COMP 3331/9331: Computer Networks and Applications

Week 5

Transport Layer (Continued)

Reading Guide: Chapter 3, Sections: 3.5 – 3.7

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

Recall: Components of a solution for reliable transport

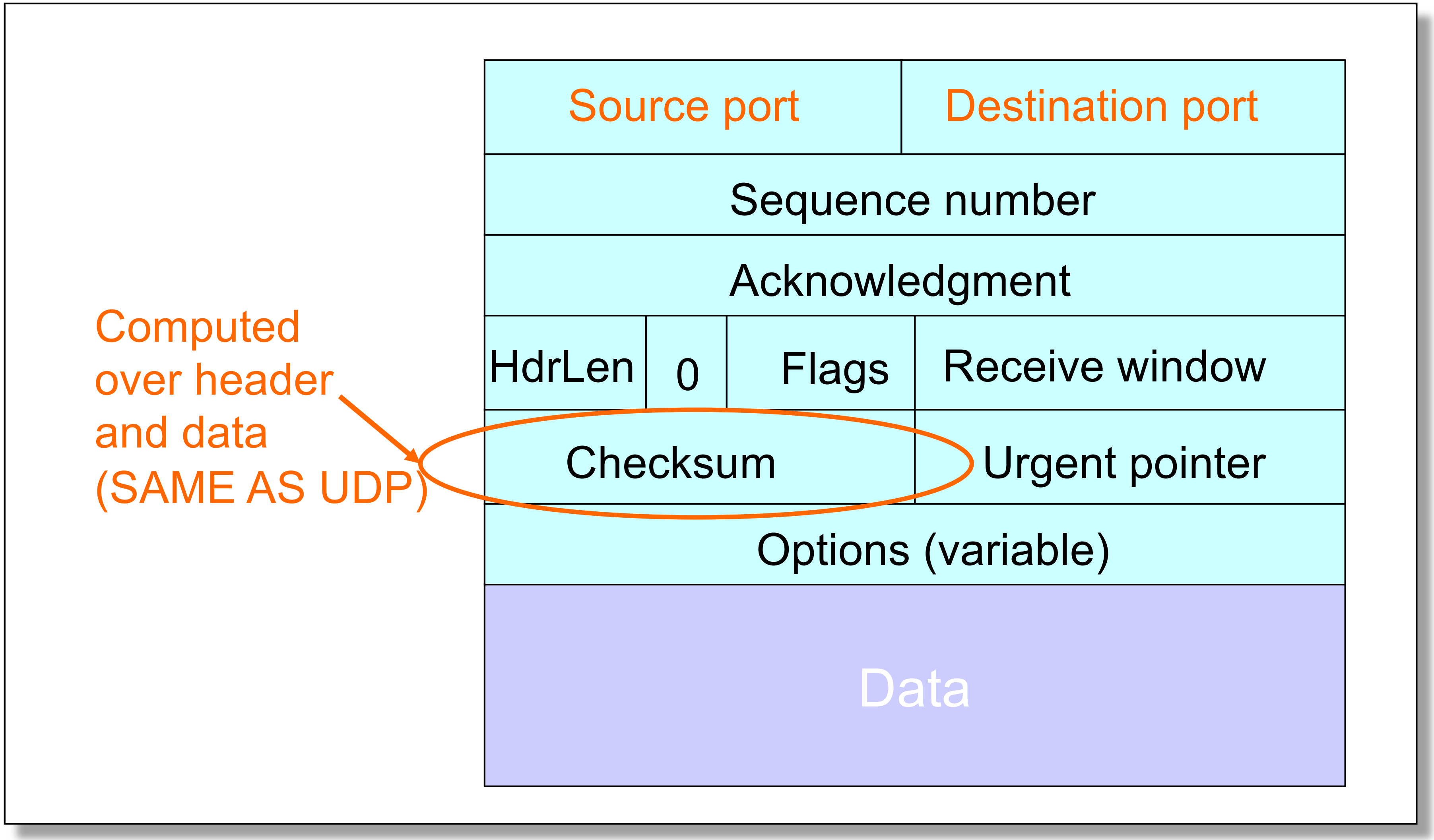
- Checksums (for error detection)
- Timers (for loss detection)
- Acknowledgments
  - cumulative
  - selective
- Sequence numbers (duplicates, windows)
- Sliding Windows (for efficiency)
  - Go-Back-N (GBN)
  - Selective Repeat (SR)

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)

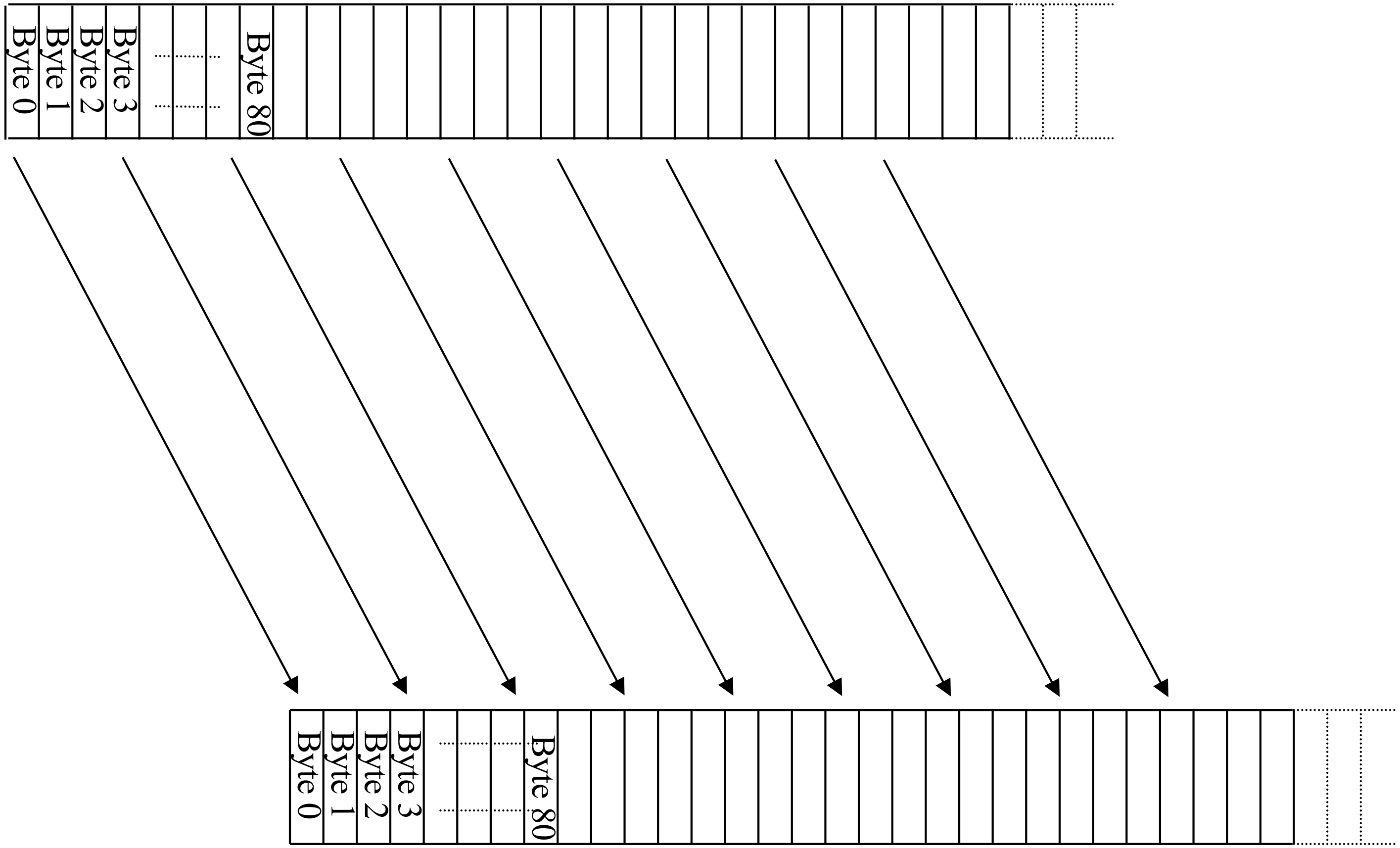
What does TCP do?

Many of our previous ideas, but some key differences

- Checksum
- **Sequence numbers are byte offsets**

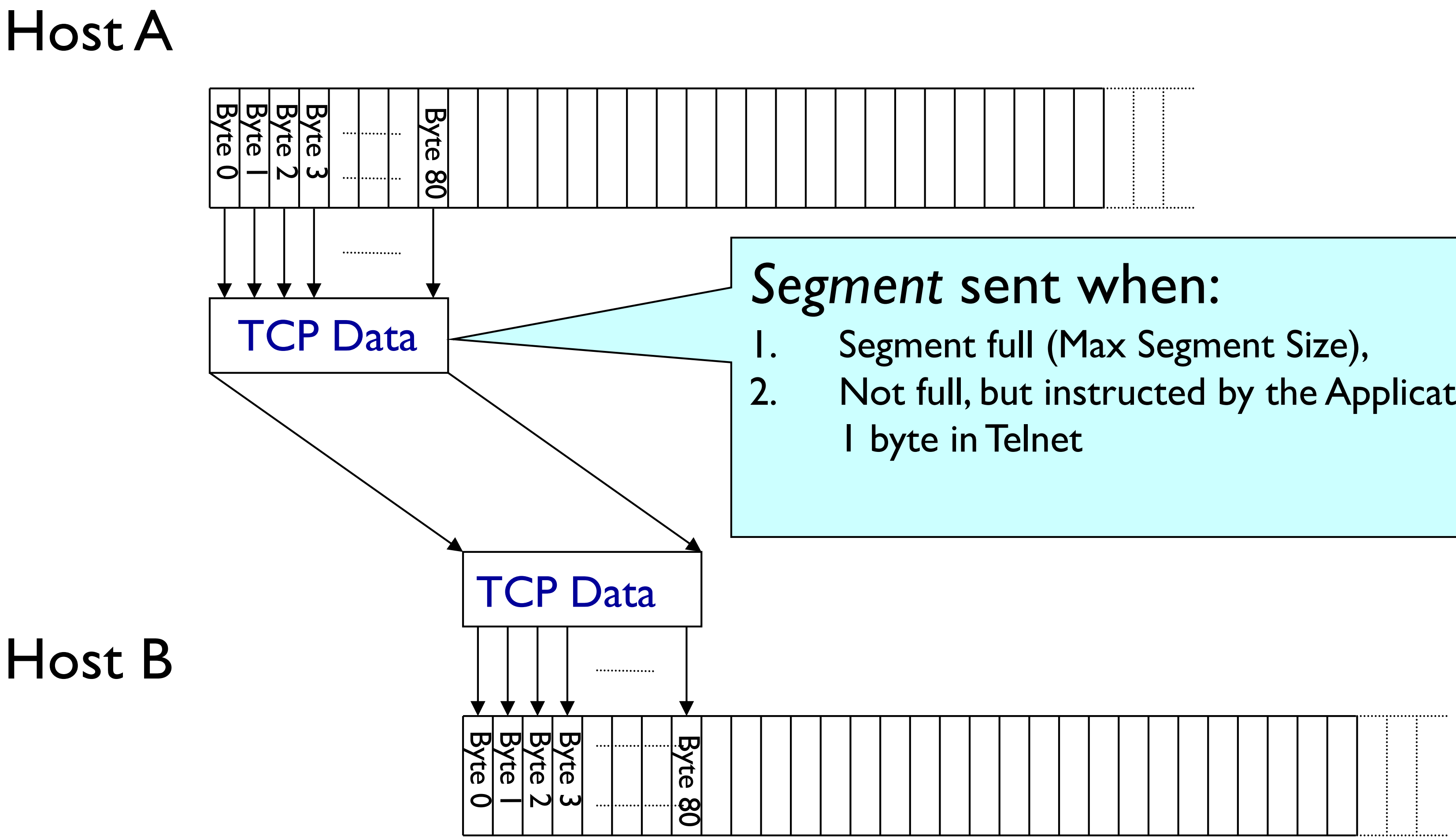
TCP "Stream of Bytes" Service ..

Application @ Host A



Application @ Host B

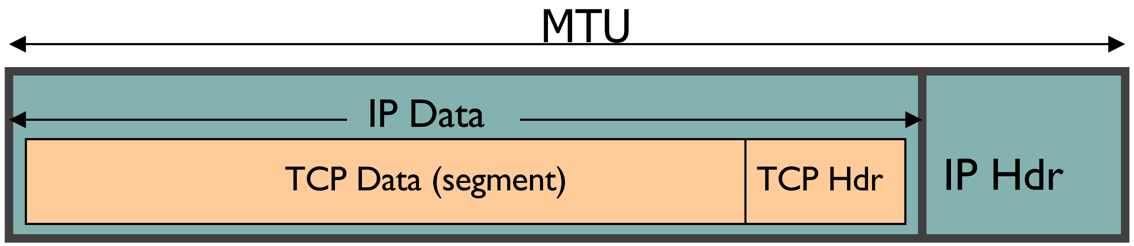
.. Provided Using TCP "Segments"



Segment sent when:

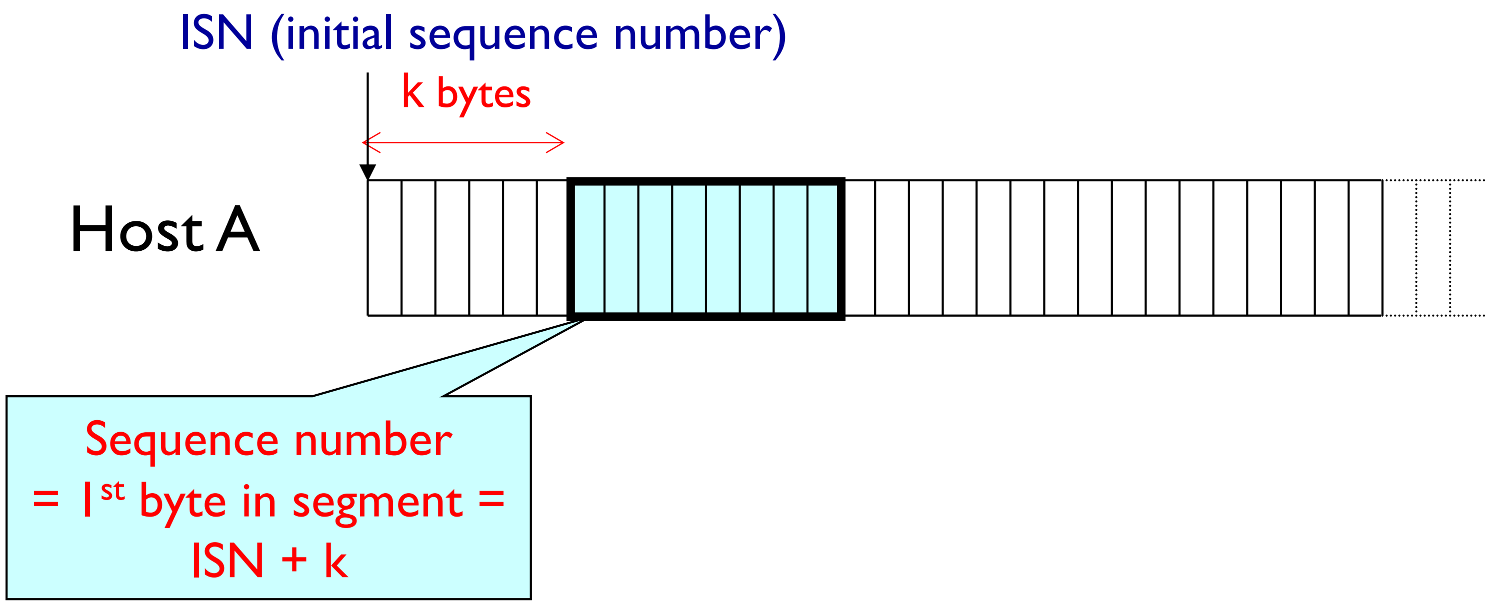
1. Segment full (Max Segment Size),
2. Not full, but instructed by the Application e.g., 1 byte in Telnet

TCP Maximum Segment Size



- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes with Ethernet
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header ≥ 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 (min IP header) – 20 (min TCP header)

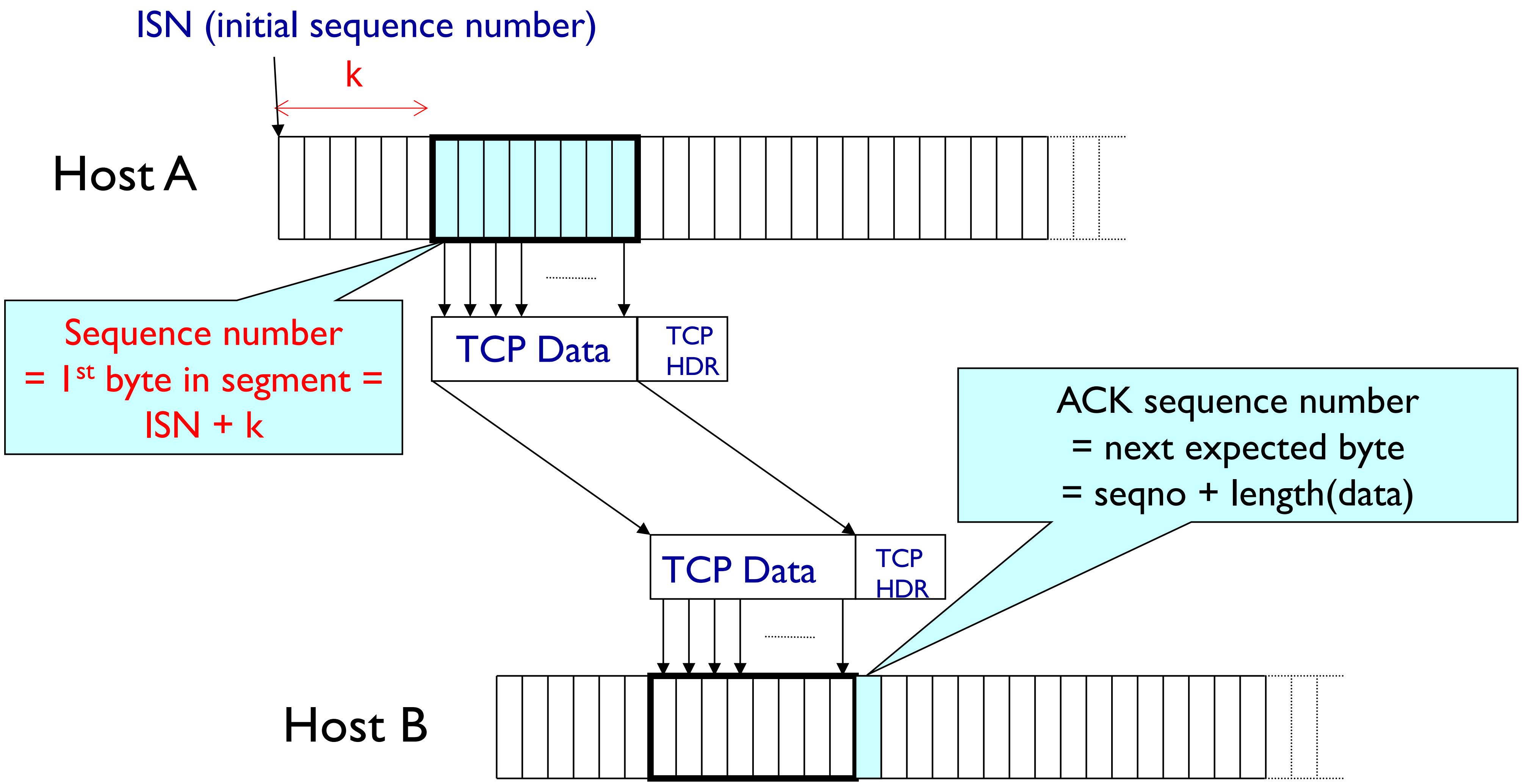
Sequence Numbers



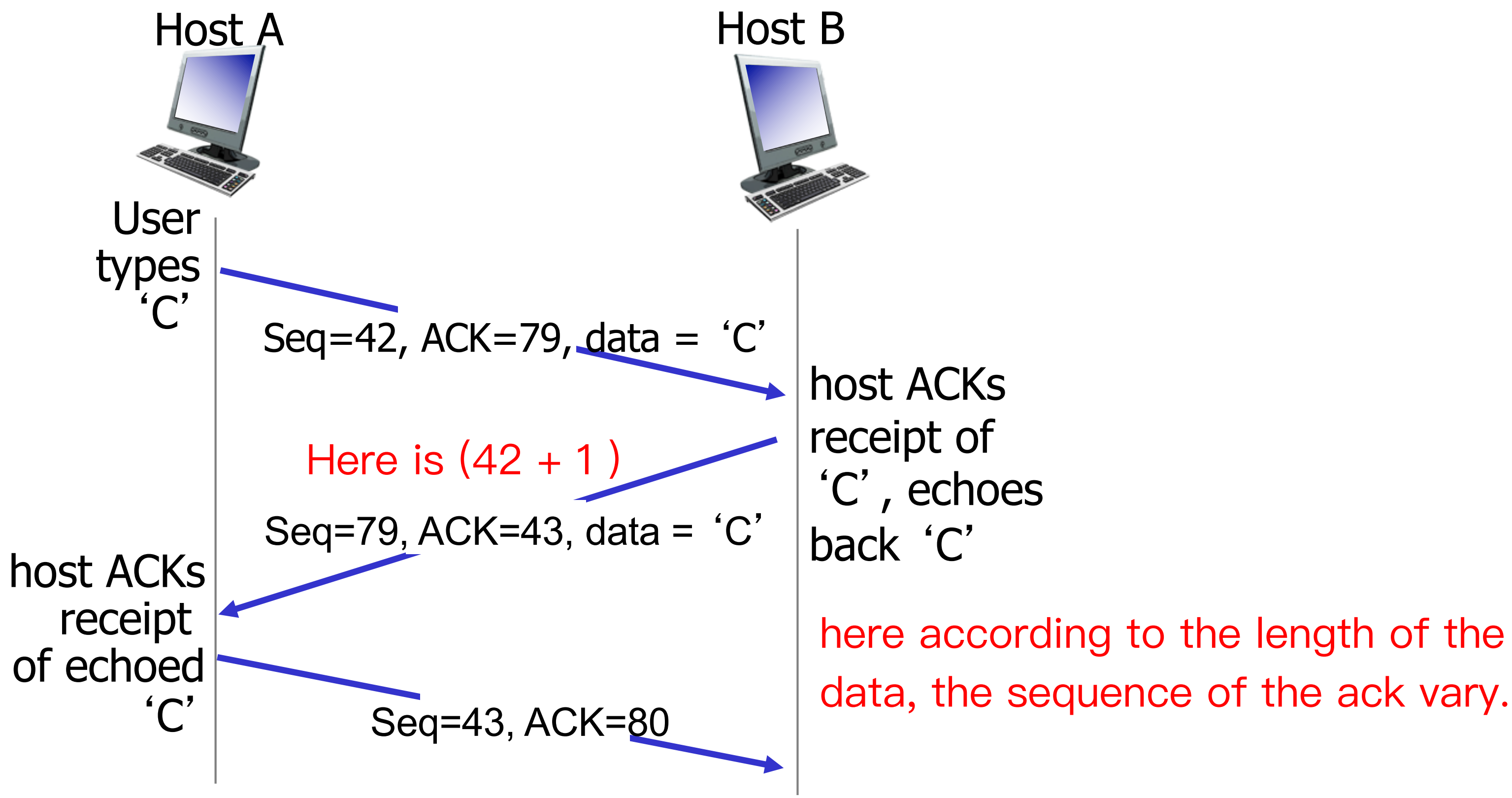
Sequence numbers:

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

Sequence & Ack Numbers



TCP seq. numbers, ACKs



simple telnet scenario

- 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 s.t. (Y+1) < X
    - ACK acknowledges Y+1
    - Even if this has been ACKed before

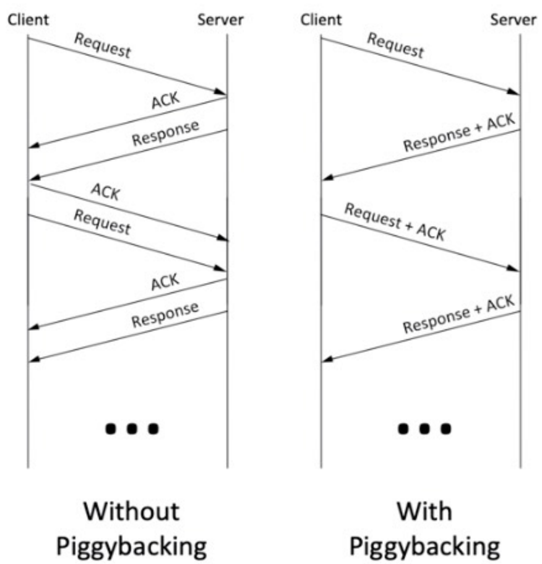
What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)

Piggybacking

- So far, we've assumed distinct "sender" and "receiver" roles
- In reality, usually both sides of a connection send some data

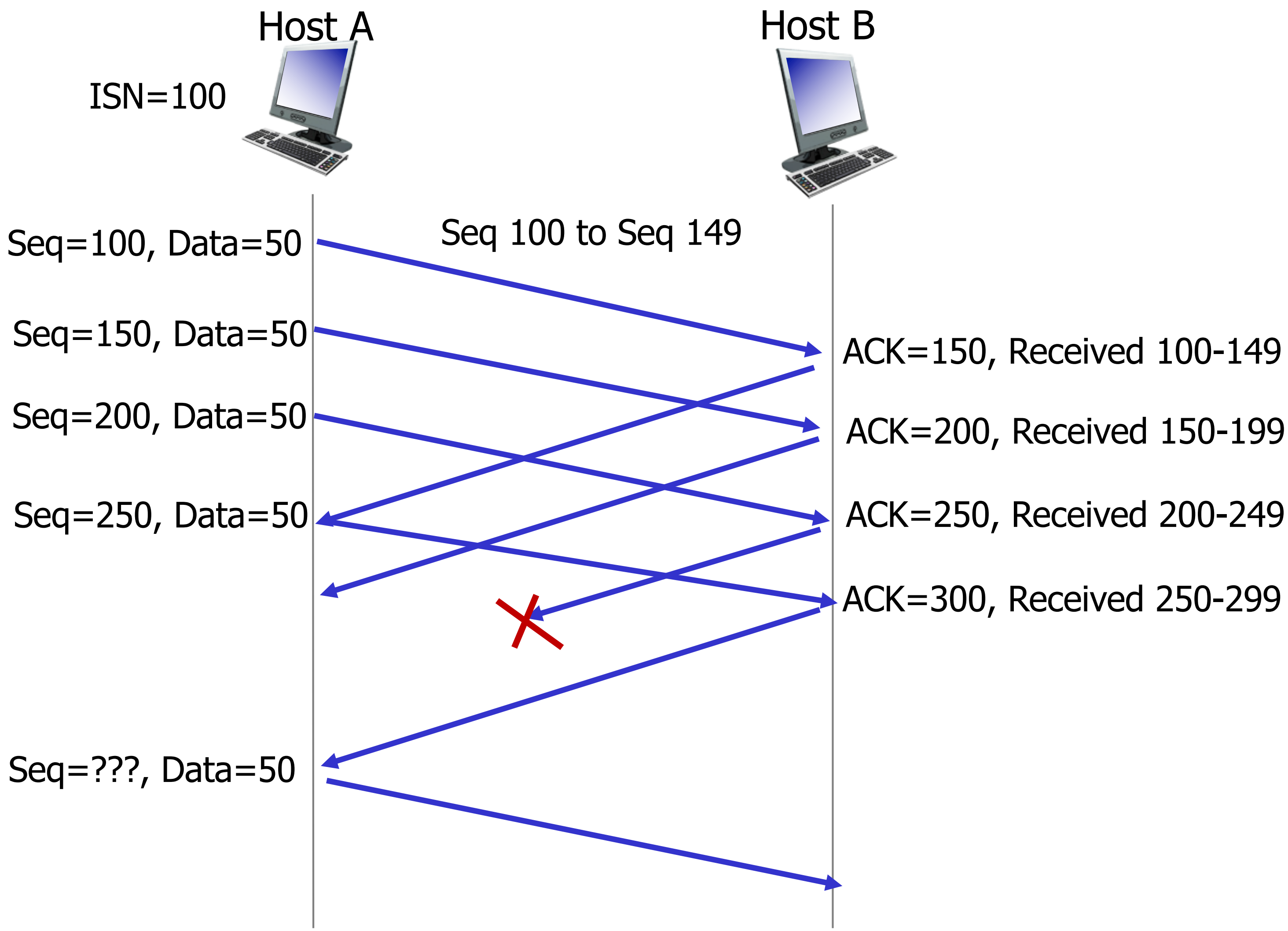


ACKing and Sequence Numbers

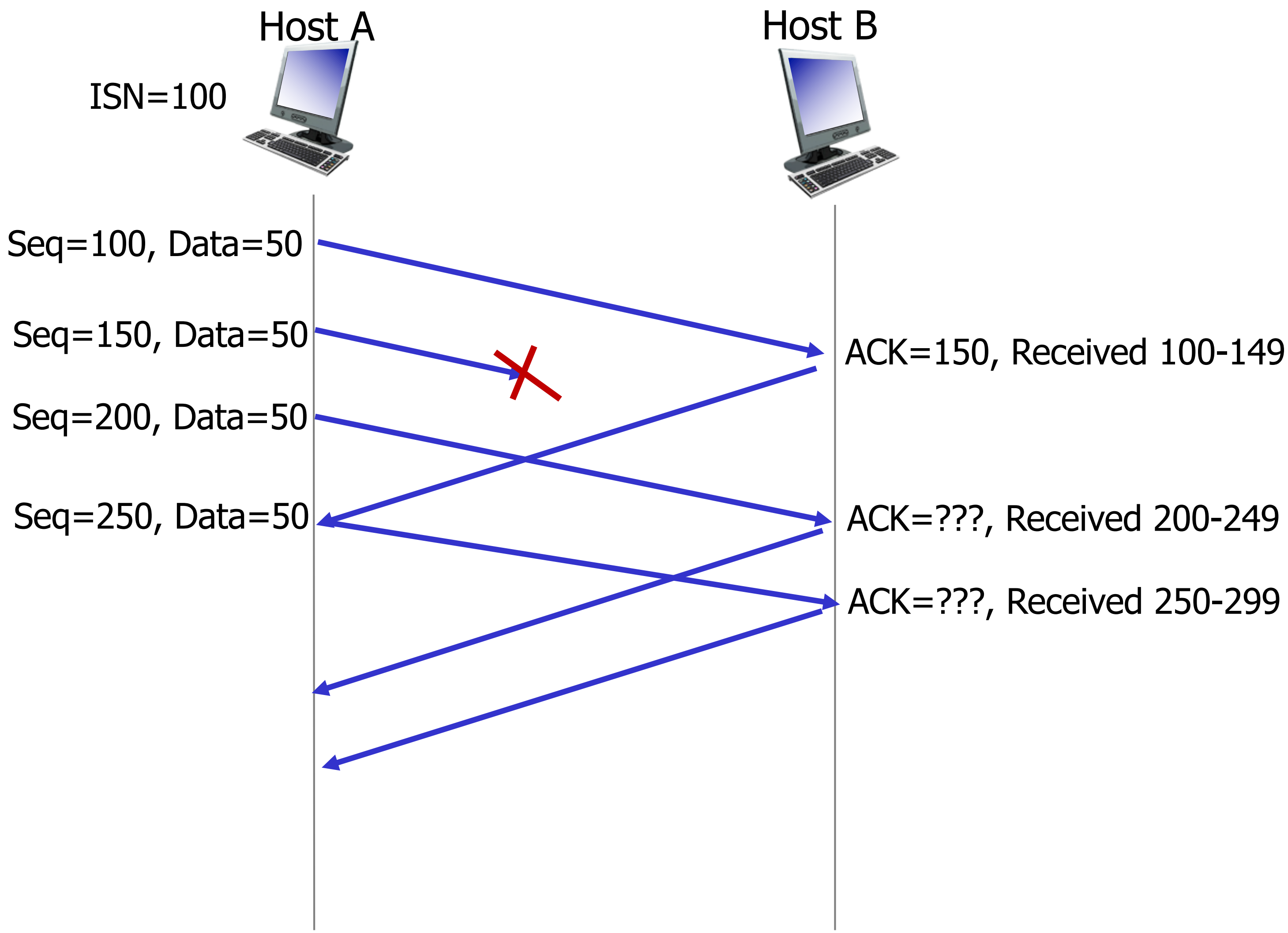
- Sender sends packet
  - Data starts with sequence number X
  - Packet contains B bytes [X, X+1, X+2, ….X+B-1]

- 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 s.t. (Y+1) < X
    - ACK acknowledges Y+1
    - Even if this has been ACKed before

TCP seq. numbers, ACKs



TCP seq. numbers, ACKs



Normal Pattern

- Sender: seqno=X, length=B
- Receiver: ACK=X+B
- Sender: seqno=X+B, length=B
- Receiver: ACK=X+2B
- Sender: seqno=X+2B, length=B

- Seqno of next packet is same as last ACK field

Packet Loss

- Sender: seqno=X, length=B
- Receiver: ACK=X+B
- Sender: ~~seqno=X+B, length=B~~ LOST

- Sender: seqno=X+2B, length=B
- Receiver: ACK = X+B

TCP Header

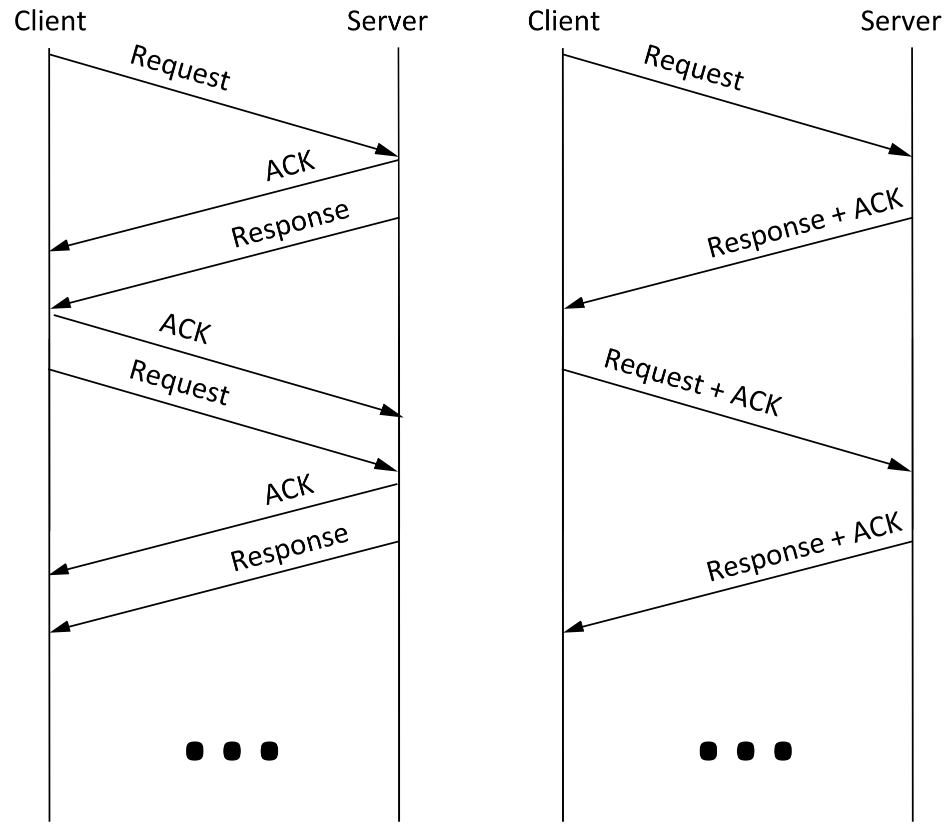
Acknowledgment gives seqno just beyond highest seqno received **in order**
("What Byte is Next")

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

TCP Acknowledgement is base on bytes

Piggybacking

- So far, we've assumed distinct "sender" and "receiver" roles
- In reality, usually both sides of a connection send some data



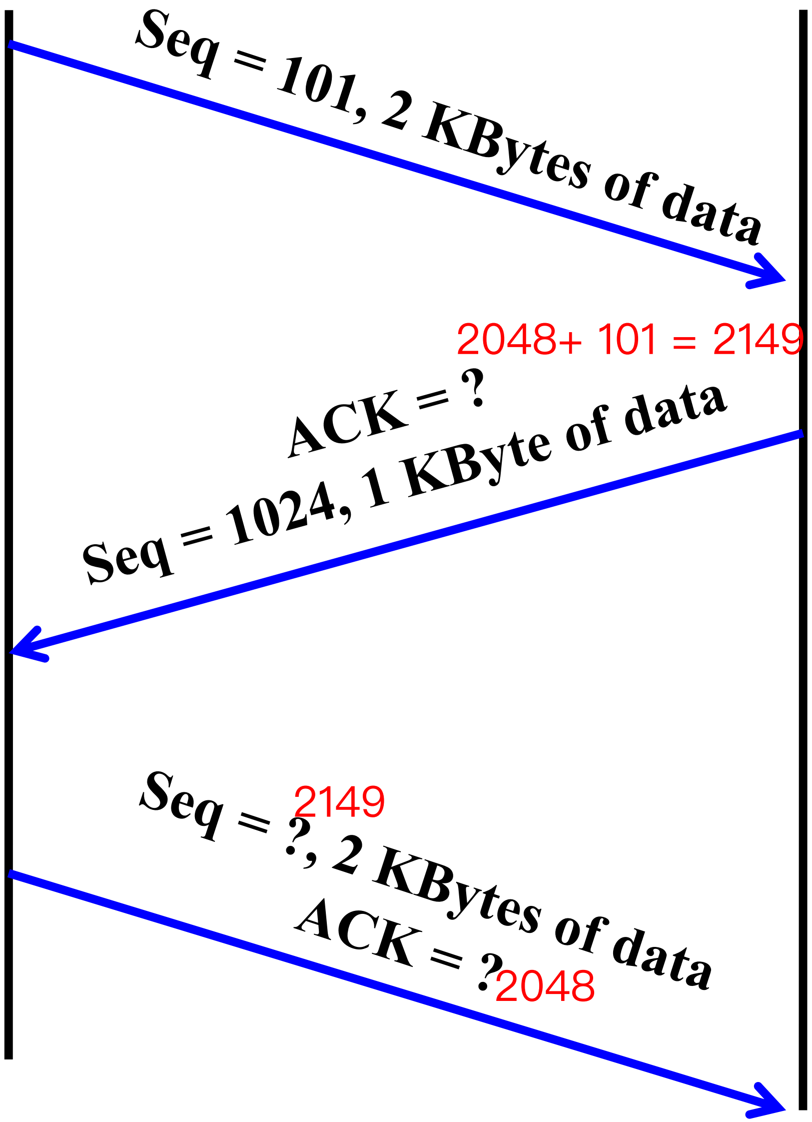
Without Piggybacking

With Piggybacking

ACK = 101 + 2048 = 2149

Quiz

ACK = 2048



Seq = 2149

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 100Bytes and sequence numbers:
  - 100, 200, 300, 400, 500, 600, 700, 800, 900, …
- Assume the fifth packet (seq. no. 500) is lost, but no others
- Stream of ACKs will be:
  - 200, 300, 400, 500, 500, 500, 500,…

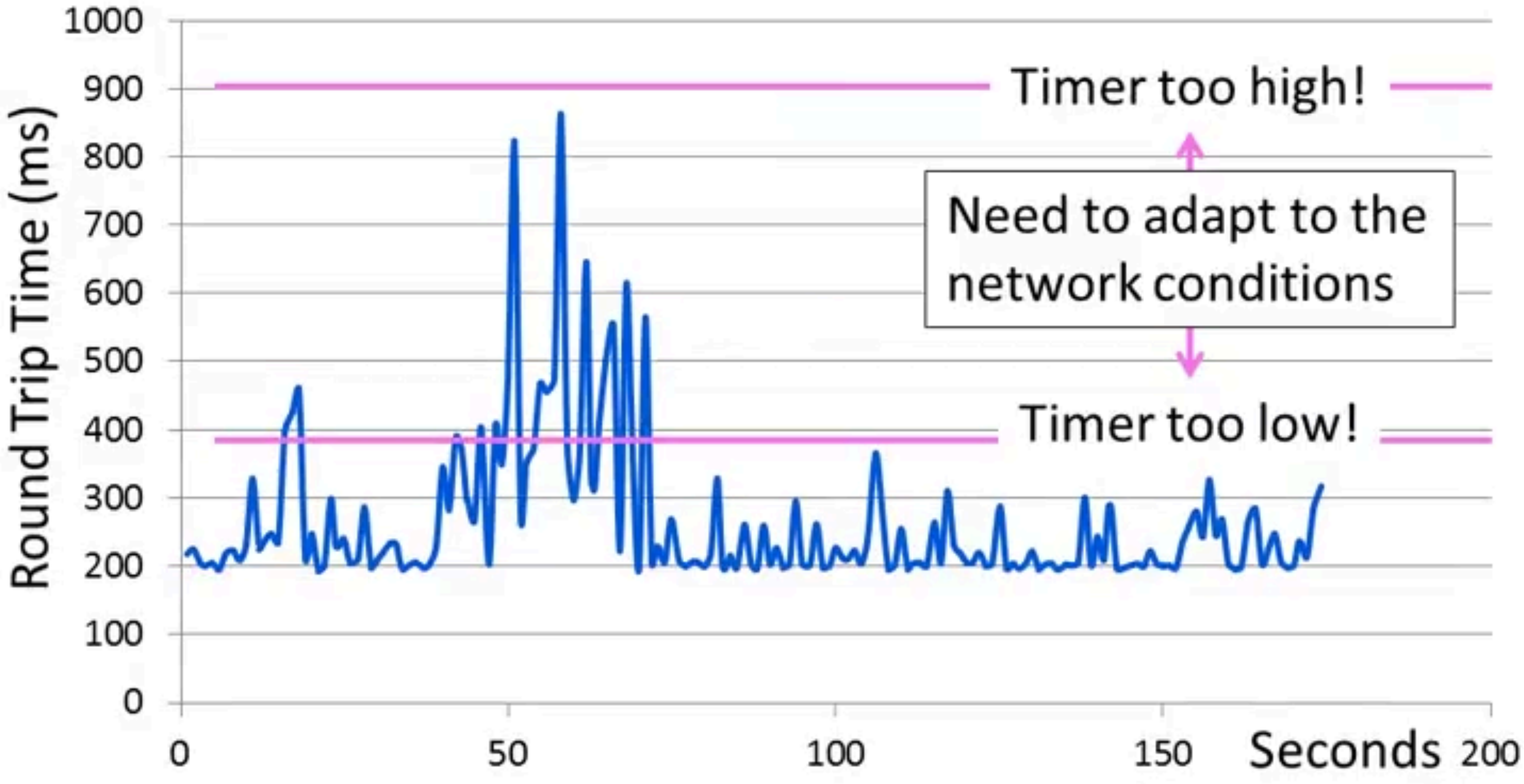
What does TCP do?

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?*) (why **single** timer?)

TCP has lots of segments, there may be a lot of timers to maintain at the sender

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 and connection has lower throughput

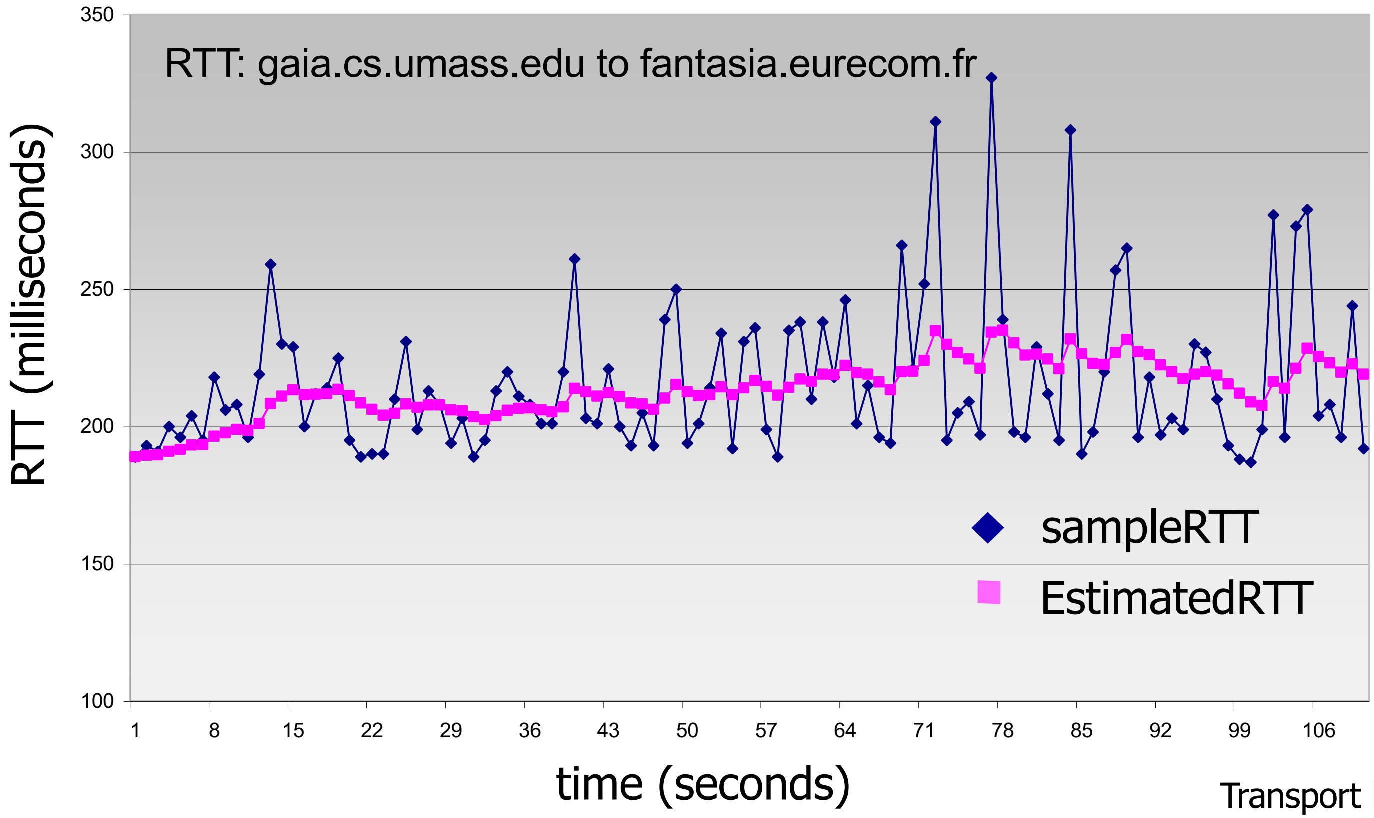
Q: how to estimate RTT?

- **`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

EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT

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



TCP round trip time, timeout

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

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

             (typically, β = 0.25)
```

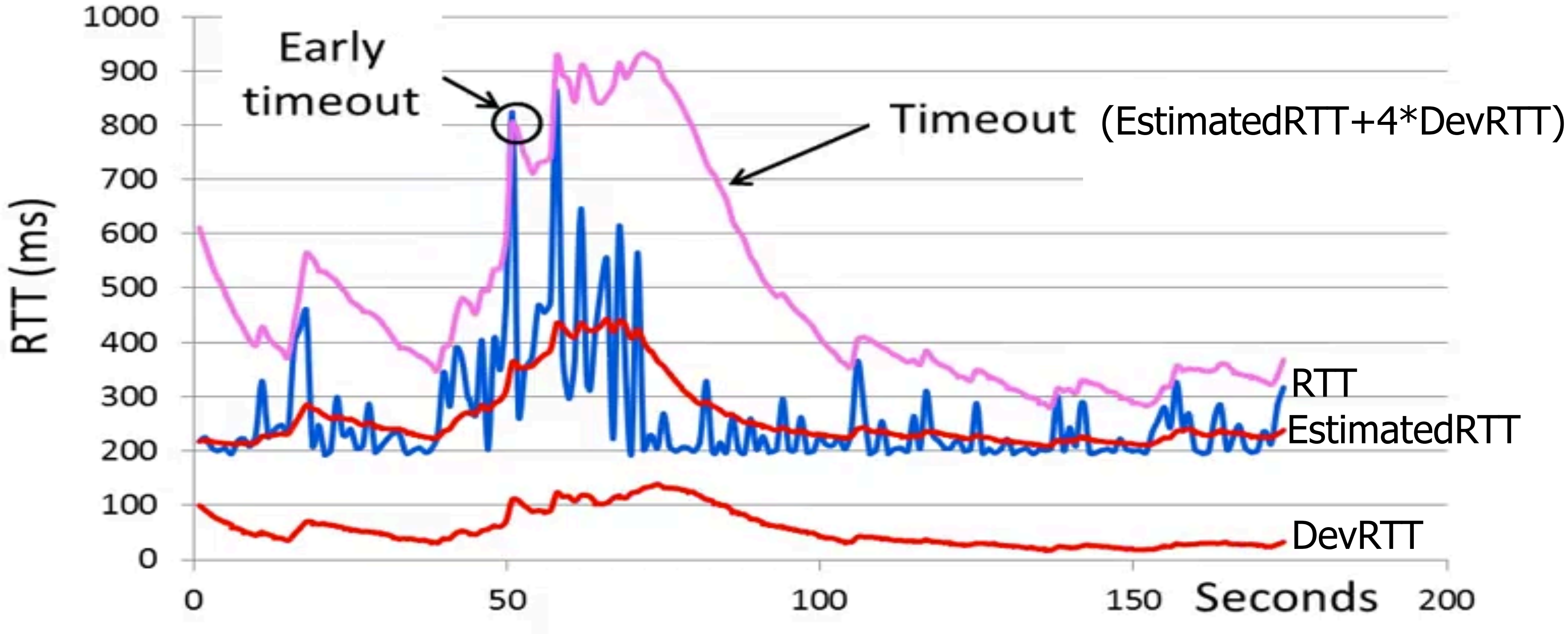
```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT (EstimatedRTT)

"safety margin" (4*DevRTT)

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

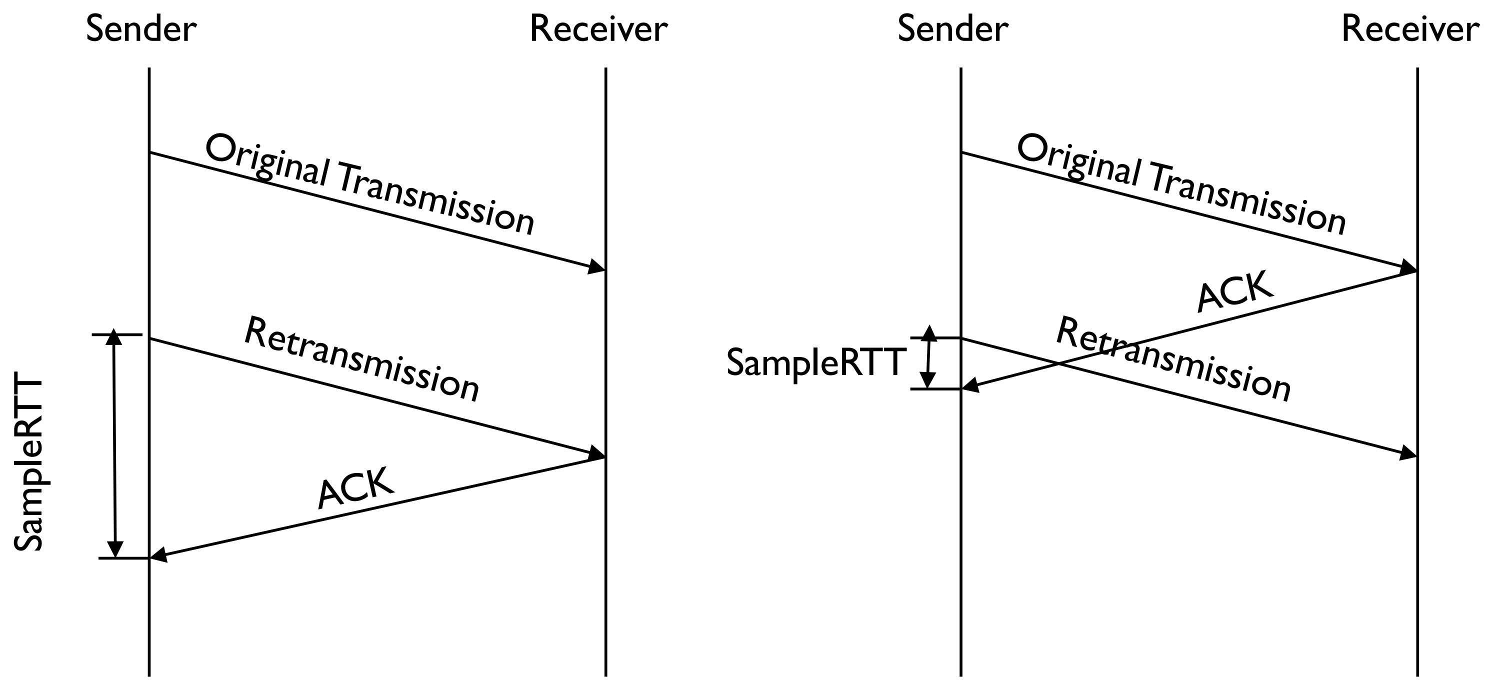
estimated RTT

"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?



TCP sender events:

PUTTING IT TOGETHER

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)

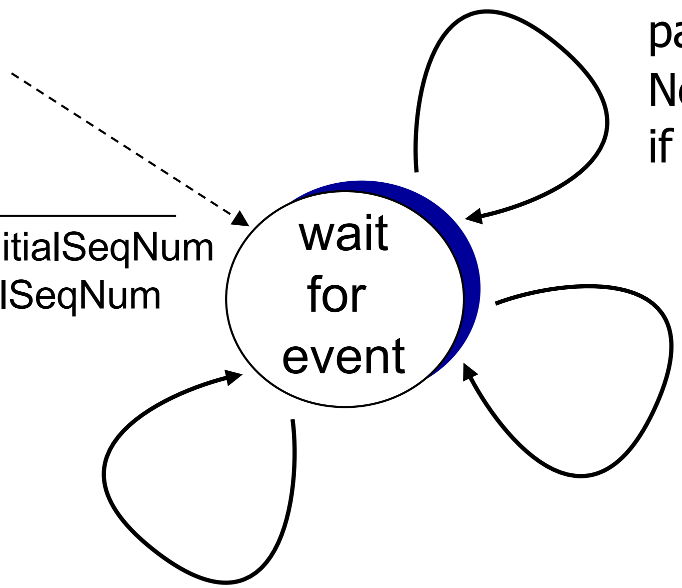
PUTTING IT TOGETHER

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



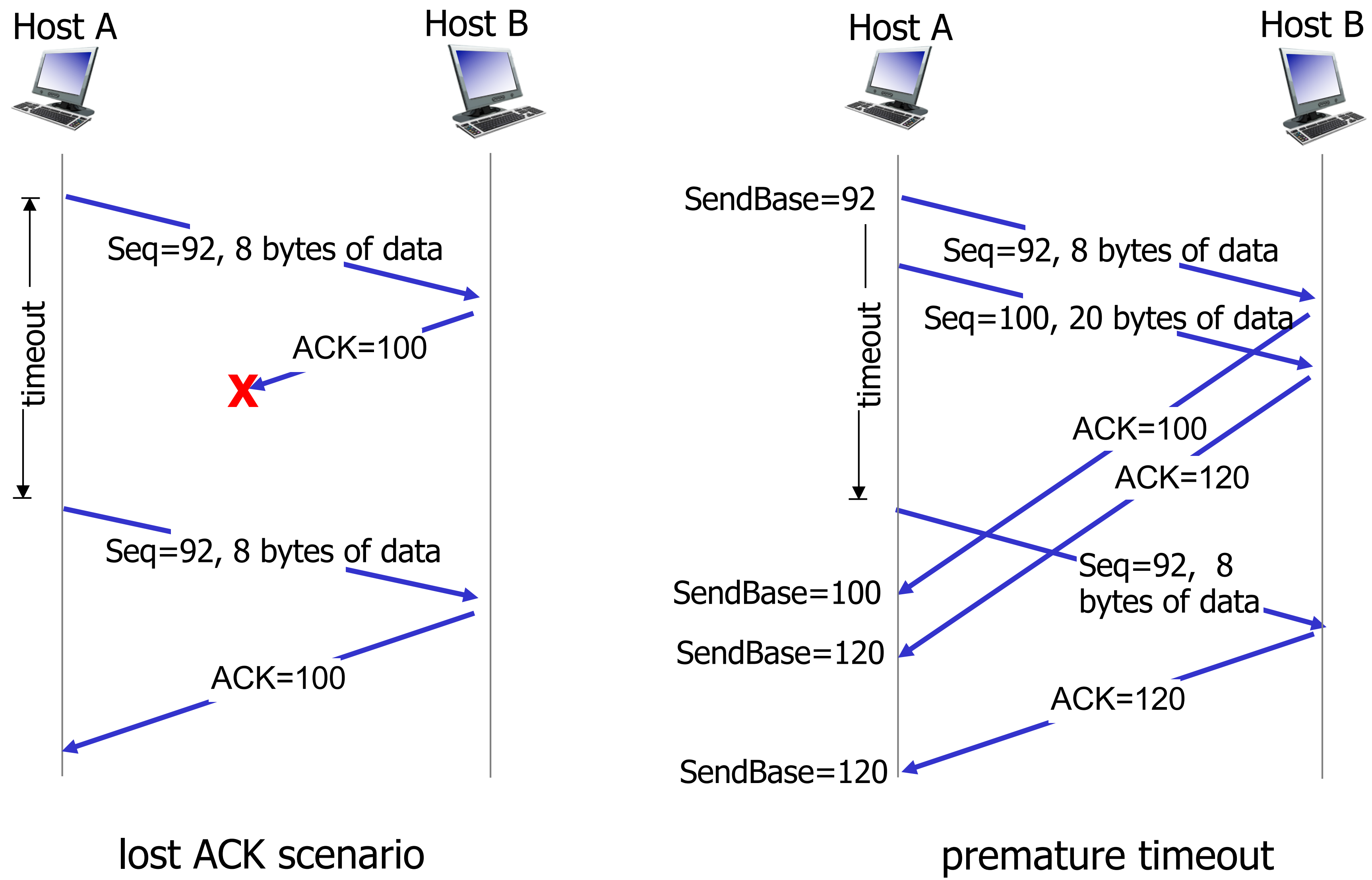
timeout

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

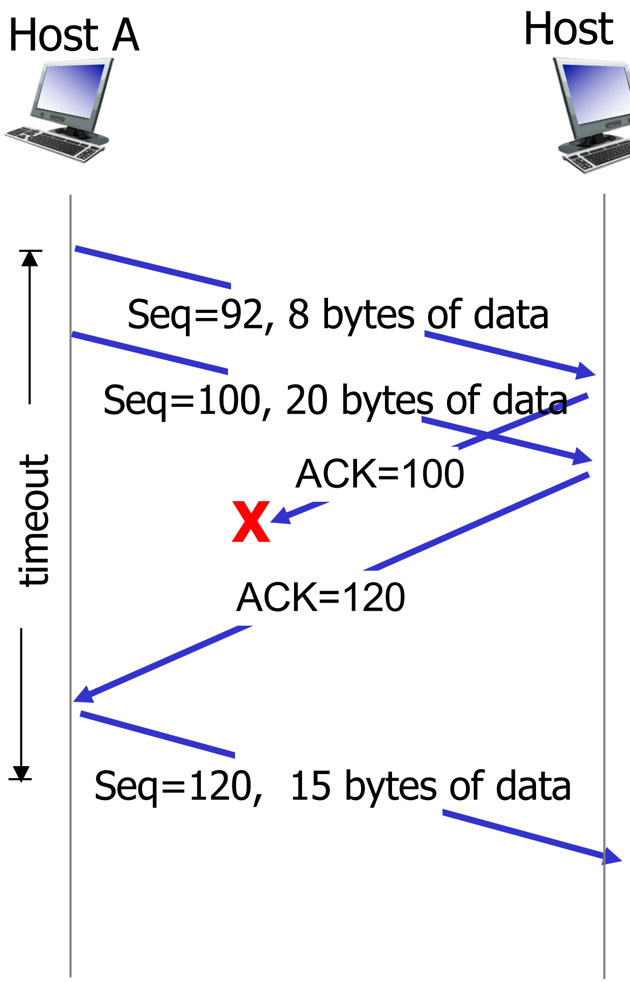
ACK received, with ACK field value y

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

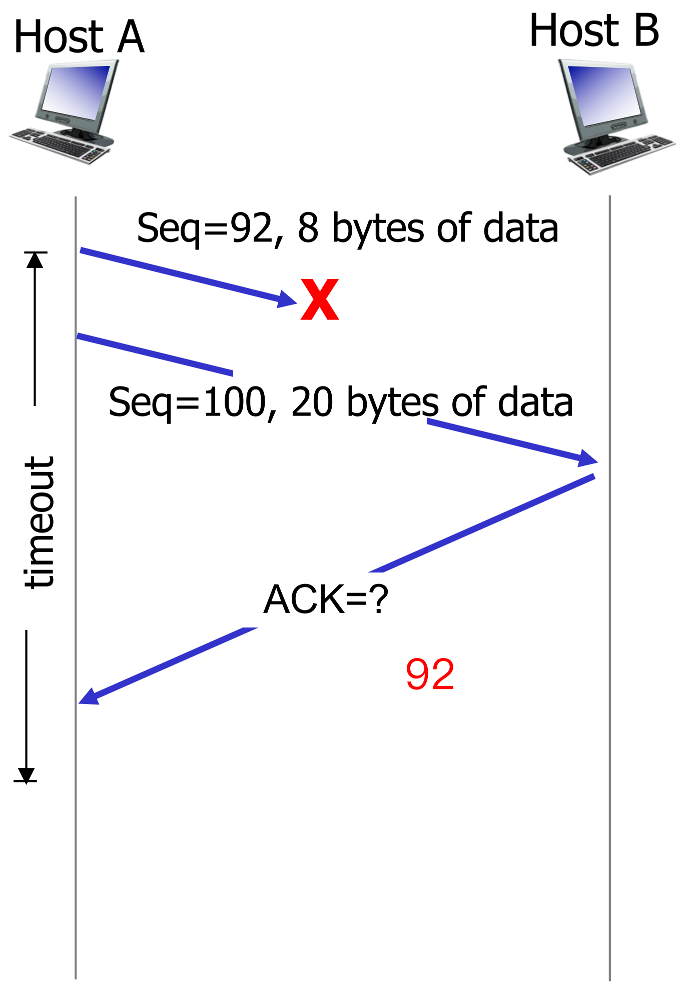
TCP: retransmission scenarios



TCP: retransmission scenarios



cumulative ACK



cumulative ACK

TCP ACK generation [RFC 1122, RFC 2581]

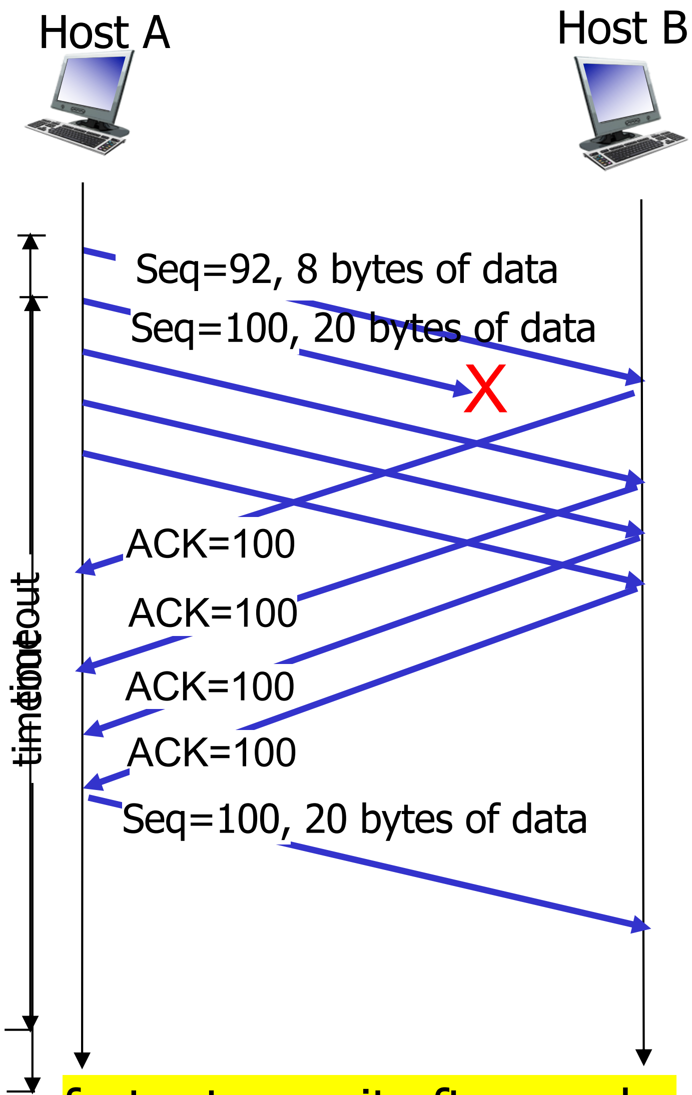
event at receiver	*TCP receiver action*
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately 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

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



Triple dup ACK = same ACK received 4 times back to back

fast retransmit after sender receipt of triple duplicate ACK

TCP fast retransmit

- time-out period often relatively long:
  - long delay before resending lost packet
- "Duplicate ACKs" are a sign of an isolated loss
  - The lack of ACK progress means that packet hasn't been delivered
  - Stream of ACKs means some packets are being delivered
  - Could trigger resend on receiving "k" duplicate ACKs (TCP uses k = 3)

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 is lost, so don't wait for timeout

What does TCP do?

Most of our previous ideas, but some key 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
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission

Quiz: TCP Sequence Numbers?

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

A. 1233

C depend on the ACK number

B. 436

C. 435

D. 1334

E. 536

Quiz: TCP Sequence Numbers?

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

A. Yes

B. No

Yes, because It can be a retransmission

A TCP Sender is just about to send a segment of size 100 bytes with sequence number 1234 and ack number 436 in the TCP header. Is it possible that the receiver has already received byte number 1333?

A. Yes

B. No

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? d

A. It will increase EstimatedRTT but leave timeout unchanged If estimatedRTT increase, timeout also increase

B. It will increase the timeout

C. Whether it increases EstimatedRTT will depend on the deviation Deviation affect the actual timeout

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

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

estimated RTT

"safety margin"

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

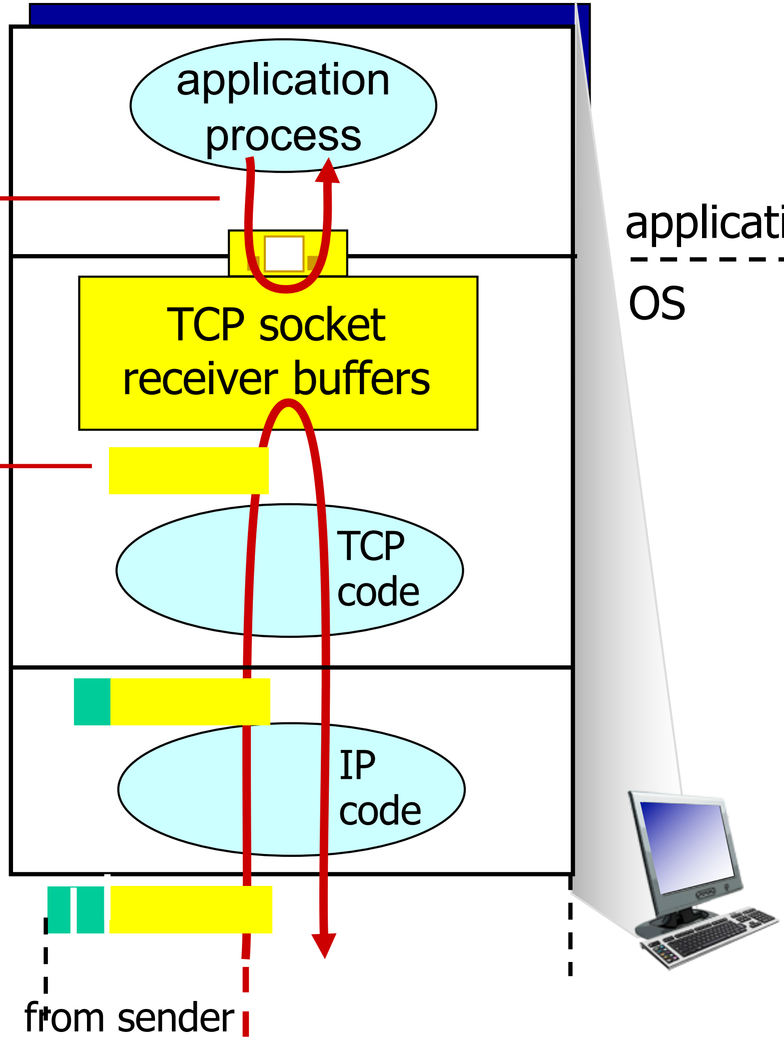
3.7 TCP congestion control

TCP flow control

So when u're receiving data very quickly, but the application layer at ur mobile phone unable to clean the data such quick at that way. So teh Tcp socket buffer may overflow

application may remove data from TCP socket buffers ….

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



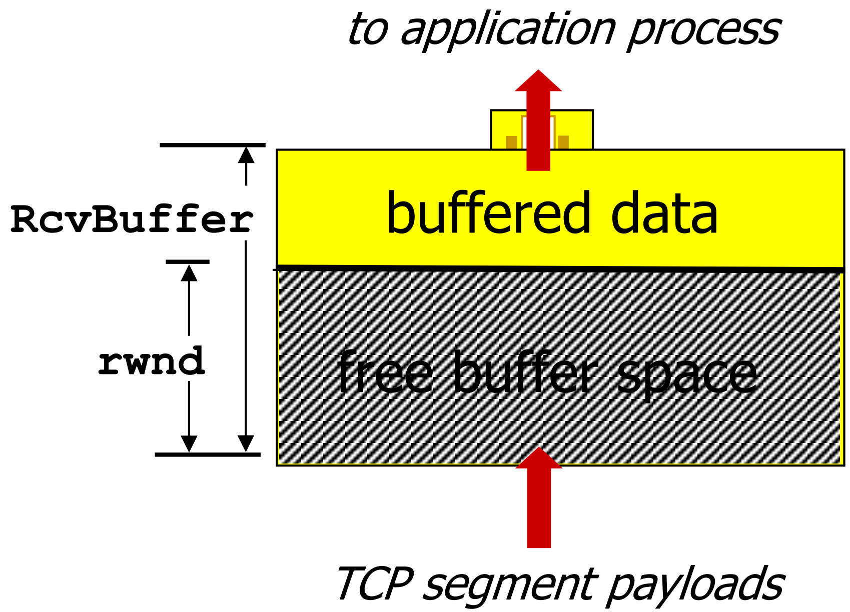
receiver protocol stack

flow control

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

TCP flow control

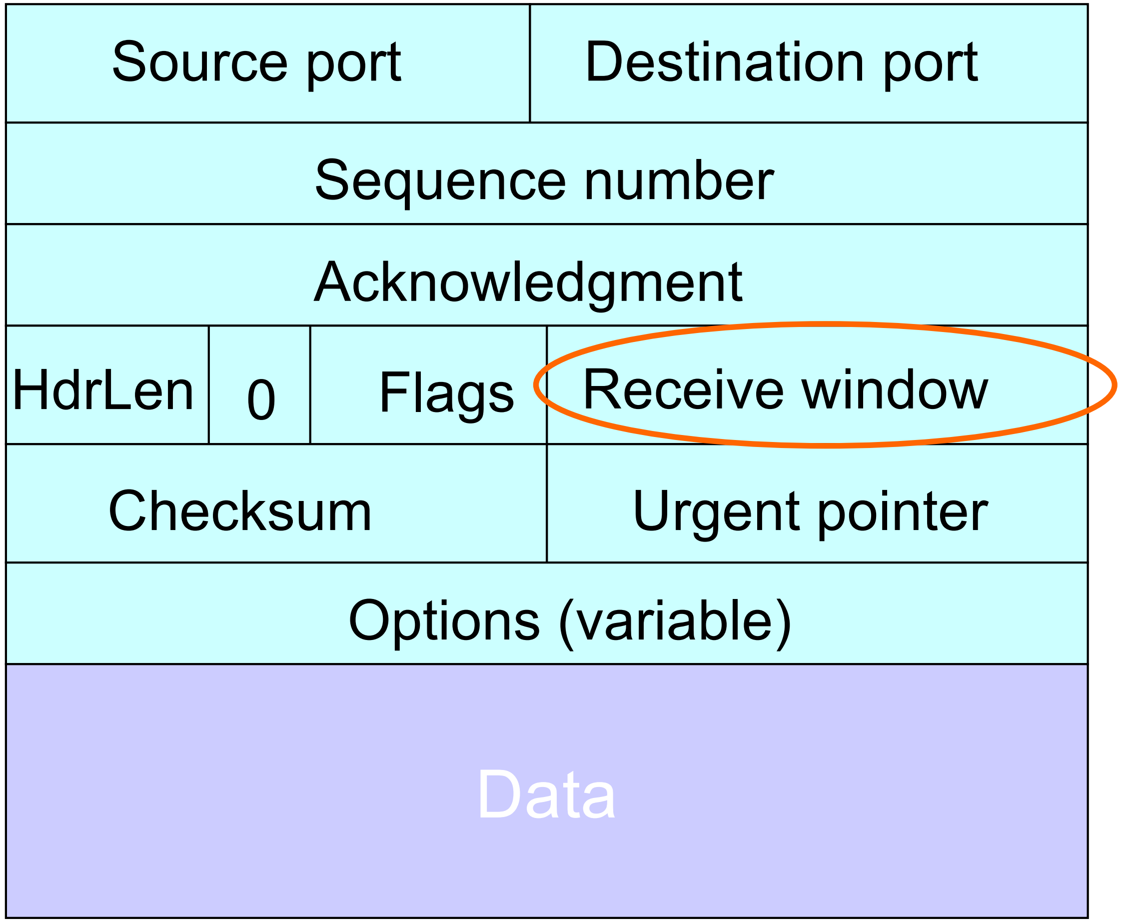
- receiver "advertises" free buffer space by including **`rwnd`** value in TCP header of receiver-to-sender segments
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unacked ("in-flight") data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



receiver-side buffering

http://media.pearsoncmg.com/aw/aw_kurose_network_4/applets/flow/FlowControl.htm

TCP Header



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

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

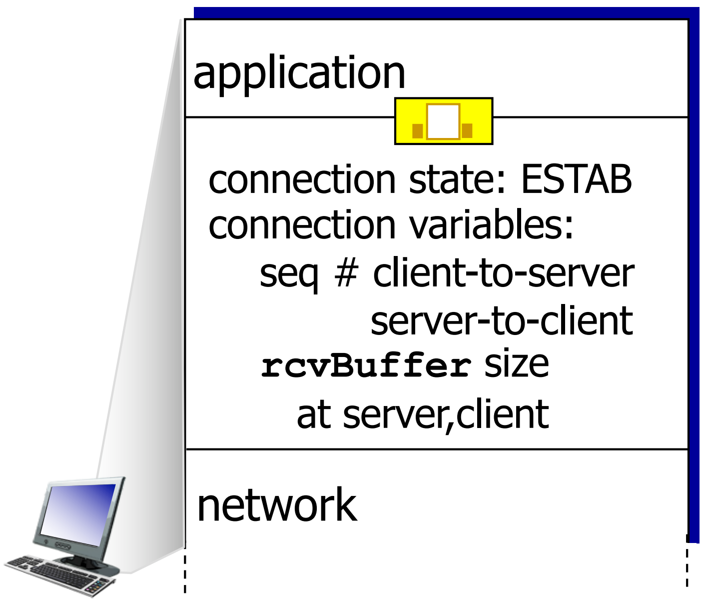
3.6 principles of congestion control

3.7 TCP congestion control

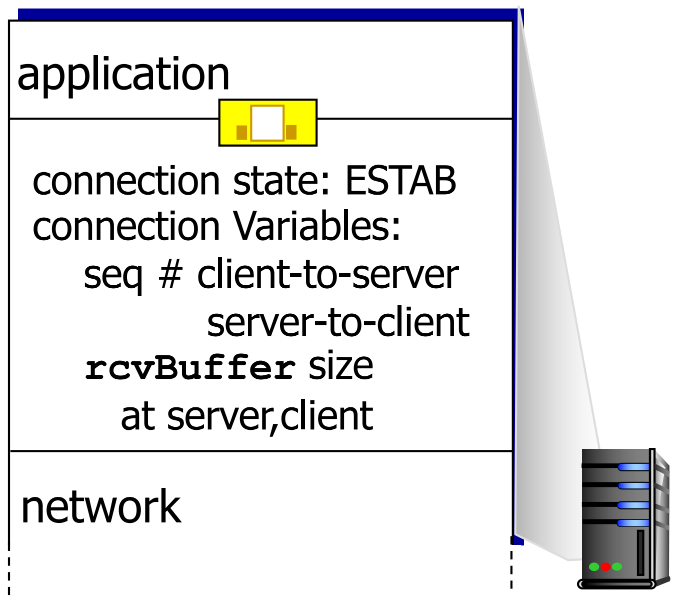
Connection Management

before exchanging data, sender/receiver "handshake":

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

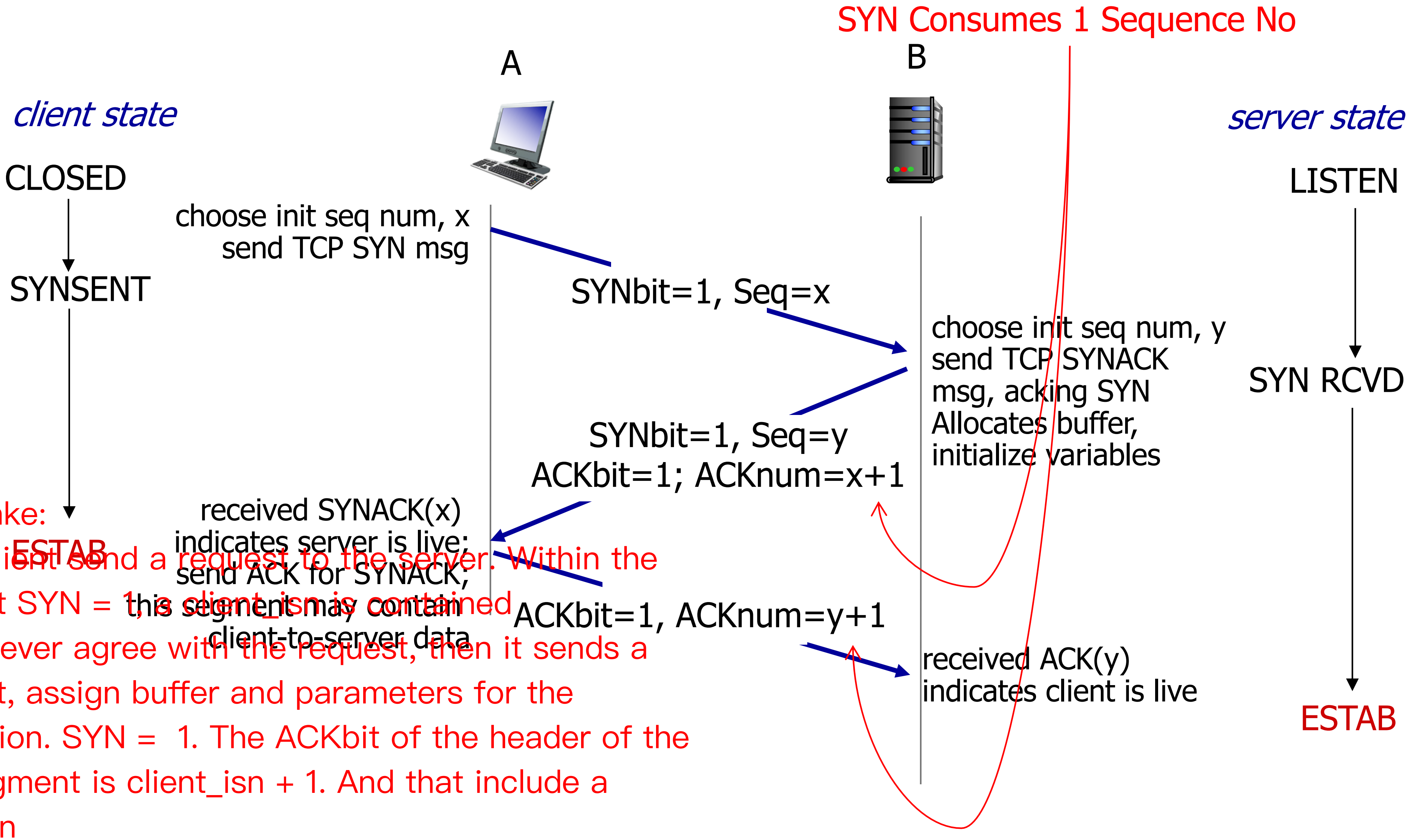



```
Socket clientSocket =
  newSocket("hostname","port
   number");
```




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

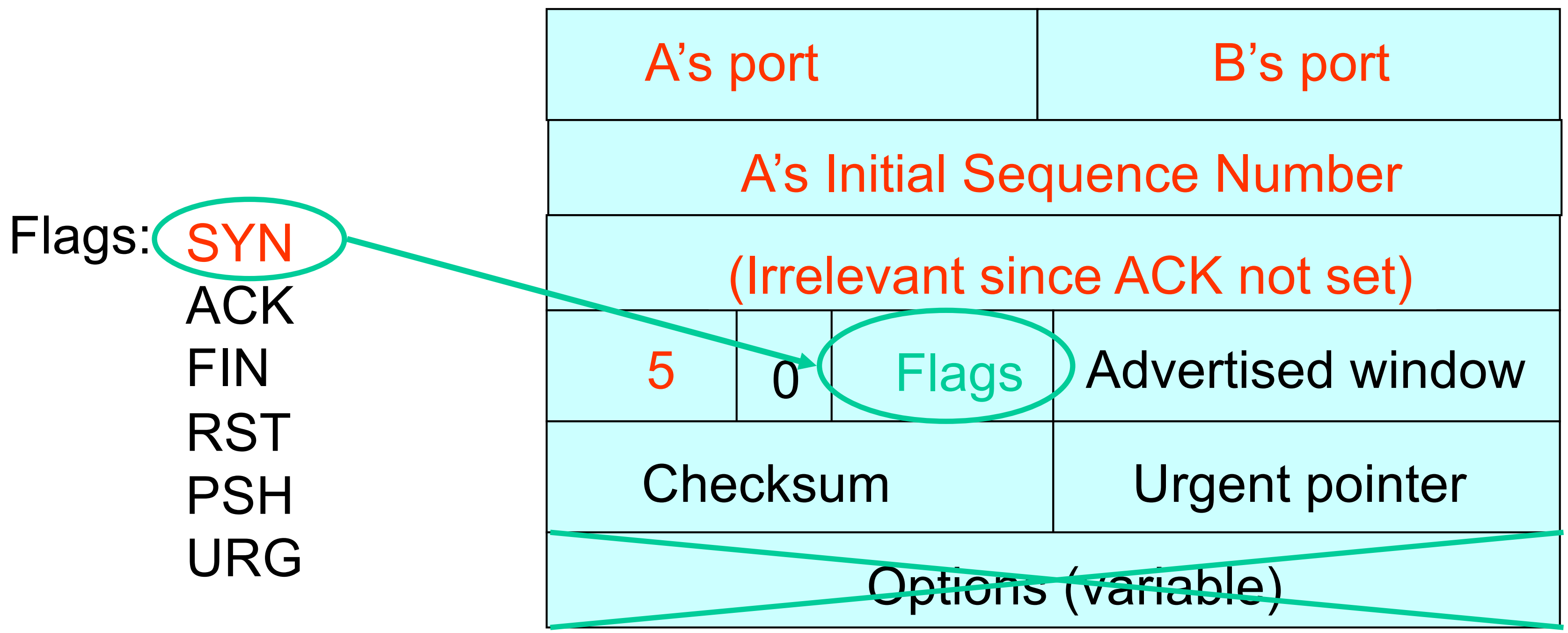
TCP 3-way handshake



handshake:

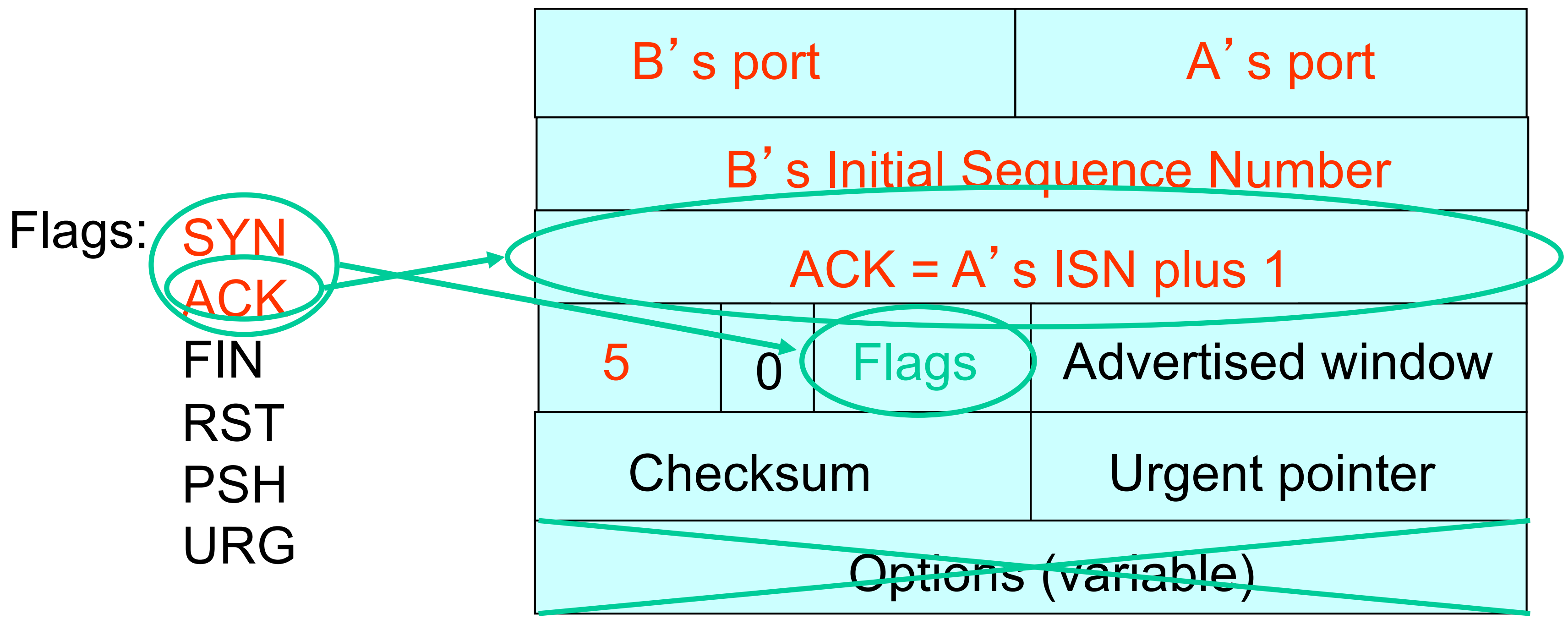
1. The client send a request to the server. Within the segment SYN = 1, a client_isn is contained
2. The sever agree with the request, then it sends a segment, assign buffer and parameters for the connection. SYN = 1. The ACKbit of the header of the TCP segment is client_isn + 1. And that include a sever_isn
3. After accept the reply, client assign buffer and parameters for the connection. And confirm the reply by placing server_isn + 1 at the ACKbit of header

Step 1: A’s Initial SYN Packet



A tells B it wants to open a connection…

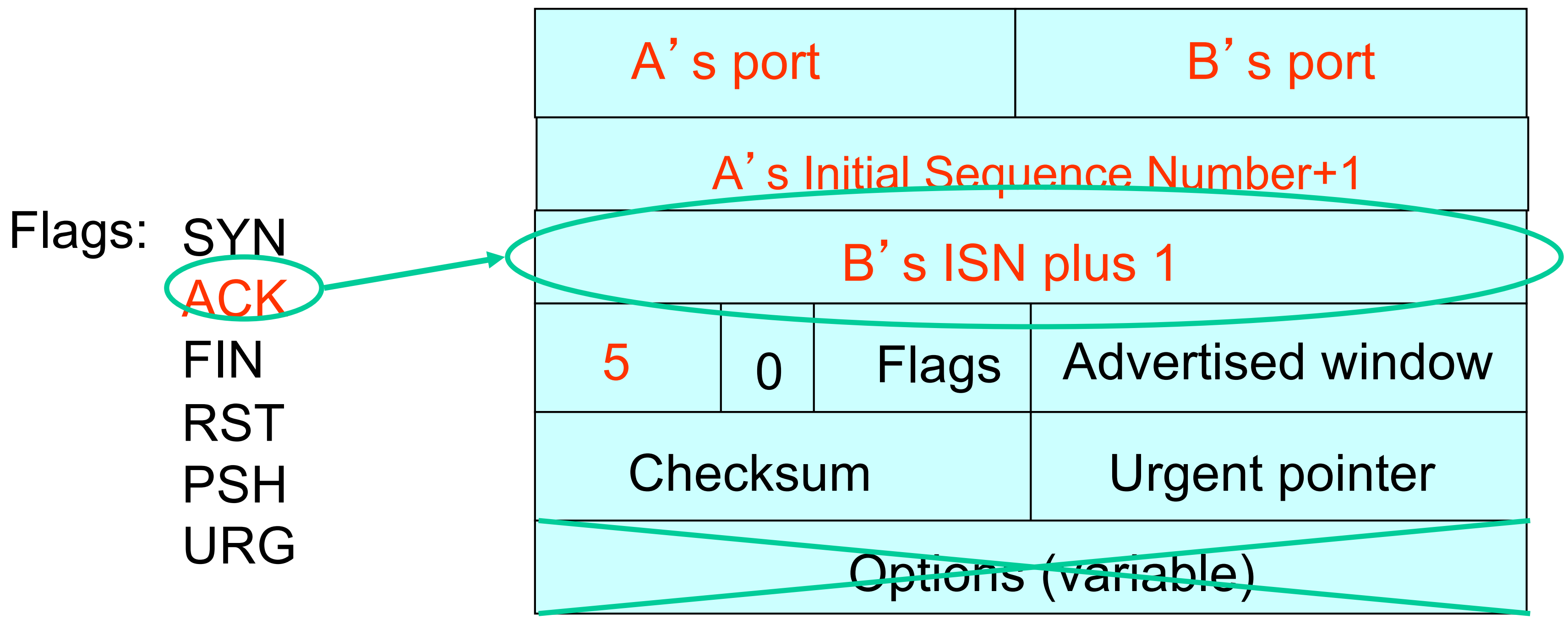
Step 2: B’s SYN-ACK Packet



B tells A it accepts, and is ready to hear the next byte…

… upon receiving this packet, A can start sending data

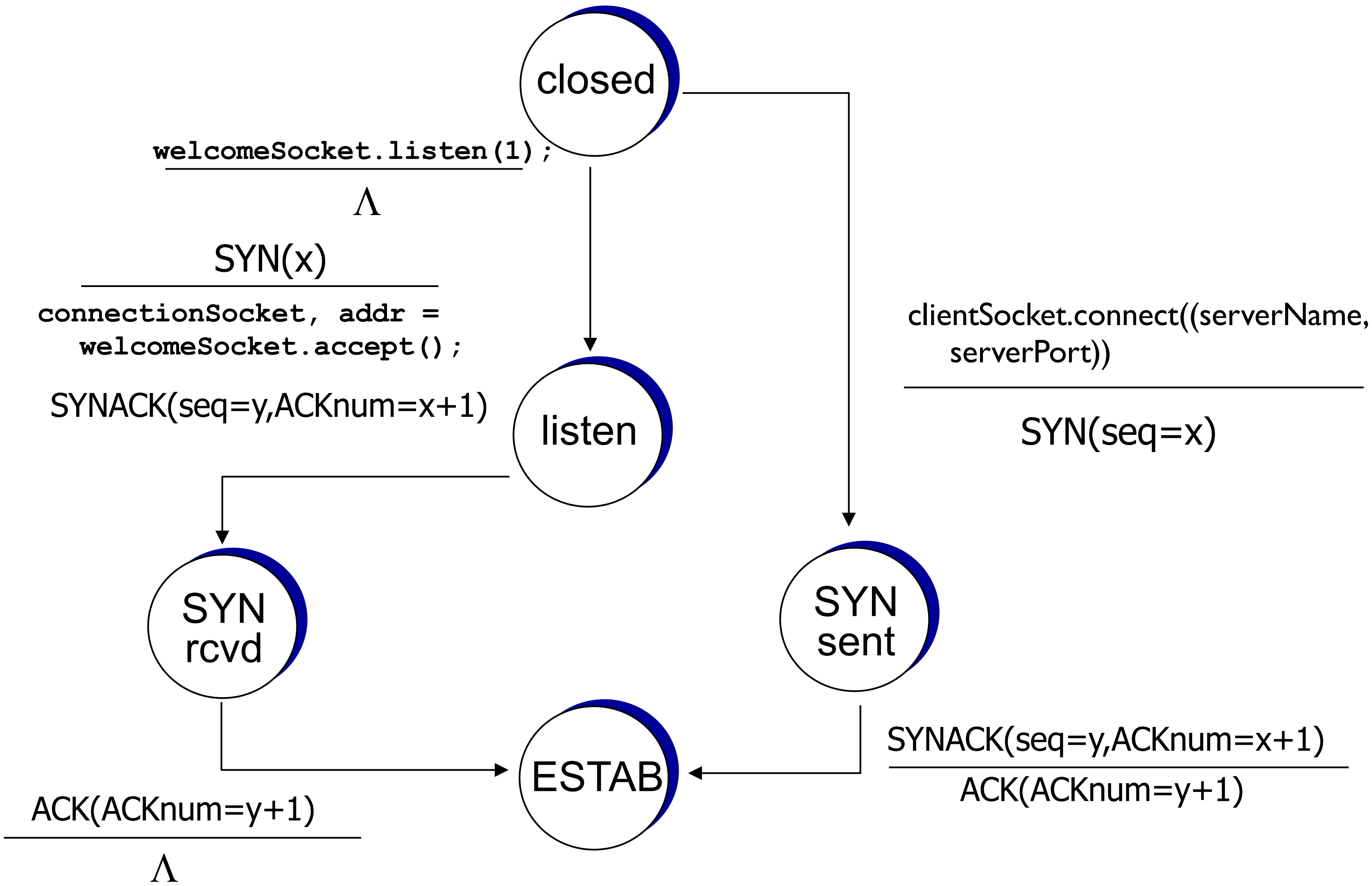
Step 3: A’s ACK of the SYN-ACK



A tells B it’s likewise okay to start sending

… upon receiving this packet, B can start sending data

TCP 3-way handshake: FSM



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

SYN Loss and Web Downloads

- User clicks on a hypertext link
  - Browser creates a socket and does a "connect"
  - The "connect" triggers the OS to transmit a SYN
- If the SYN is lost…
  - 1-3 seconds of delay: can be very long
  - User may become impatient
  - … and click the hyperlink again, or click "reload"

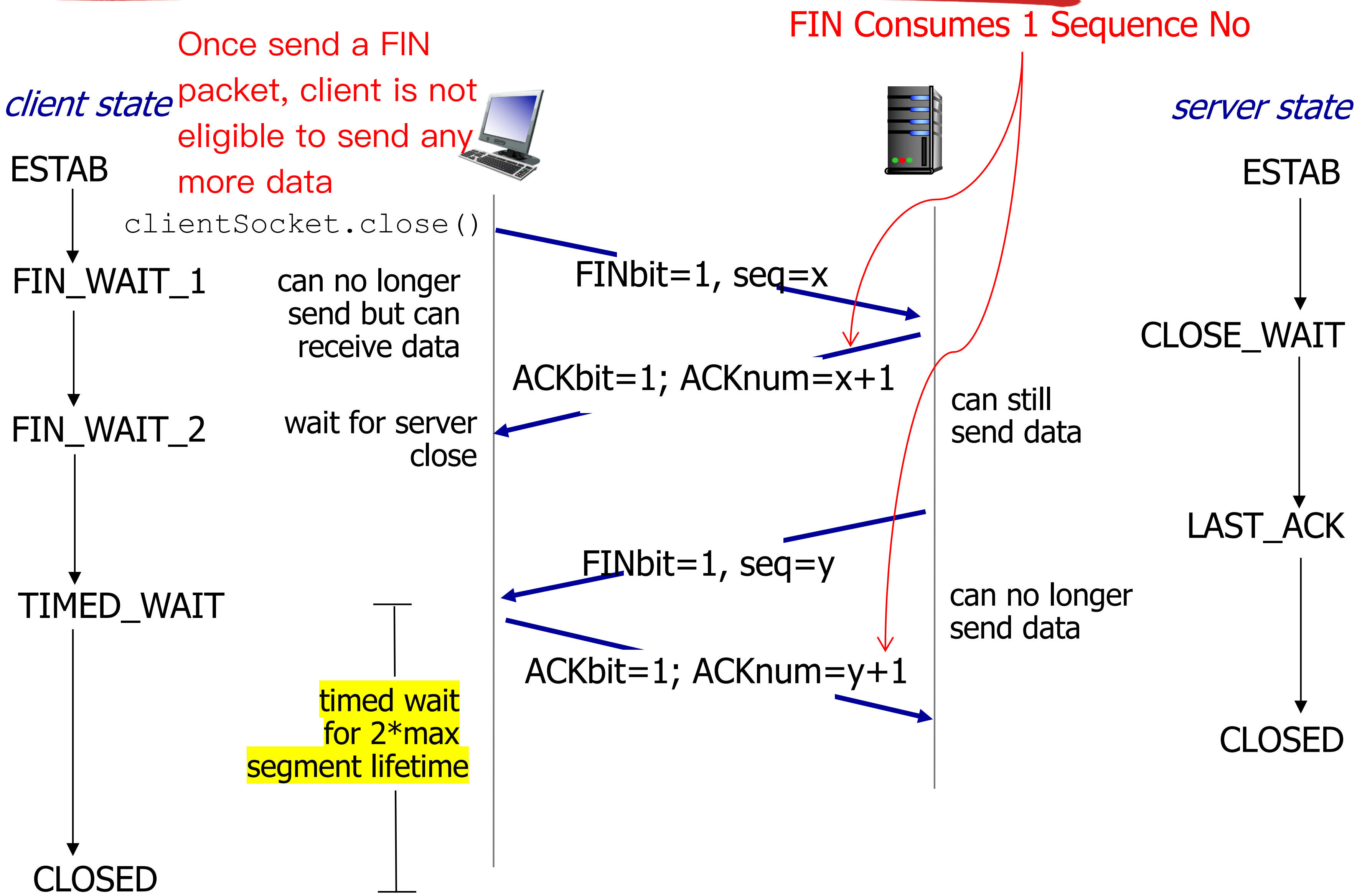
That trigger a retransmission

- User triggers an "abort" of the "connect"
  - Browser creates a new socket and another "connect"
  - Essentially, forces a faster send of a new SYN packet!
  - Sometimes very effective, and the page comes quickly

TCP: closing a connection

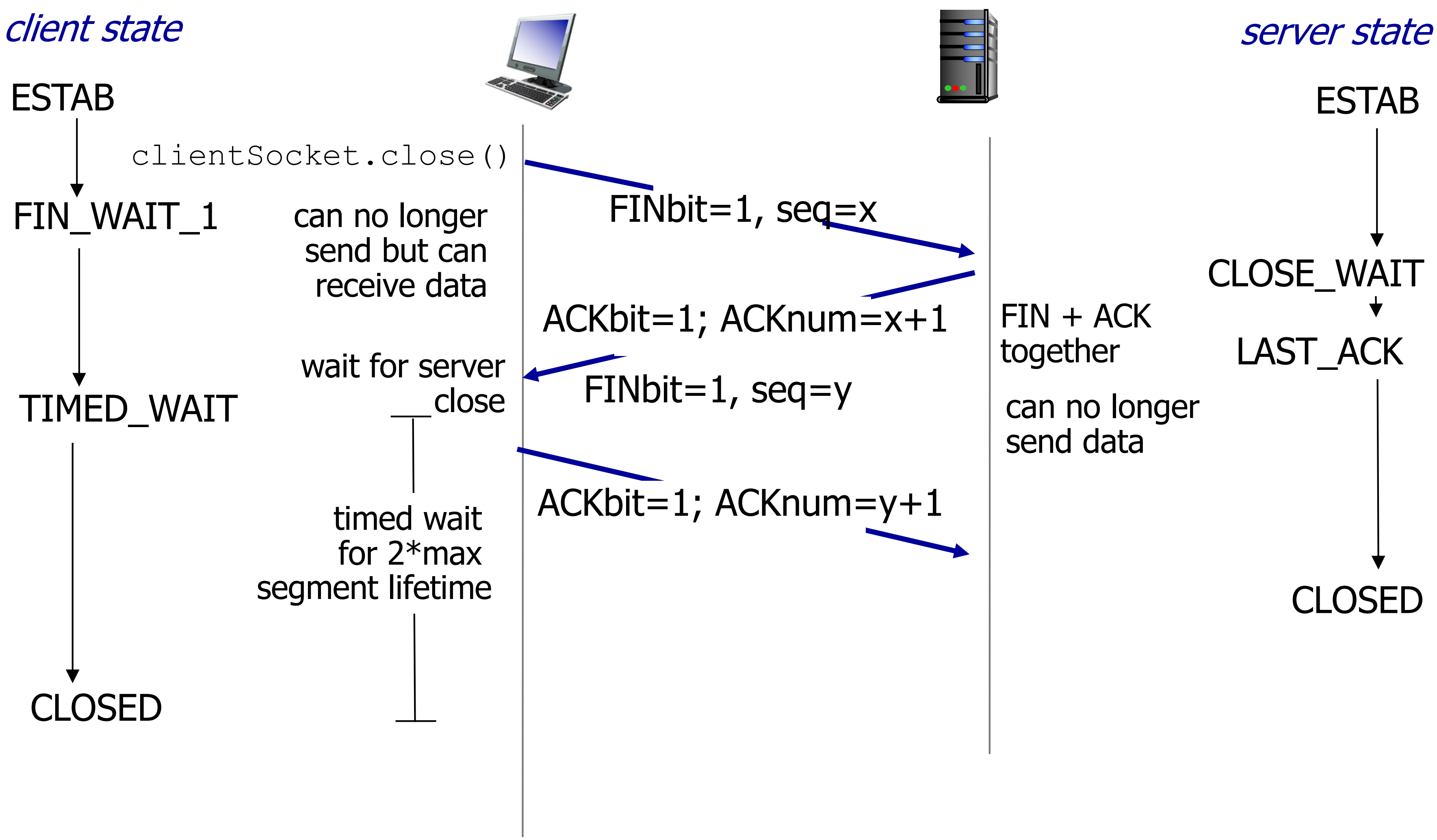
- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

Normal Termination, One at a Time

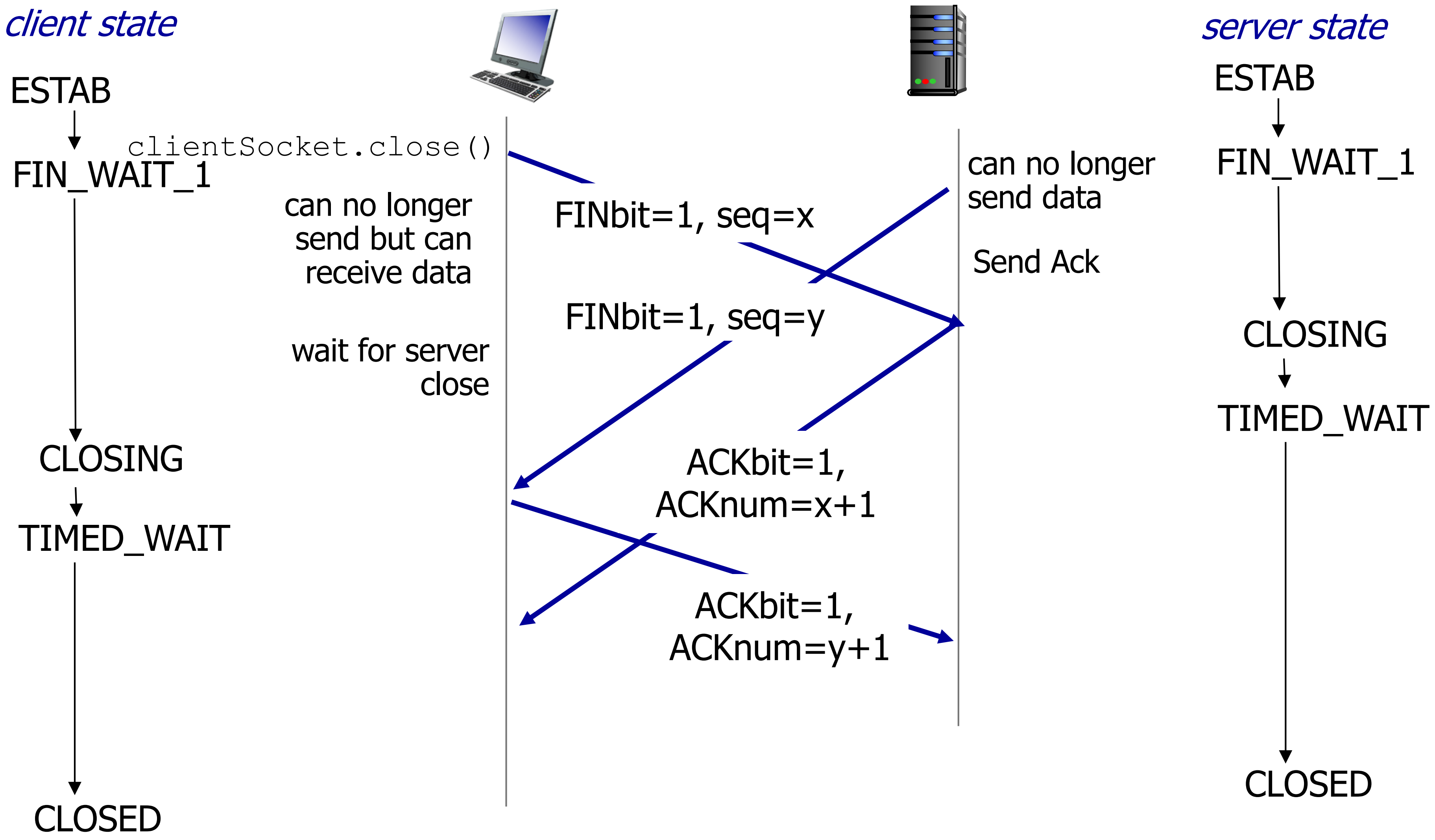


TIMED_WAIT: Can retransmit ACK if last ACK is lost !!!!

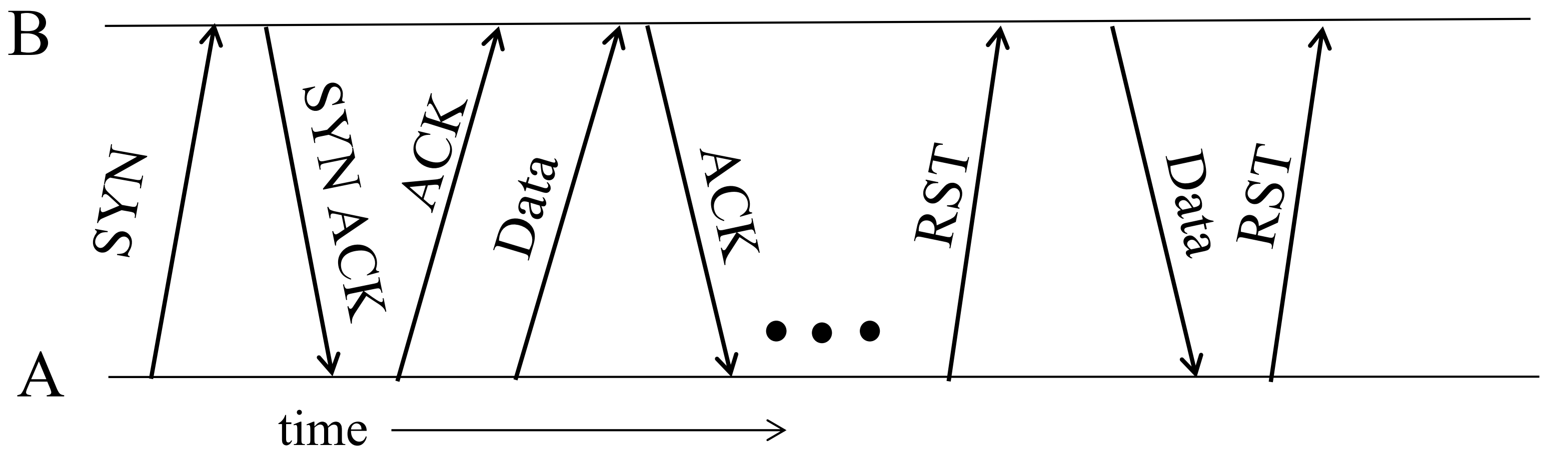
Normal Termination, Both Together



Simultaneous Closure

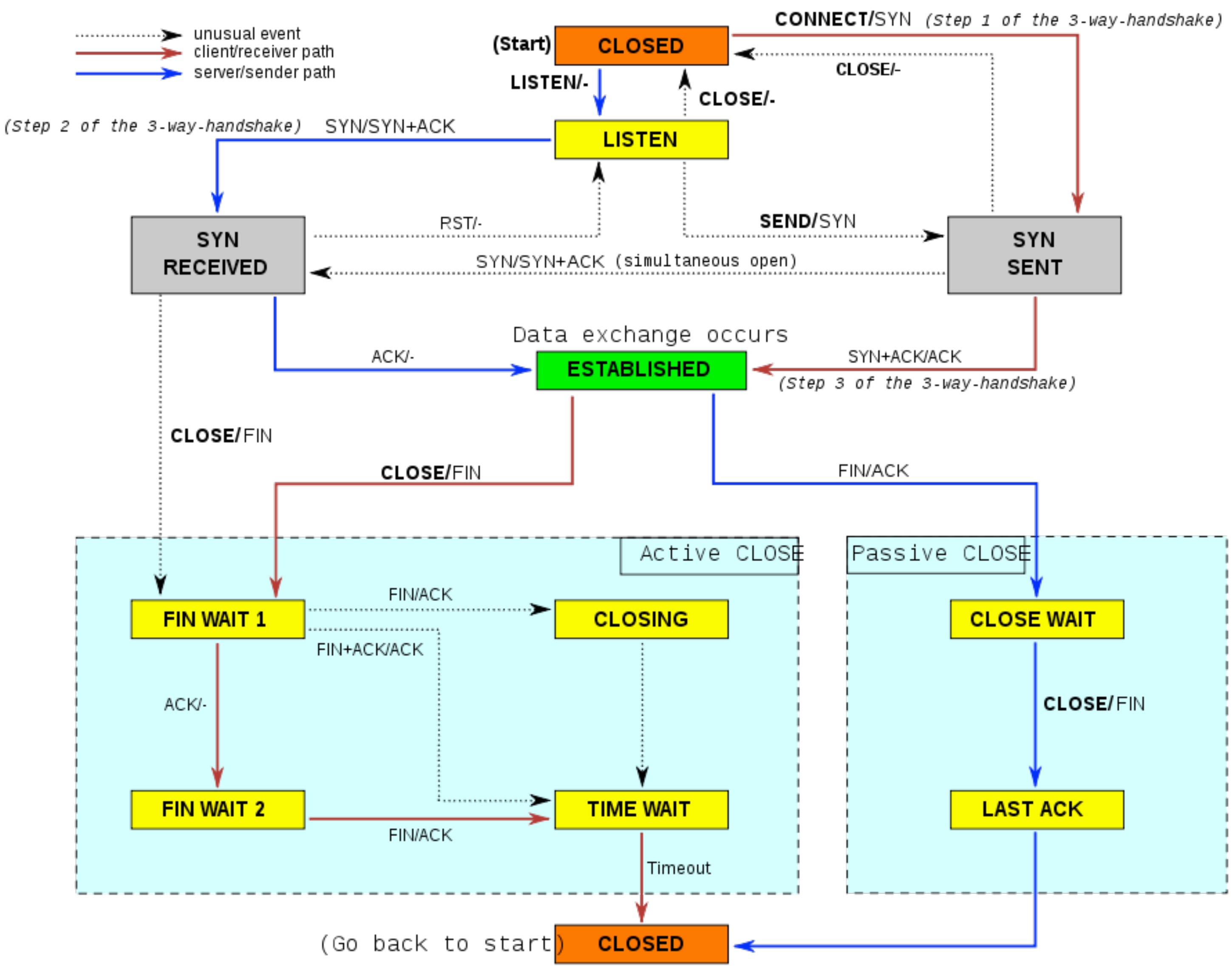


Abrupt Termination



- 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　　引出
  - But: if B sends anything more, will elicit another **RST**

TCP Finite State Machine



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?

Roughly how much time does it take for both the TCP Sender and Receiver to establish connection state since the connect() call?

A. RTT

B. 1.5RTT

C. 2RTT

D. 3RTT

3 hand-shake

B

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?

B

A. Per-byte sequence and acknowledgement numbers

B. Triple Duplicate ACKs

C. Receiver window-based flow control

D. Timeout estimation algorithm

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

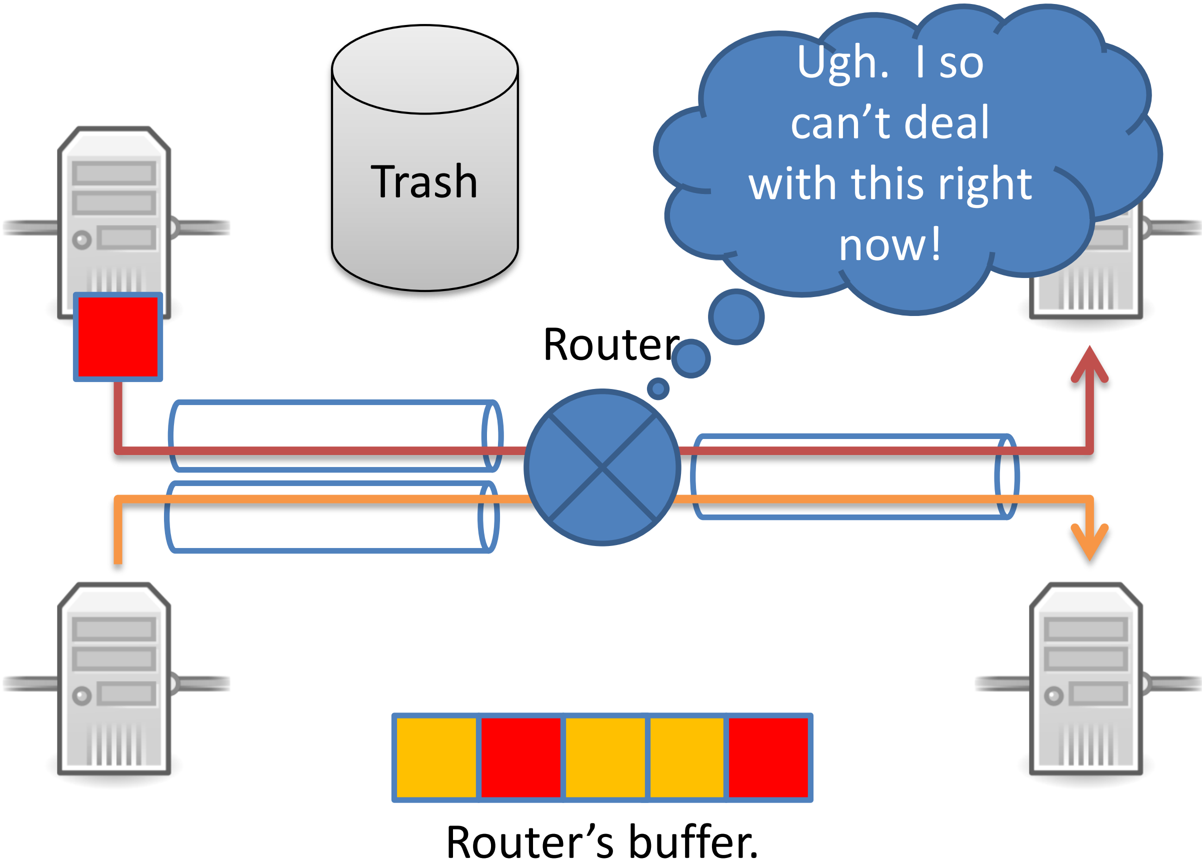
congestion:

- 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!

Difference on flow and congestion control:

1. Flow: to prevent overflow of receiver and buffer at the TCP socket
2. Congestion: overwhelm on the network, not the devices

Congestion



Incoming rate is faster than outgoing link can support.

Quiz: What's the worst that can happen?

B –> C –> D

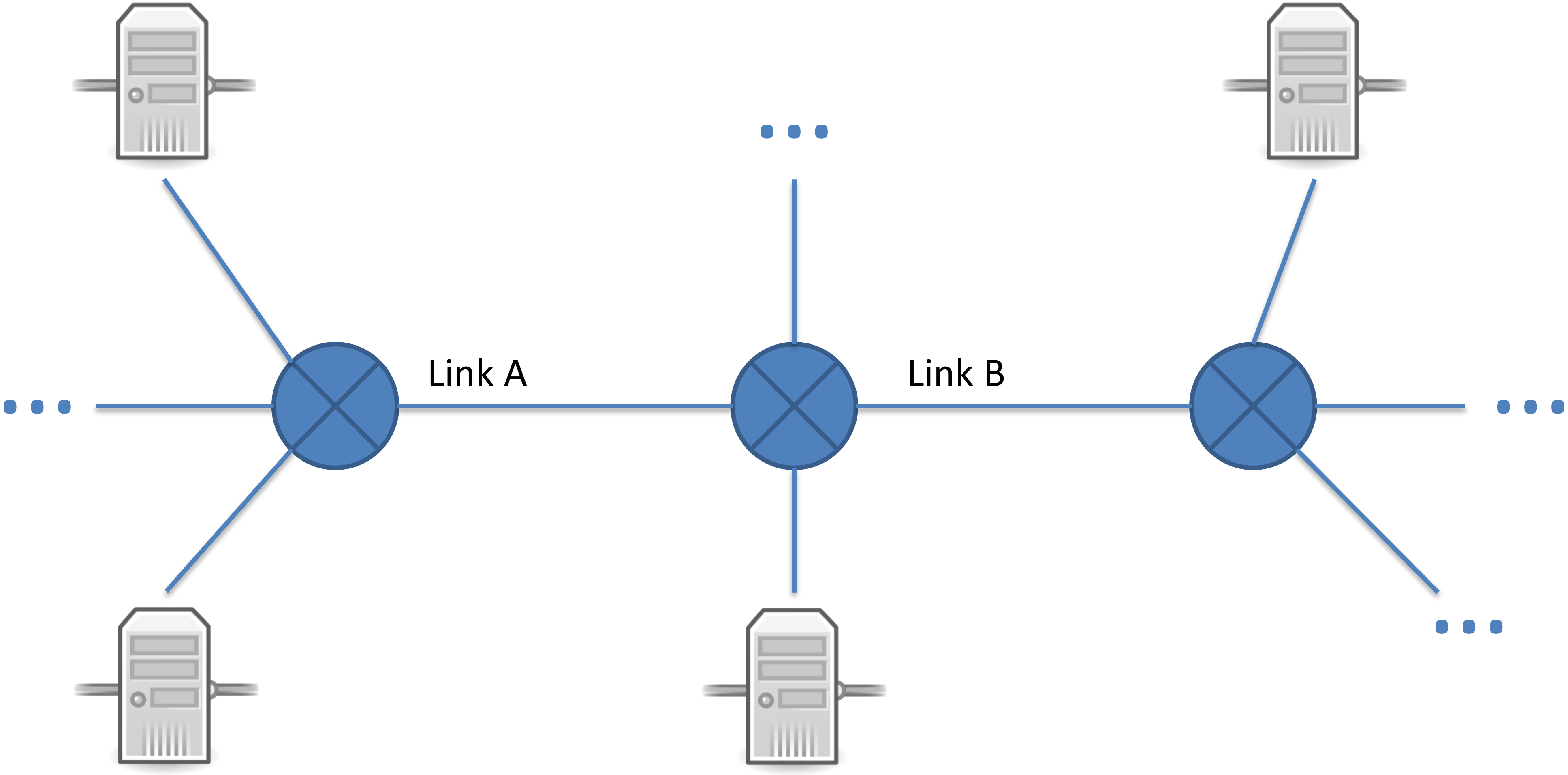
A: This is no problem. Senders just keep transmitting, and it'll all work out.

B: There will be retransmissions, but the network will still perform without much trouble.

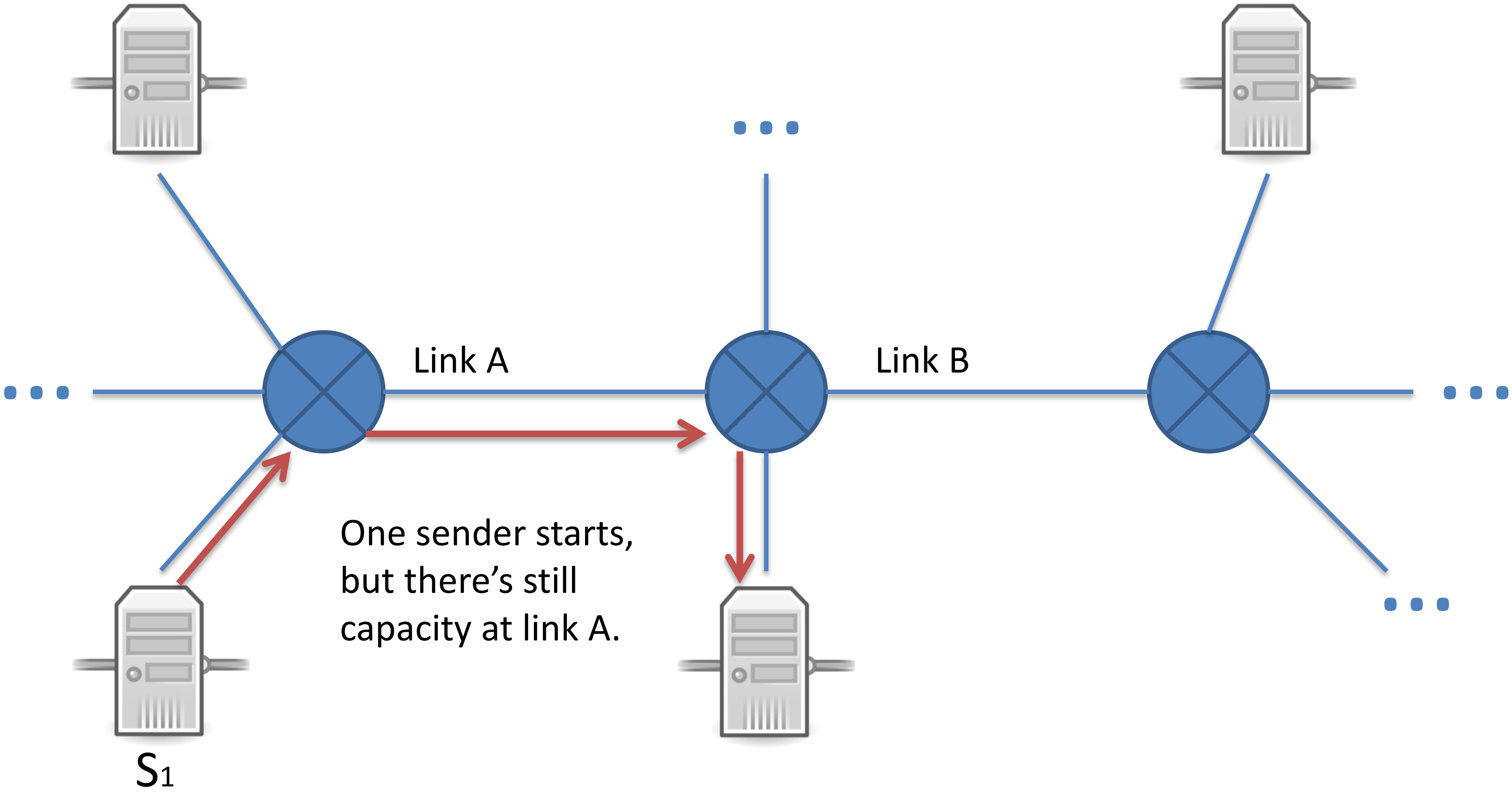
C: Retransmissions will become very frequent, causing a serious loss of efficiency

D: The network will become completely unusable

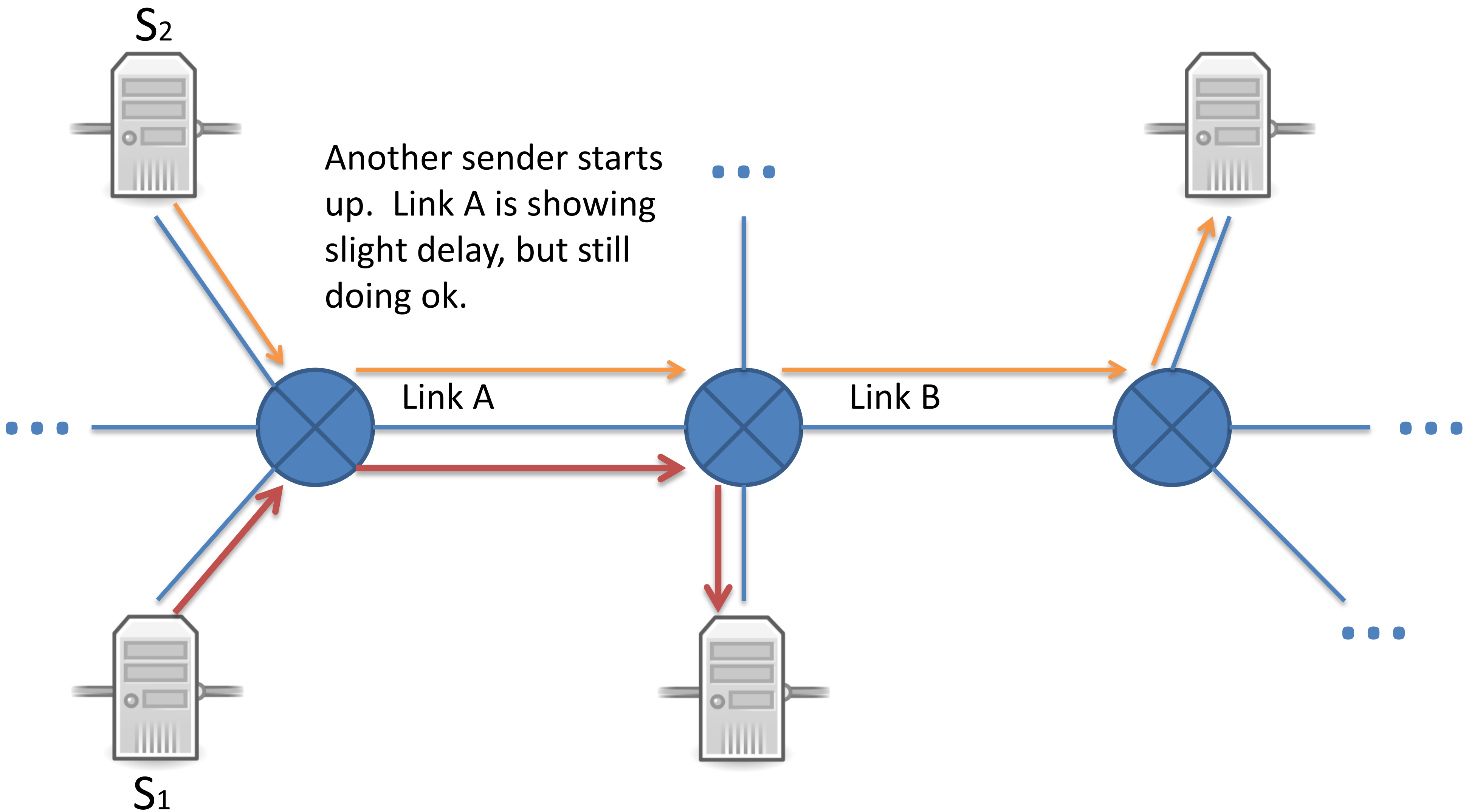
Congestion Collapse



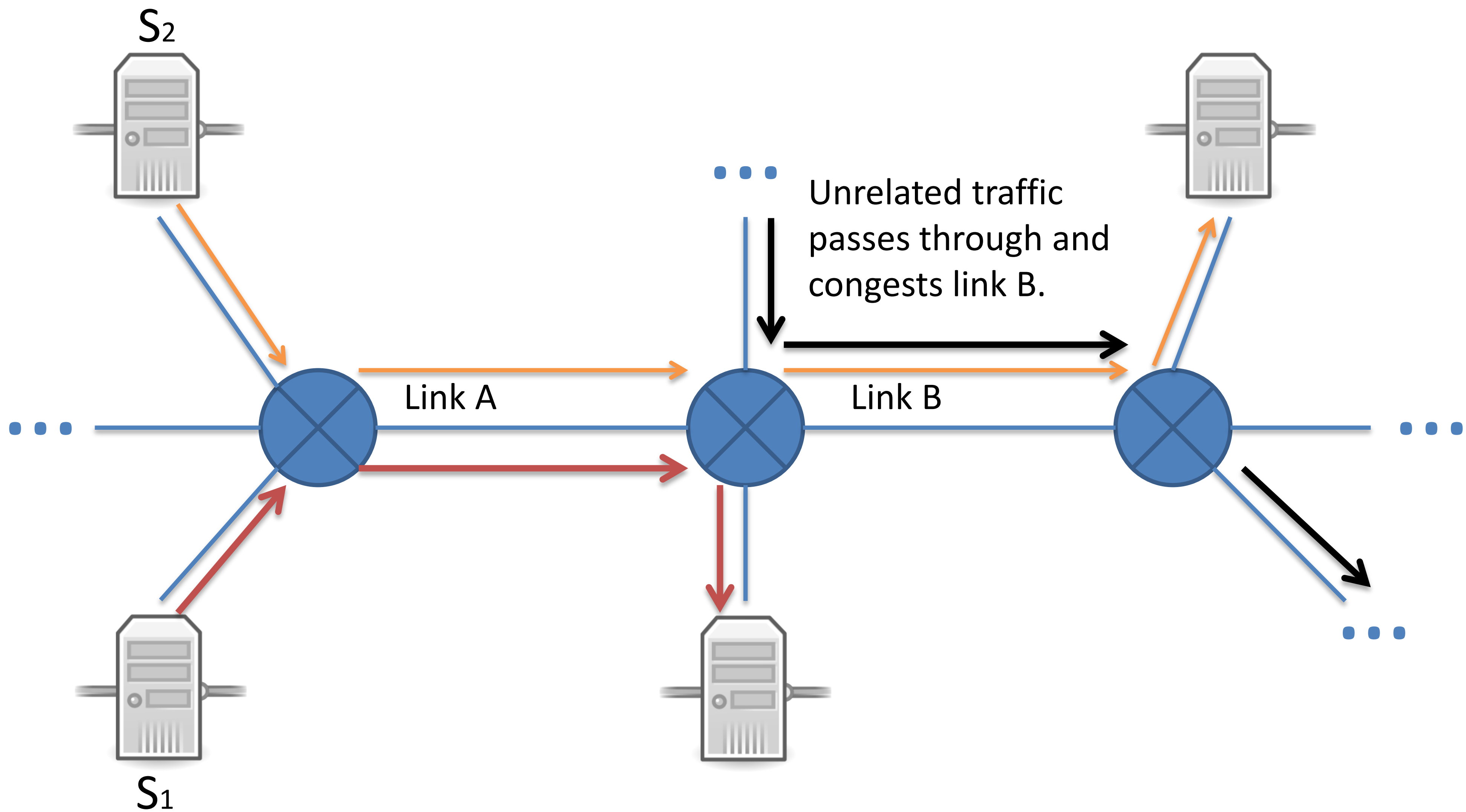
Congestion Collapse



Congestion Collapse

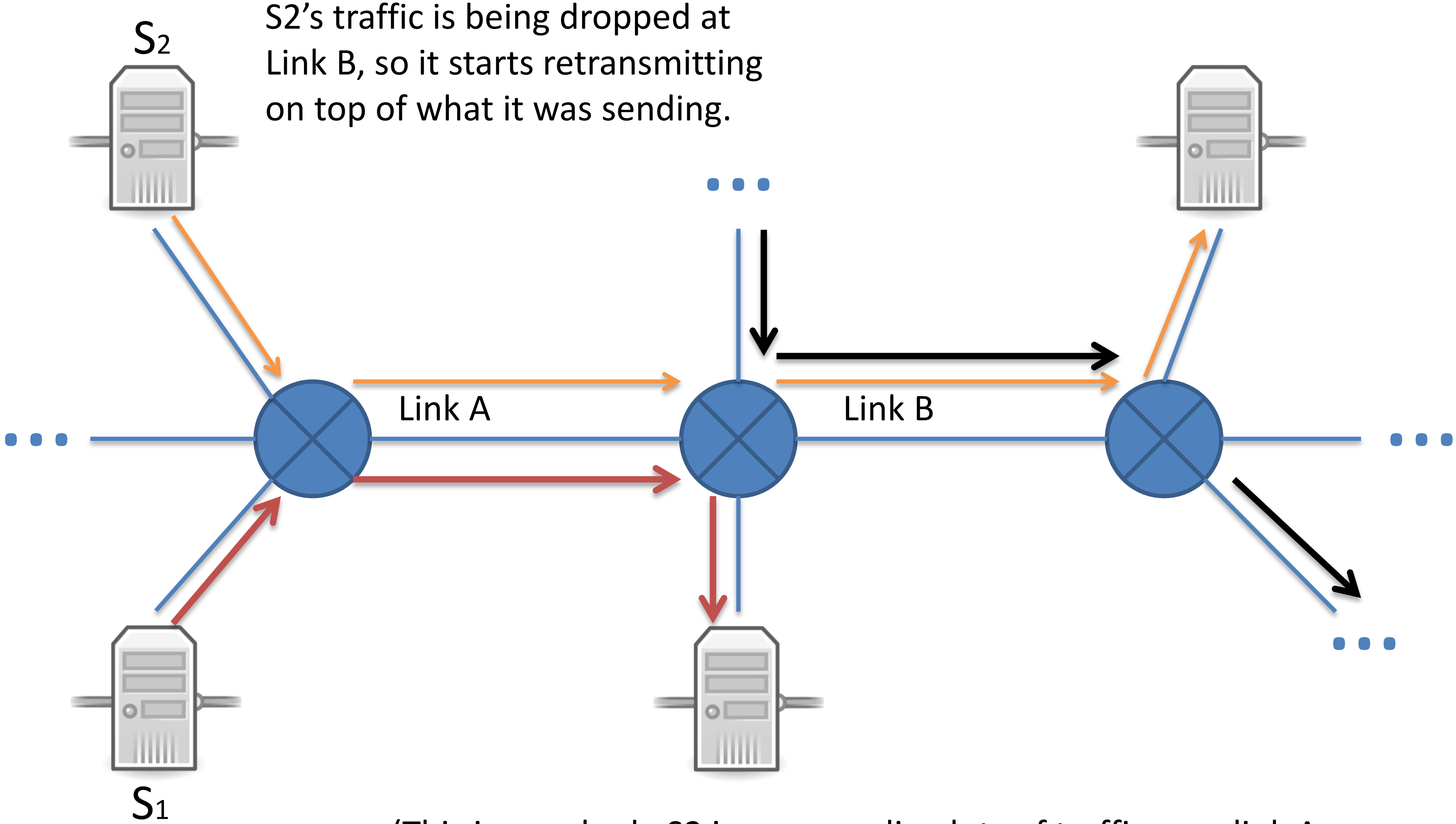


Congestion Collapse



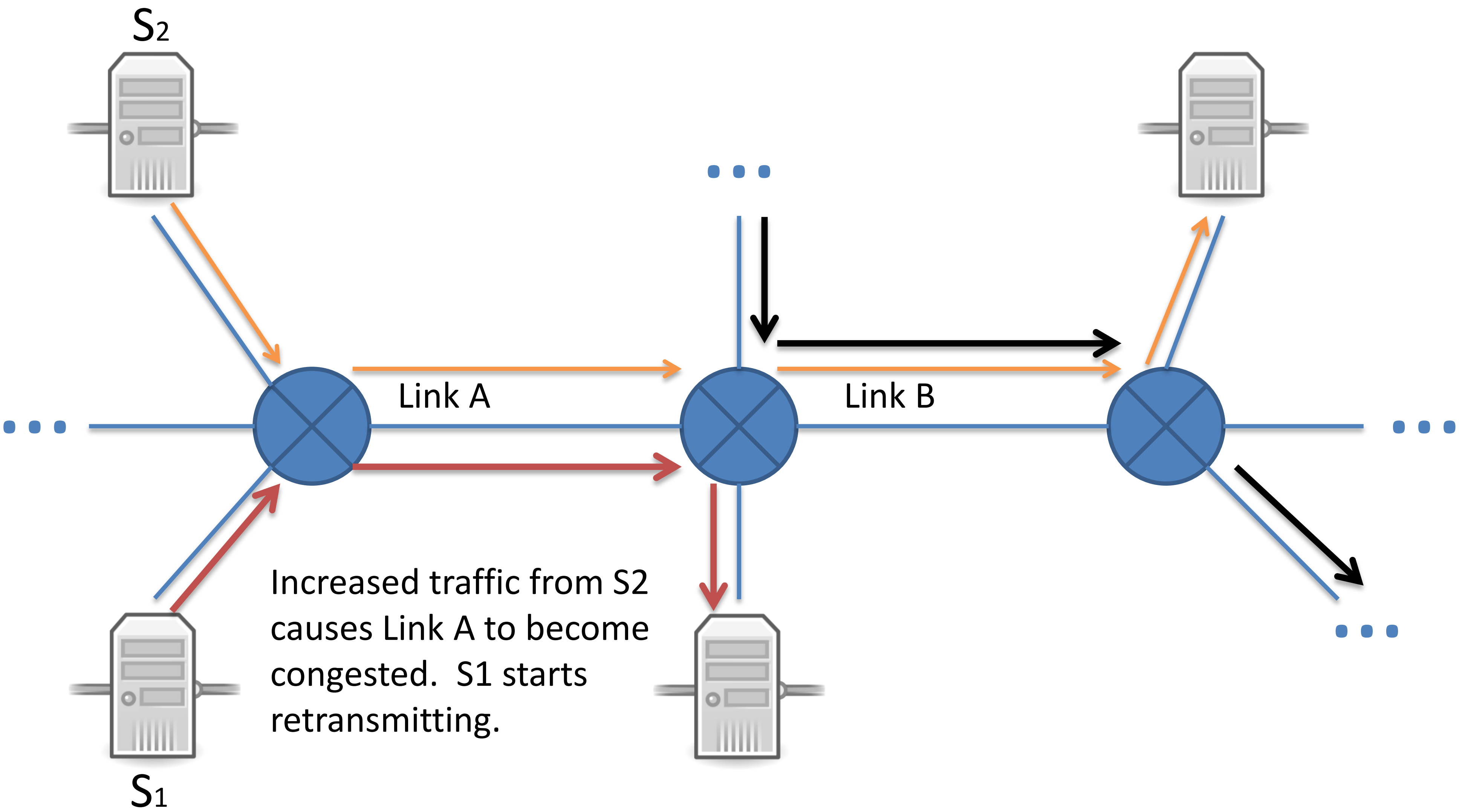
Congestion Collapse

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



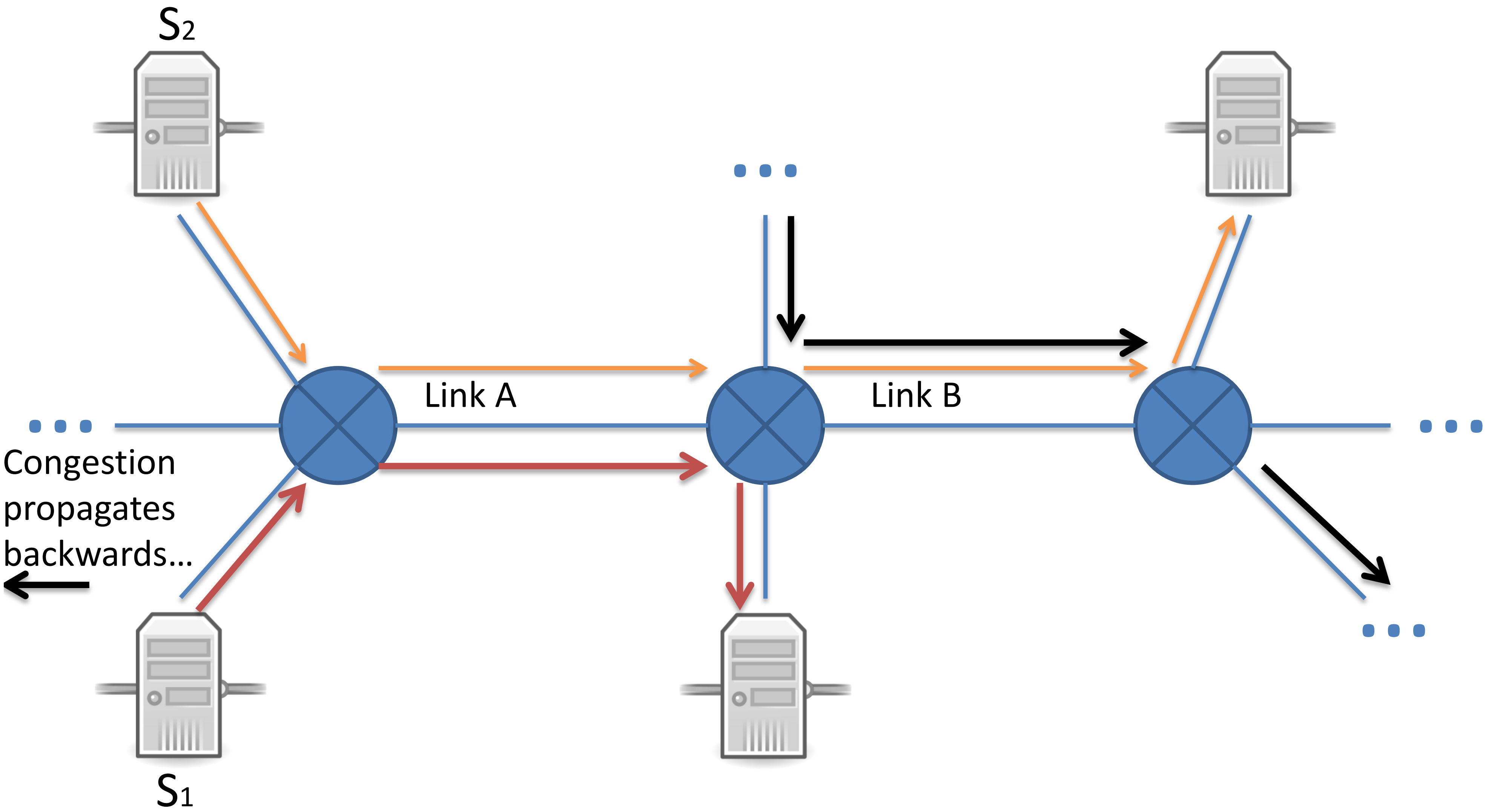
(This is very bad. S2 is now sending lots of traffic over link A that has no hope of crossing link B.)

Congestion Collapse



Increased traffic from S2 causes Link A to become congested. S1 starts retransmitting.

Congestion Collapse



Without congestion control

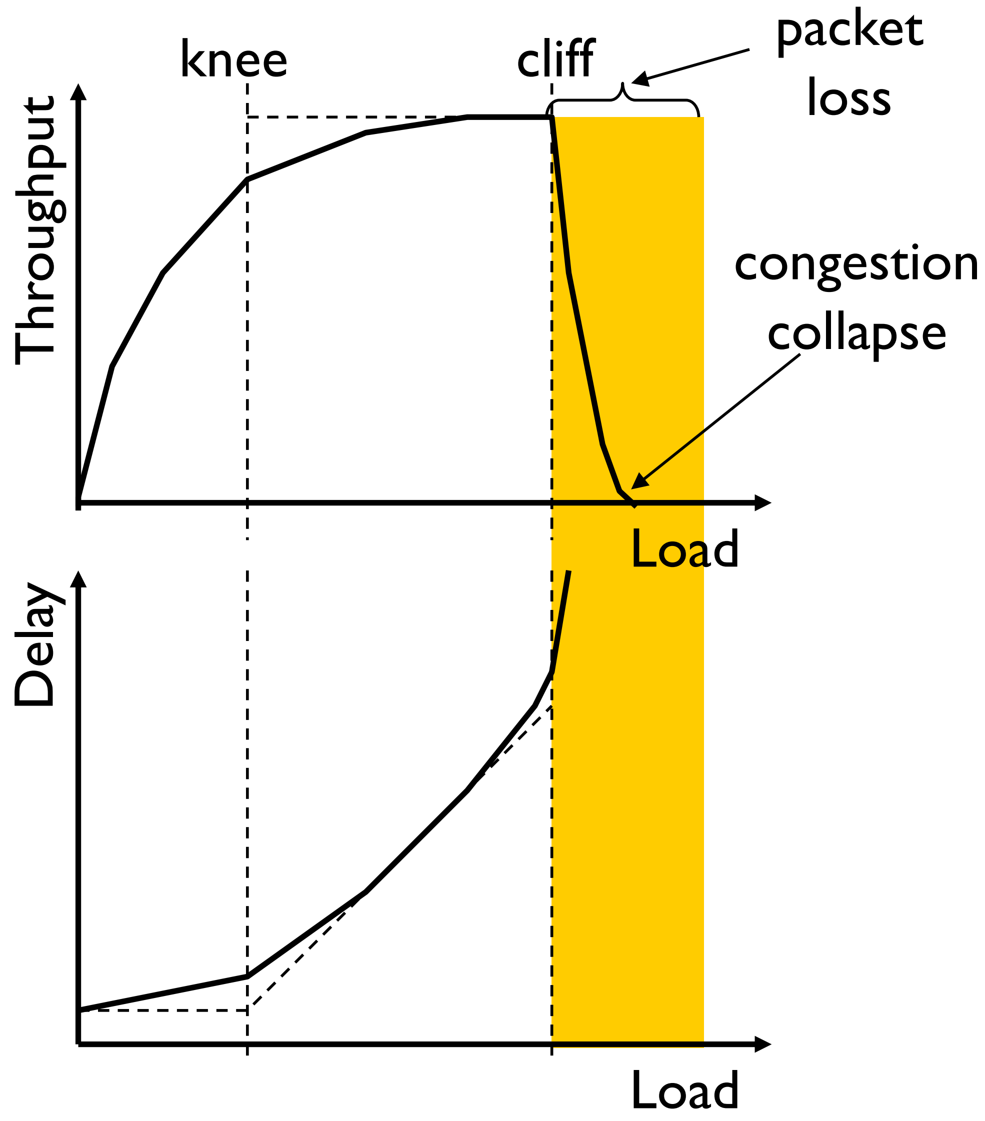
Unnecessary retransmission lead to negative loop

congestion:

- Increases delivery latency
  - Variable delays retransmission timeout
  - 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



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

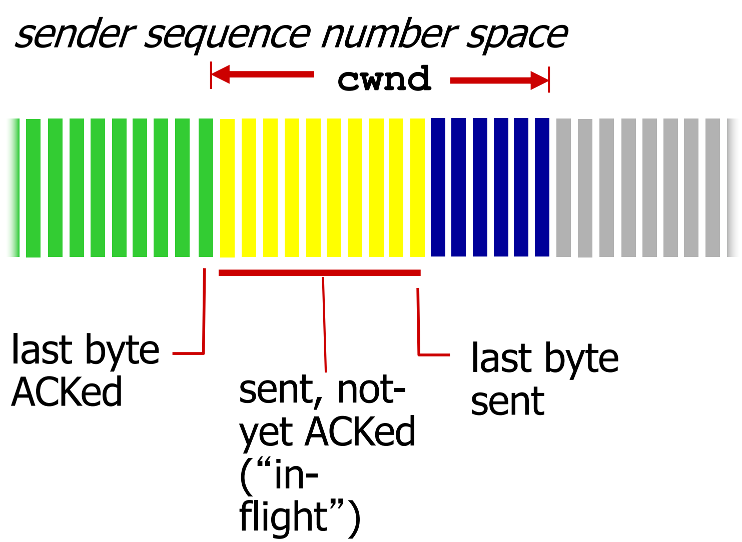
TCP's Approach in a Nutshell

- TCP connection has window
  - Controls number of packets in flight

In practice, we need different size of window to deal with different level of congestion

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

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



- Vary window size to control sending rate

All These Windows…

Congestion window is controlled by the sender

- 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 lecture that RWND >> CWND

Either CWND and RWND can congest the network

CWND

Congestion window

- 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?

BCD

Quiz: What is a "congestion event"

A: A segment loss (but how can the sender be sure of this?)

This is actually cannot known by the sender (because it can be bit error or overflow of the recv)

B: Increased delays

C: Receiving duplicate acknowledgement(s)

D: A retransmission timeout firing

E: Some subset of A, B, C & D (what is the subset?)

Quiz: How should we set CWND?

BC

A: We should keep raising it until a "congestion event" then back off slightly until we notice no more events

B: We should raise it until a "congestion event", then go back to 1 and start raising it again

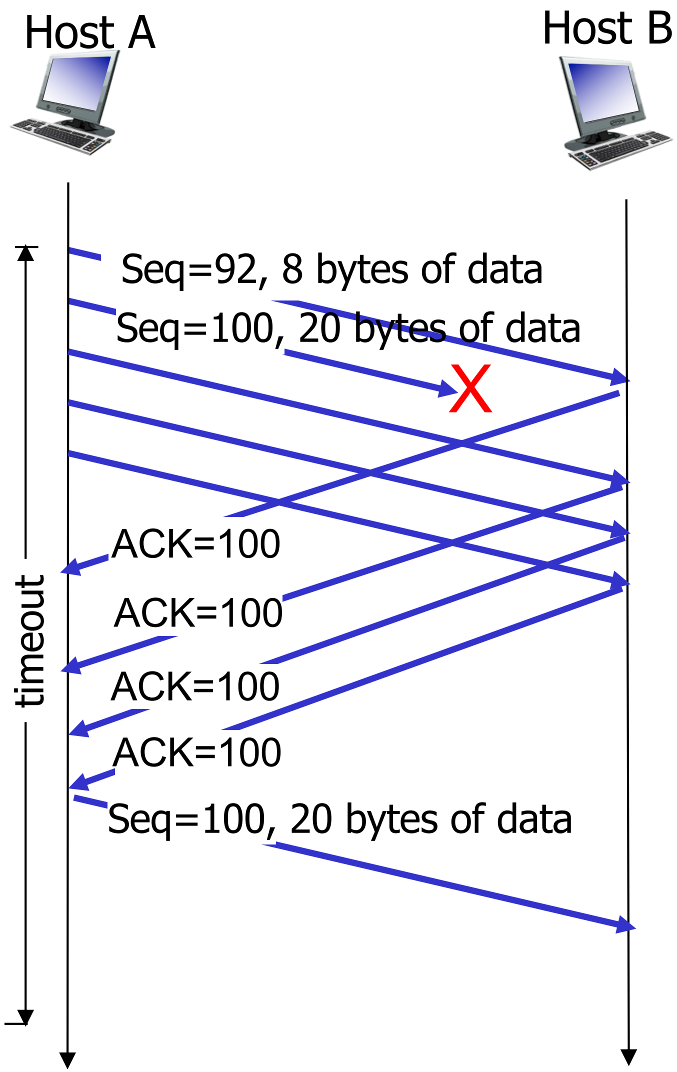
C: We should raise it until a "congestion event", then go back to median value and start raising it again

D: We should sent as fast as possible at all times

Not All Losses the Same

- 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



fast retransmit after sender receipt of triple duplicate ACK

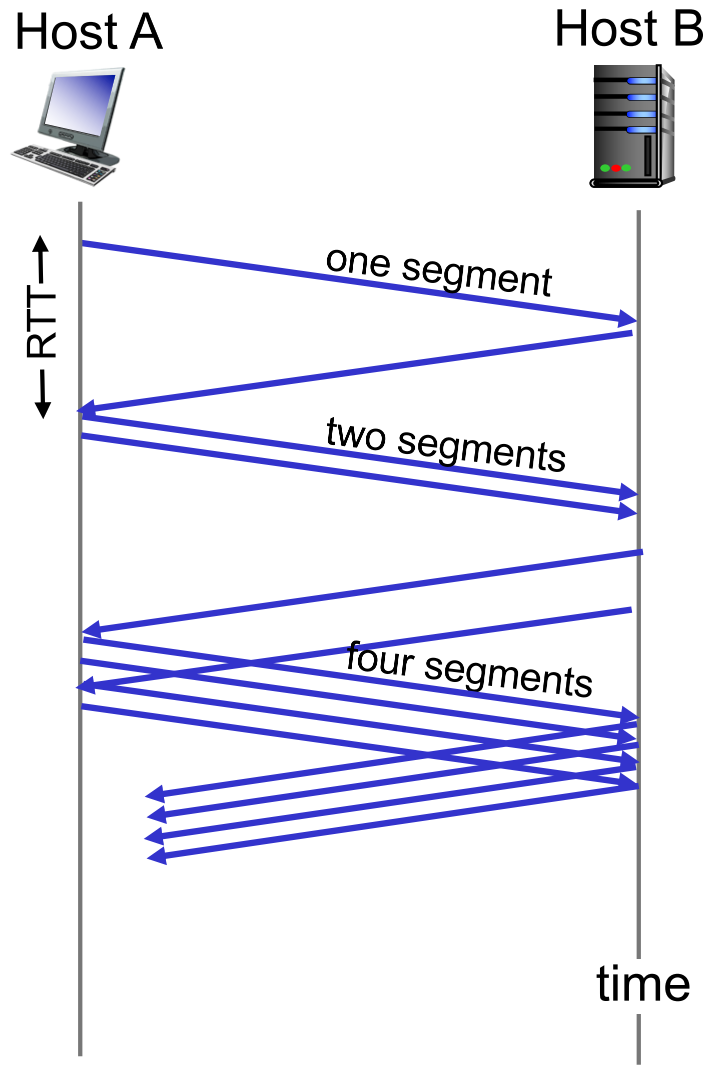
Rate Adjustment

- 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 (full 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



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 backoff (rate decrease)
  - Known as Congestion Avoidance (CA)

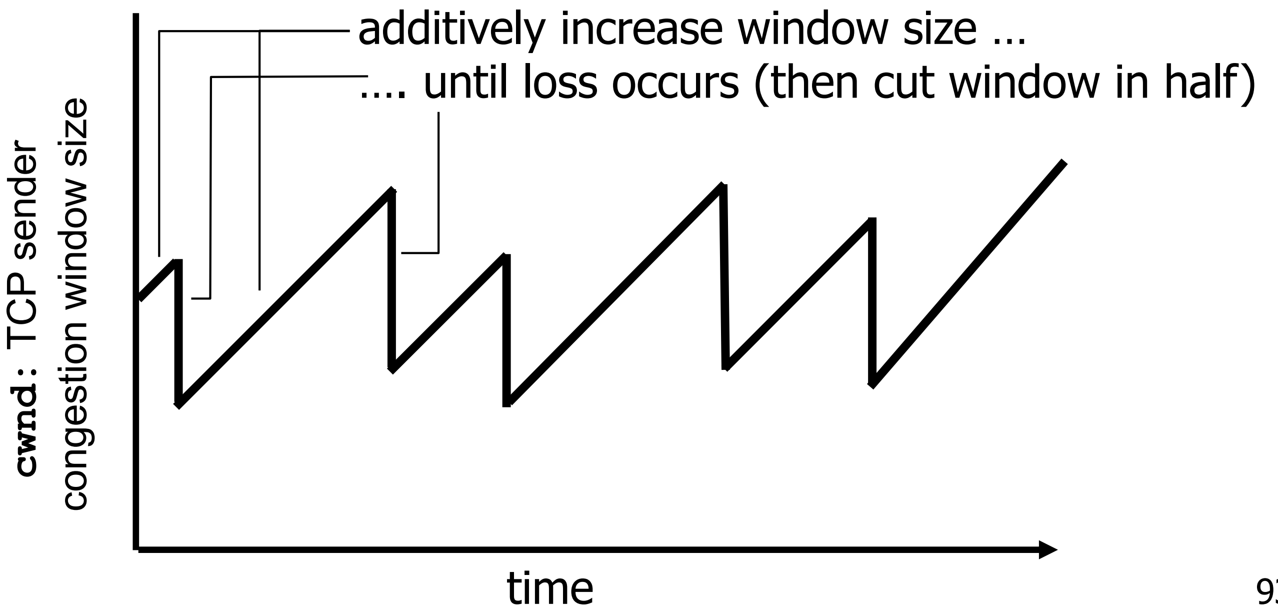
- TCP uses: "Additive Increase Multiplicative Decrease" (AIMD)
  - We'll see why shortly…

AIMD

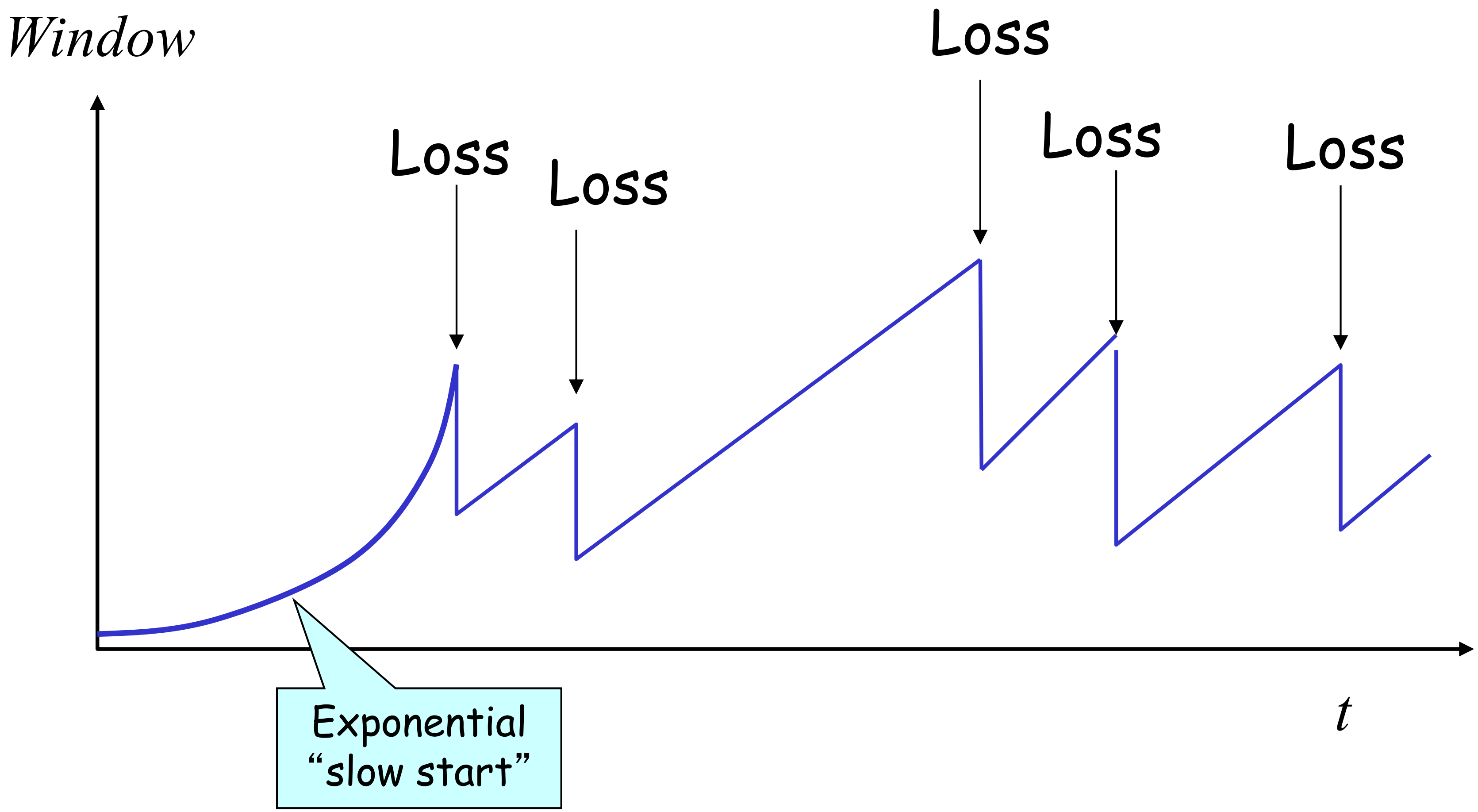
Additive increase/multiplicative decrease

- *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`**
    - Simple implementation: for each ACK, **`cwnd = cwnd + 1/cwnd`**
  - *multiplicative decrease*: cut **`cwnd`** in half after loss

AIMD saw tooth behavior: probing for bandwidth



Leads to the TCP "Sawtooth"



Slow-Start vs. AIMD

- When does a sender stop Slow-Start and start Additive Increase?

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

- Convert to AI when cwnd = ssthresh, sender switches from slow-start to AIMD-style increase
  - On timeout, ssthresh = CWND/2

Implementation

- State at sender
  - CWND (initialized to a small constant)
  - 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

- *2*MSS packets per ACK*
- *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

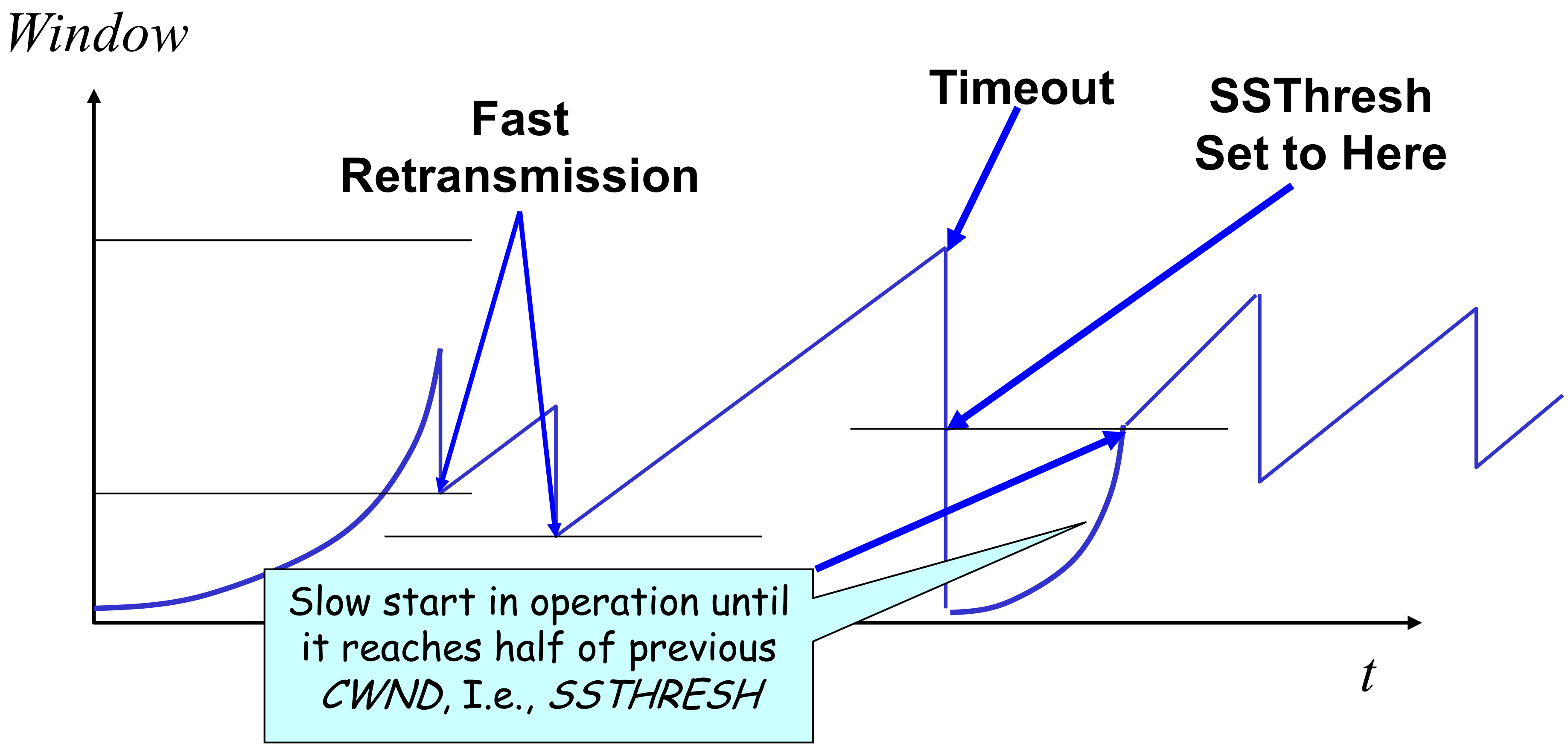
- dupACKcount ++

- 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

One Final Phase: Fast Recovery

- The problem: Fast retransmit slow in recovering from an isolated loss

Example (window in units of MSS, not bytes)

- Consider a TCP connection with:
  - CWND=10 packets (of size MSS, which is 100 bytes)
  - Last ACK was for byte # 101
    - i.e., receiver expecting next packet to have seq. no. 101

- 10 packets [101, 201, 301,…, 1001] are in flight
  - Packet 101 is dropped
  - What ACKs do they generate?
  - And how does the sender respond?

Timeline

- ACK 101 (due to 201) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 301) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 401) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 501) cwnd=5 + 1/5 (no xmit)
- ACK 101 (due to 601) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 701) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 801) cwnd=5 + 4/5 (no xmit)
- ACK 101 (due to 901) cwnd=5 + 5/5 (no xmit)
- ACK 101 (due to 1001) cwnd=6 + 1/6 (no xmit)
- ACK 1101 (due to 101) ← only now can we transmit new packets
- Plus no packets in flight so ACK "clocking" (to increase CWND) stalls for another RTT

Solution: Fast Recovery

Idea: Grant the sender temporary "credit" for each dupACK so as to keep packets in flight

- If dupACKcount = 3
  - ssthresh = cwnd/2
  - cwnd = ssthresh + 3
- While in fast recovery
  - cwnd = cwnd + 1 for each additional duplicate ACK
- Exit fast recovery after receiving new ACK
  - set cwnd = ssthresh

Example

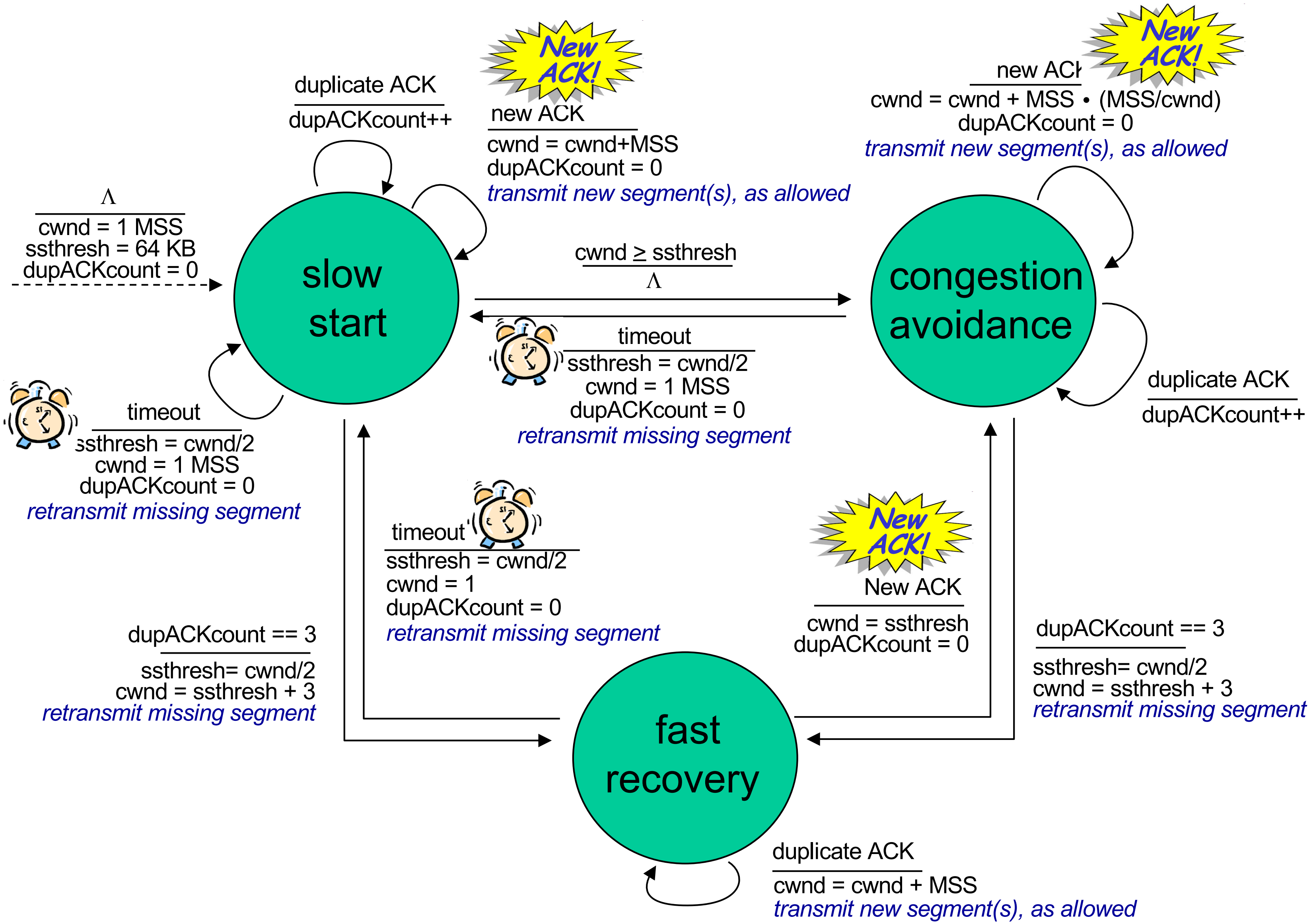
- Consider a TCP connection with:
  - CWND=10 packets (of size MSS = 100 bytes)
  - Last ACK was for byte # 101
    - i.e., receiver expecting next packet to have seq. no. 101

- 10 packets [101, 201, 301,…, 1001] are in flight
  - Packet 101 is dropped

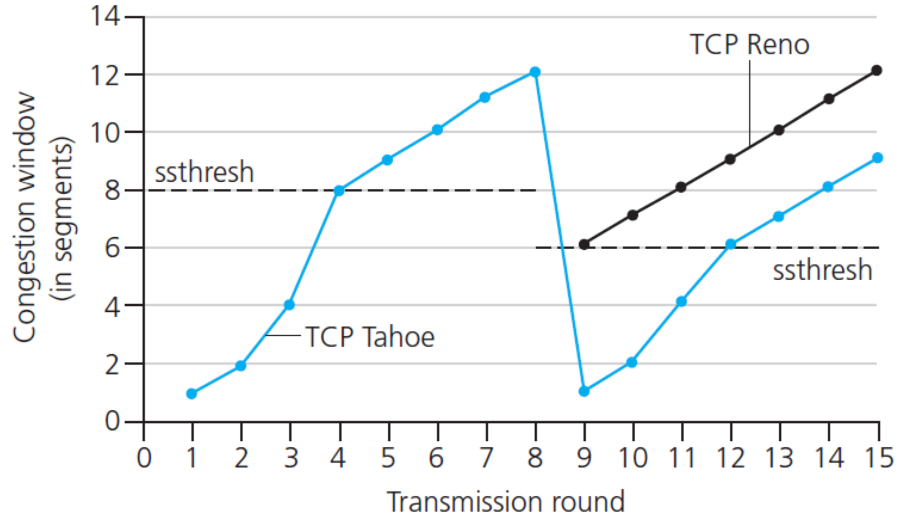
Timeline

- ACK 101 (due to 201) cwnd=10 dup#1
- ACK 101 (due to 301) cwnd=10 dup#2
- ACK 101 (due to 401) cwnd=10 dup#3
- REXMIT 101 ssthresh=5 cwnd= 8 (5+3)
- ACK 101 (due to 501) cwnd= 9 (no xmit)
- ACK 101 (due to 601) cwnd=10 (no xmit)
- ACK 101 (due to 701) cwnd=11 (xmit 1101)
- ACK 101 (due to 801) cwnd=12 (xmit 1201)
- ACK 101 (due to 901) cwnd=13 (xmit 1301)
- ACK 101 (due to 1001) cwnd=14 (xmit 1401)
- ACK 1101 (due to 101) cwnd = 5 (xmit 1501) ← exiting fast recovery
- Packets 1101-1401 already in flight
- ACK 1201 (due to 1101) cwnd = 5 + 1/5 ← back in congestion avoidance

Summary: TCP Congestion Control



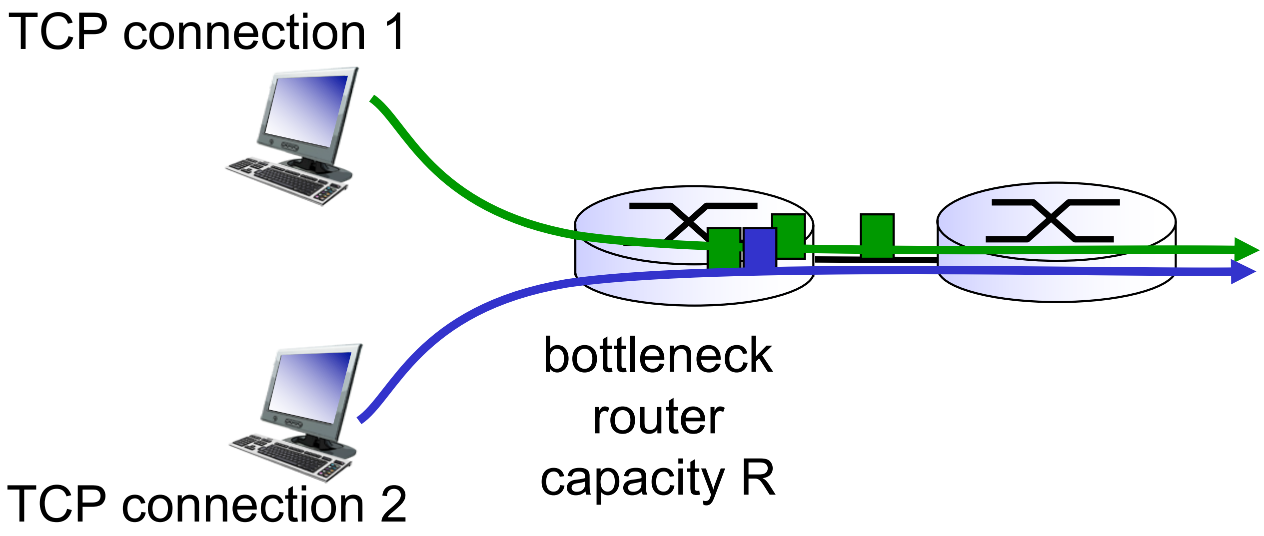
TCP Flavours



- 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
- TCP-SACK (NOT COVERED IN THE COURSE)
  - incorporates selective acknowledgements

TCP Fairness

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



Why AIMD?

- Some rate adjustment options: Every RTT, we can
  - Multiplicative increase or decrease: CWND$\rightarrow$ a*CWND
  - Additive increase or decrease: CWND$\rightarrow$ CWND + b

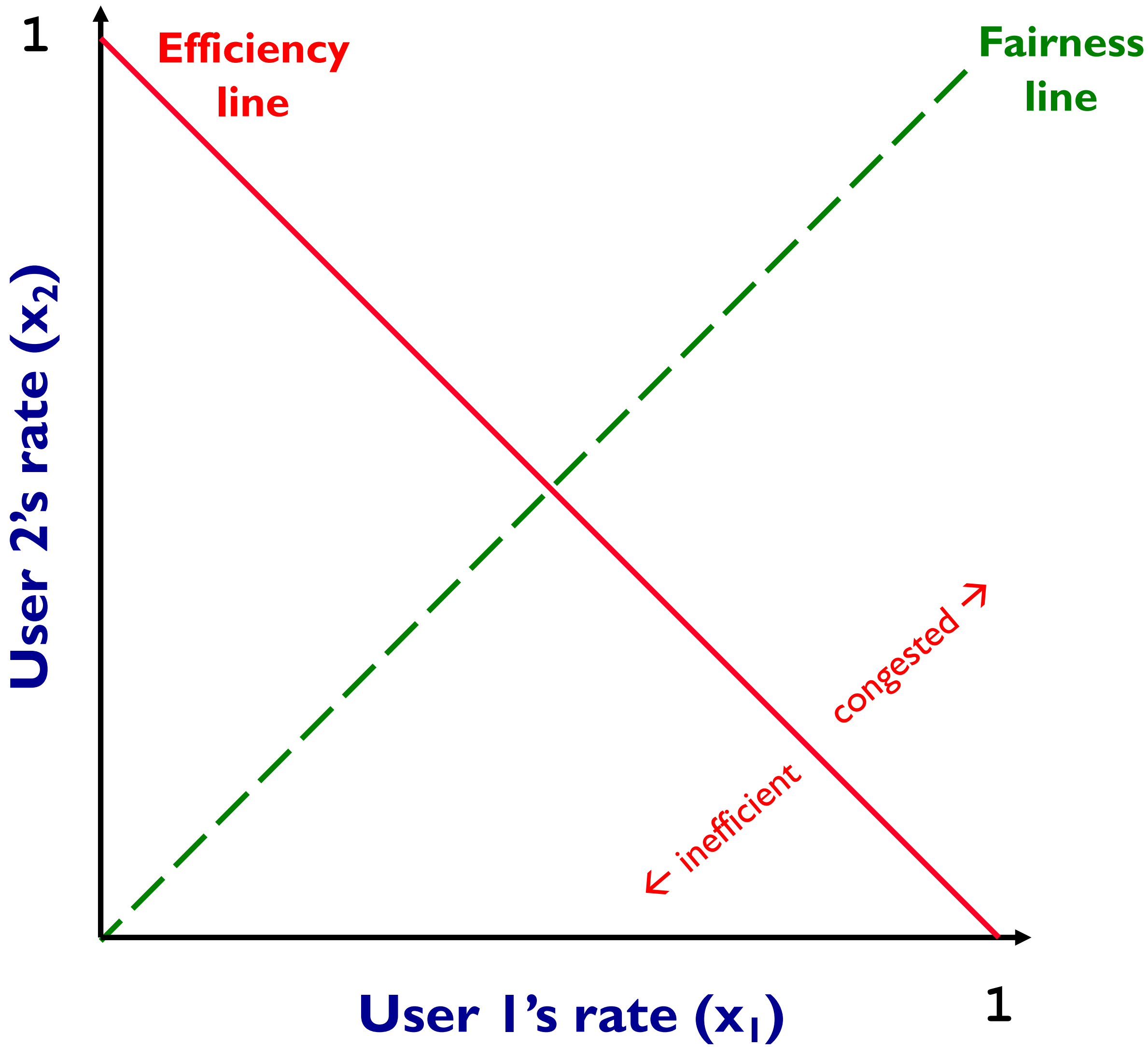
- Four alternatives:
  - AIAD: gentle increase, gentle decrease
  - AIMD: gentle increase, drastic decrease
  - MIAD: drastic increase, gentle decrease
  - MIMD: drastic increase and decrease

Simple Model of Congestion Control

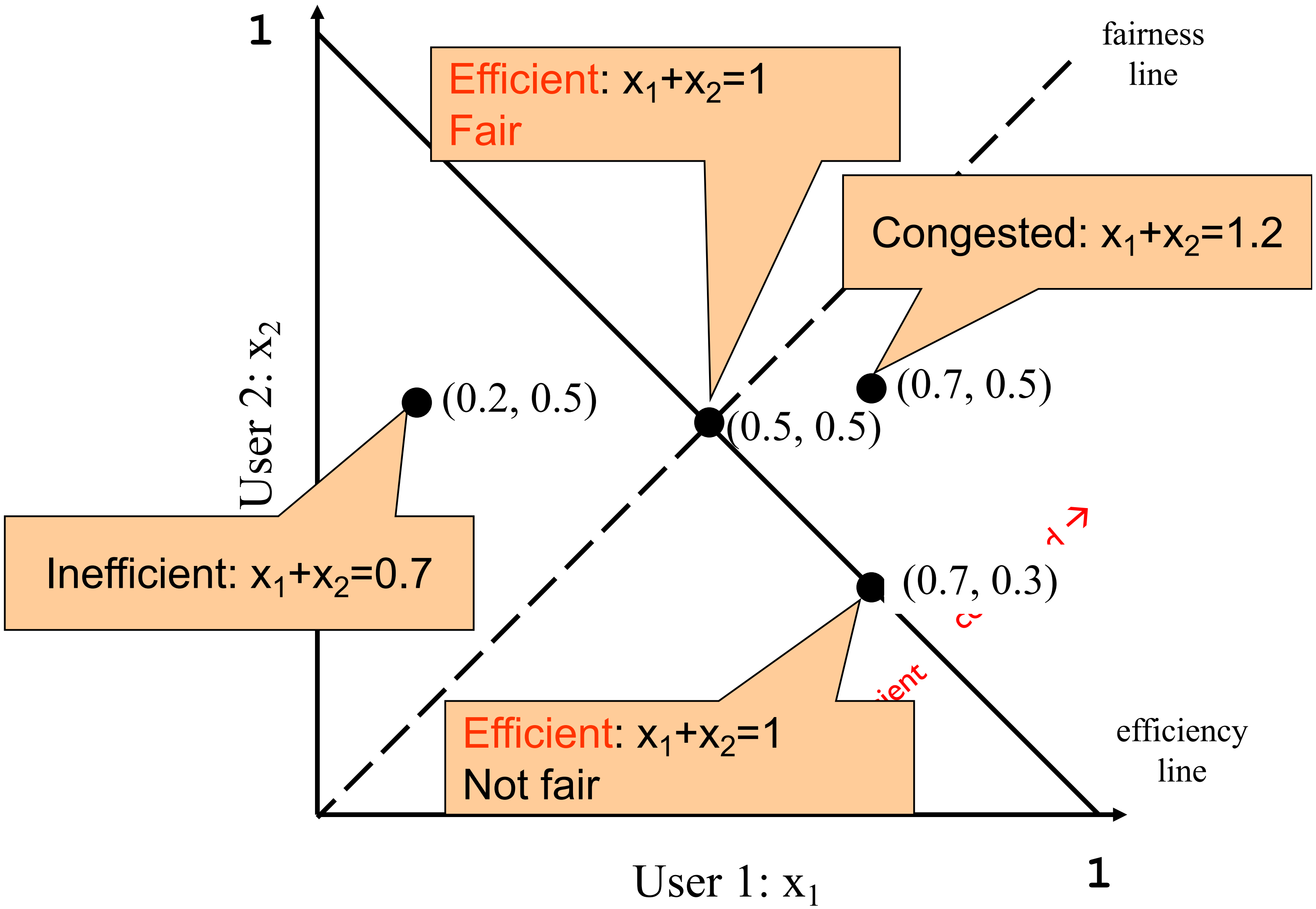
- Two users
  - rates x_1 and x_2

- Congestion when $x_1+x_2 > 1$
- Unused capacity when $x_1+x_2 < 1$

- Fair when $x_1 = x_2$



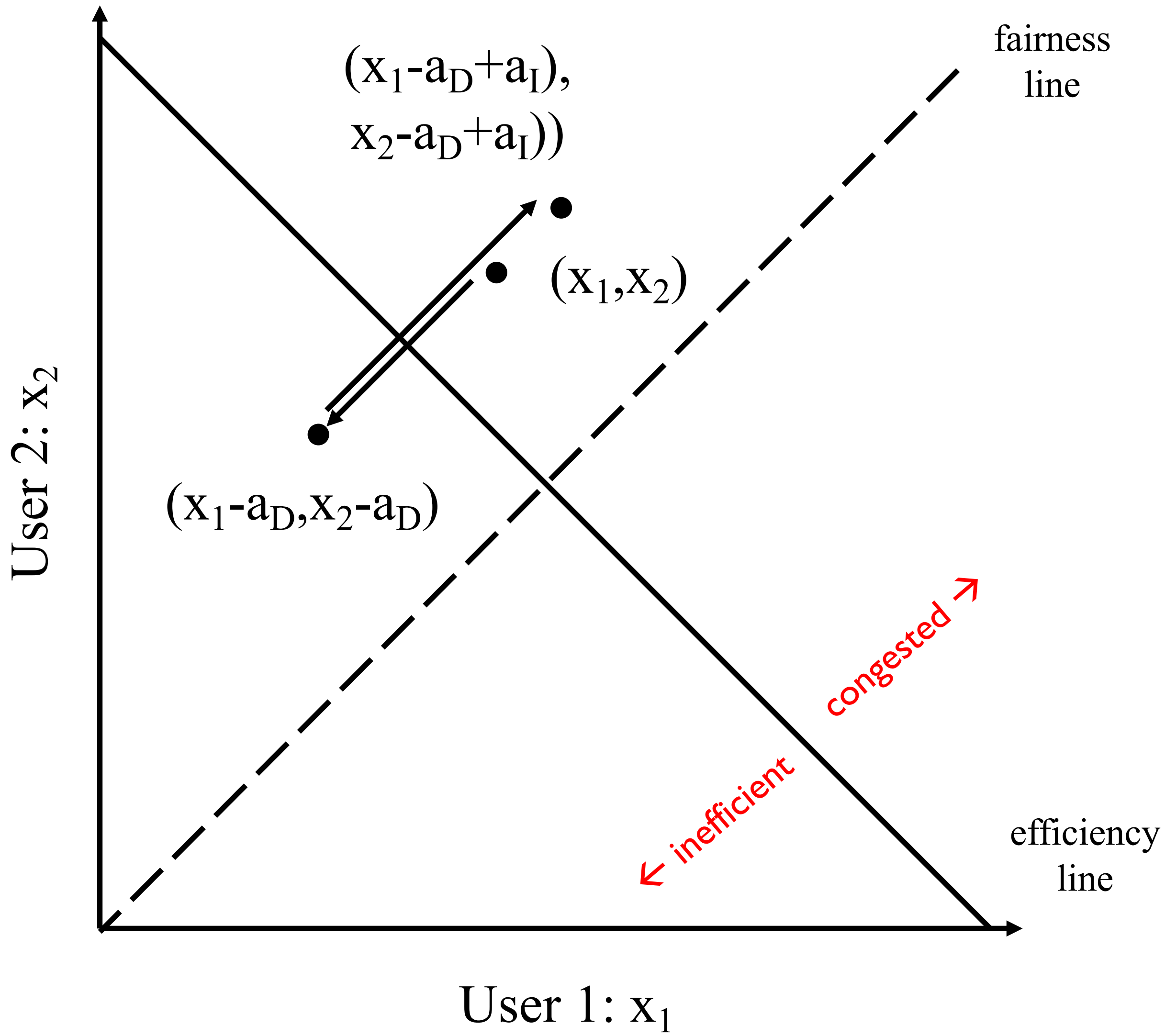
Example



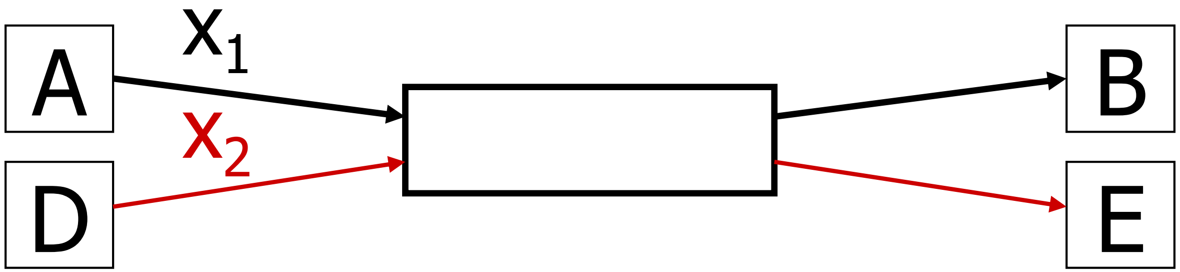
1
fairness line
Efficient: $x_1+x_2=1$
Fair
Congested: $x_1+x_2=1.2$
(0.2, 0.5)
(0.5, 0.5)
(0.7, 0.5)
User 2: x_2
Inefficient: $x_1+x_2=0.7$
(0.7, 0.3)
Efficient: $x_1+x_2=1$
Not fair
efficiency line
User 1: x_1
1

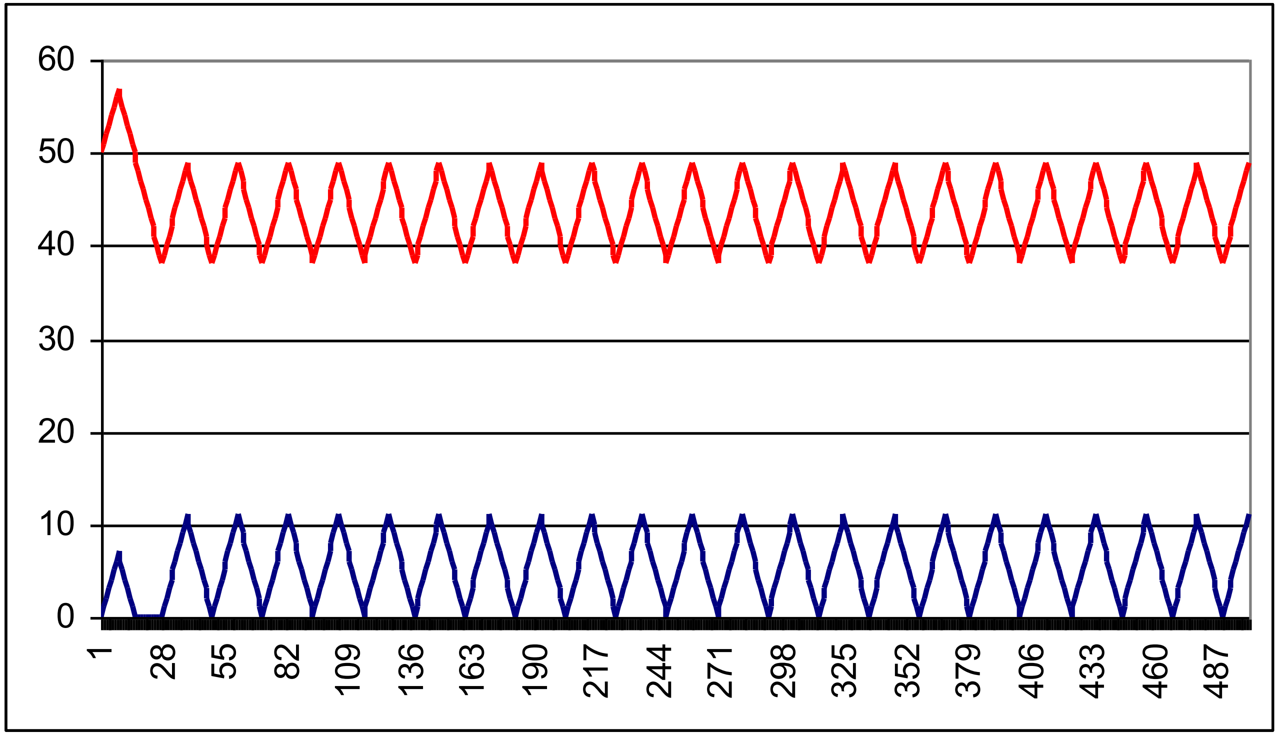
AIAD

- Increase: $x + a_I$
- Decrease: $x - a_D$
- Does not converge to fairness



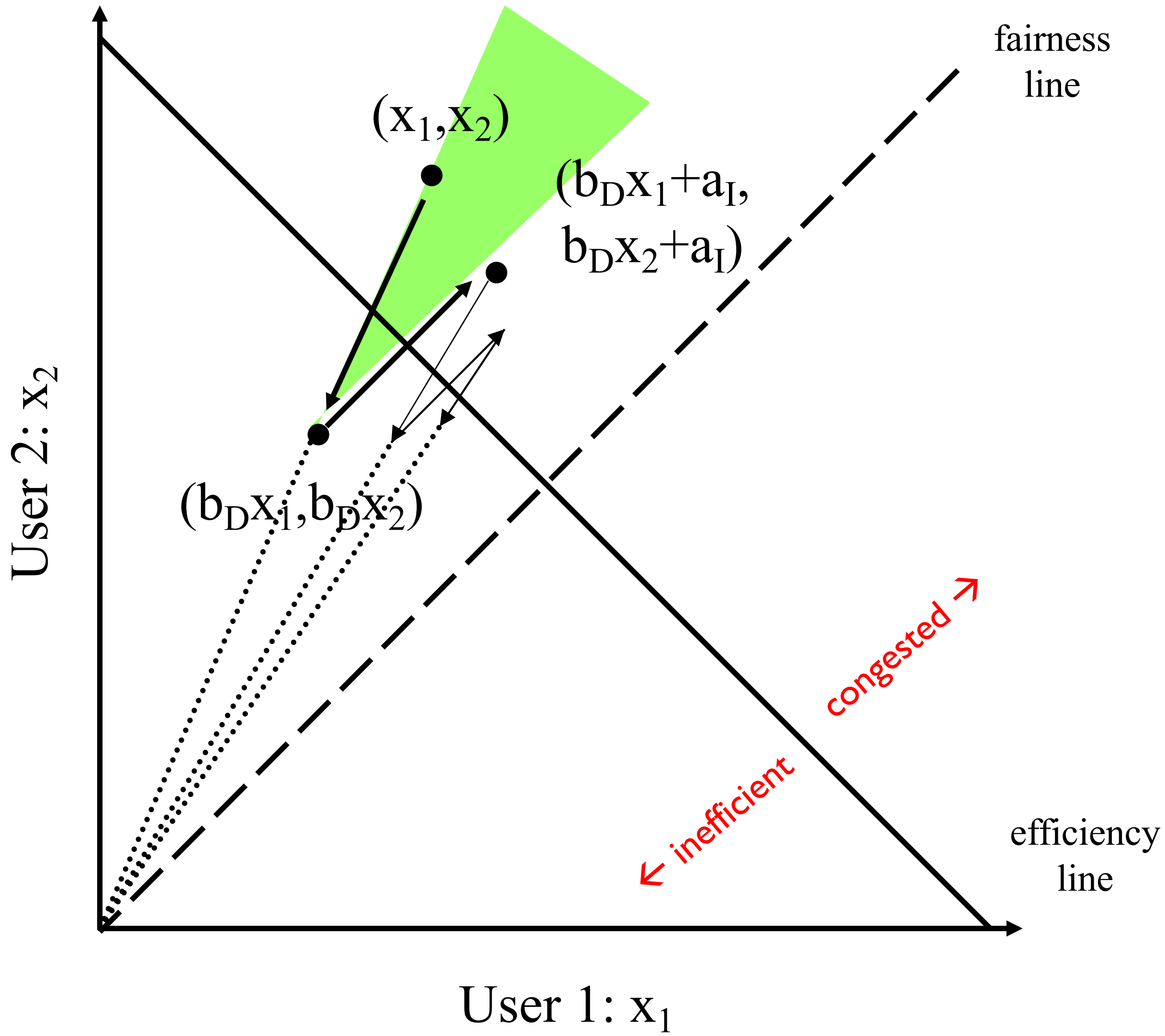
AIAD Sharing Dynamics



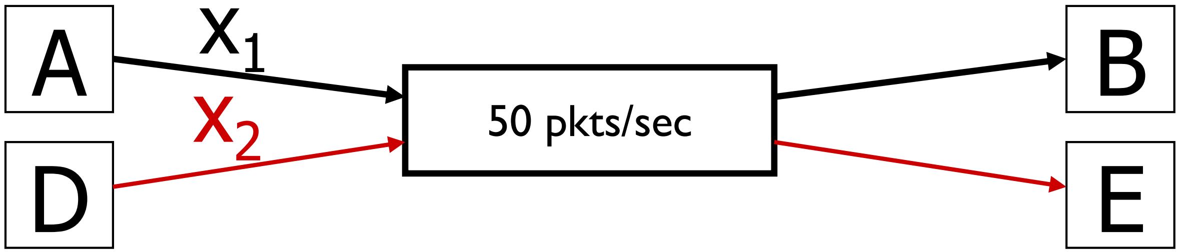


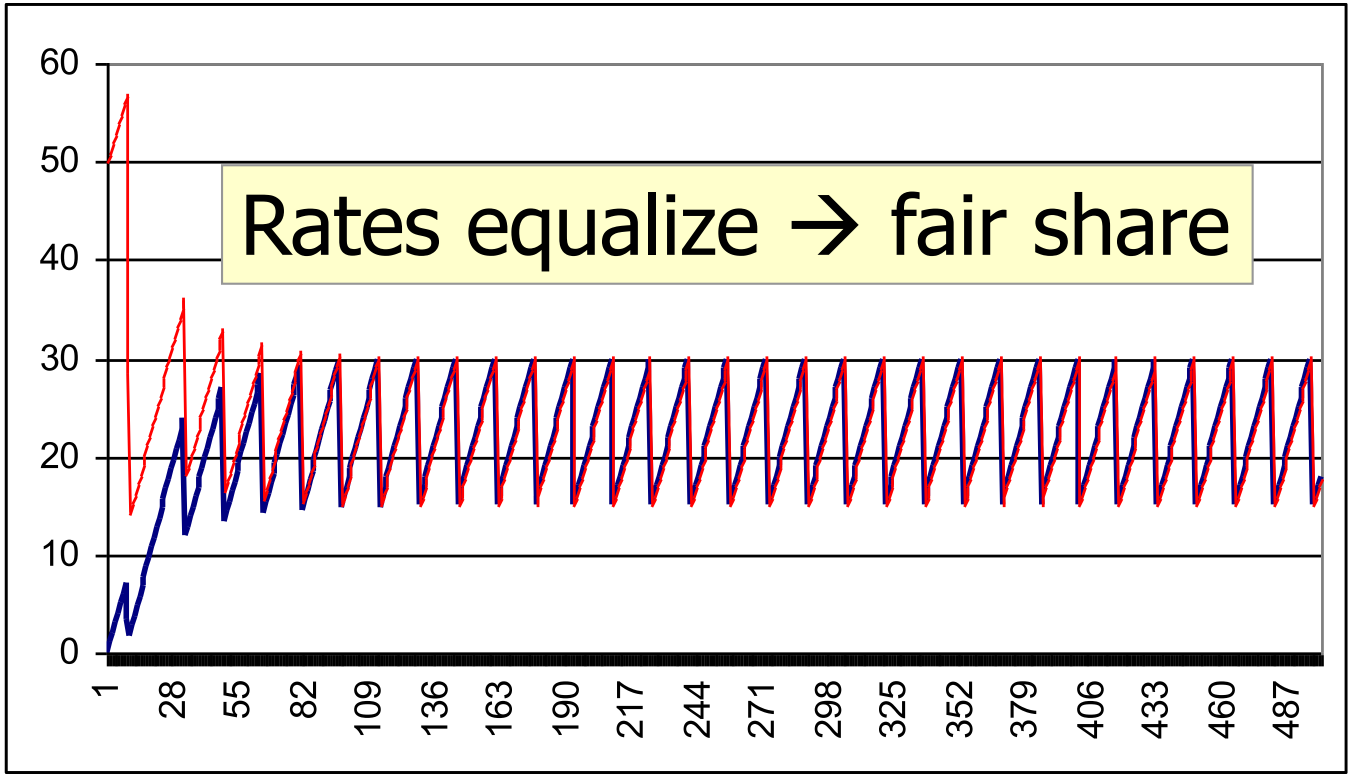
AIMD

- Increase: $x+a_I$
- Decrease: $x*b_D$
- Converges to fairness

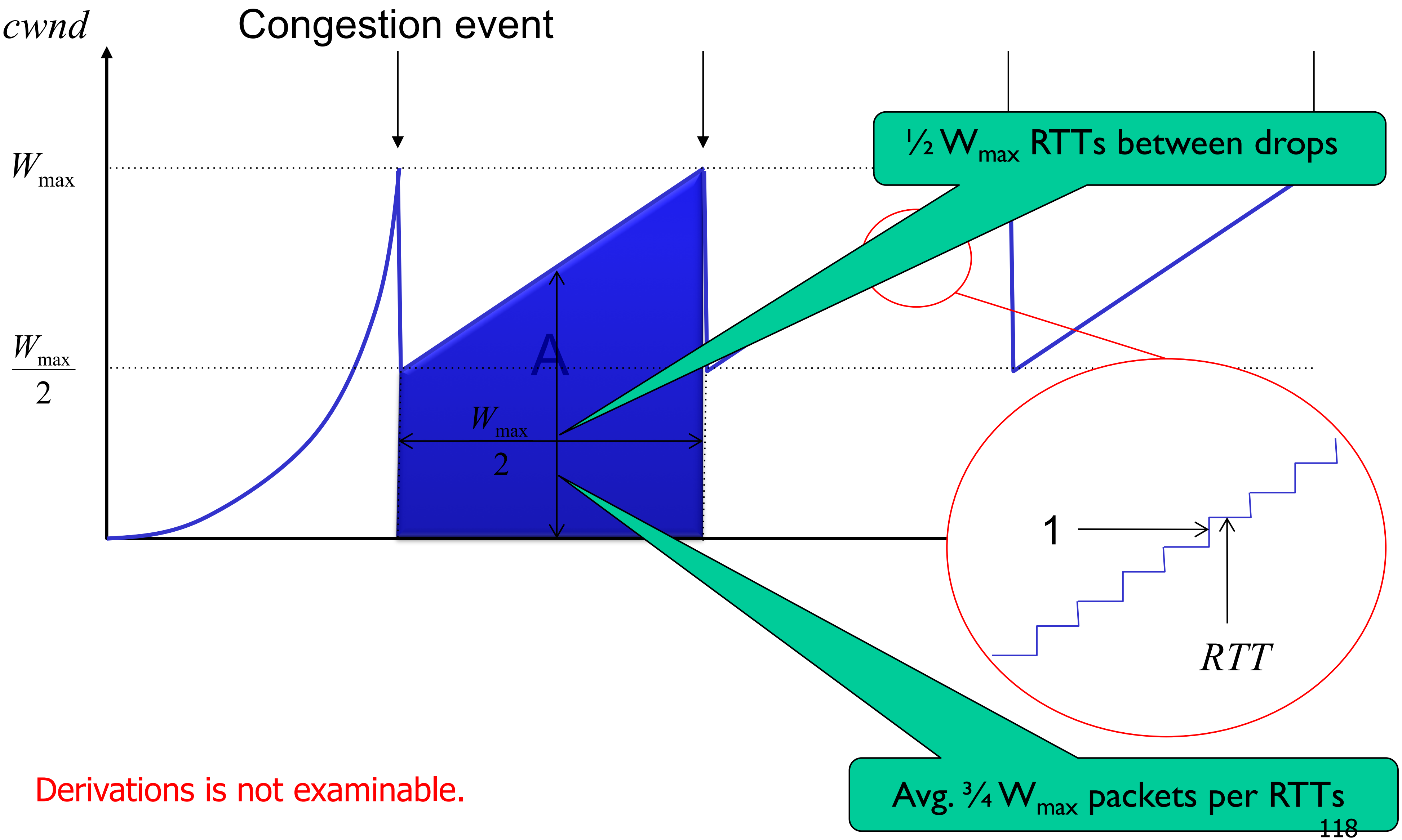


AIMD Sharing Dynamics



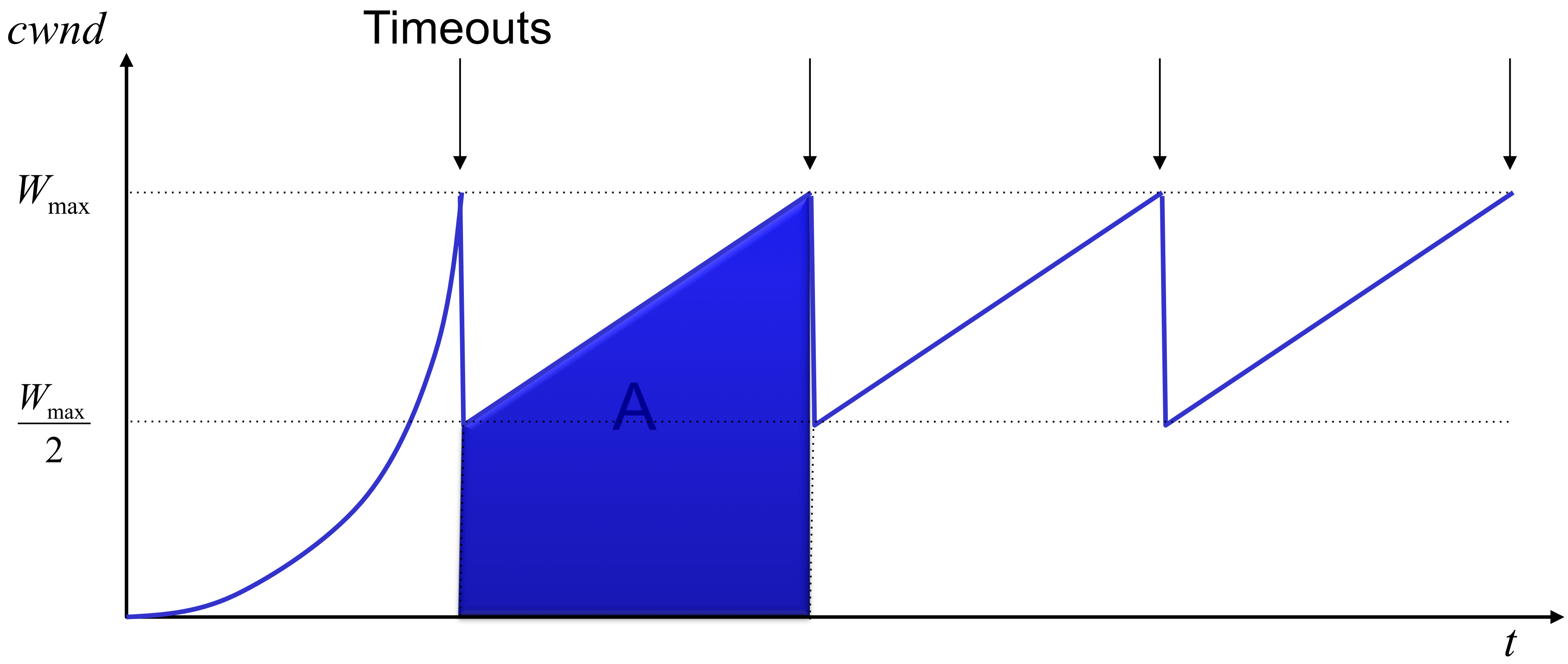


A Simple Model for TCP Throughput



Derivations is not examinable.

A Simple Model for TCP Throughput



$$\text{Packet drop rate, } p = 1/A, \text{ where } A = \frac{3}{8}W_{\max}^2$$

$$\text{Throughput, } B = \frac{A}{\left(\frac{W_{\max}}{2}\right)RTT} = \sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$$

Quiz: TCP Congestion Control?

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

A. 0

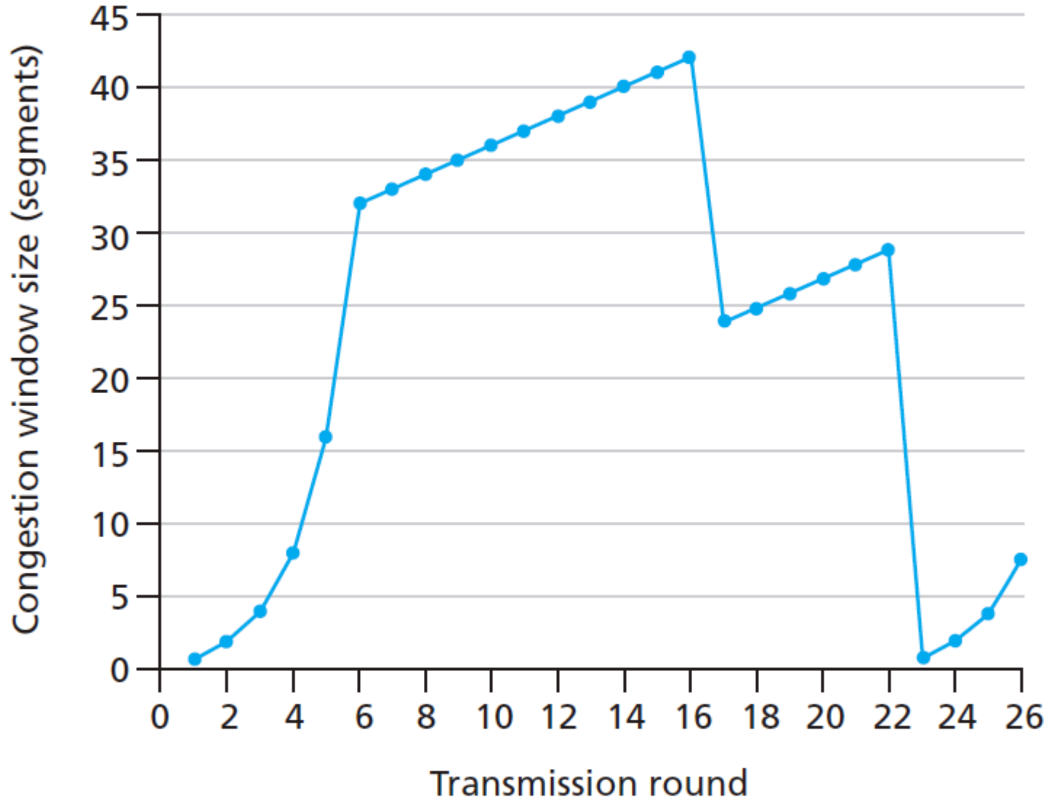
B. 1

C. 2

D. 3

E. 4

C



Quiz: TCP Congestion Control?

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

A. 0

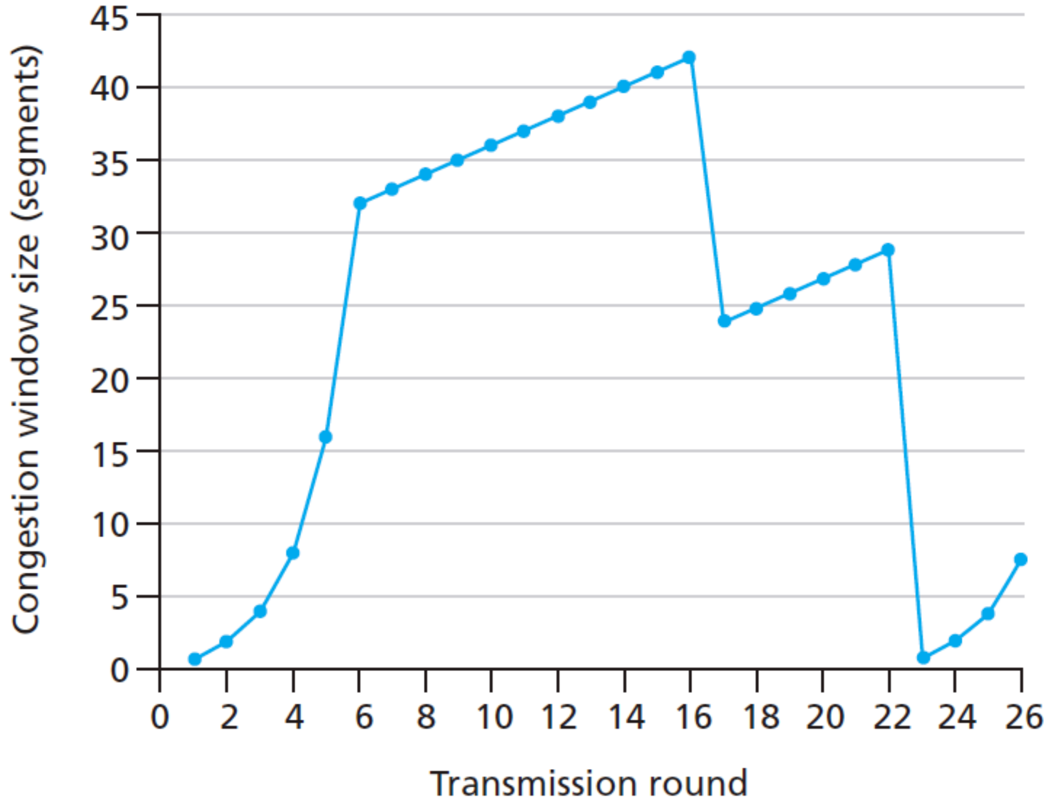
B. 1

C. 2

D. 3

E. 4

C



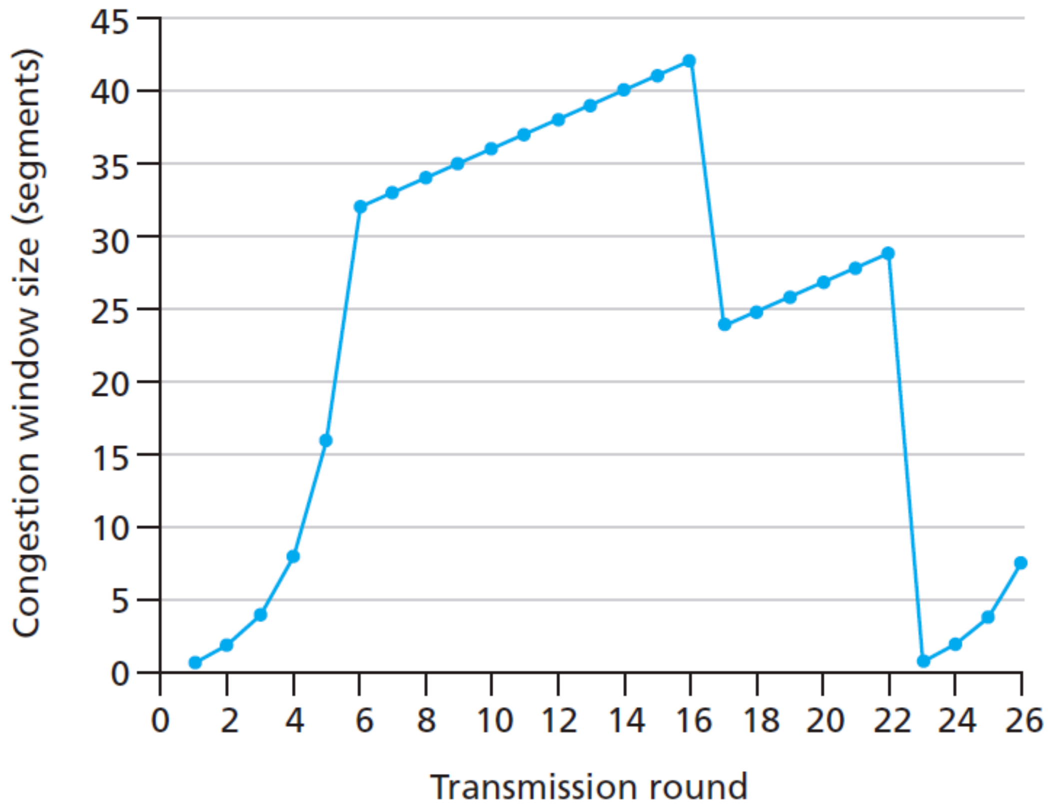
Quiz: TCP Congestion Control?

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

A

A. Triple Dup Ack

B. Timeout

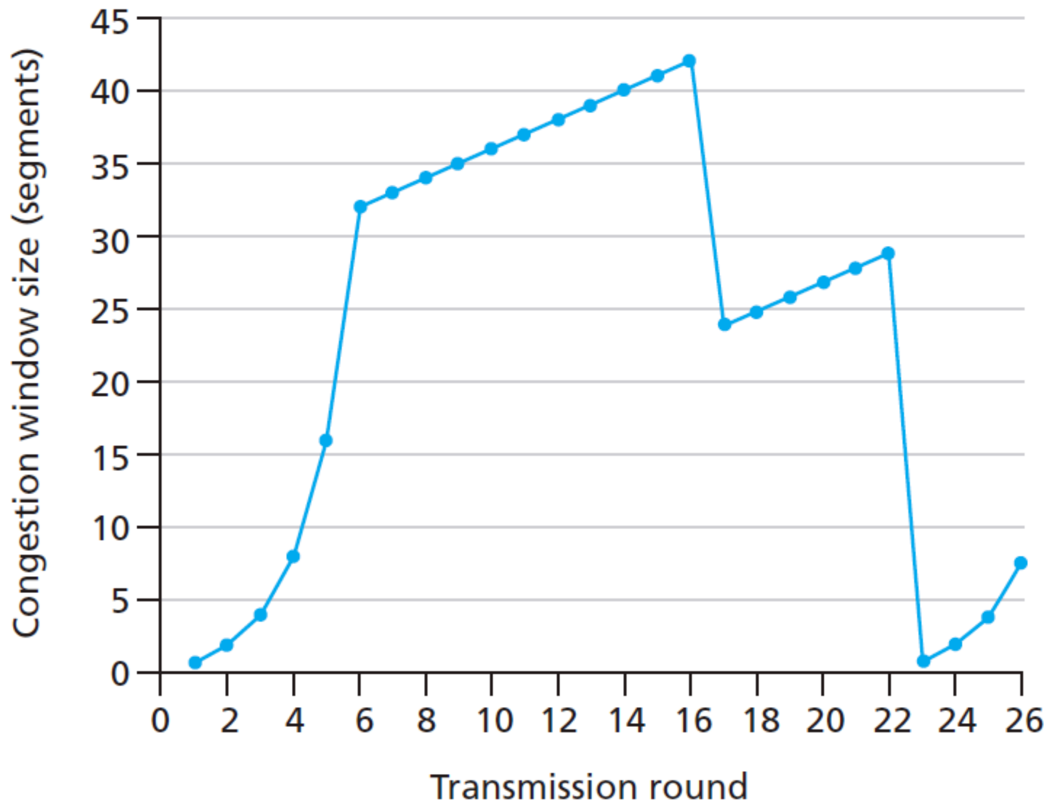


Quiz: TCP Congestion Control?

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

A. 0
B. 28
C. 32
D. 42
E. 64

C



Quiz: TCP Congestion Control?

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

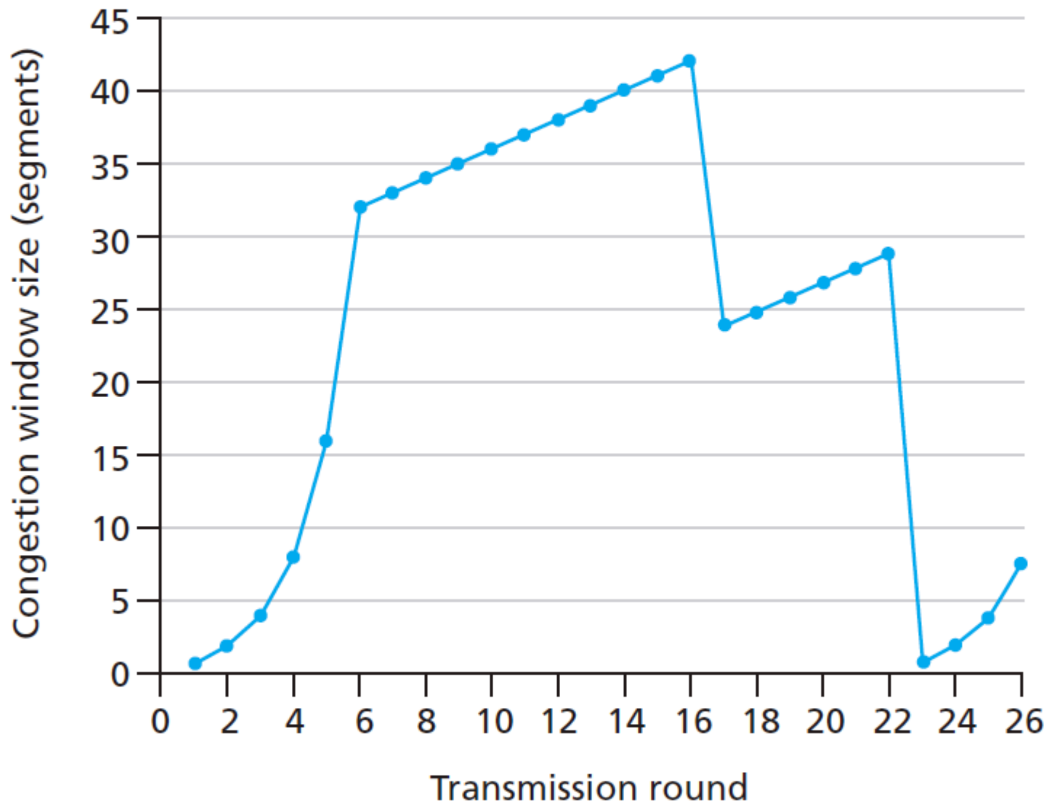
A. 1

B. 32

C. 42

D. 21 D

E. 20



Transport Layer: 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"