP Connection Management?

Assume that one end of a TCP connection selects an initial sequence number 120. The first TCP segment containing data sent by this end point will have a sequence number of _____

A. 120

B. 121

C. 122

D. 128

E. 0

# Quiz: TCP Connection Management

Assume that one end point of the TCP connection sends a FIN segment. If it never receives an ACK, what should it do?

A. Assume that the connection is closed and do nothing

B. Retransmit the FIN

C. Transmit an ACK

D. Start crying

# Transport Layer: 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
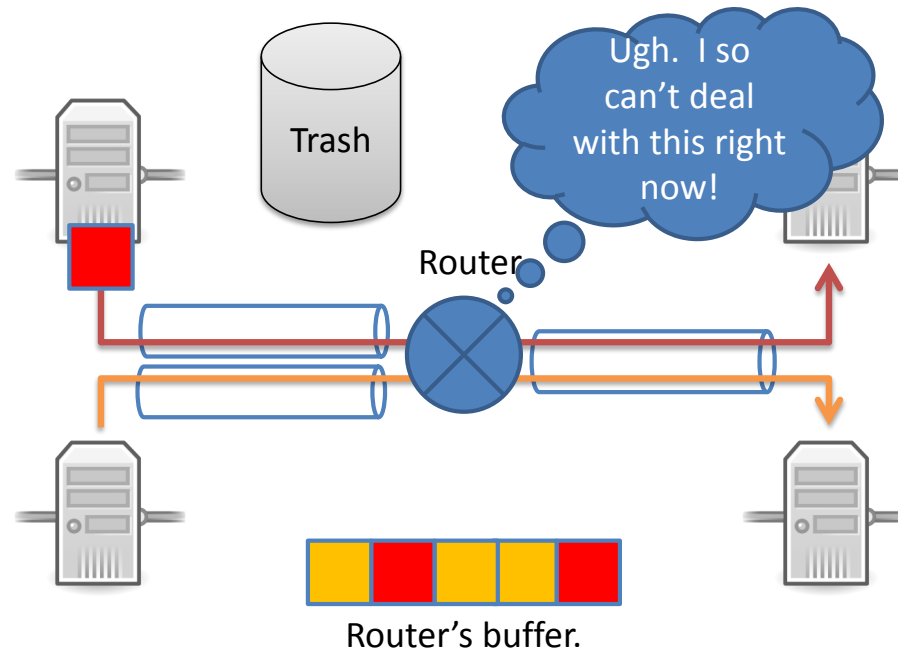
3.6 principles of congestion control

3.7 TCP congestion control

# Principles of congestion control

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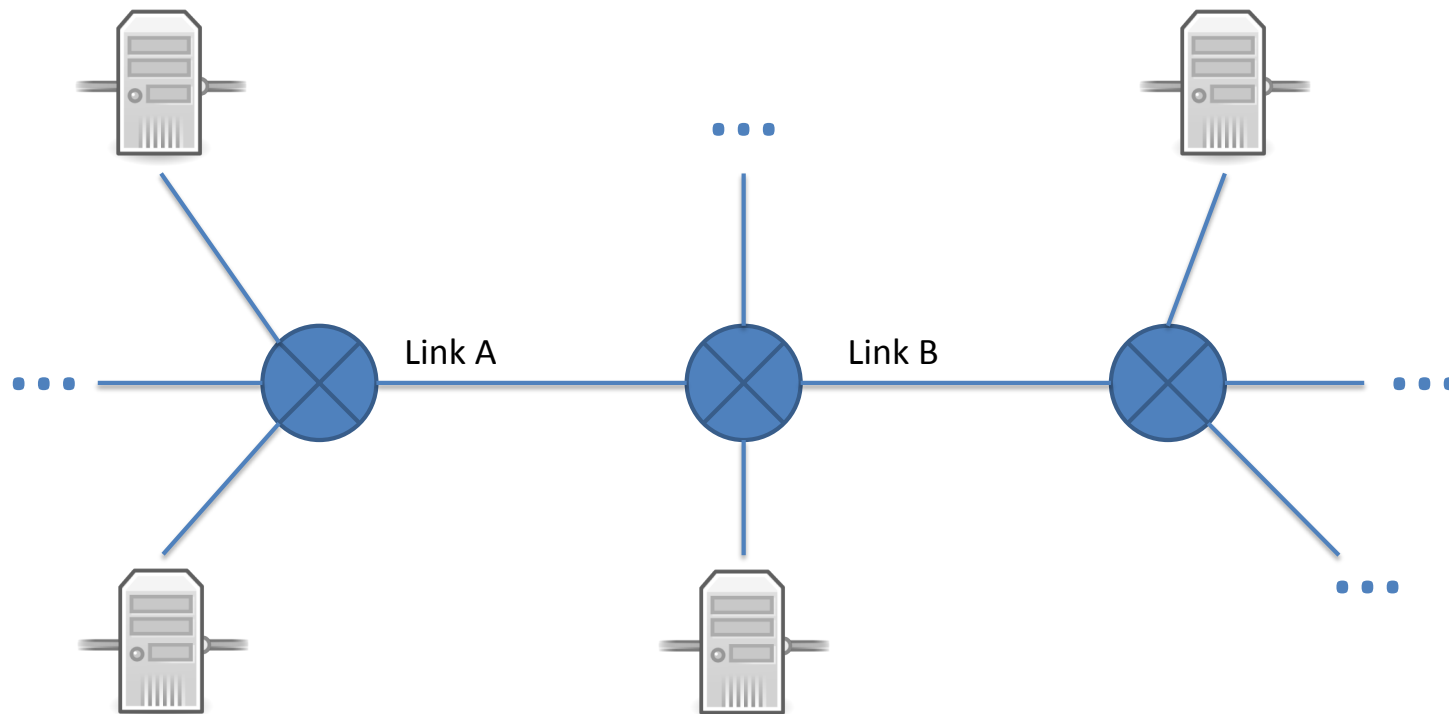
- ❖ informally: "too many sources sending too much data too fast for *network* to handle"
- ❖ different from flow control!
- ❖ manifestations:
  - ▪ lost packets (buffer overflow at routers)
  - ▪ long delays (queueing in router buffers)
- ❖ a top-10 problem!

# Congestion



Router's buffer.

Incoming rate is faster than
outgoing link can support.

# Congestion Collapse



Link A

Link B

# Congestion Collapse



Link A

Link B

One sender starts,
but there's still
capacity at link A.

$S_1$

# Congestion Collapse



S₂

Another sender starts up.  Link A is showing slight delay, but still doing ok.

Link A

Link B

S₁

# Congestion Collapse



S₂

Unrelated traffic
passes through and
congests link B.

Link A

Link B

S₁

# Congestion Collapse

S2's traffic is being dropped at Link B, so it starts retransmitting on top of what it was sending.



S2

S1
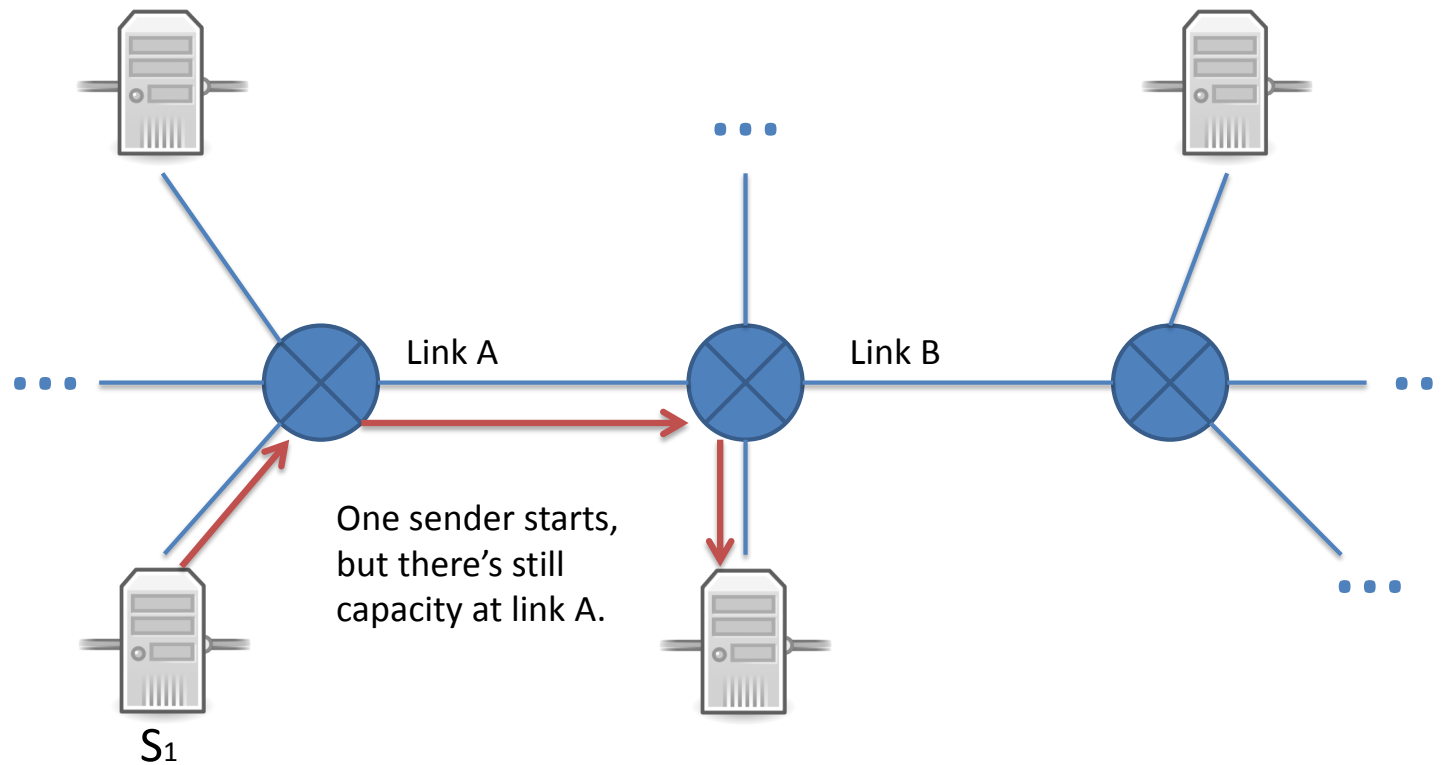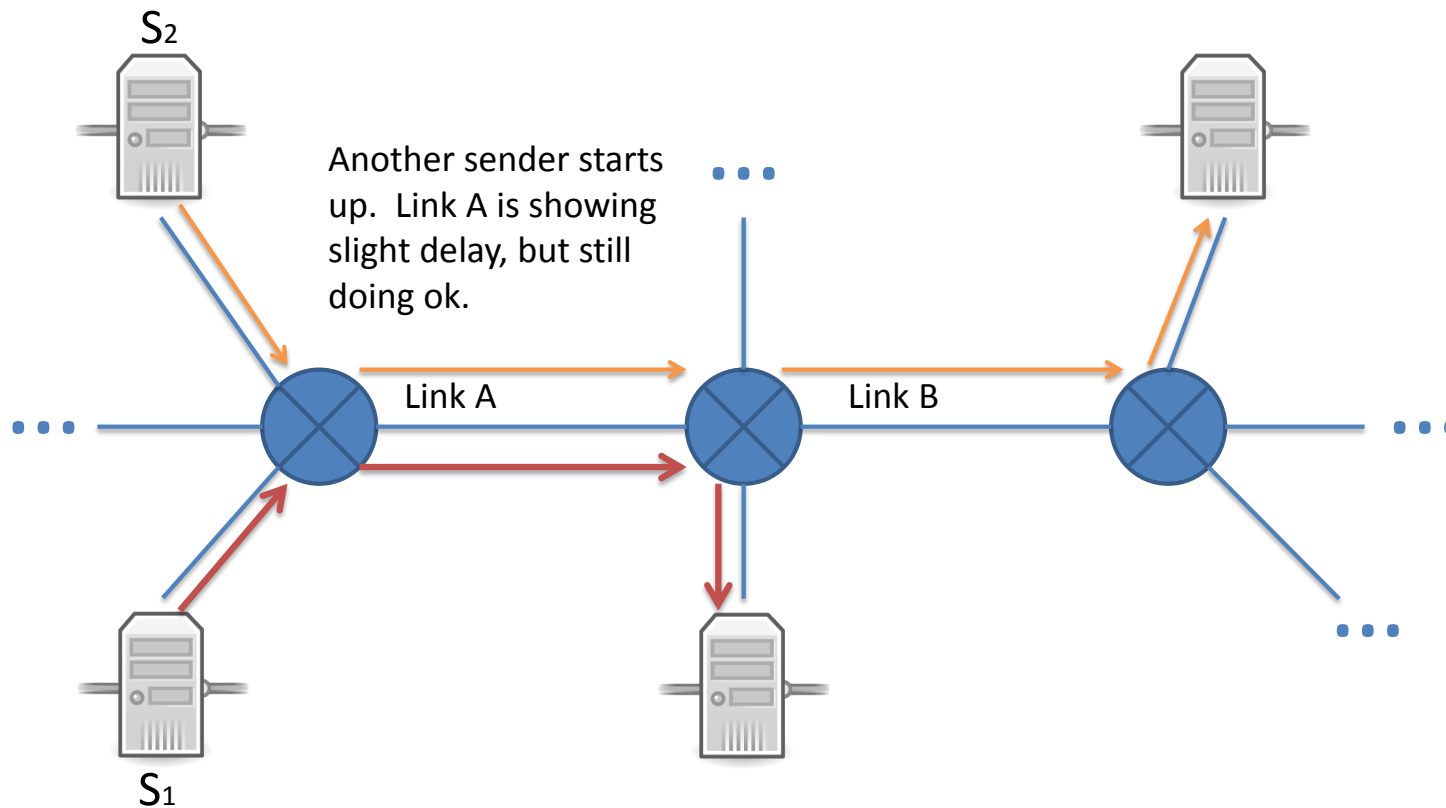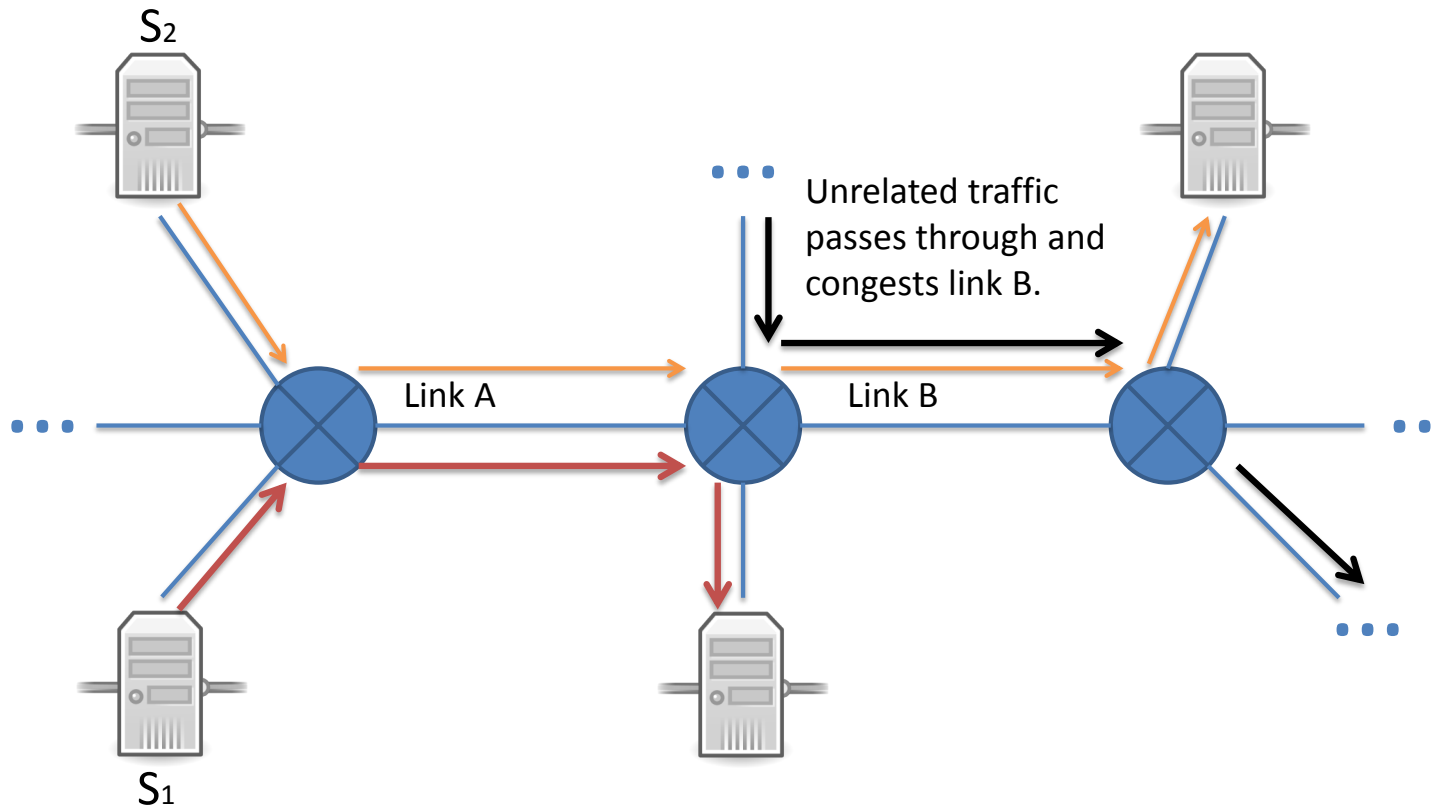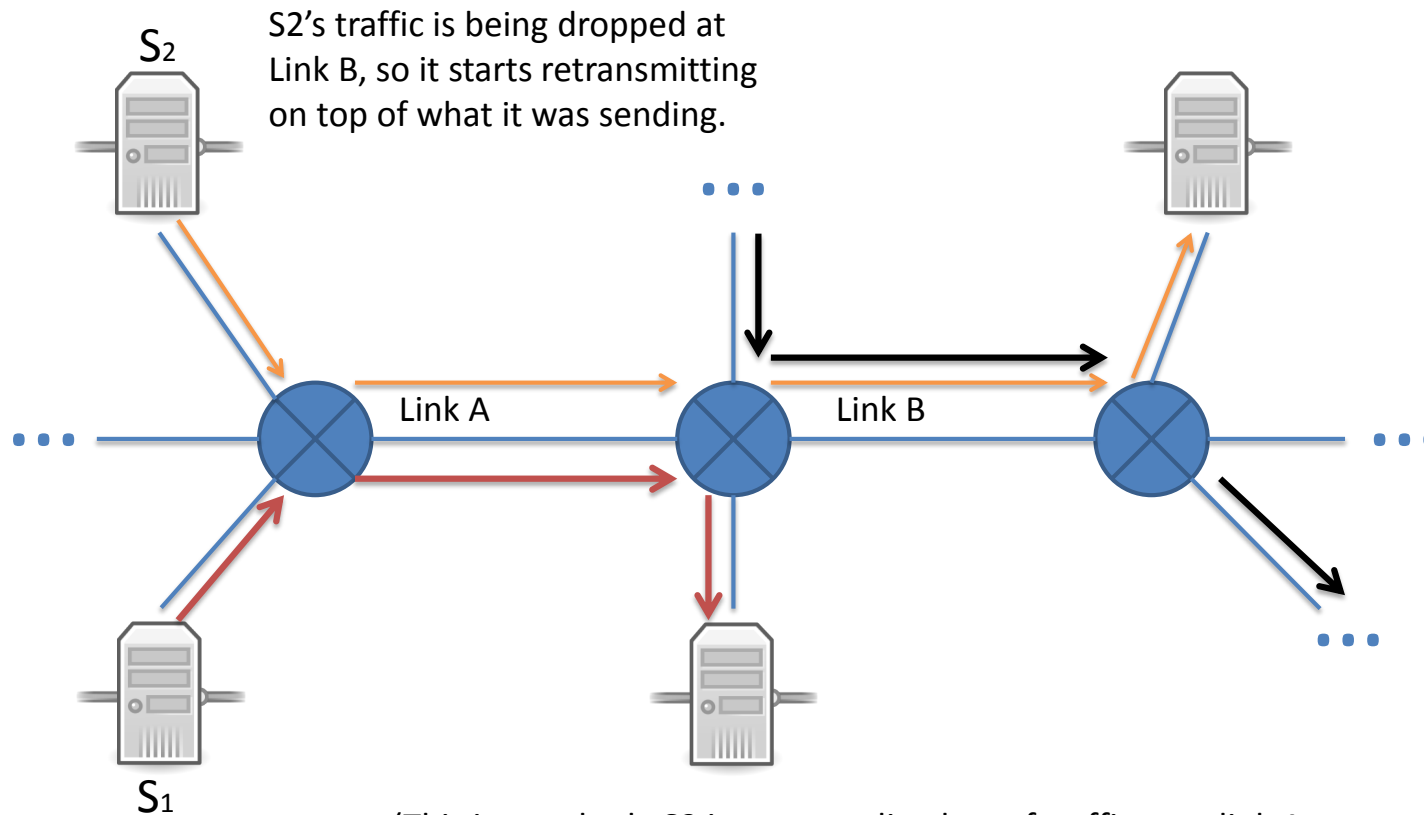
Link A

Link B

(This is very bad. S2 is now sending lots of traffic over link A that has no hope of crossing link B.)

74

# Congestion Collapse



S₂

S₁

Link A

Link B

Increased traffic from S2 causes Link A to become congested. S1 starts retransmitting.
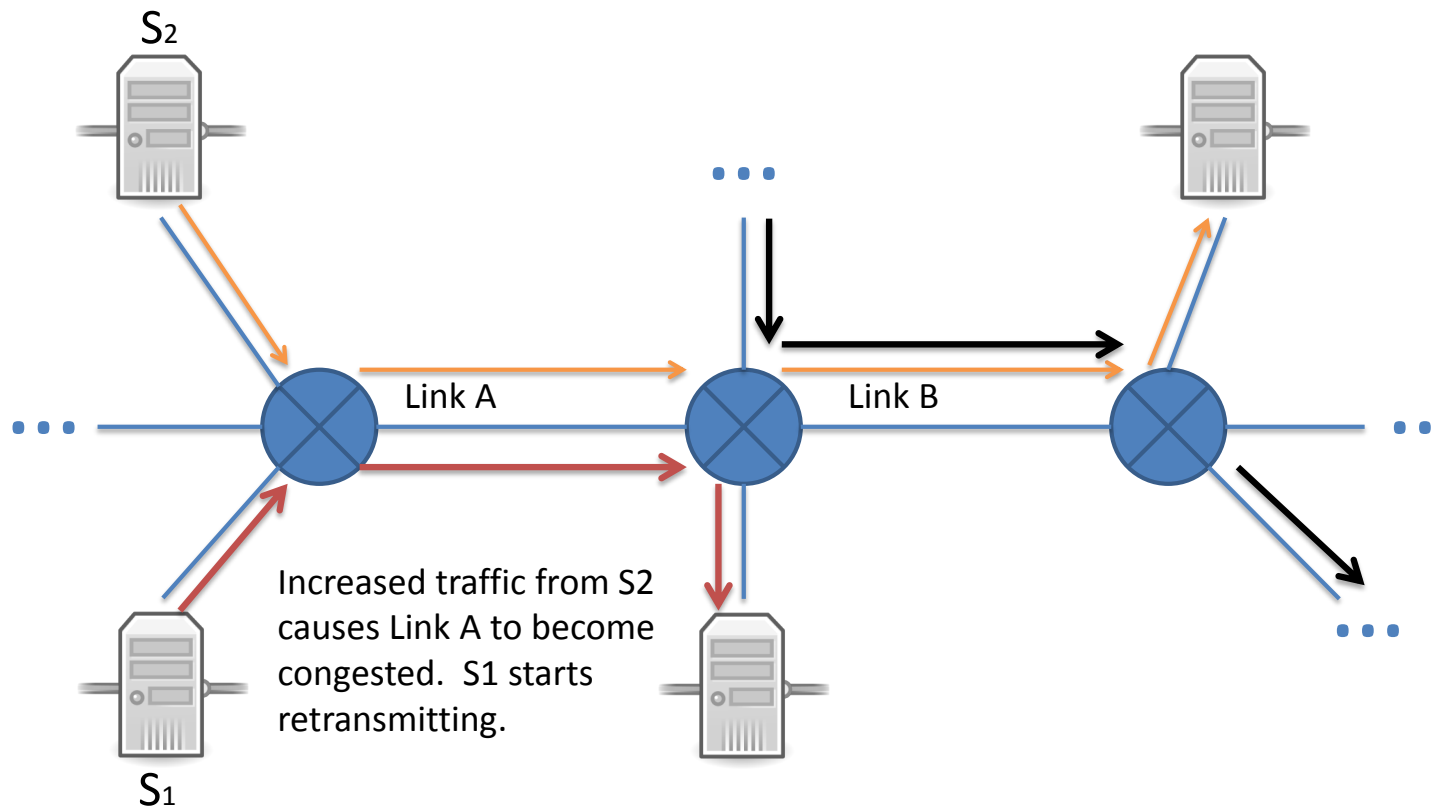
# Congestion Collapse
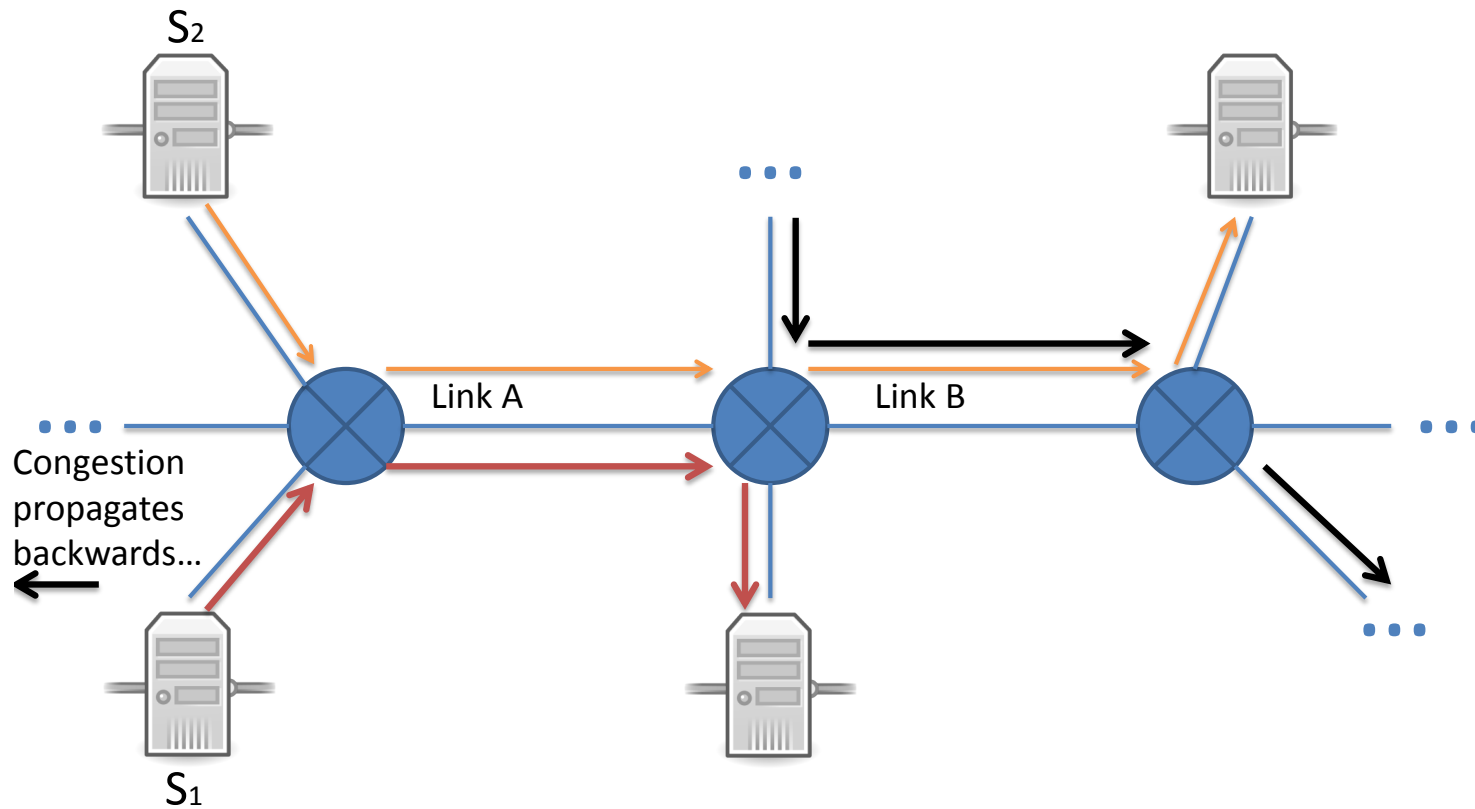
# Without congestion control

*congestion*:

- Increases delays
  - If delays > RTO, sender retransmits
- Increases loss rate
  - Dropped packets also retransmitted
- Increases retransmissions, many unnecessary
  - Wastes capacity of traffic that is never delivered
  - Increase in load results in decrease in useful work done
- Increases congestion, cycle continues …

# Cost of Congestion

❖ Knee – point after which
  ▪ Throughput increases slowly
  ▪ Delay increases fast

❖ Cliff – point after which
  ▪ Throughput starts to drop to zero (congestion collapse)
  ▪ Delay approaches infinity

knee    cliff    packet loss

Throughput

congestion collapse

Load

Delay

Load
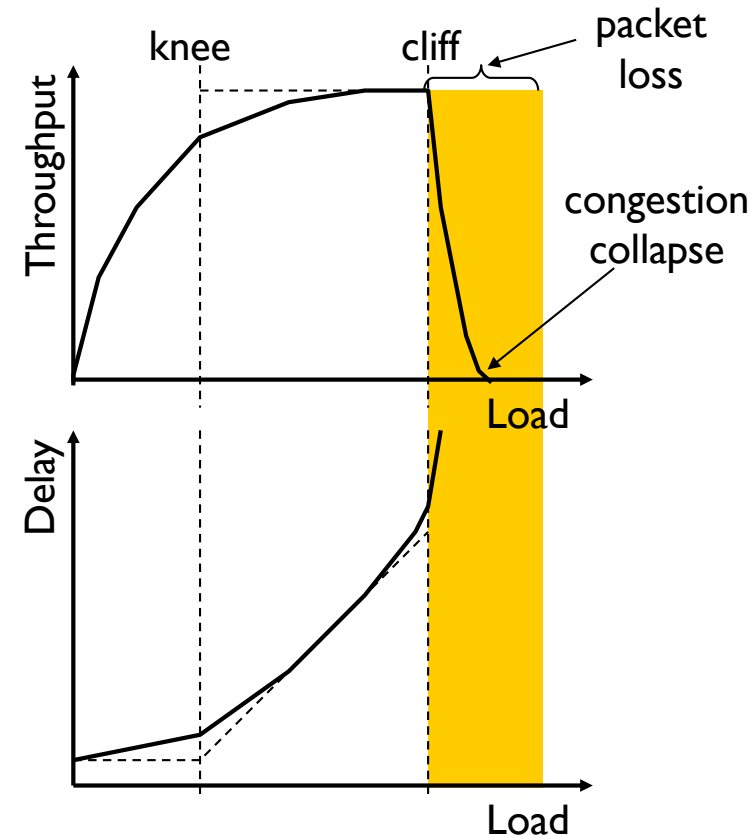
# Congestion Collapse

*This happened to the Internet (then NSFnet) in 1986*

- ❖ Rate dropped from a *blazing* 32 Kbps to 40bps
- ❖ This happened on and off for *two years*
- ❖ In 1988, Van Jacobson published "Congestion Avoidance and Control"
- ❖ The fix: senders voluntarily limit sending rate

# 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

# Transport Layer: 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's Approach in a Nutshell

❖ TCP connection maintains a window
- Controls number of packets in flight

❖ *TCP sending rate:*
- *roughly:* send cwnd bytes, wait RTT for ACKs, then send more bytes

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

sender sequence number space

cwnd

last byte ACKed

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

last byte sent
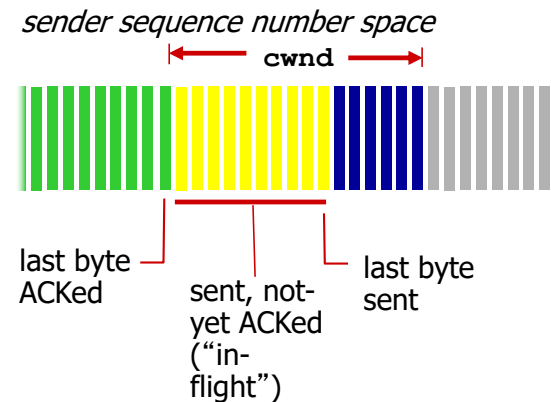
❖ Vary window size to control sending rate

# All These Windows…

❖ Congestion Window: CWND
  - How many bytes can be sent without overflowing routers
  - Computed by the sender using congestion control algorithm

❖ Flow control window: Advertised / Receive Window (RWND)
  - How many bytes can be sent without overflowing receiver's buffers
  - Determined by the receiver and reported to the sender

❖ Sender-side window = minimum{CWND, RWND}
  - Assume for this discussion that RWND >> CWND

# CWND

❖ This lecture will talk about CWND in units of MSS

- (Recall MSS: Maximum Segment Size, the amount of payload data in a TCP packet)
- This is only for pedagogical purposes

❖ Keep in mind that real implementations maintain CWND in bytes

# Two Basic Questions

❖ How does the sender detect congestion?

❖ How does the sender adjust its sending rate?

# Detection Congestion: Infer Loss

- ❖ Duplicate ACKs: isolated loss
  - ▪ dup ACKs indicate network capable of delivering some segments

- ❖ Timeout: much more serious
  - ▪ Not enough dup ACKs
  - ▪ Must have suffered several losses

- ❖ Will adjust rate differently for each case

# RECAP: TCP fast retransmit (dup acks)



Host A                                    Host B

Seq=92, 8 bytes of data
Seq=100, 20 bytes of data
X

ACK=100
ACK=100
ACK=100
ACK=100
Seq=100, 20 bytes of data

timeout

fast retransmit after sender
receipt of triple duplicate ACK
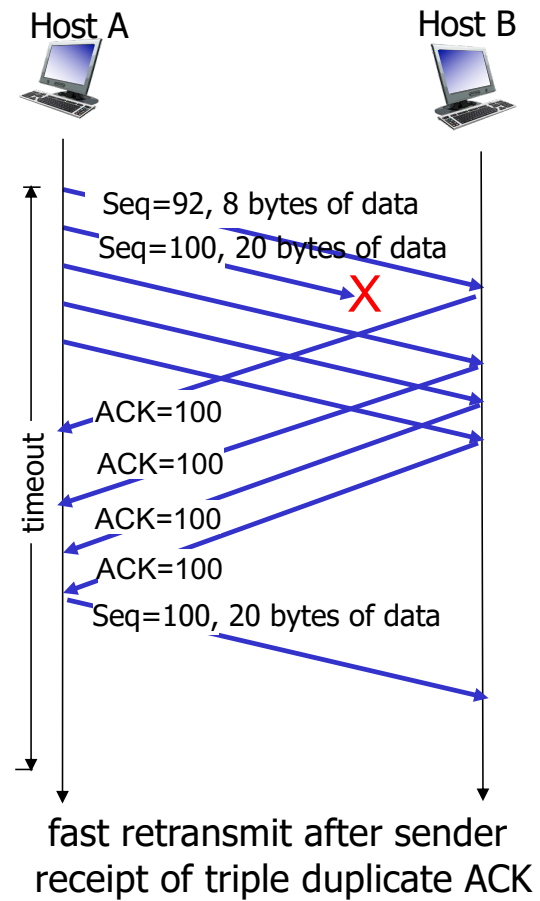
# Rate Adjustment

❖ Basic structure:
  ▪ Upon receipt of ACK (of new data): increase rate
  ▪ Upon detection of loss: decrease rate

❖ How we increase/decrease the rate depends on the phase of congestion control we're in:
  ▪ Discovering available bottleneck bandwidth *vs.*
  ▪ Adjusting to bandwidth variations

# TCP Slow Start (Bandwidth discovery)

- ❖ when connection begins, increase rate exponentially until first loss event:
  - ▪ initially `cwnd` = 1 MSS
  - ▪ double `cwnd` every RTT (all ACKs)
  - ▪ Simpler implementation achieved by incrementing `cwnd` for every ACK received
    - ▪ cwnd += 1 for each ACK
- ❖ *summary:* initial rate is slow but ramps up exponentially fast

Host A                    Host B

RTT

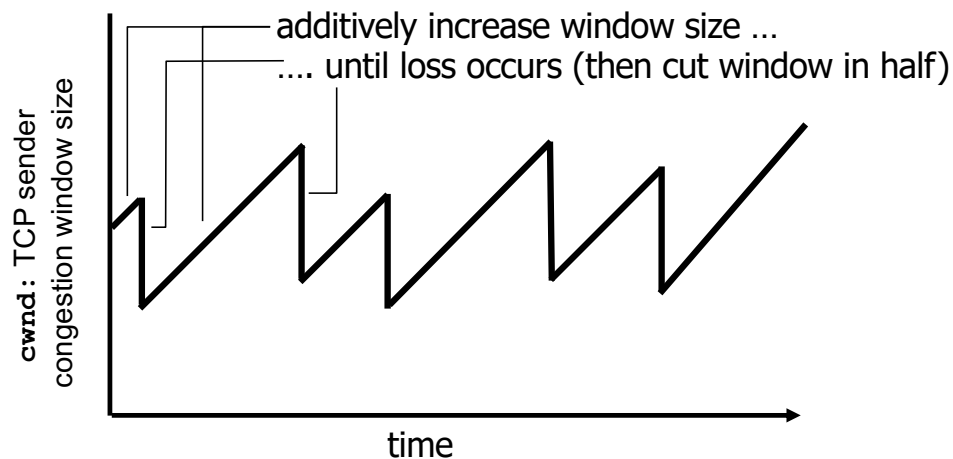one segment

two segments

four segments

time
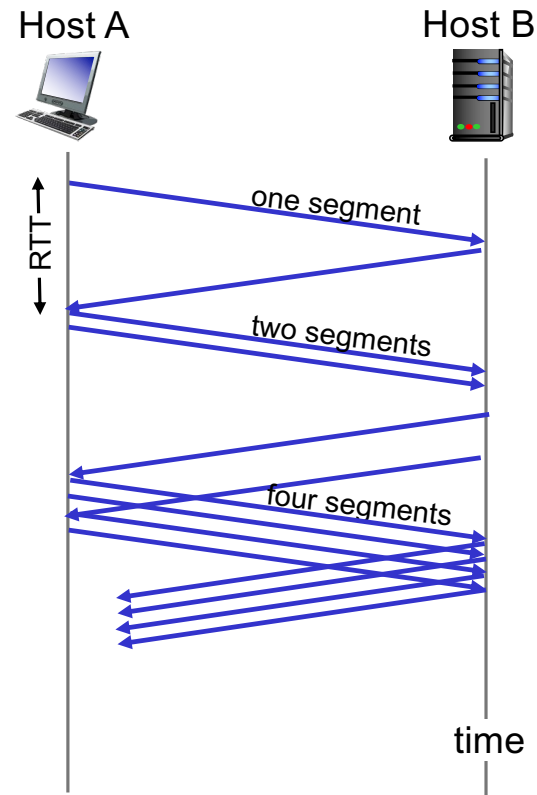
# Adjusting to Varying Bandwidth

❖ Slow start gave an estimate of available bandwidth

❖ Now, want to track variations in this available bandwidth, oscillating around its current value
  ▪ Repeated probing (rate increase) and back-off (rate decrease)
  ▪ Known as Congestion Avoidance (CA)

❖ TCP uses: "Additive Increase Multiplicative Decrease" (AIMD)

# AIMD

❖ *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until another congestion event occurs

  ▪ *additive increase:* increase `cwnd` by 1 MSS every RTT until loss detected

   • For each successful RTT (all ACKs), `cwnd = cwnd +1` (in multiples of MSS)

   • Simple implementation: for each ACK, `cwnd = cwnd + 1/cwnd` (since there are `cwnd/MSS` packets in a window)

  ▪ *multiplicative decrease:* cut `cwnd` in half after loss

AIMD saw tooth behavior: probing for bandwidth

additively increase window size ...
.... until loss occurs (then cut window in half)

`cwnd`: TCP sender congestion window size

time

# Leads to the TCP "Sawtooth"
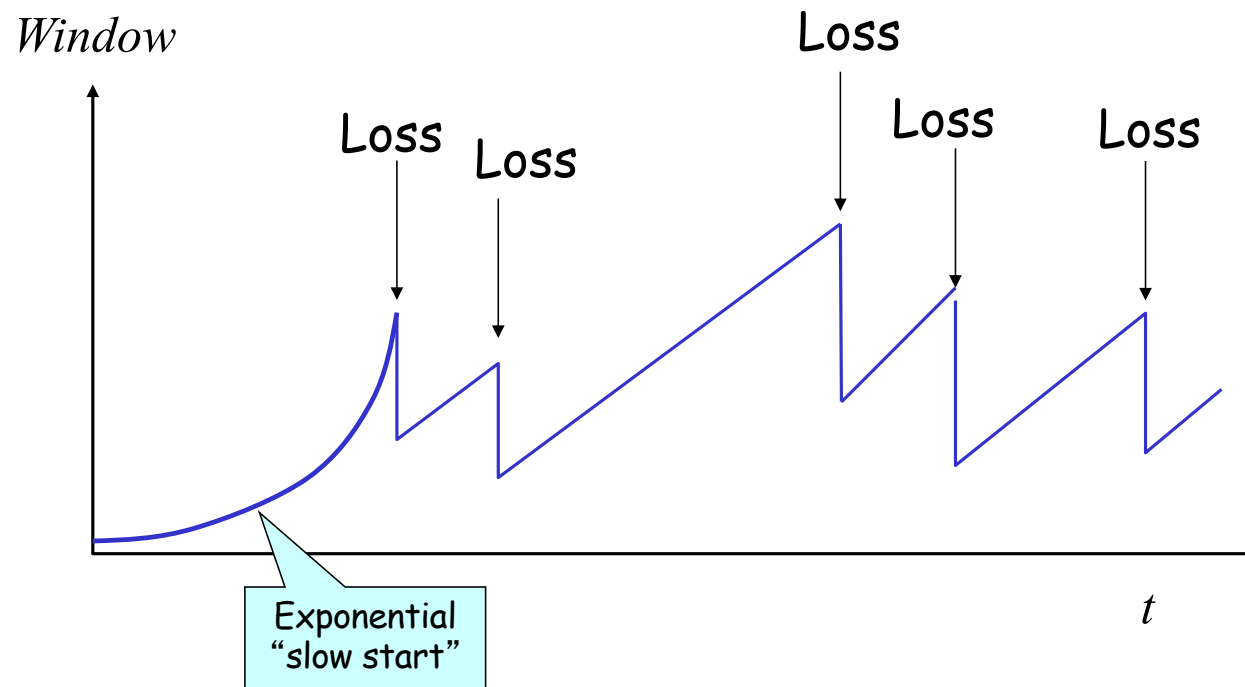
# Slow-Start vs. AIMD

❖ When does a sender stop Slow-Start and start Congestion Avoidance?

❖ Introduce a "slow start threshold" (ssthresh)
  ▪ Initialized to a large value

❖ Convert to CA when cwnd = ssthresh, sender switches from slow-start to AIMD-style increase
  ▪ On loss, ssthresh = CWND/2

# Implementation

❖ **State at sender**
  ▪ CWND (initialized to a small constant)
    • the slides use multiple of MSS
  ▪ ssthresh (initialized to a large constant)
  ▪ [Also dupACKcount and timer, as before]

❖ **Events**
  ▪ ACK (new data)
  ▪ dupACK (duplicate ACK for old data)
  ▪ Timeout

# Event: ACK (new data)

❖ If CWND < ssthresh
  ▪ CWND += 1

> • *Hence after one RTT (All ACKs with no drops):*
>   *CWND = 2xCWND*

# Event: ACK (new data)

❖ **If CWND < ssthresh**
- CWND += 1

*Slow start phase*

❖ **Else**
- CWND = CWND + 1/CWND

*"Congestion Avoidance" phase (additive increase)*

- Hence after one RTT (All ACKs with no drops):
  CWND = CWND + 1

# Event: dupACK

❖ dupACKcount ++

❖ If dupACKcount = 3 /* fast retransmit  */
  ▪ ssthresh = CWND/2
  ▪ CWND = CWND/2

# Event: TimeOut

❖ **On Timeout**
- ▪ ssthresh ← CWND/2
- ▪ CWND ← 1

# Example
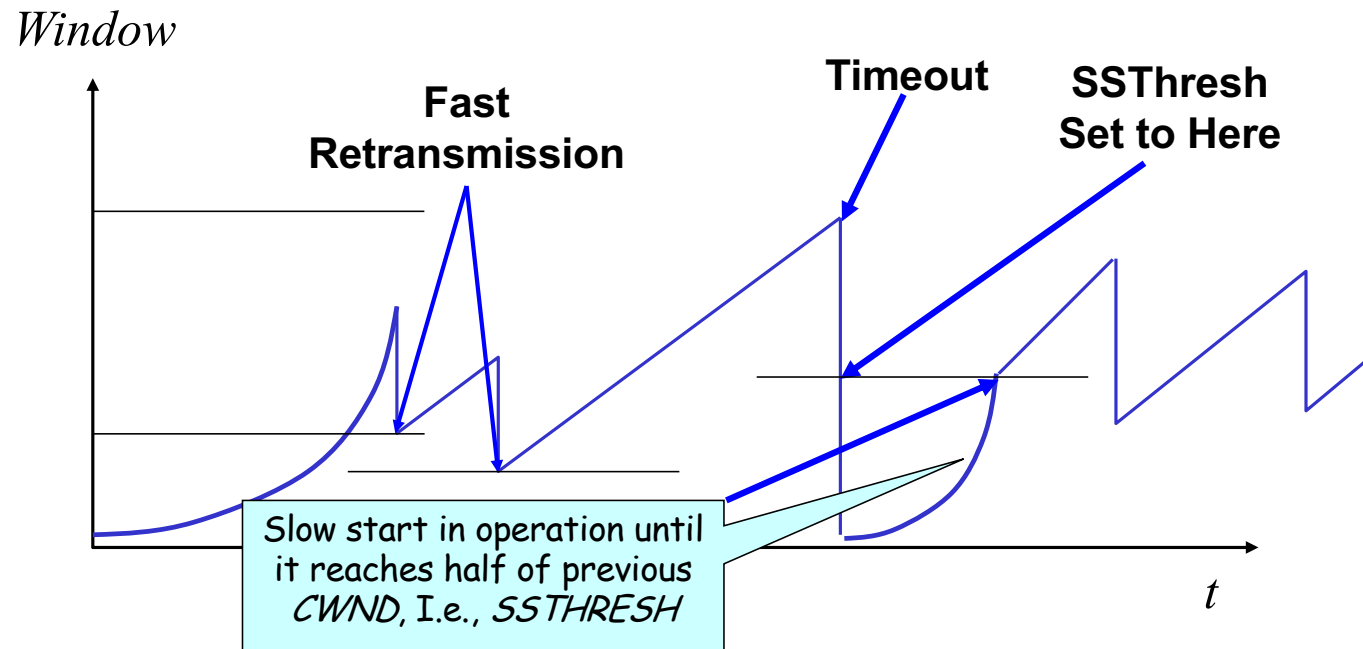


Slow-start restart: Go back to CWND = 1 MSS, but take advantage of knowing the previous value of CWND

# TCP Flavours

❖ TCP-Tahoe
  ▪ cwnd =1 on triple dup ACK & timeout

❖ TCP-Reno
  ▪ cwnd =1 on timeout
  ▪ cwnd = cwnd/2 on triple dup ACK

❖ TCP-newReno
  ▪ TCP-Reno + improved fast recovery (SKIPPED AND NOT ON EXAM)

❖ TCP-SACK (NOT COVERED IN THE COURSE)
  ▪ incorporates selective acknowledgements
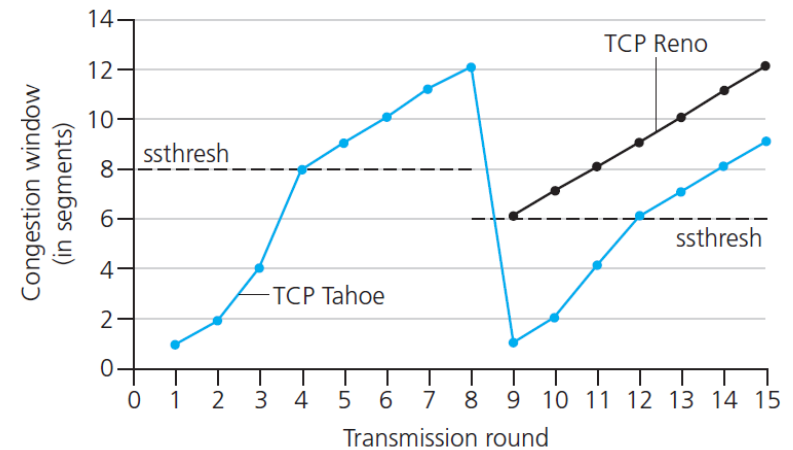
# Quiz: TCP Congestion Control?

In the figure how many congestion avoidance intervals can you identify?
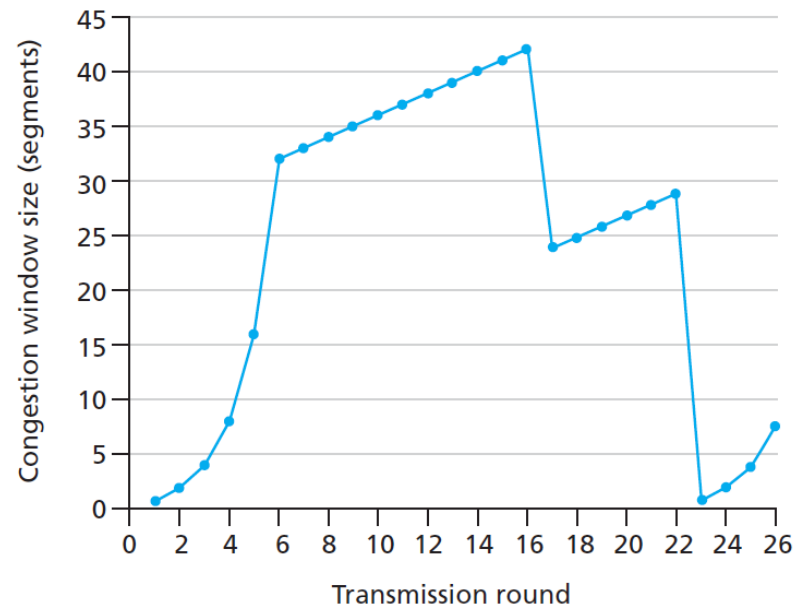
A.  0

B.  1

C.  2

D.  3

E.  4



Note: the transition at round 17 is not entirely acurrate, the window should reduce to 21 (currently 24)

# Quiz: TCP Congestion Control?

In the figure how many slow start intervals can you identify?

A. 0

B. 1

C. 2

D. 3

E. 4



Note: the transition at round 17 is not entirely acurrate, the window should reduce to 21 (currently 24)

# Quiz: TCP Congestion Control

In the figure after the 16th transmission round, segment loss is detected by _____ ?

A. Triple Dup Ack

B. Timeout
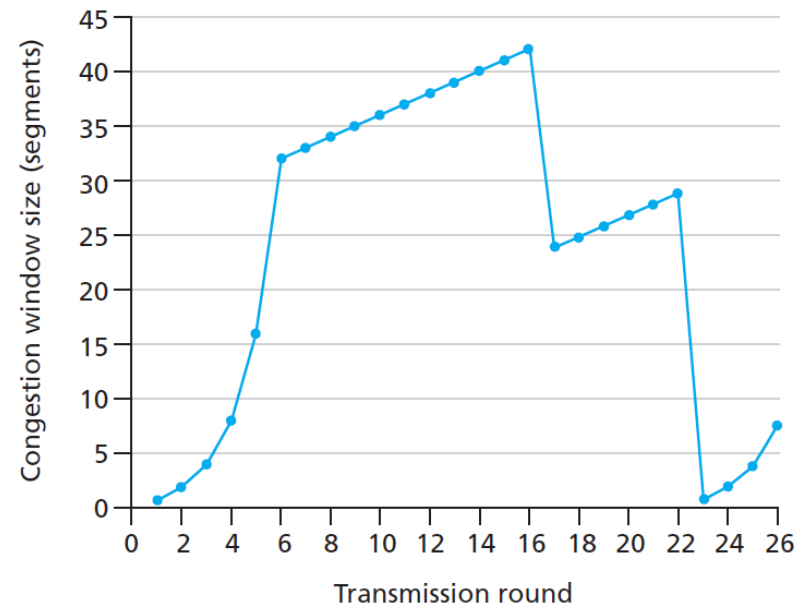


Note: the transition at round 17 is not entirely acurrate, the window should reduce to 21 (currently 24)

# Quiz: TCP Congestion Control

In the figure what is the initial value of sstresh (steady state threshold)?

A. 0

B. 28

C. 32

D. 42

E. 64
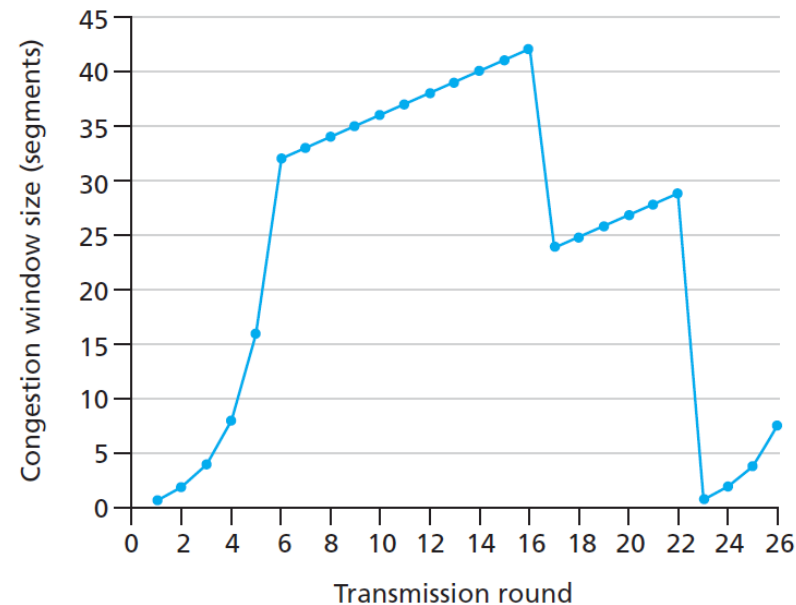


Note: the transition at round 17 is
not entirely acurrate, the window
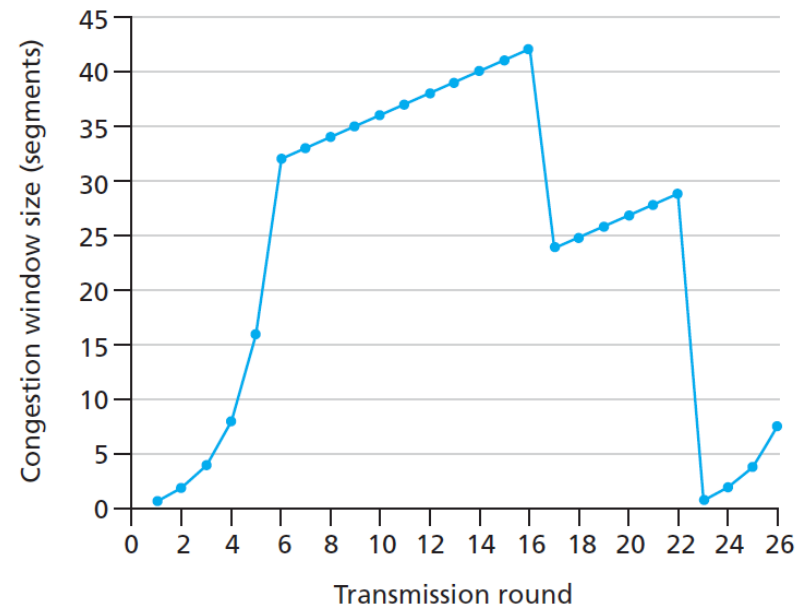should reduce to 21 (currently 24)

# Quiz: TCP Congestion Control

In the figure what is the value of sstresh (steady state threshold) at the 18$^{th}$ round?

A. 1

B. 32

C. 42

D. 21

E. 20
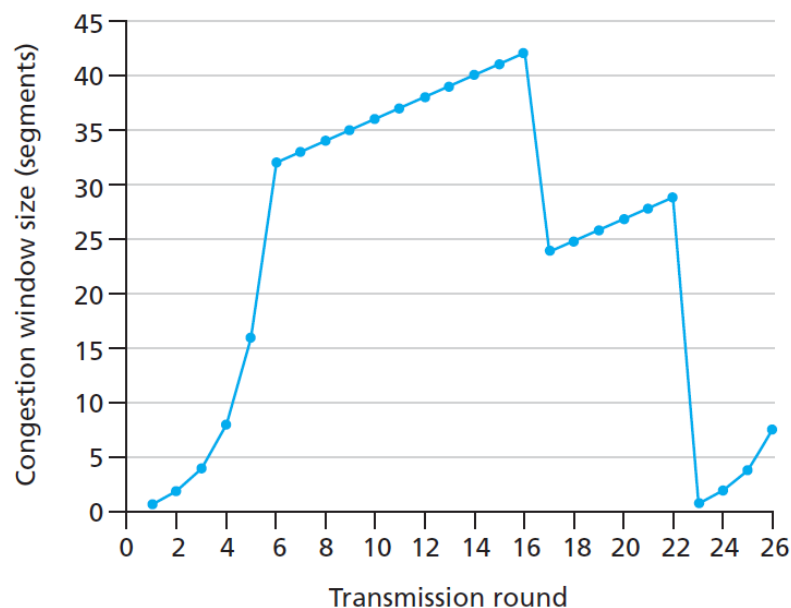


Note: the transition at round 17 is not entirely acurrate, the window should reduce to 21 (currently 24)

# Quiz: TCP Reliability

TCP uses cumulative ACKs like Go-Back-N but does not retransmit the entire window of outstanding packets upon a timeout. What mechanism lets TCP get away with this?

A. Per-byte sequence and acknowledgement numbers

B. Triple Duplicate ACKs

C. Receiver window-based flow control

D. Timeout estimation algorithm

# Quiz: TCP Timeout

A TCP Sender maintains an EstimatedRTT of 100ms. Suppose the next SampleRTT is 108. Which of the following is true about the sender?

A. It will increase EstimatedRTT but leave timeout unchanged

B. It will increase the timeout

C. Whether it increases EstimatedRTT will depend on the deviation

D. Whether it increases the timeout will depend on the deviation

# TCP seq. numbers, ACKs

Host A

Host B

Seq=42, ACK = 79, Data=571 bytes

Seq=c, ACK =d, Data=435 bytes

Seq = a, ACK=b, data = 0 byte

Seq = e, ACK=f, data = 0 byte

Seq=g, ACK =h

# TCP seq. numbers, ACKs

Host A                                      Host B

Seq=42, ACK = 55, Data=432 bytes

Seq=c, ACK =d, Data=319 bytes

Seq = a, ACK=b, data = 0 byte

Seq = e, ACK=f, data = 0 byte

Seq=g, ACK =h

# TCP seq. numbers, ACKs

Host A                                    Host B

Seq=32, ACK = 121, Data=389 bytes

Seq=c, ACK =d, Data=513 bytes

Seq = a, ACK=b, data = 0 byte

Seq = e, ACK=f, data = 0 byte

Seq=g, ACK =h

Seq=32, ACK = 121, Data=389 bytes     Seq=c, ACK =d, Data=513 bytes

time ⟶

Host A

Host B     Connection Setup     Data Transfer     Connection Teardown

Host A

Host B

cwnd = 4 MSS
sstresh = 8 MSS

Seq=1000
Seq=2000
Seq=3000
Seq=4000

X

Ack =a
Ack =b

cwnd = w

Ack =c

cwnd = x

Seq=p
Seq=q
Seq=r
Seq=s

Ack =d
Ack =e
Ack =f
Ack =g

timeout

cwnd = y
ssthresh =z

Seq=t

Ack =h

Host A                Host B                    Host A                Host B

ACK = 250, Seq=32, 50 data bytes              ACK = 200, Seq=55, 70 data bytes

ACK = 250, Seq=**o**, 60 data bytes           ACK = 200, Seq=**o**, 60 data bytes

ACK= **p**, Seq = 250, 100 data bytes         ACK= **p**, Seq = 200, 100 data bytes

**X**                                         **X**

ACK=**q**, Seq = **r**, 120 data bytes        ACK=**q**, Seq = **r**, 100 data bytes

ACK= **s**, Seq = **t**, 50 data bytes        ACK= **s**, Seq = **t**, 70 data bytes

ACK= **u**, Seq = **v**, no data              ACK= **u**, Seq = **v**, no data

ACK=**w**, Seq = **x**, 100 data bytes        ACK=**w**, Seq = **x**, 100 data bytes

ACK= **y**, Seq = **z**, no data              ACK= **y**, Seq = **z**, no data

timeout

Host A                    Host B

ACK = 300, Seq=32, 60 data bytes

ACK = 300, Seq=**o**, 60 data bytes

ACK= **p**, Seq = 300, 100 data bytes

**X**

ACK=**q**, Seq = **r**, 150 data bytes

ACK= **s**, Seq = **t**, 60 data bytes

ACK= **u**, Seq = **v**, no data

ACK=**w**, Seq = **x**, 100 data bytes

ACK= **y**, Seq = **z**, no data

timeout