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

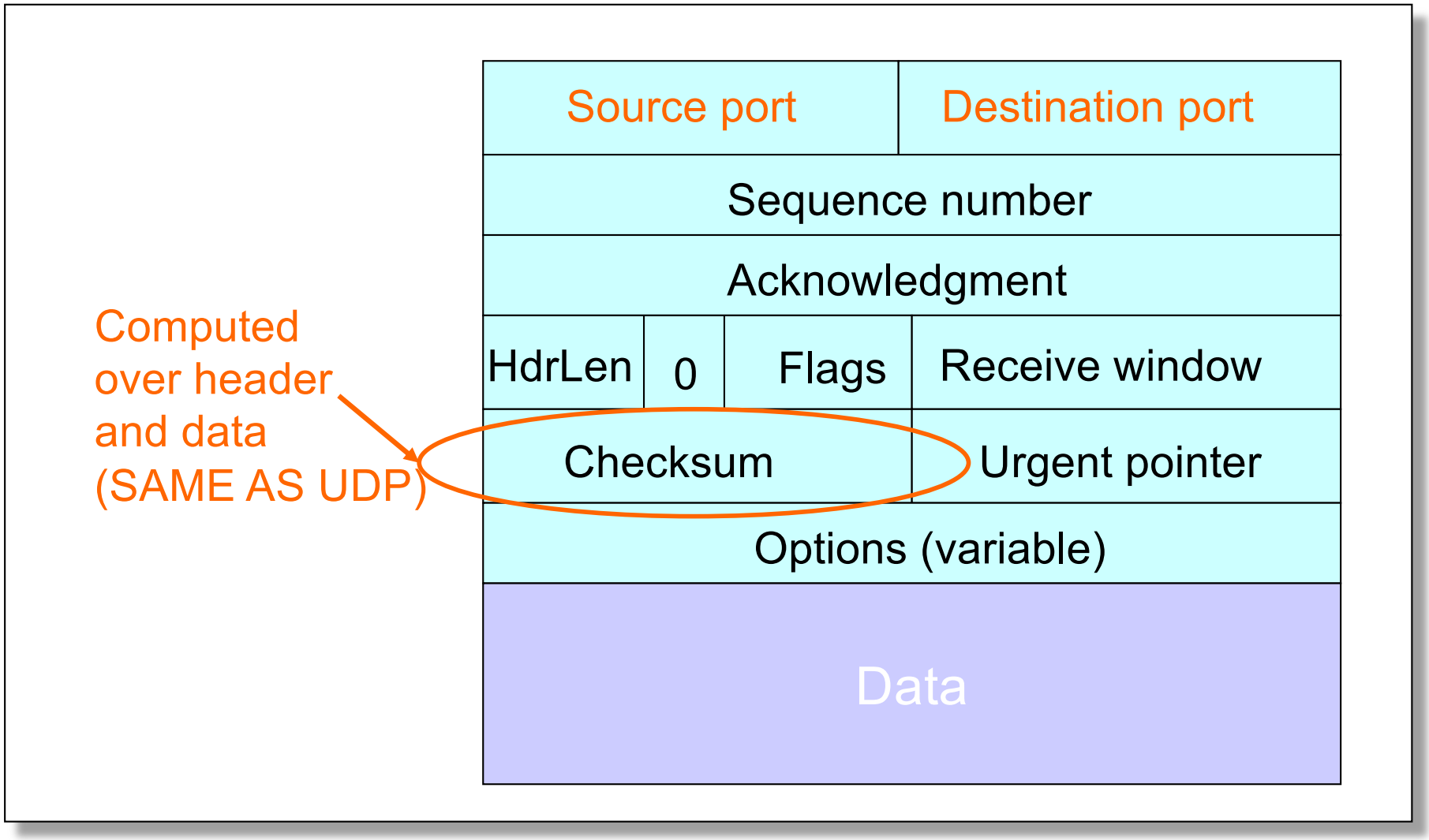
- Checksums (for error detection)
- Timers (for loss detection)
- Acknowledgments
  - cumulative
  - selective
- Sequence numbers (duplicates, windows)
- Sliding Windows (for efficiency)
  - Go-Back-N (GBN)
  - Selective Repeat (SR)

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)

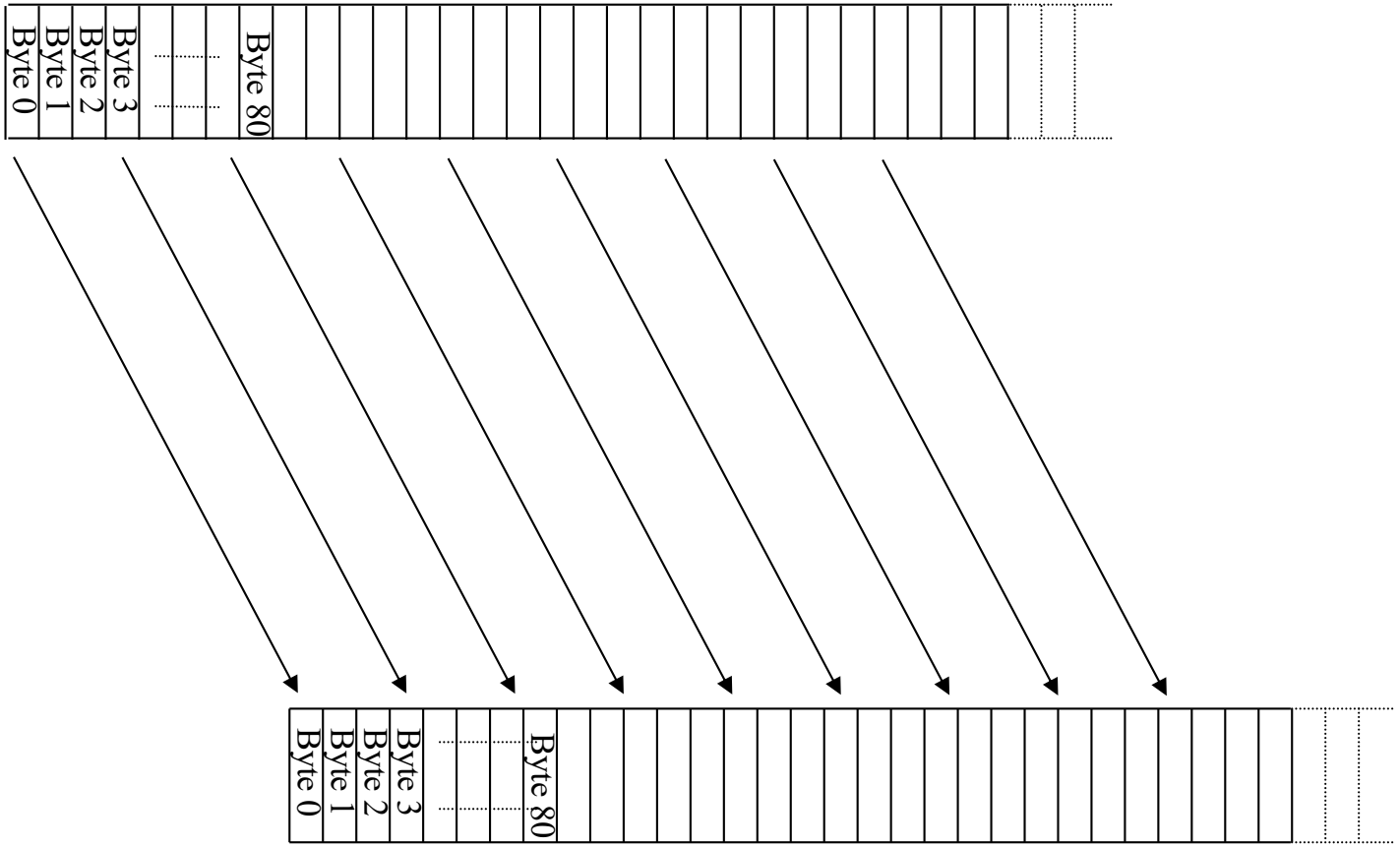
What does TCP do?

Many of our previous ideas, but some key differences

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

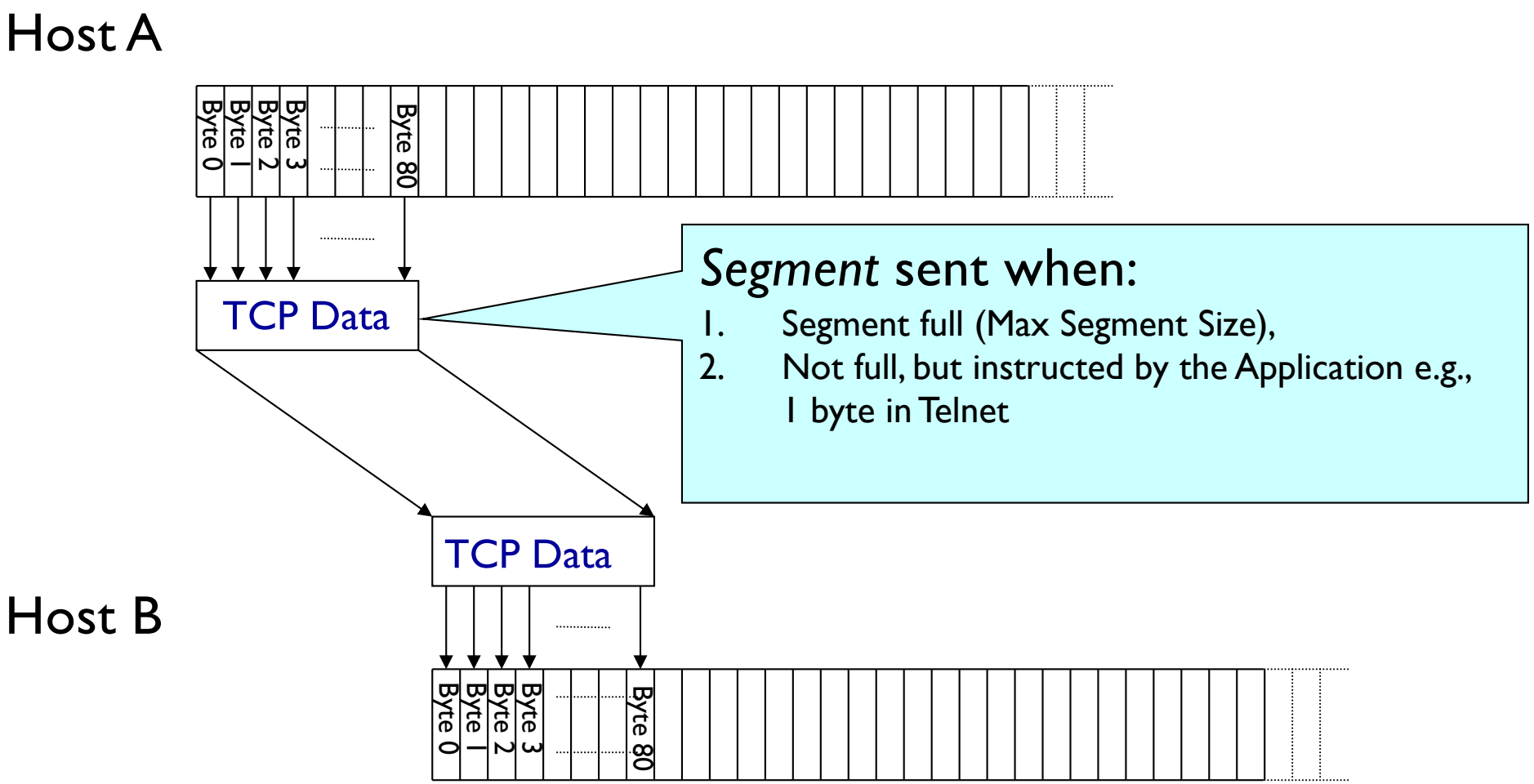
TCP "Stream of Bytes" Service ..

Application @ Host A

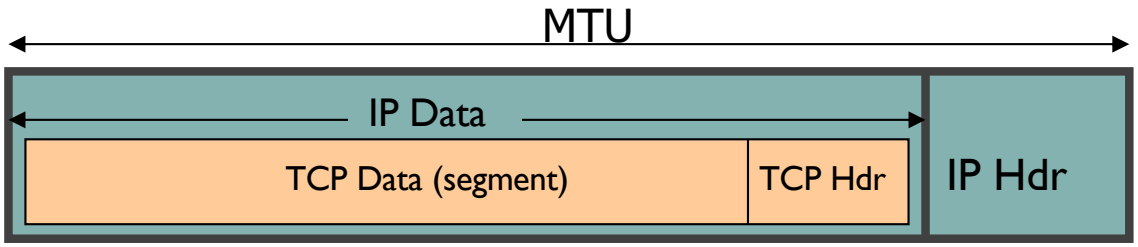


Application @ Host B

.. Provided Using TCP "Segments"

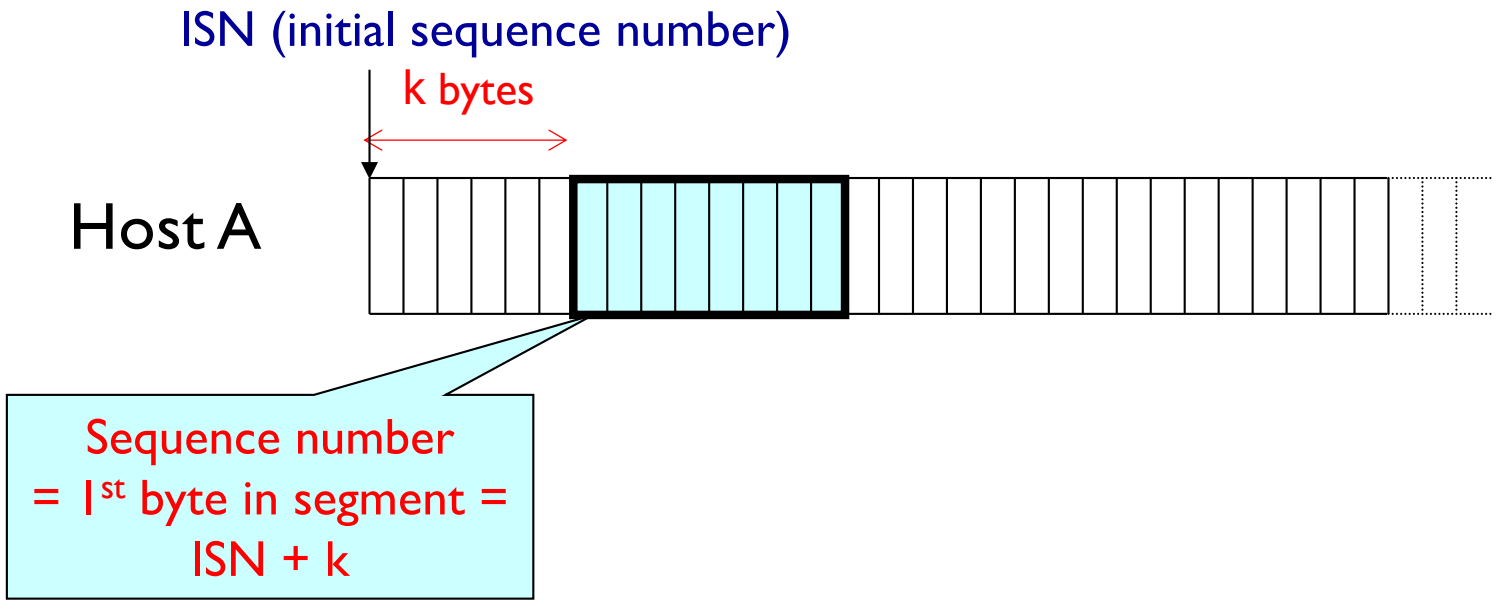


TCP Maximum Segment Size



- 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)

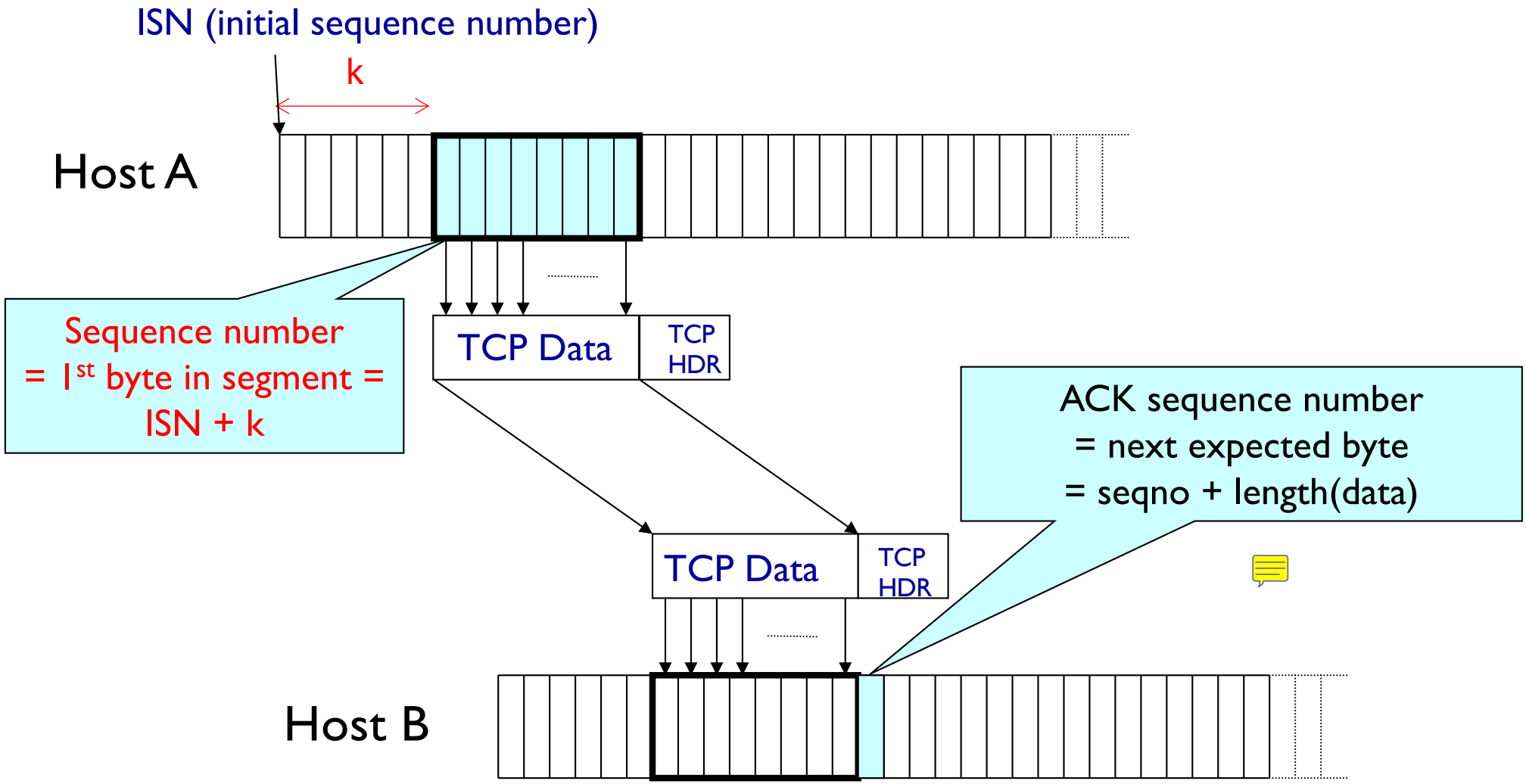
Sequence Numbers



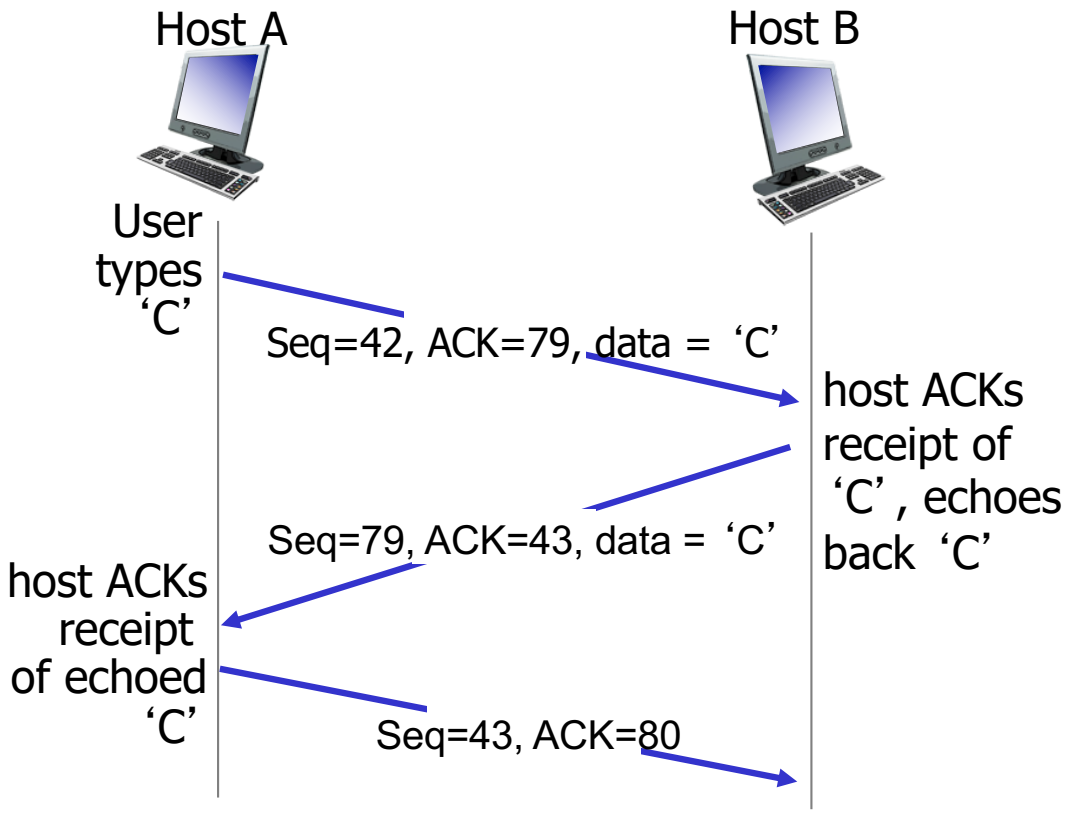
Sequence numbers:

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

Sequence & Ack Numbers



TCP seq. numbers, ACKs



simple telnet scenario

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 i.e. seq# is index of first byte
  - Packet contains B bytes [X, X+1, X+2, ….X+B-1]

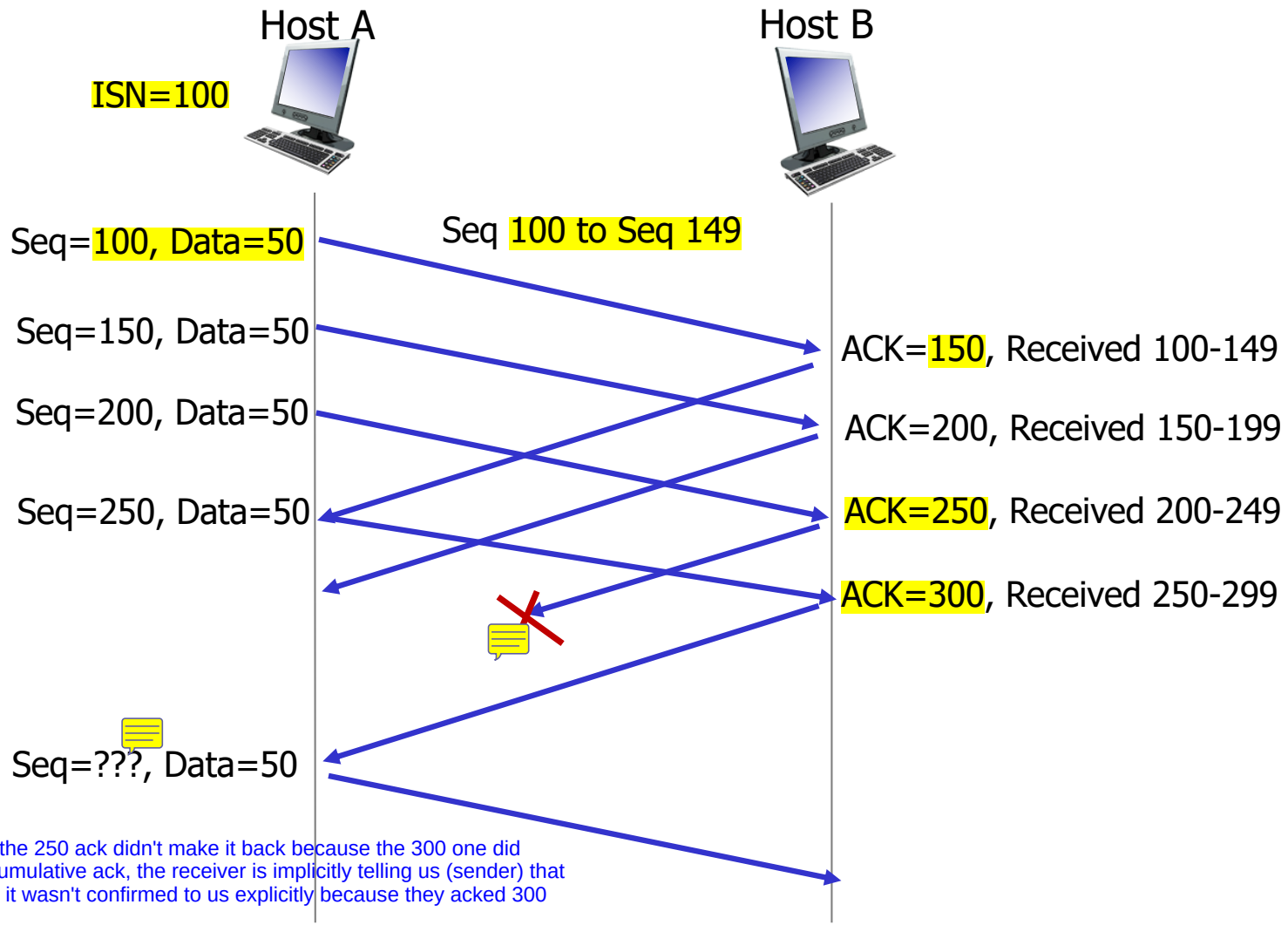
- Upon receipt of packet, receiver sends an ACK
  - If all data prior to X already received:
    - ACK acknowledges X+B (because that is next expected byte)
  - If highest in-order byte received is Y s.t. (Y+1) < X
    - ACK acknowledges Y+1
    - Even if this has been ACKed before

could have acked it before but the ack got dropped

note that above the packet ends at byte X+B-1, so X+B is the next byte after the packet i.e. the next expected byte

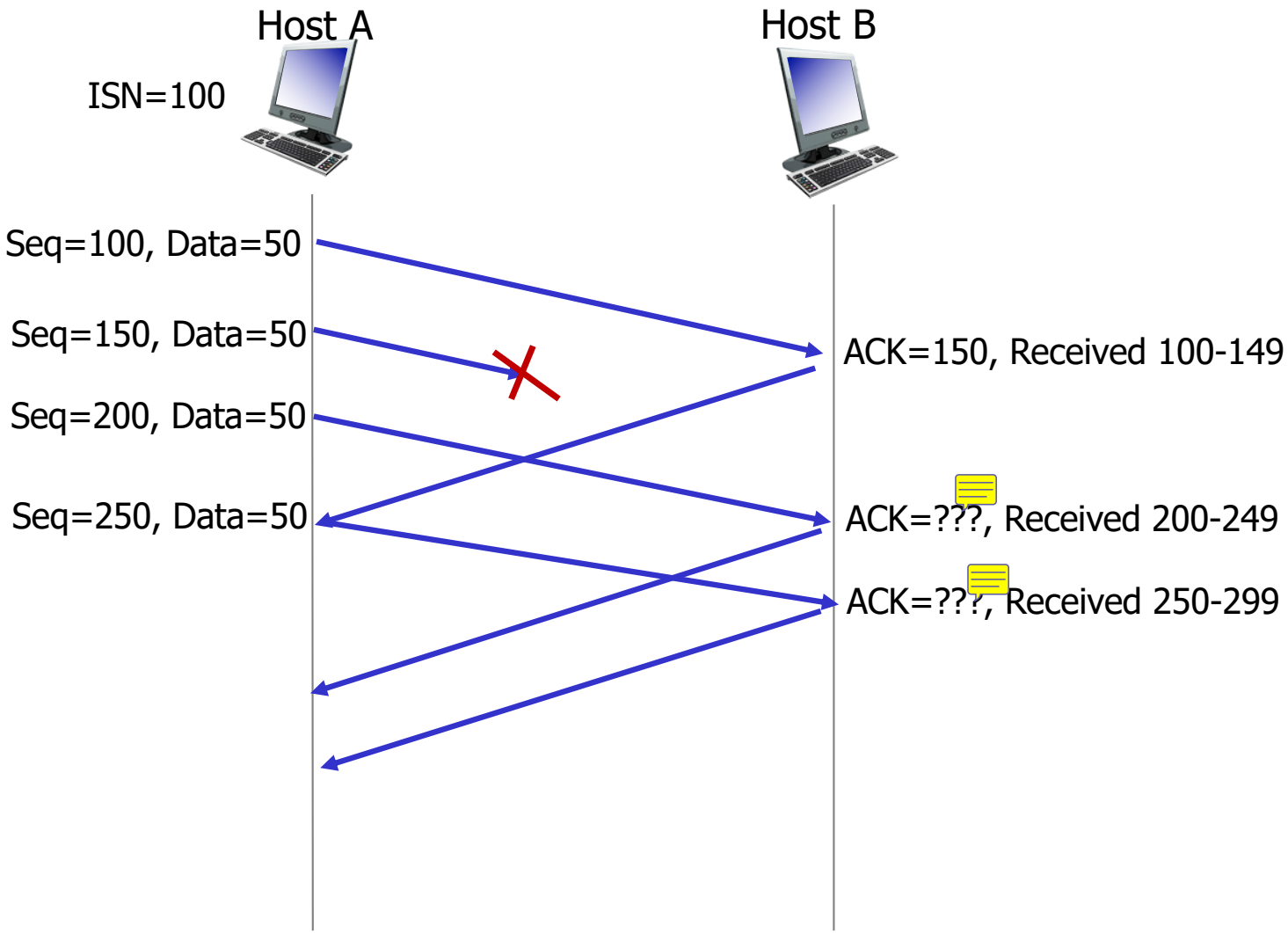
if the last byte of the highest packet is not the byte directly before the byte at the start of the new incoming packet, then the new incoming packet is not in order (it must be further in the future). Because TCP is cumulative ACK, it will ack Y+1 because this is the packet we need

TCP seq. numbers, ACKs



seq#300, it doesn't matter that the 250 ack didn't make it back because the 300 one did which trumps it (because it is cumulative ack, the receiver is implicitly telling us (sender) that they do have 250, even though it wasn't confirmed to us explicitly because they acked 300

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

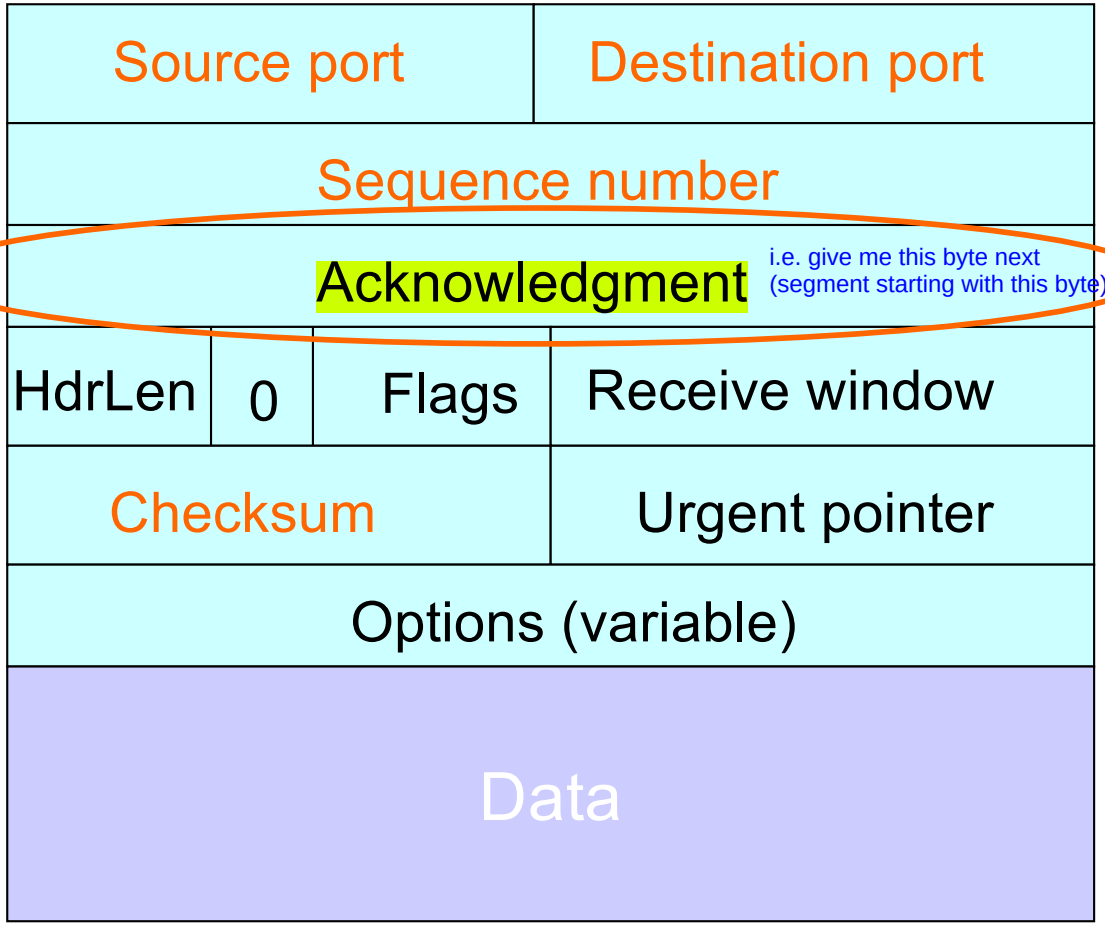
Packet Loss

- 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 still haven't received X+B so ask for it again

TCP Header

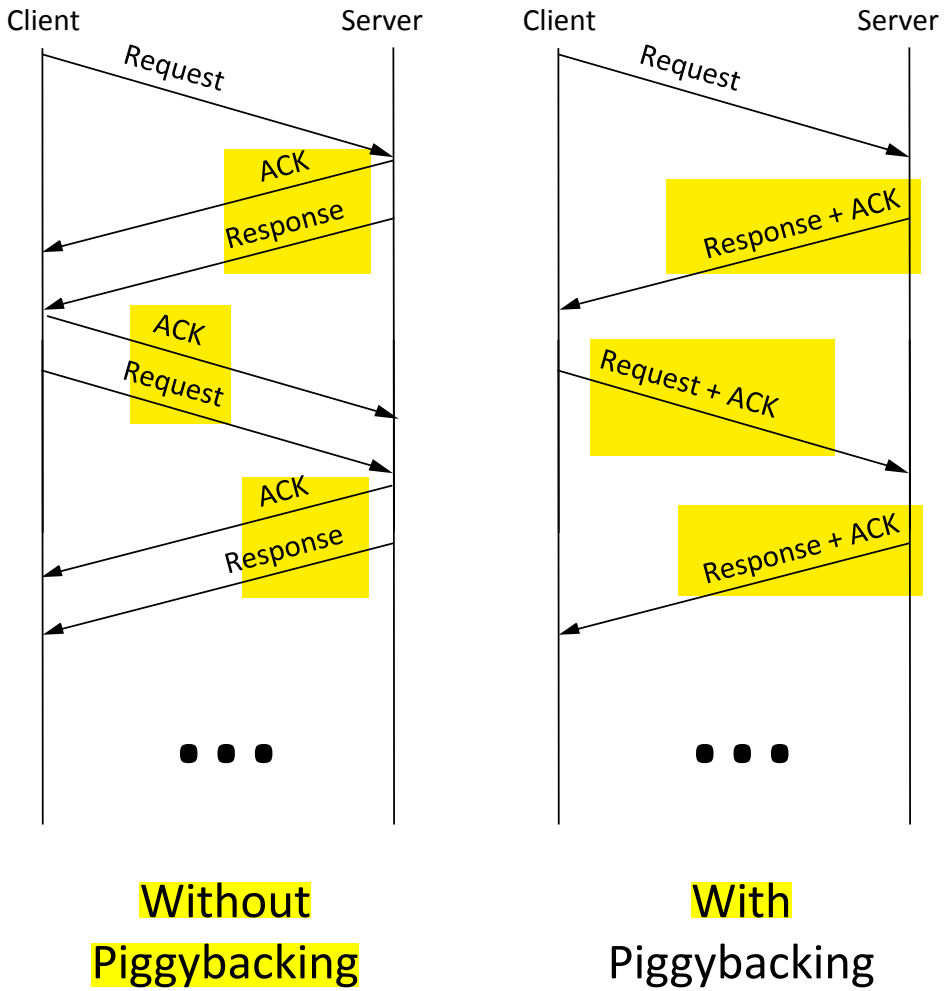
Acknowledgment gives seqno just beyond highest seqno received **in order** (*"What Byte is Next"*)



Piggybacking

i.e. you need to send back an ACK but also some data, may as well send together rather than separately

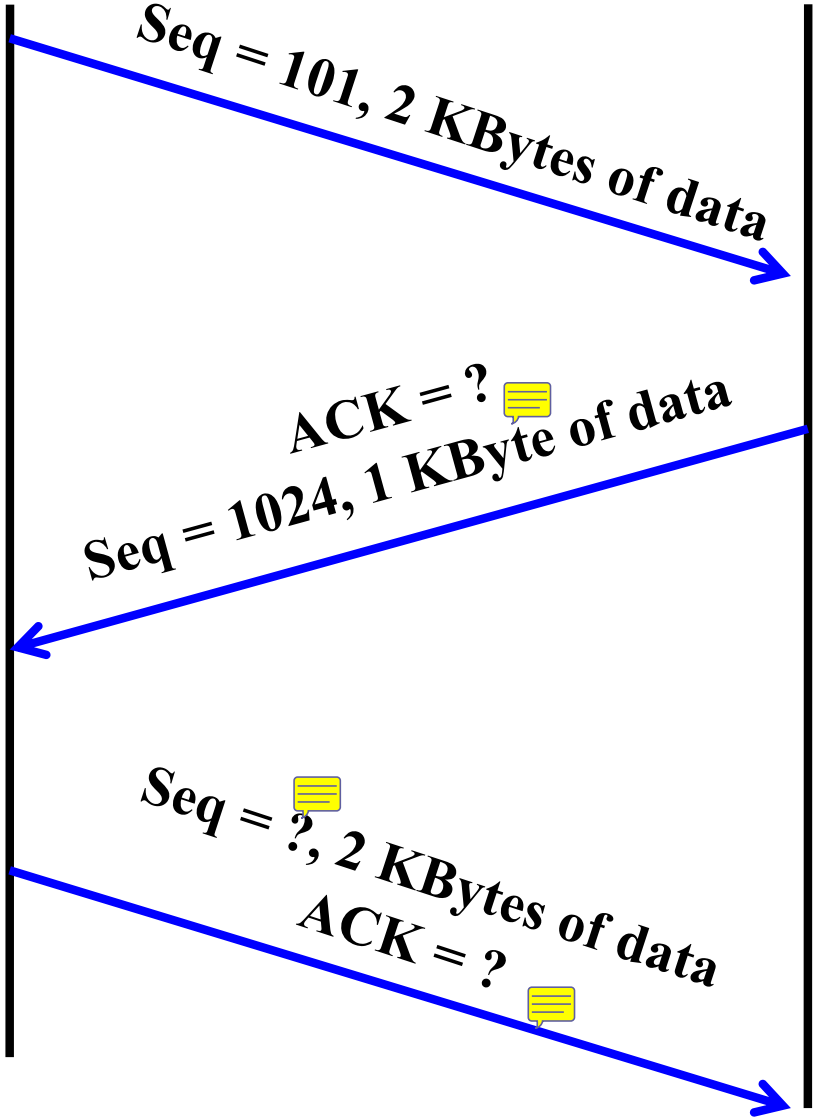
- So far, we've assumed distinct "sender" and "receiver" roles
- In reality, usually both sides of a connection send some data



Without Piggybacking

With Piggybacking

Quiz



1KB = 1024 bytes
2KB = 2048

seq#101, so bytes 101 to 101+2048-1 = bytes 101-2148
so ACK will be for 2149

next seq# will hence be 2149, the ack is for 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)

BEST OF BOTH WORLDS!

Loss with cumulative ACKs

- Sender sends packets with 100Bytes and sequence numbers:
  - 100, 200, 300, 400, 500, 600, 700, 800, 900, …

so bytes 100-199 for packet 1, bytes 200-299 for packet 2, and so on...

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

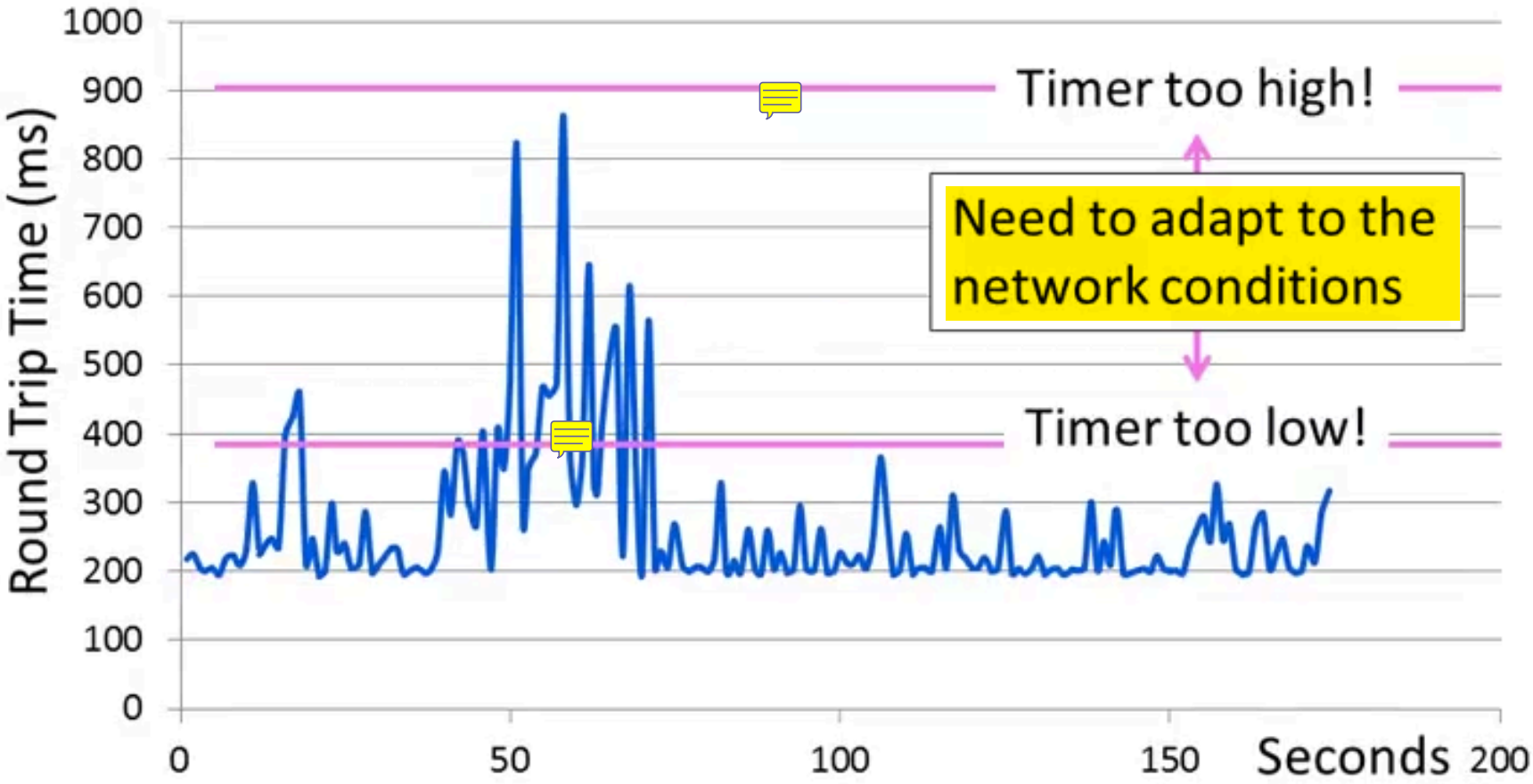
- Stream of ACKs will be:
  - 200, 300, 400, 500, 500, 500, 500,…

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?*)

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

because network layer can take different routes, and because of traffic fluctuations, round trip times vary

you want it to be longer than the RTT but not by much. This prevents both re-sending packets when you don't need to and busy waiting for timeout when a packet has dropped

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`**

they obvs don't give a reliable RTT estimate, especially if the original wasn't dropped but was just taking a long time and happens to arrive right after re-transmission, giving an RTT of 0 essentially.

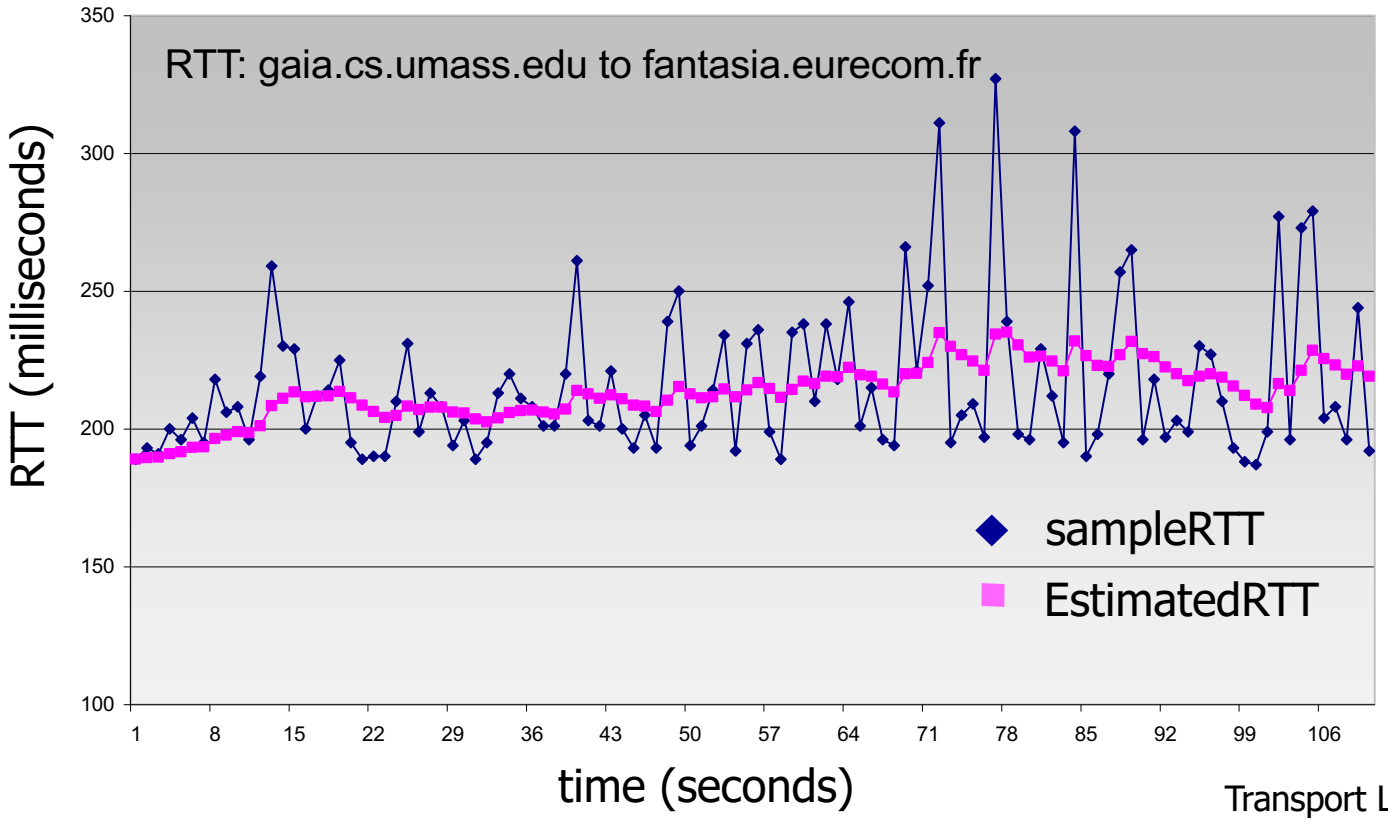
TCP round trip time, timeout

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

weighted because of the alpha, moving because the average is not over the entire thing, but instead the average at each point (based on current and past, with more emphasis on past to protect against outliers)

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

i.e. weigh more based on historical time and not the current sample time, but also each sample decreases in influence as time goes on



TCP round trip time, timeout

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

as seen in the previous slide, the estimated RTT is in the middle (which makes sense because it is a moving average). But we need to rise it so it is over the sample RTT, but not by much. So take into account the current sample RTT and see how much it deviates from the corresponding estimated RTT (the deviation) and scale the timer value based on this deviation.

$$DevRTT = (1-\beta)*DevRTT + \beta*|SampleRTT-EstimatedRTT|$$

difference in sample RTT and estimated RTT

$$(typically,\ \beta = 0.25)$$

again, weight more based on the historical deviation

most sample fit within 4 times your standard deviation e.g. if the average mark was 60/100, and the standard deviation was 9, then pretty much everyones marks will be in the range of 24-96 (obvs there may be exceptions, really good students in the 97+ range and bad students/people who missed the exam in the 23 and - range)

$$TimeoutInterval = EstimatedRTT + 4*DevRTT$$

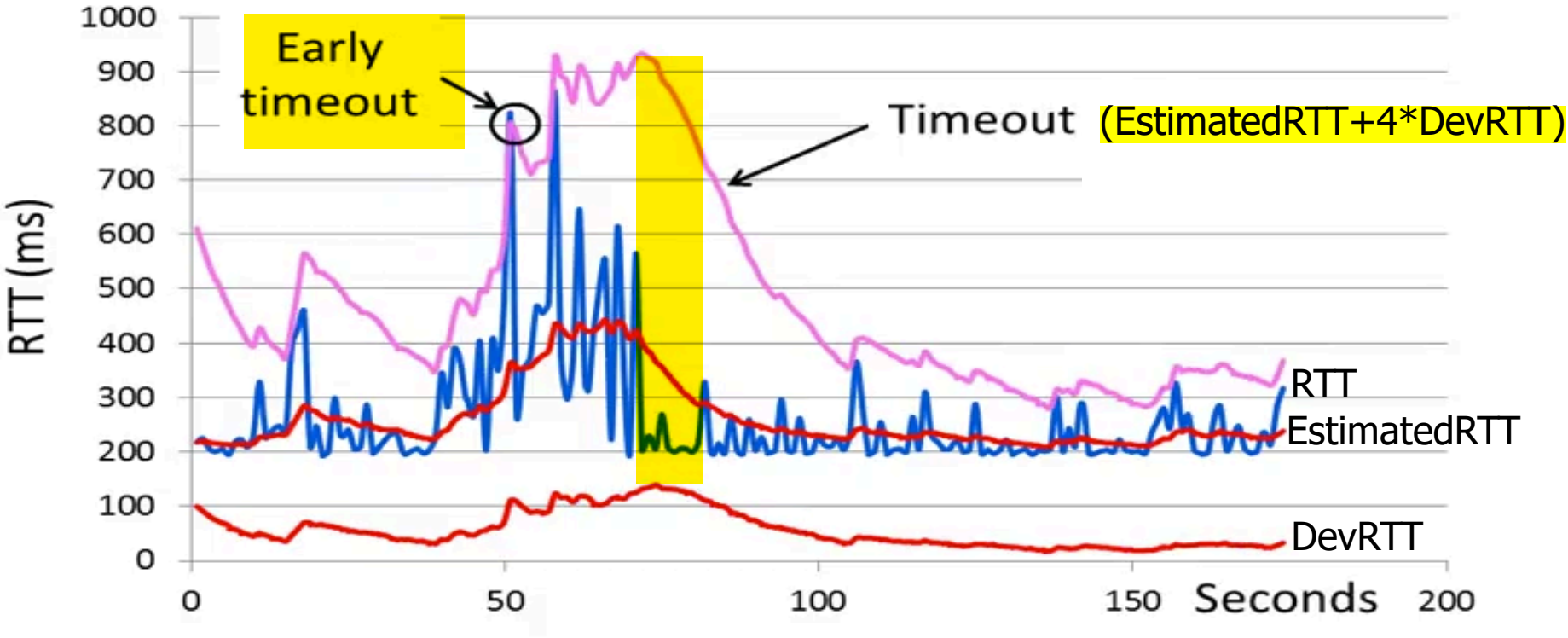
estimated RTT

"safety margin"

Practice Problem: this is probs on exam then, in tute and a practice here, he wants us to know this

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

TCP round trip time, timeout



`TimeoutInterval = EstimatedRTT + 4*DevRTT`

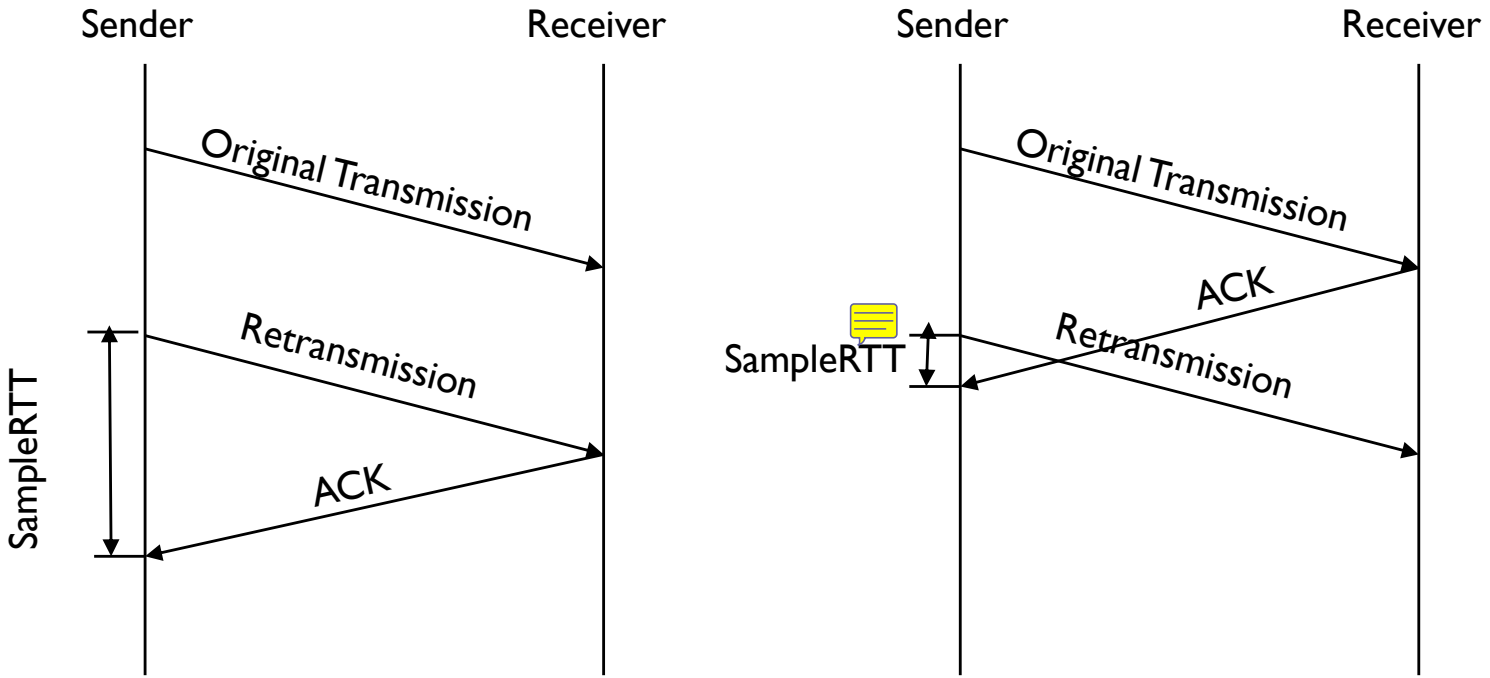
estimated RTT

"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`

it is not for the latest unacked packet, but for the oldest, so first sent unacked packet

timeout:

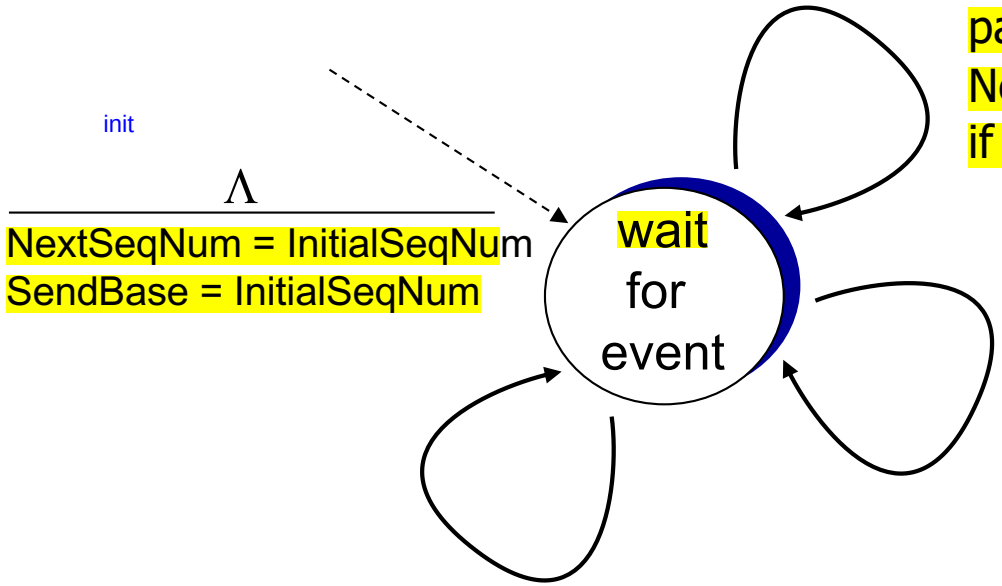
- retransmit segment that caused timeout
- restart timer

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

because if it wasn't running, then we are now the oldest unacked packet (because we are the only one, otherwise time would have been running!)

timeout

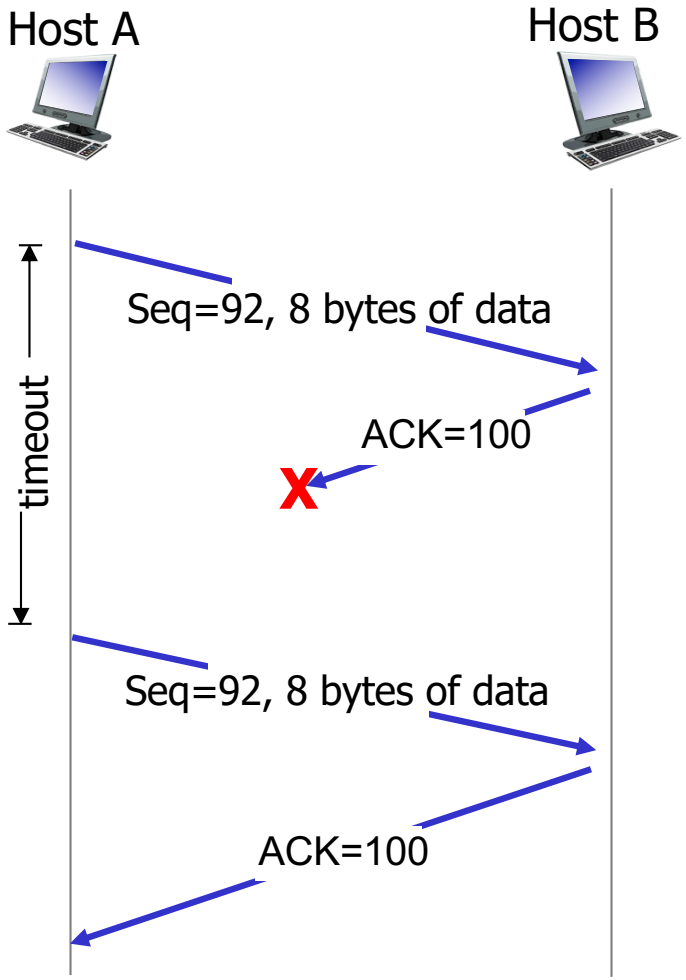
retransmit not-yet-acked segment with smallest seq. #
start timer

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
    }
```

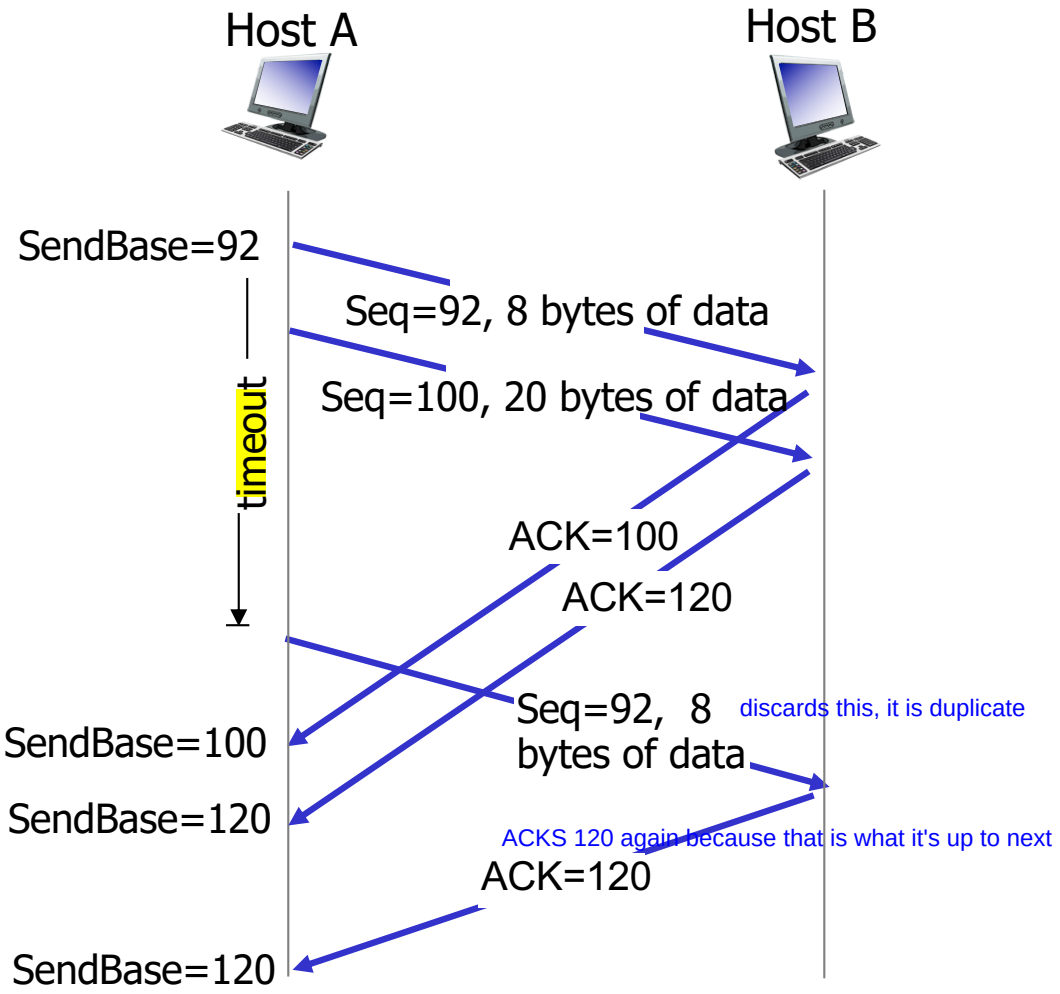
if the receiver has requested the next chunk, then set the send base to be that chunk, and also, if there are unacked in flight segments, start the timer for them, if there isn't stop the timer (if there is nothing in flight there is nothing to time)

TCP: retransmission scenarios



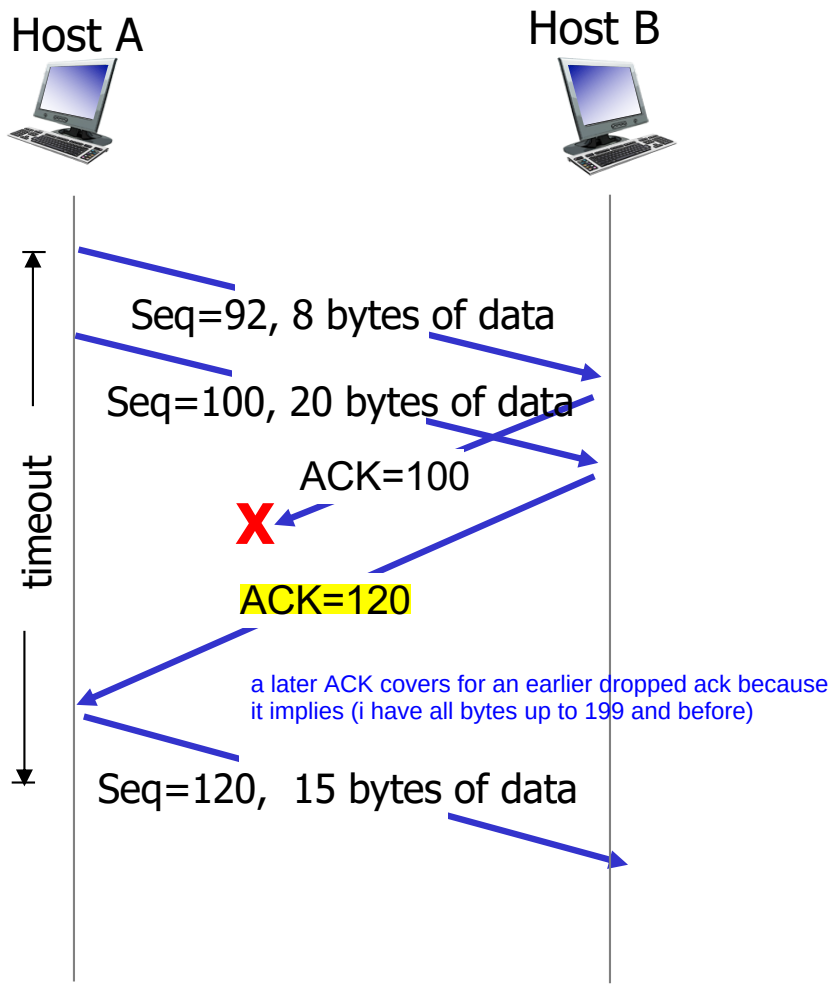
lost ACK scenario

saved by timer timeout

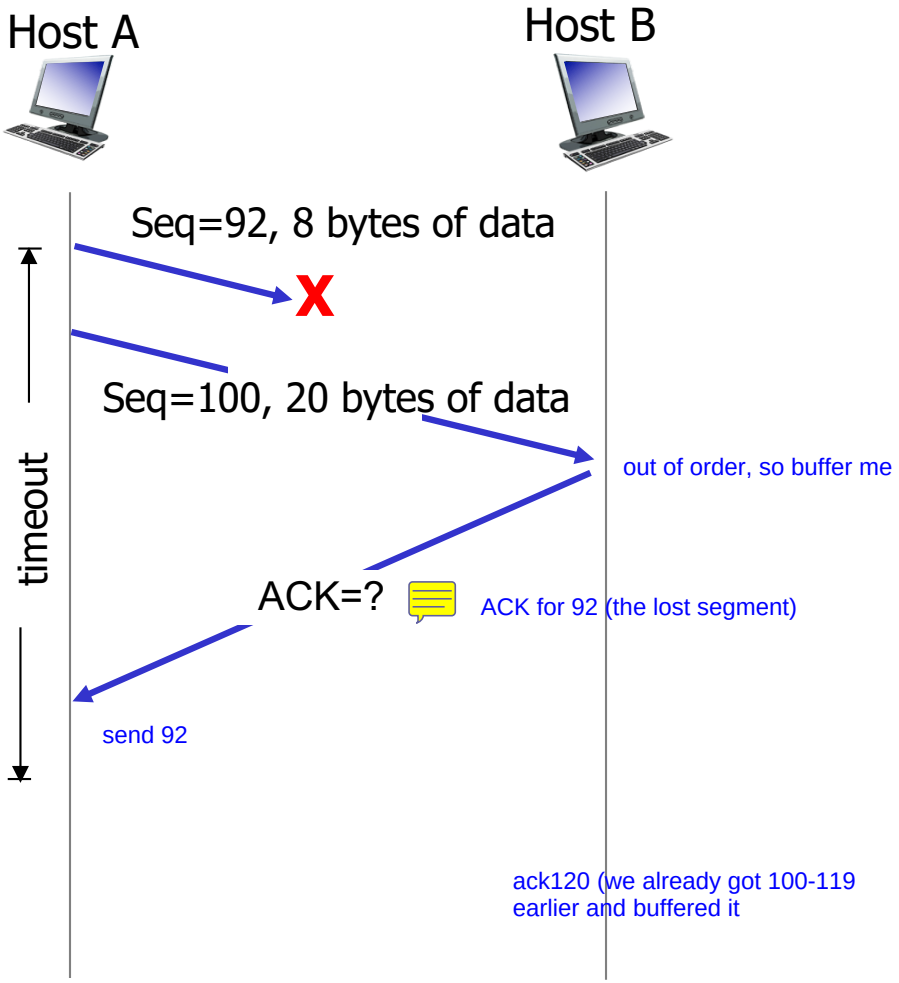


premature timeout

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 the delay is because you are hoping a later ACK comes in and you can ACK 2 for the price of 1
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments this is the 2 for the price of 1 scenario, everything was in order, but the first segment has ACK pending because it is still within the 500ms wait time
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *duplicate ACK*, indicating seq. # of next expected byte AND buffer store this current packet!
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

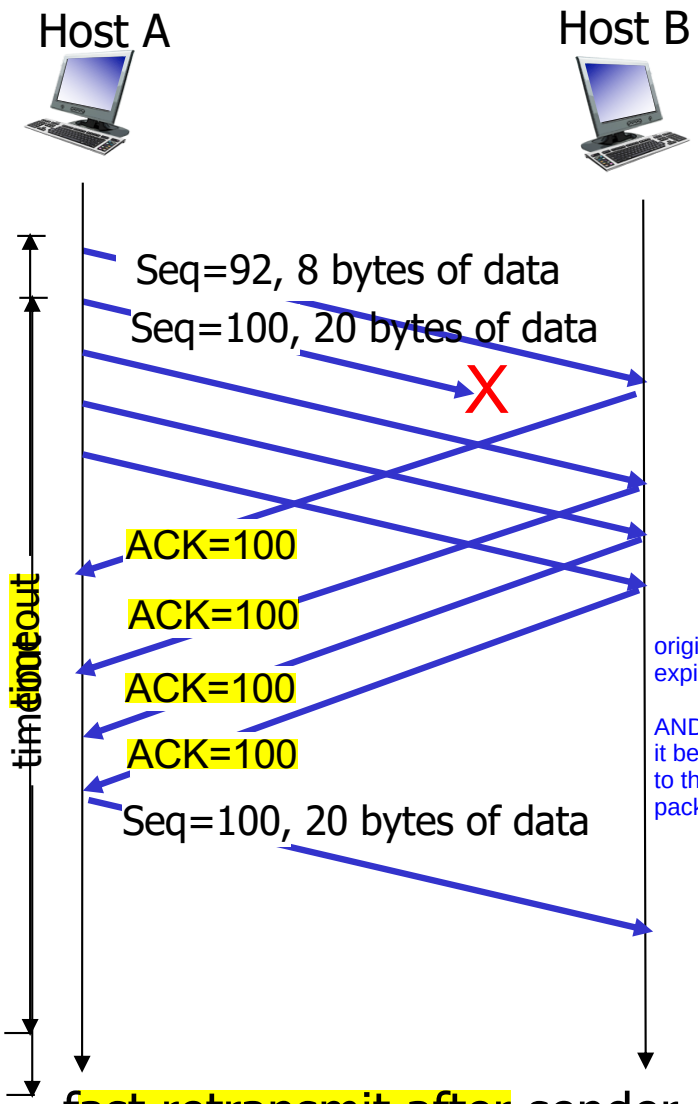
if its not the lower end of the gap then that's the above situation (out of order)
remember, receiver always requests lowest seq# segment that it doesn't yet have!

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



fast retransmit after sender receipt of triple duplicate ACK

original ack, then dupes 1, 2, and 3, then premature retransmit before timer even expires

AND the good thing here is that we know the retransmitted packet is likely to make it because the fact we are getting these duplicate acks means packets are arriving to the host B (it cannot send an ack back whenever it likes, only when it receives a packet)

original + 3*dupe = resend without timer interrupt

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)

this is the busy wait part of the graph earlier

so the networks is actually working because we are getting ACKS back which means segments are making it to the receiver. They just didn't get the lost one

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

B. 436

C. 435

the senders ACK is what he wants next from the receiver, meaning sender has received 435 bytes successfully already from the receiver

D. 1334

E. 536

www.zeetings.com/salil

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

segment staring at 1234 could be a re-transmit, meaning 1335 could already have been sent in a pipelined batch and buffered at the receiver

B. No

www.zeetings.com/salil

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

www.zeetings.com/salil

EstRTT = (1-alpha)*EstRTT + alpha*SampleRTT

StdDev = (1-beta)*StdDev + beta*([SampleRTT - EstRTT])

timeout = EstRTT + 4*StdDev

now, because sample RTT has increased, and EstRTT equation uses mult and add, EstRtt will increase. By the same argument, stdDev and timeout will both increase. Hence the timeout will increase.

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

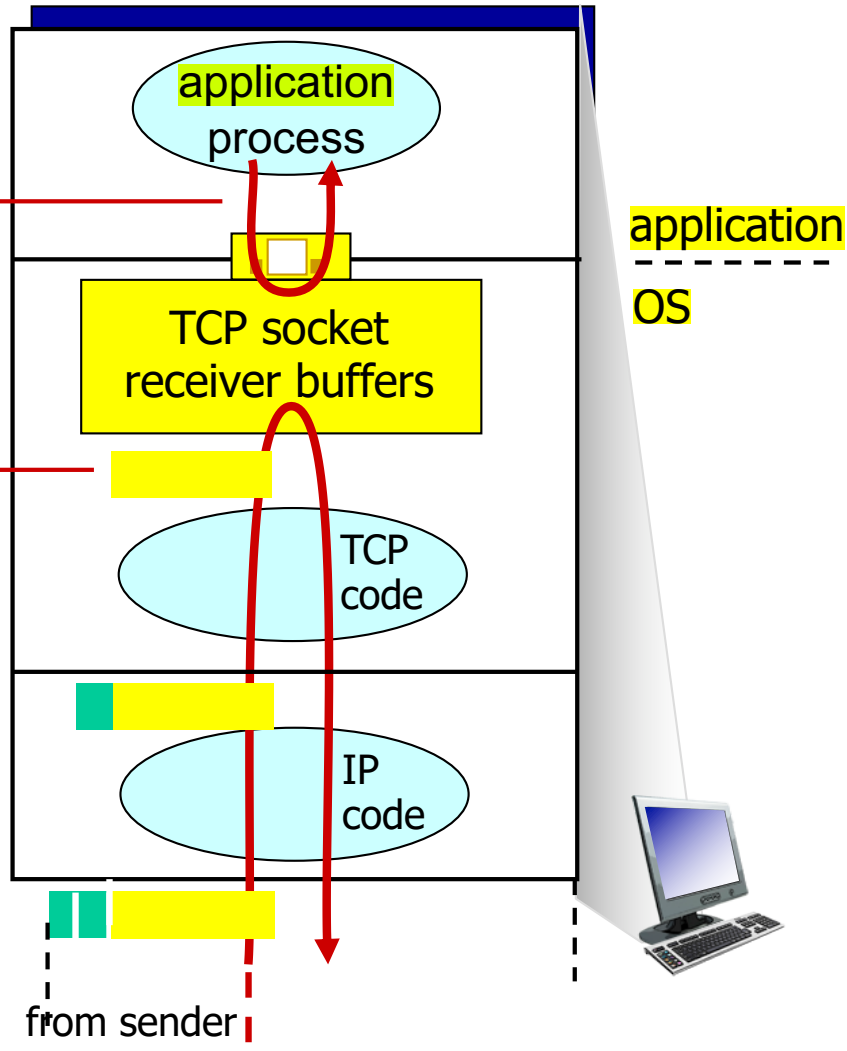
3.6 principles of congestion control

3.7 TCP congestion control

TCP flow control

application may remove data from TCP socket buffers ....

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



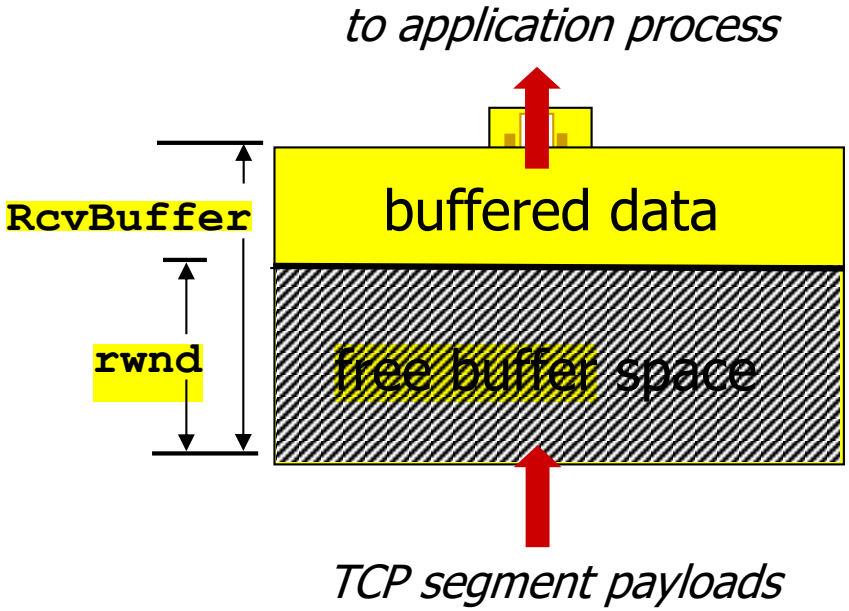
receiver protocol stack

flow control

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

TCP flow control

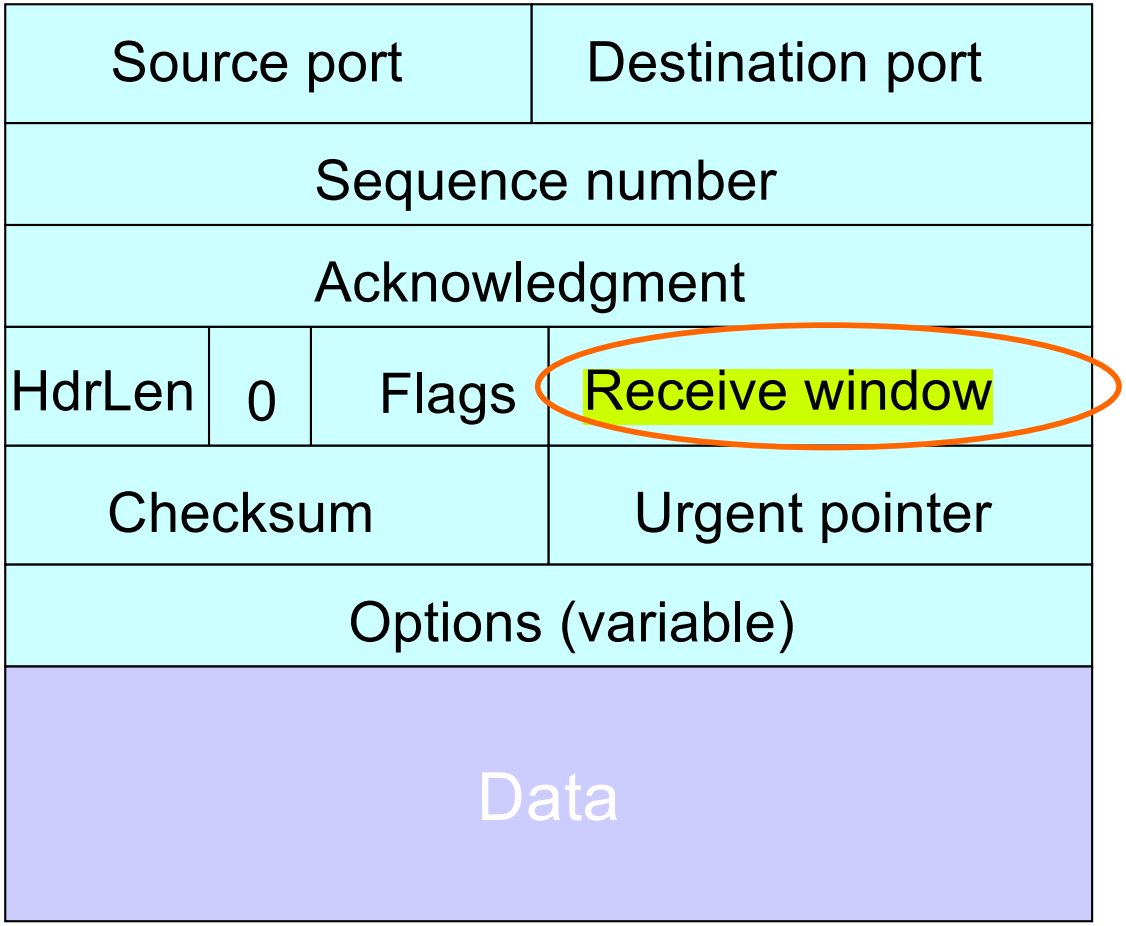
- receiver "advertises" free buffer space by including **`rwnd`** value in TCP header of receiver-to-sender segments
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unacked ("in-flight") data 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

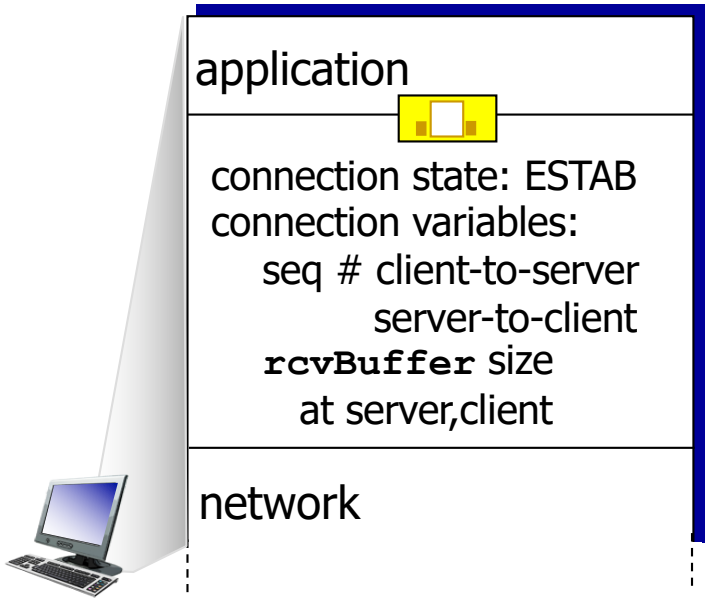
3.6 principles of congestion control

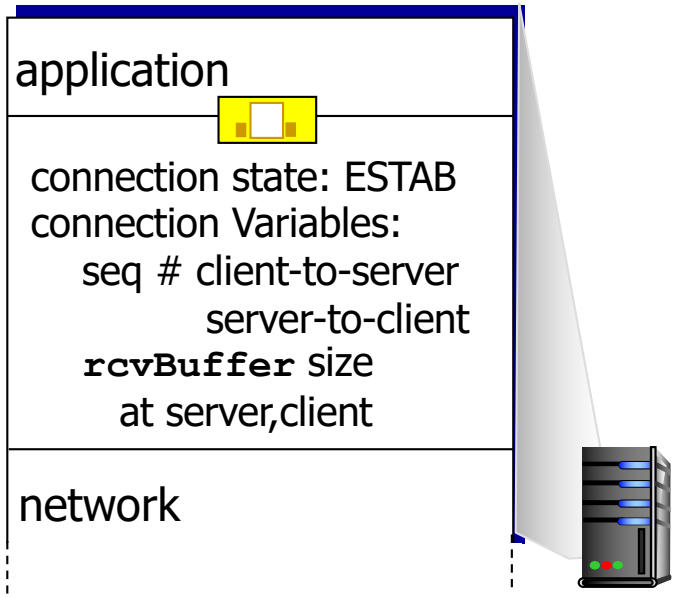
3.7 TCP congestion control

Connection Management

before exchanging data, sender/receiver "handshake":

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

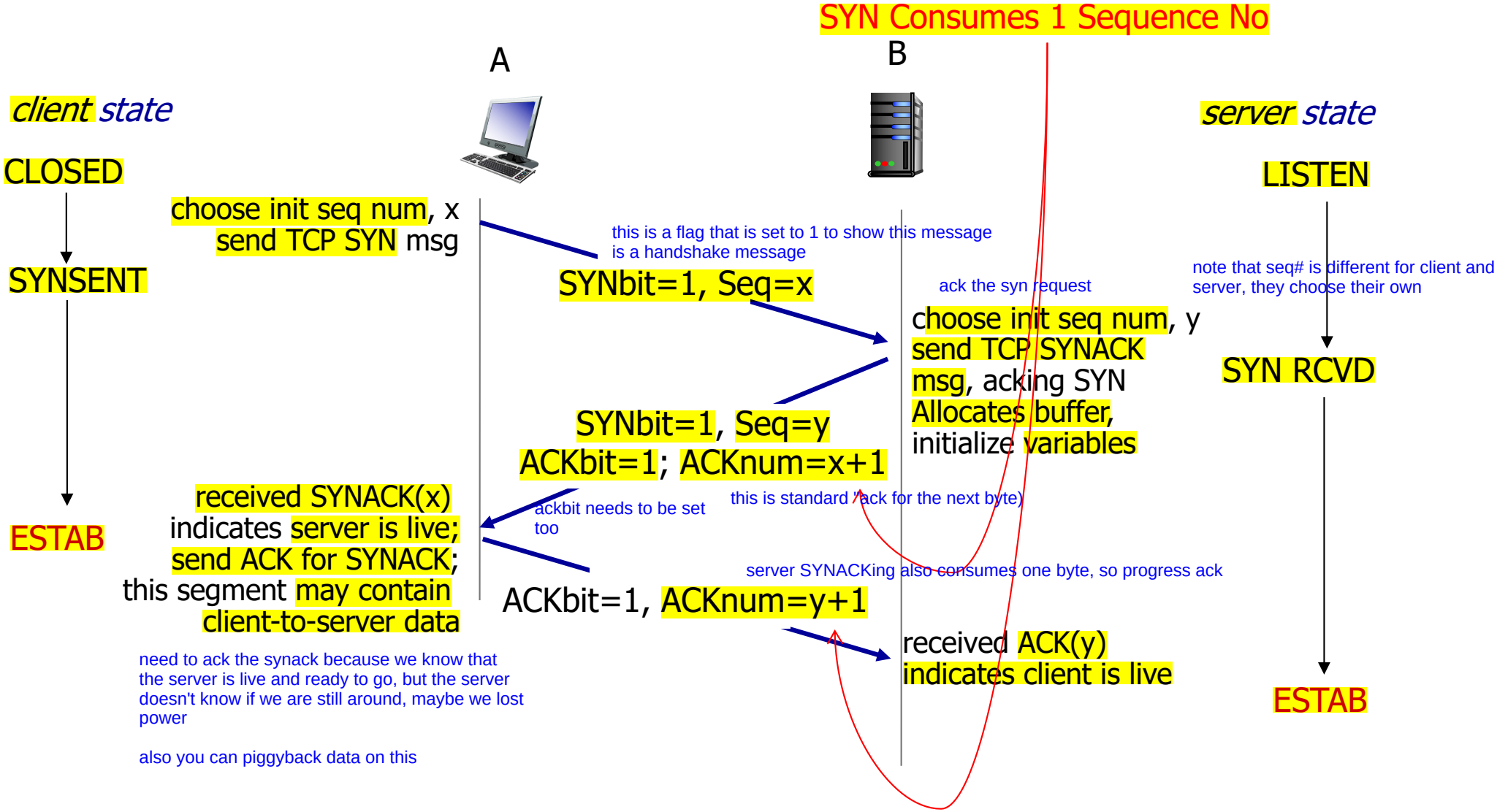




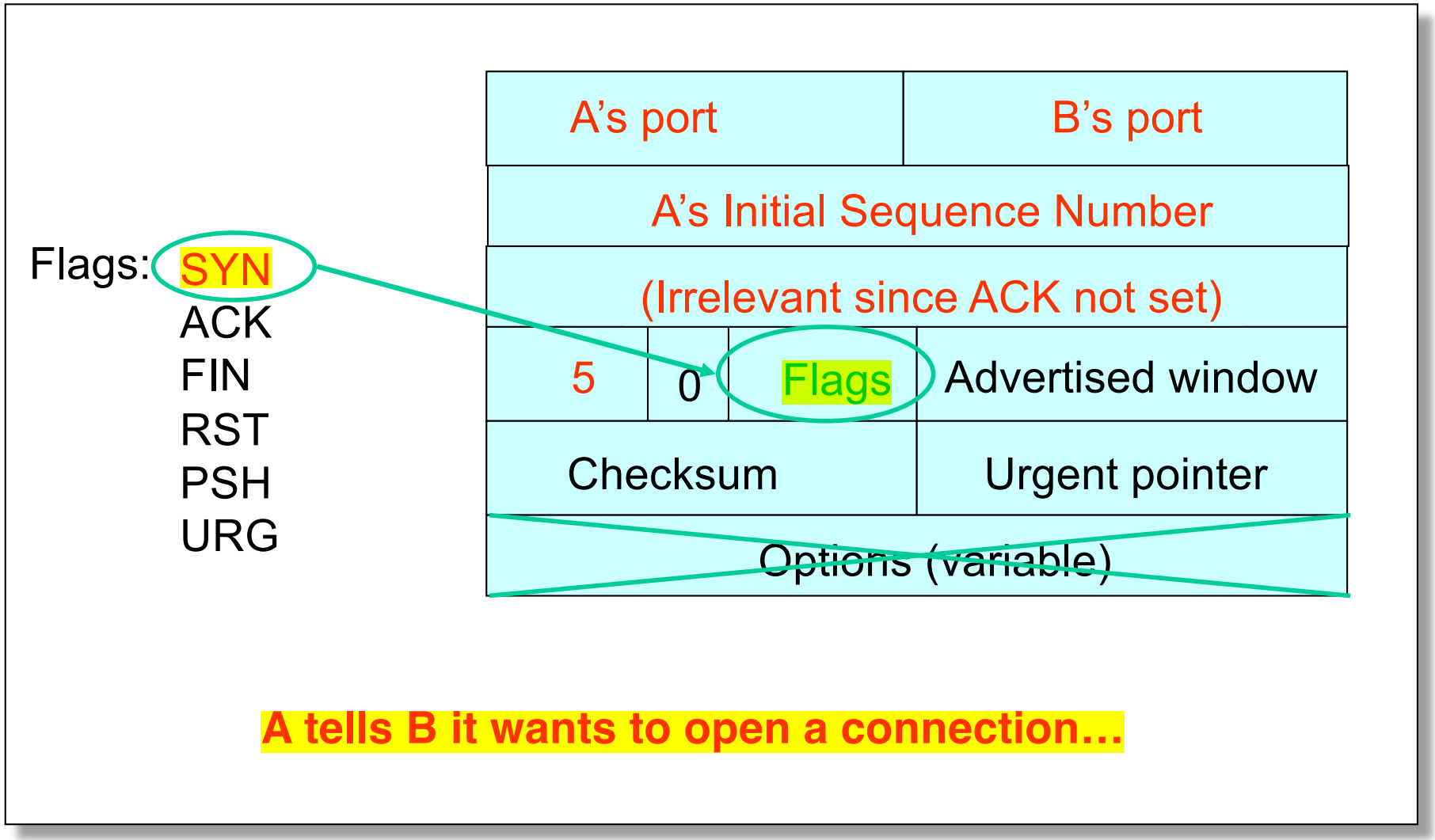

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

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

TCP 3-way handshake

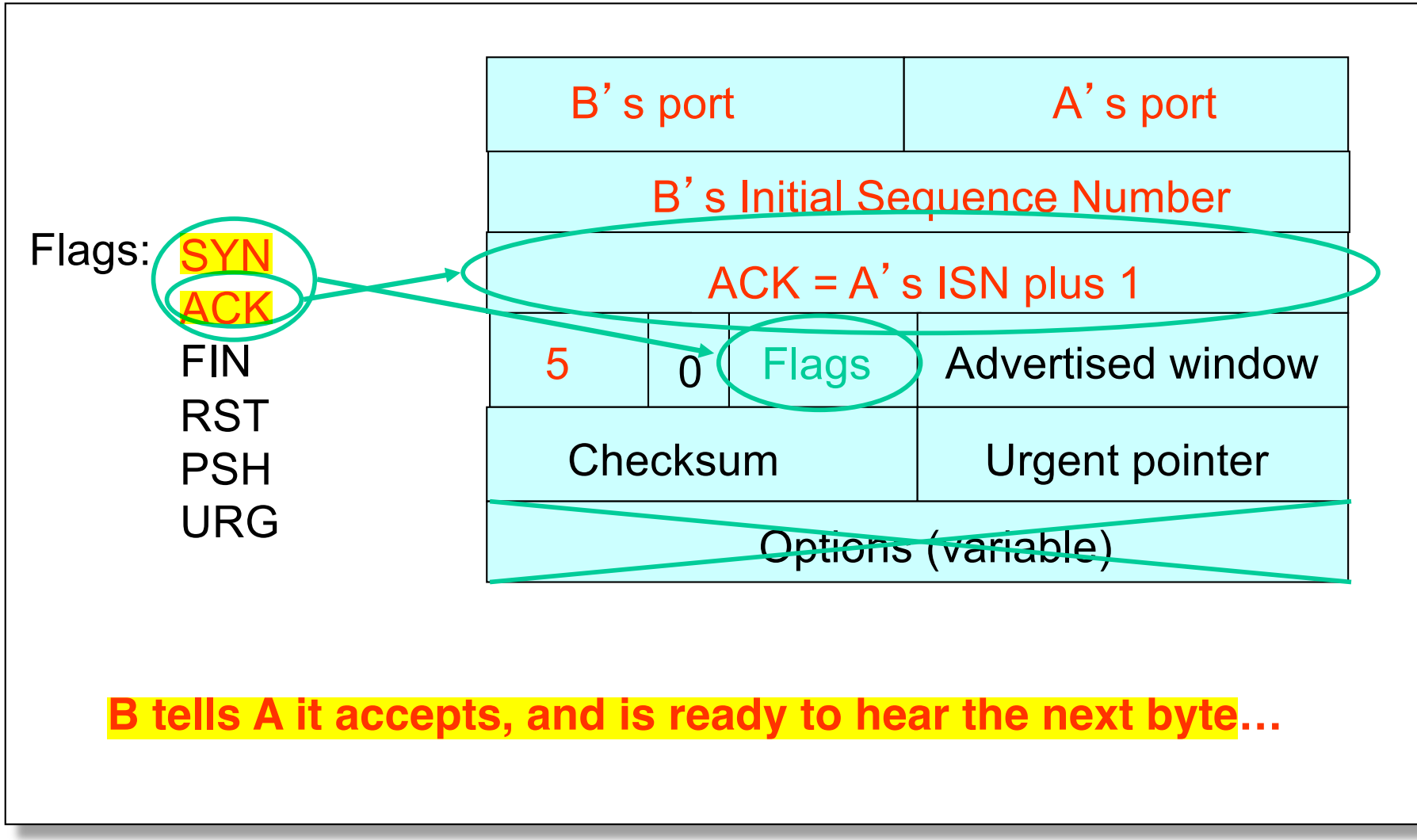


Step 1: A's Initial SYN Packet



A tells B it wants to open a connection…

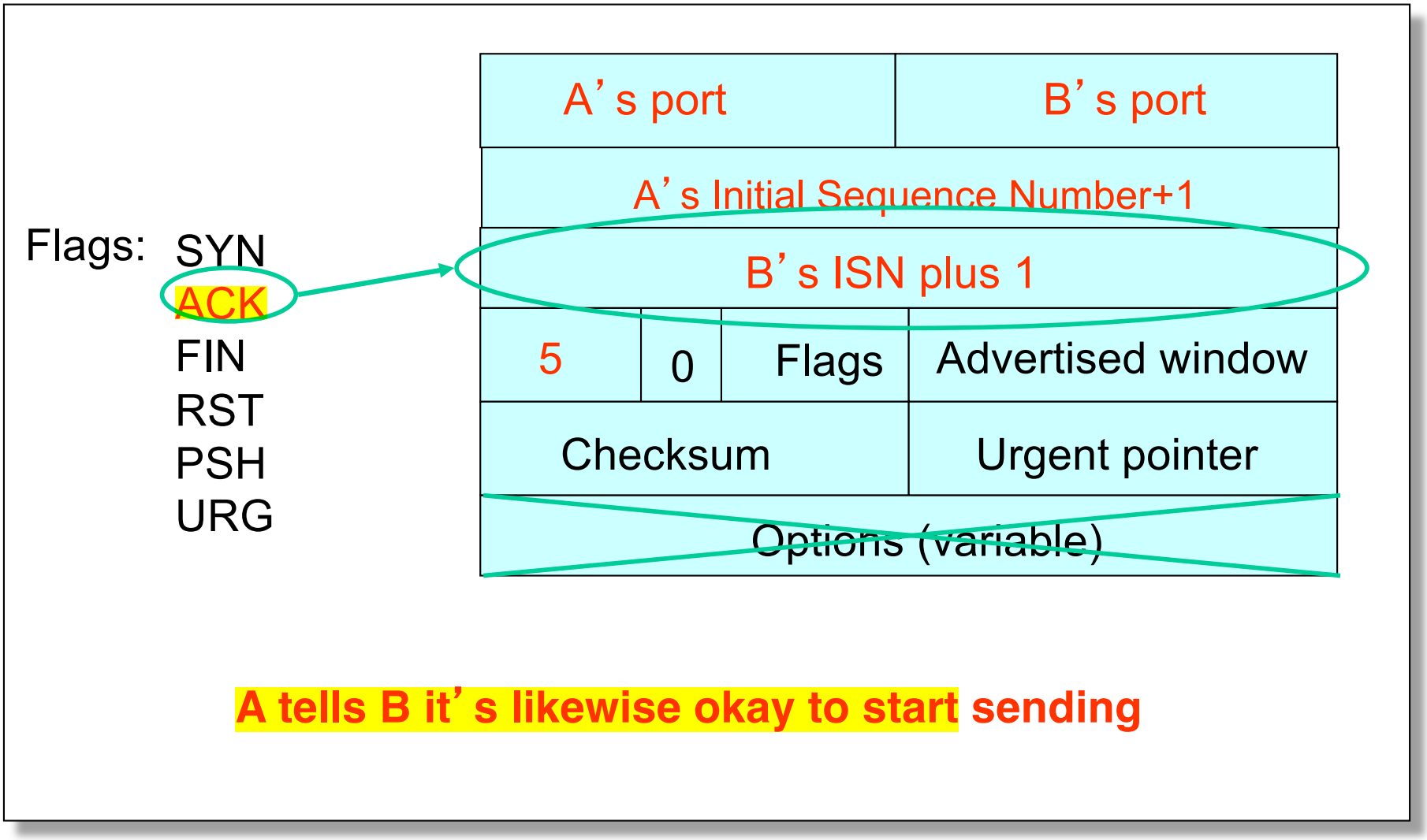
Step 2: B's SYN-ACK Packet



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

… upon receiving this packet, A can start sending data

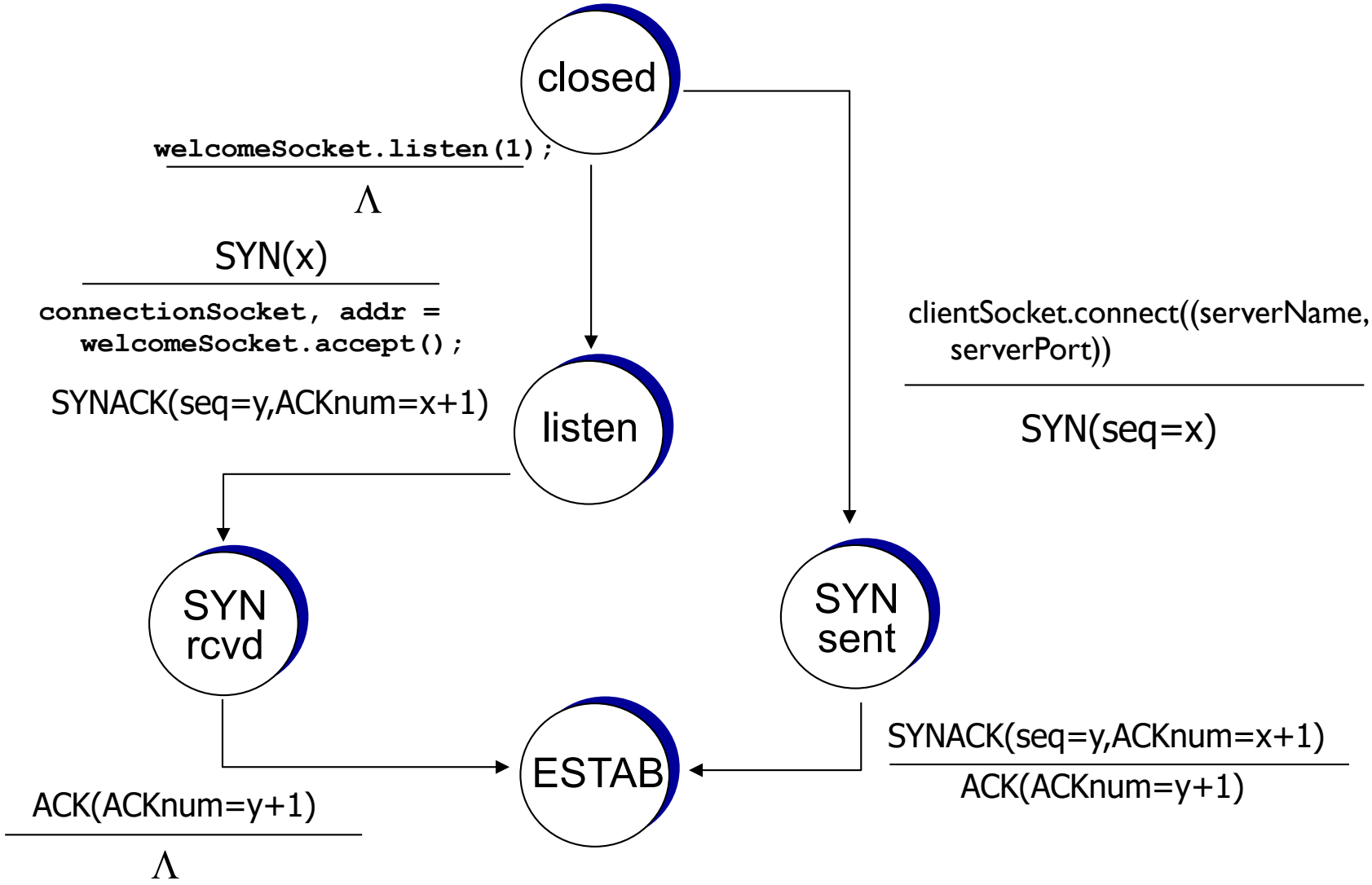
Step 3: A's ACK of the SYN-ACK



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"
- 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

we know people are impatient, so on a reload or re-click, abort the current connection and retry but with faster SYN packet

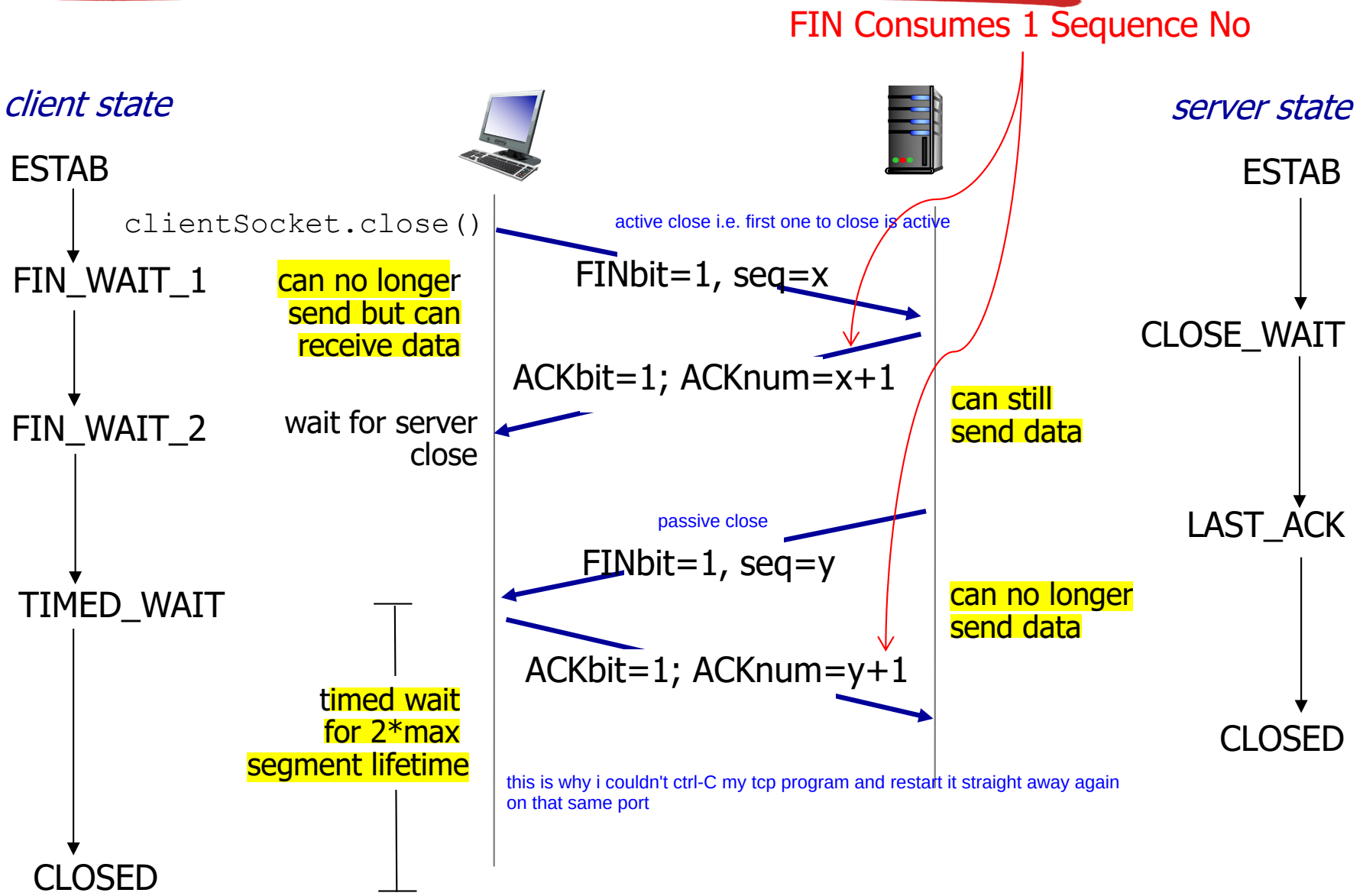
TCP: closing a connection

- 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

FIN means, "I will not send any more data to you, BUUUT you can still send more to me if you like"

so it means, "i've FINished sending"

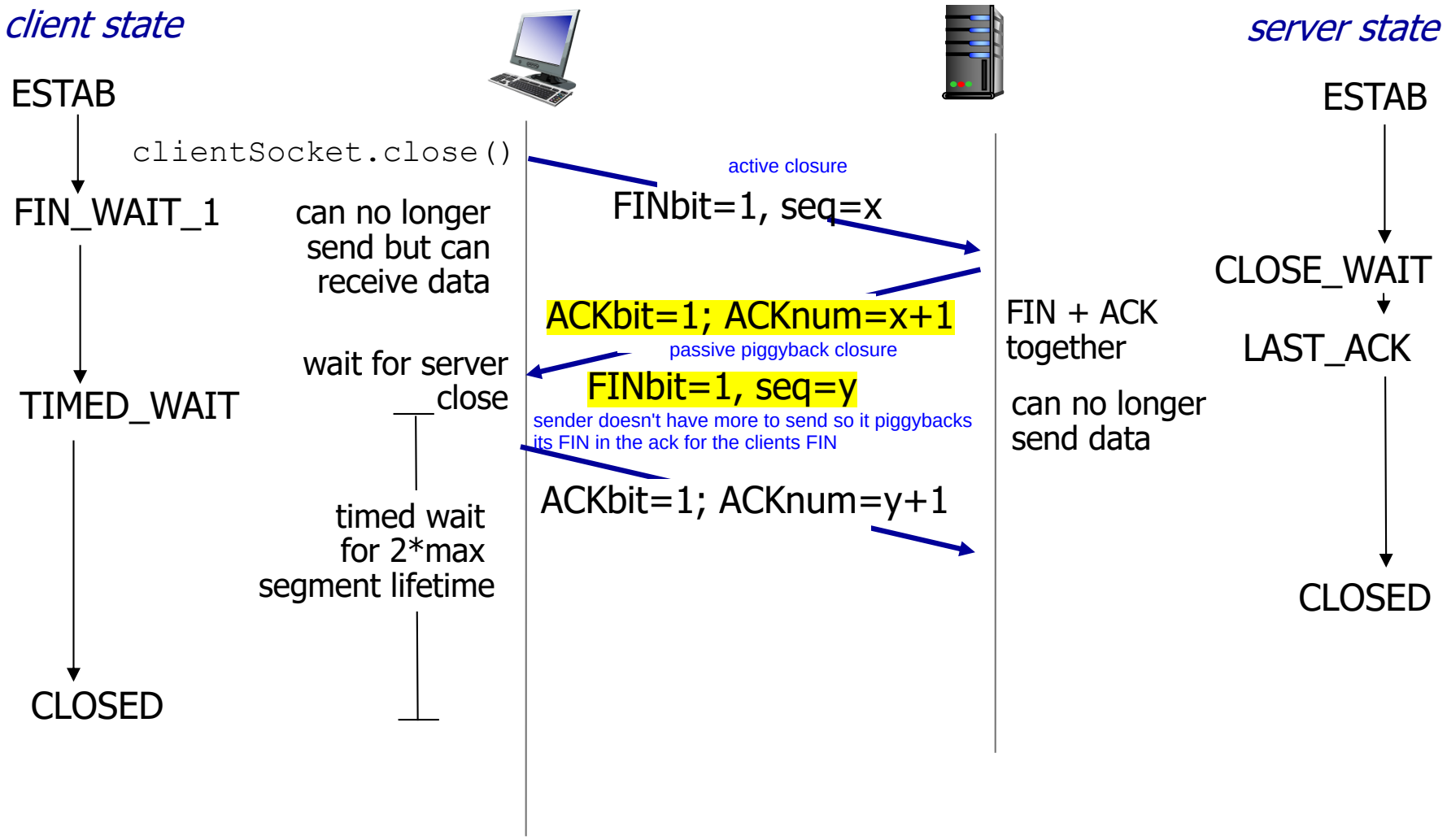
Normal Termination, One at a Time



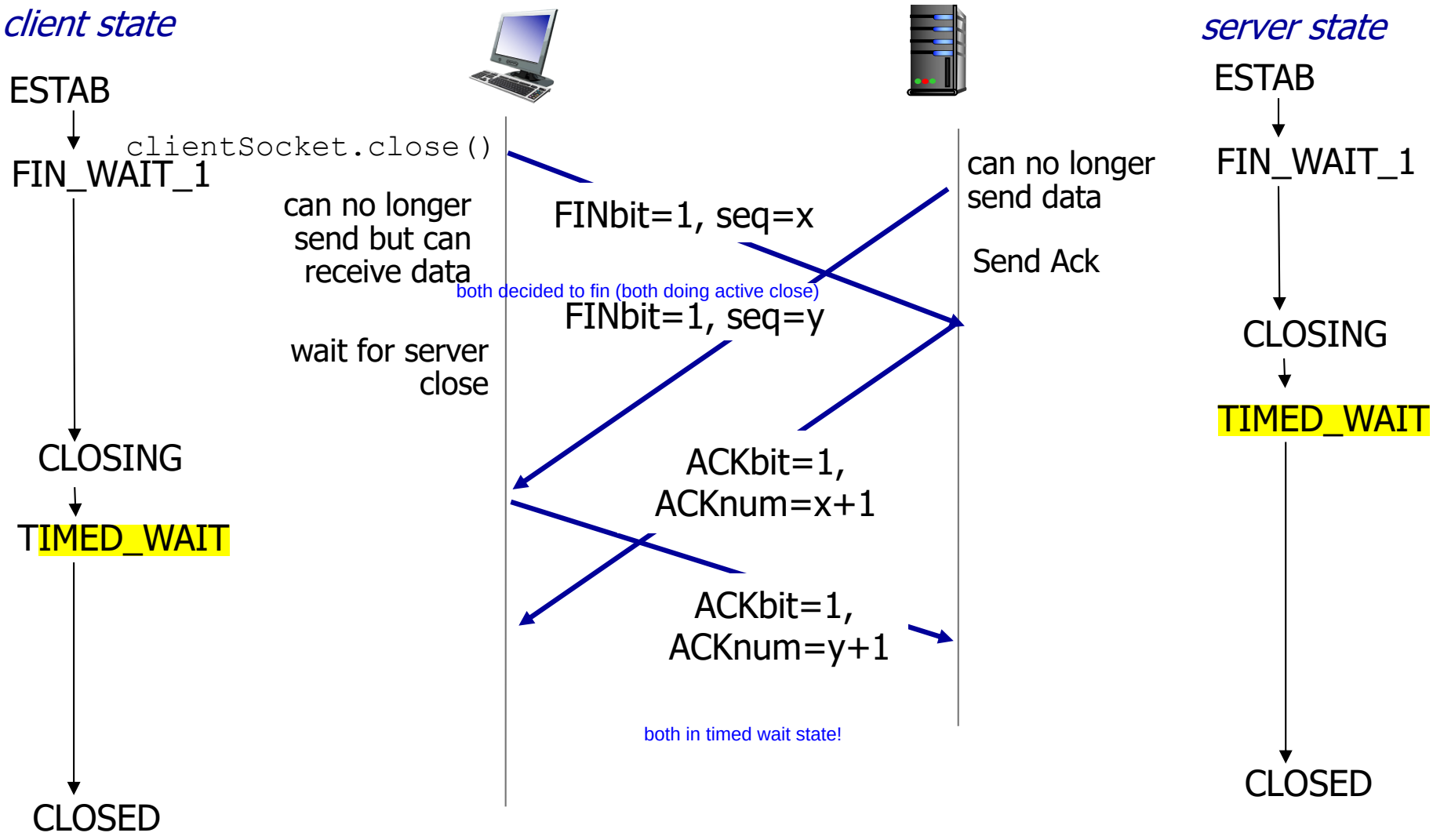
TIMED_WAIT: Can retransmit ACK if last ACK is lost

because the server needs to see our last ACK, it it doesn't, it will have a timer timout and will re-send that fin packet again, but we are now closed so cannot ack again, so the timer will timeout again, and so the server will now re-send the fin pack over and over again forever! :(

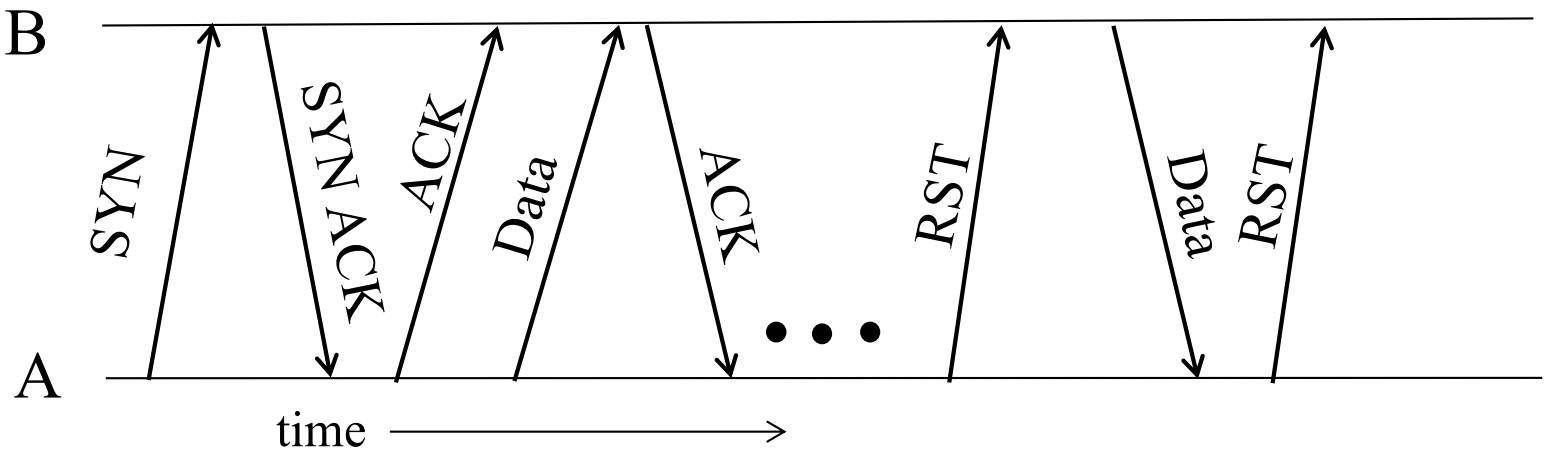
Normal Termination, Both Together



Simultaneous Closure

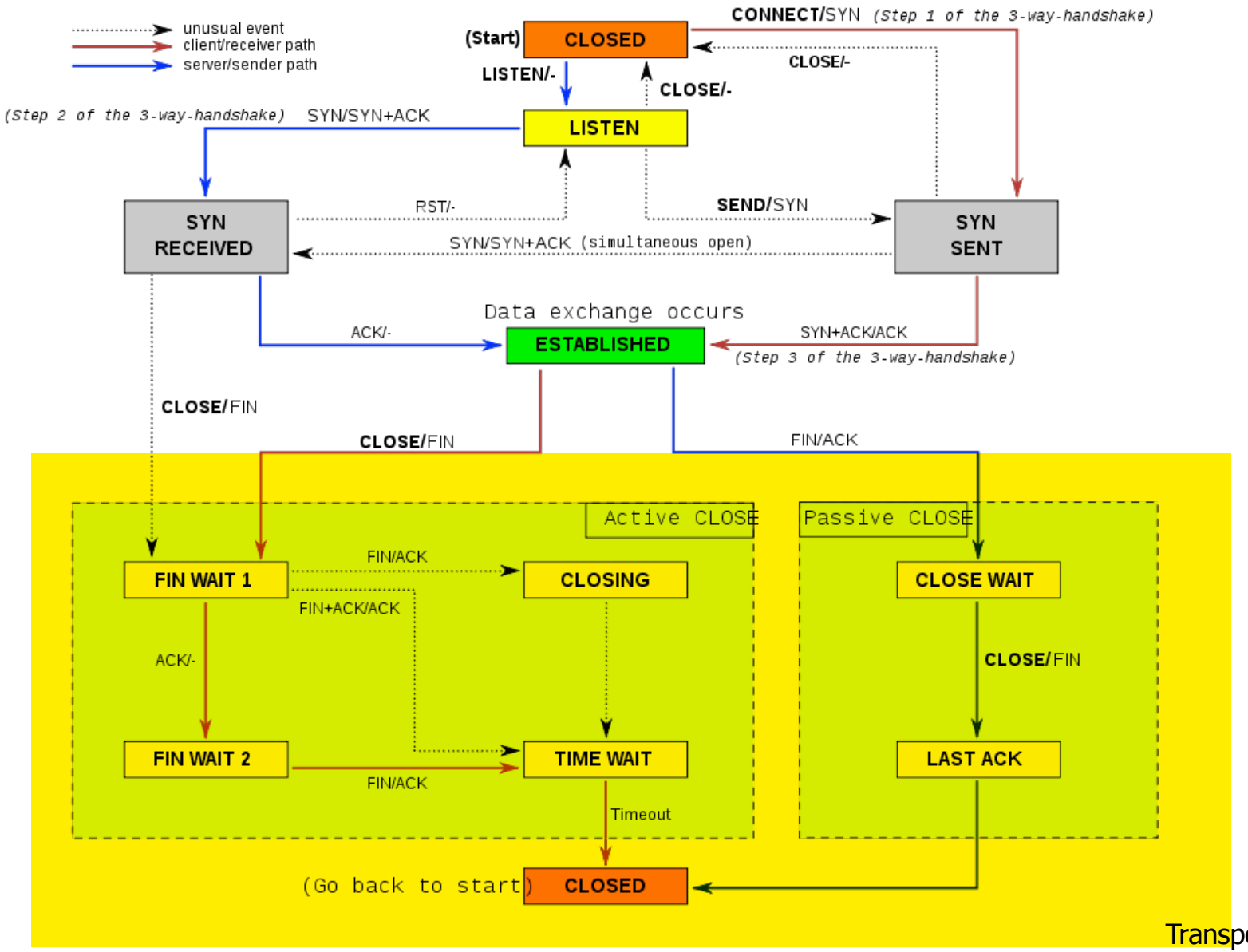


Abrupt Termination



- 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
  - 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 — will still eventually lose, this is a half measure
  - Decrease timeout wait for the 3-way handshake
  - Firewalls: list of known bad source IP addresses — very easy to spoof IP address though
  - 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 remember it is 1, 12, 2 i.e. SYN, SYNACK, ACK, where each take 1/2 RTT, hence 1.5RTT total to both be in established state

C. 2RTT

D. 3RTT

www.zeetings.com/salil

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 — this must be the answer. the receiver will ack for the seq# of the segment it needs, allowing the sender to send just that one rather than all

B. Triple Duplicate ACKs — this is for fast re-send of lost packet

C. Receiver window-based flow control — this is to stop sender overwhelming the receiver

D. Timeout estimation algorithm — this is just for getting a nice timer value

NOTE: the LECTURE HAS OPTION B AS CORRECT, but in the lecture the option B is that the receiver can buffer. This is obviously correct as it allows for storing out of order segments and so only having to retransmit specific ones instead of the whole window, but as you can see, that is not what option B says here, and A is the closest as per the arguments i have given

www.zeetings.com/salil

STOP EXAM STUDY HERE! CONGESTION CONTROL IS NOT ON THE MID TERM EXAM!

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

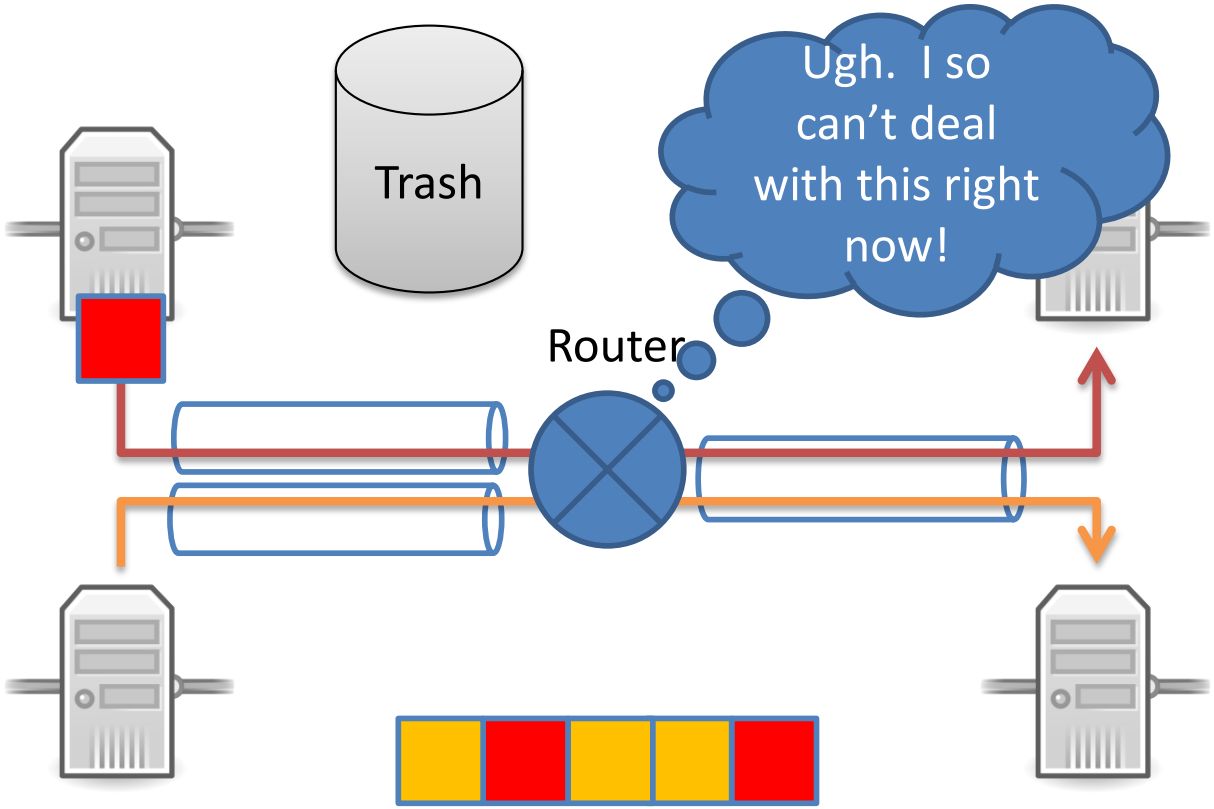
3.7 TCP congestion control

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



Incoming rate is faster than outgoing link can support.

Quiz: What's the worst that can happen?

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

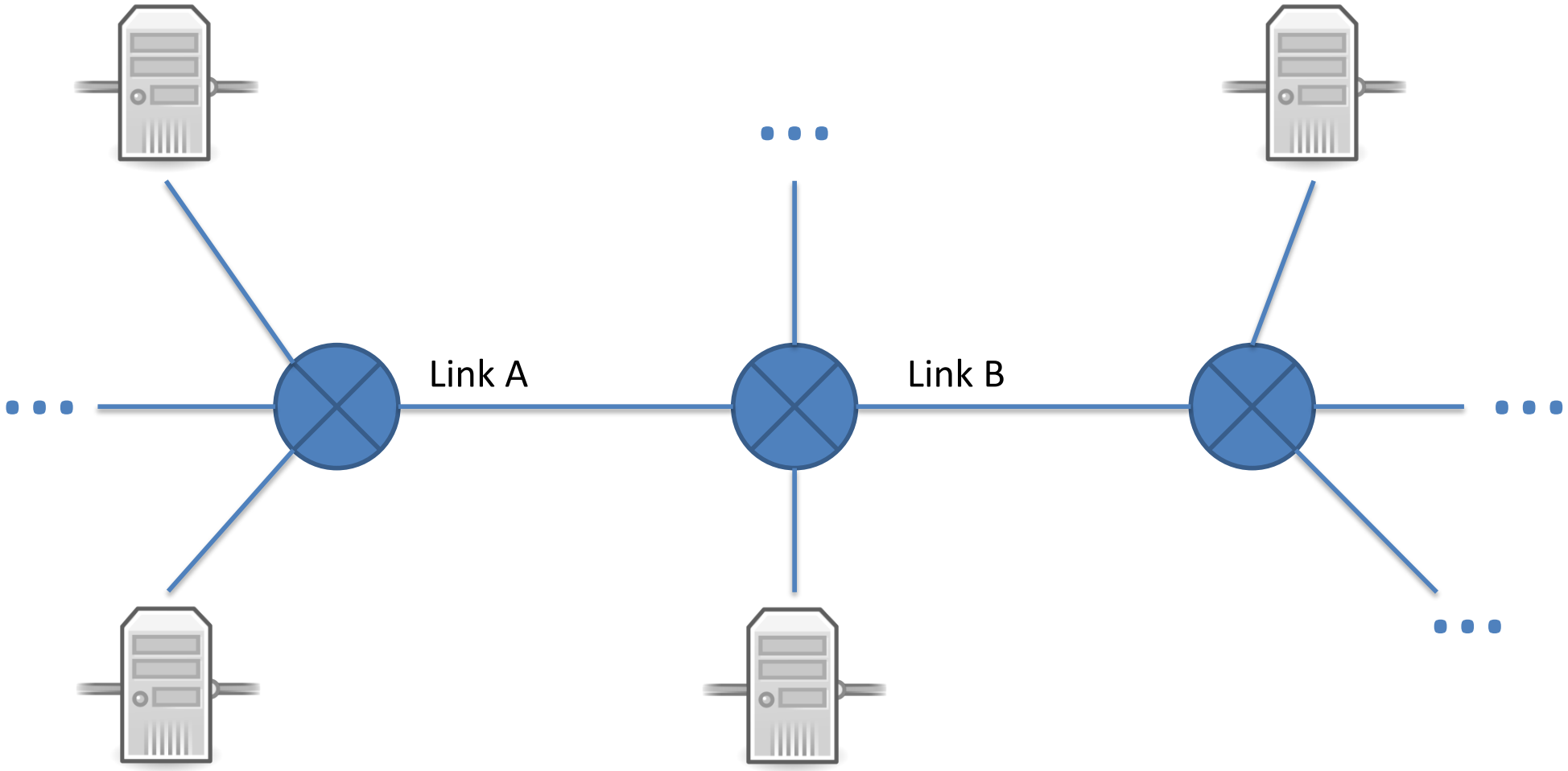
B: There will be retransmissions, but the network will still perform without much trouble.

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

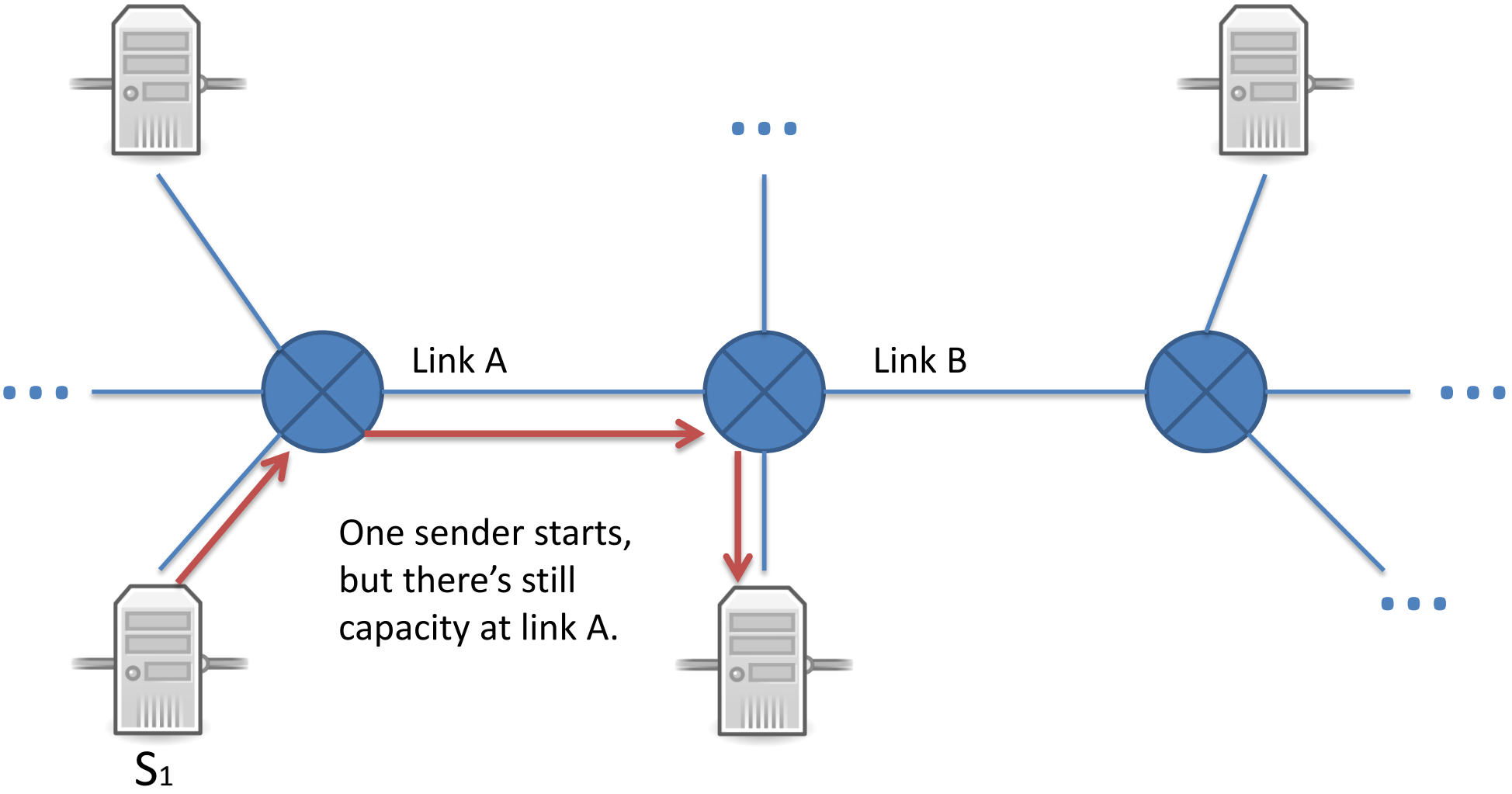
D: The network will become completely unusable

www.zeetings.com/salil

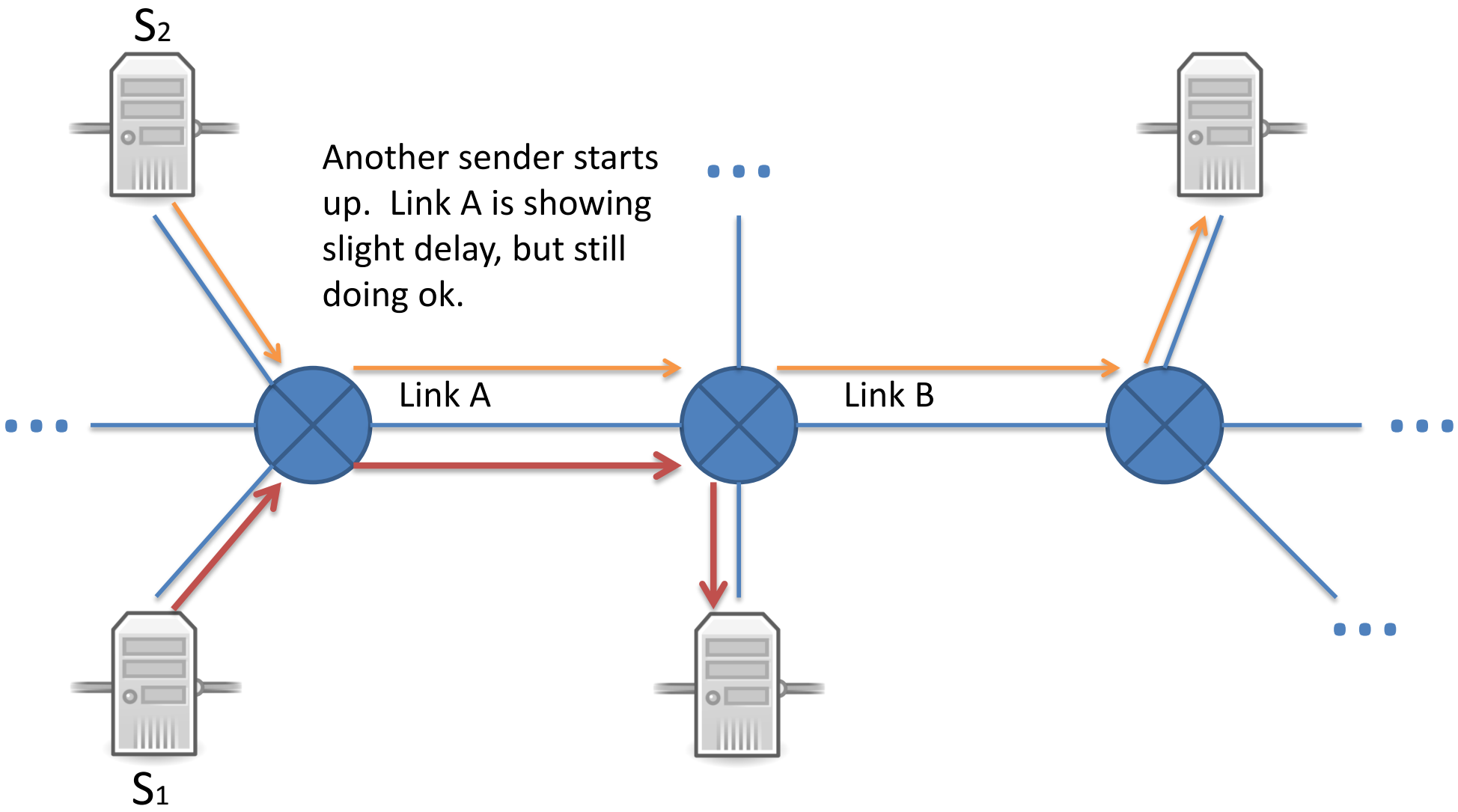
Congestion Collapse



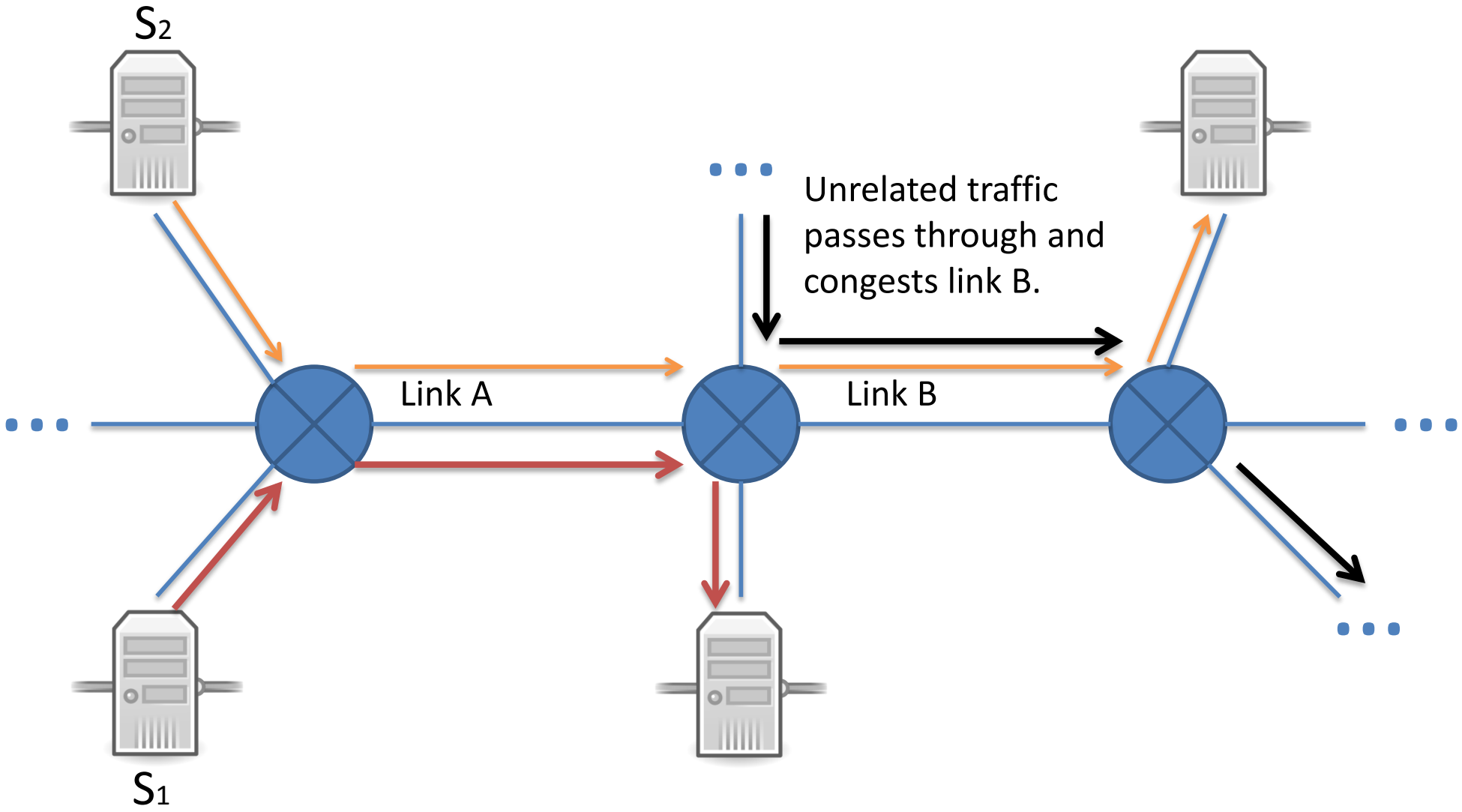
Congestion Collapse



Congestion Collapse

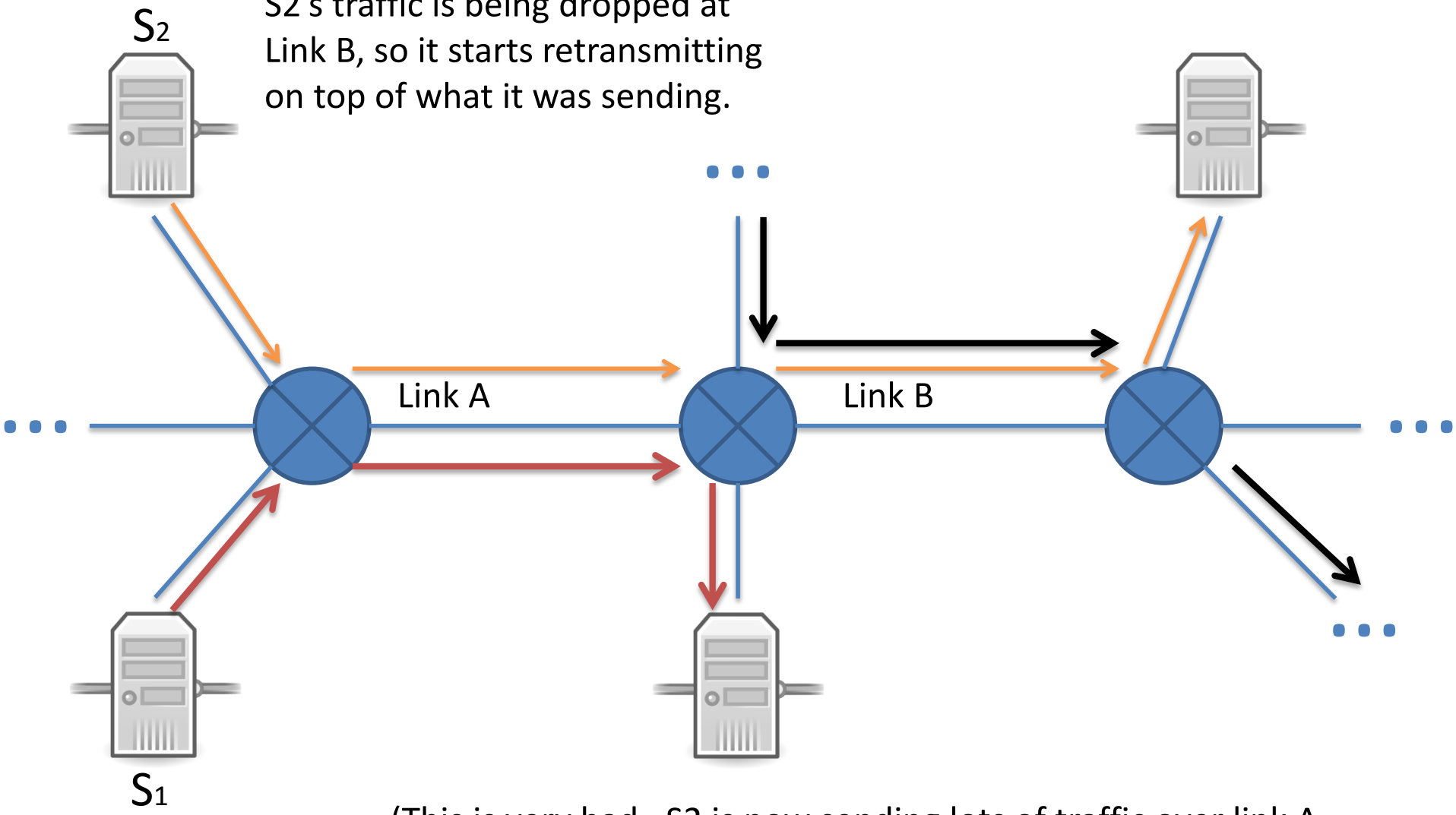


Congestion Collapse



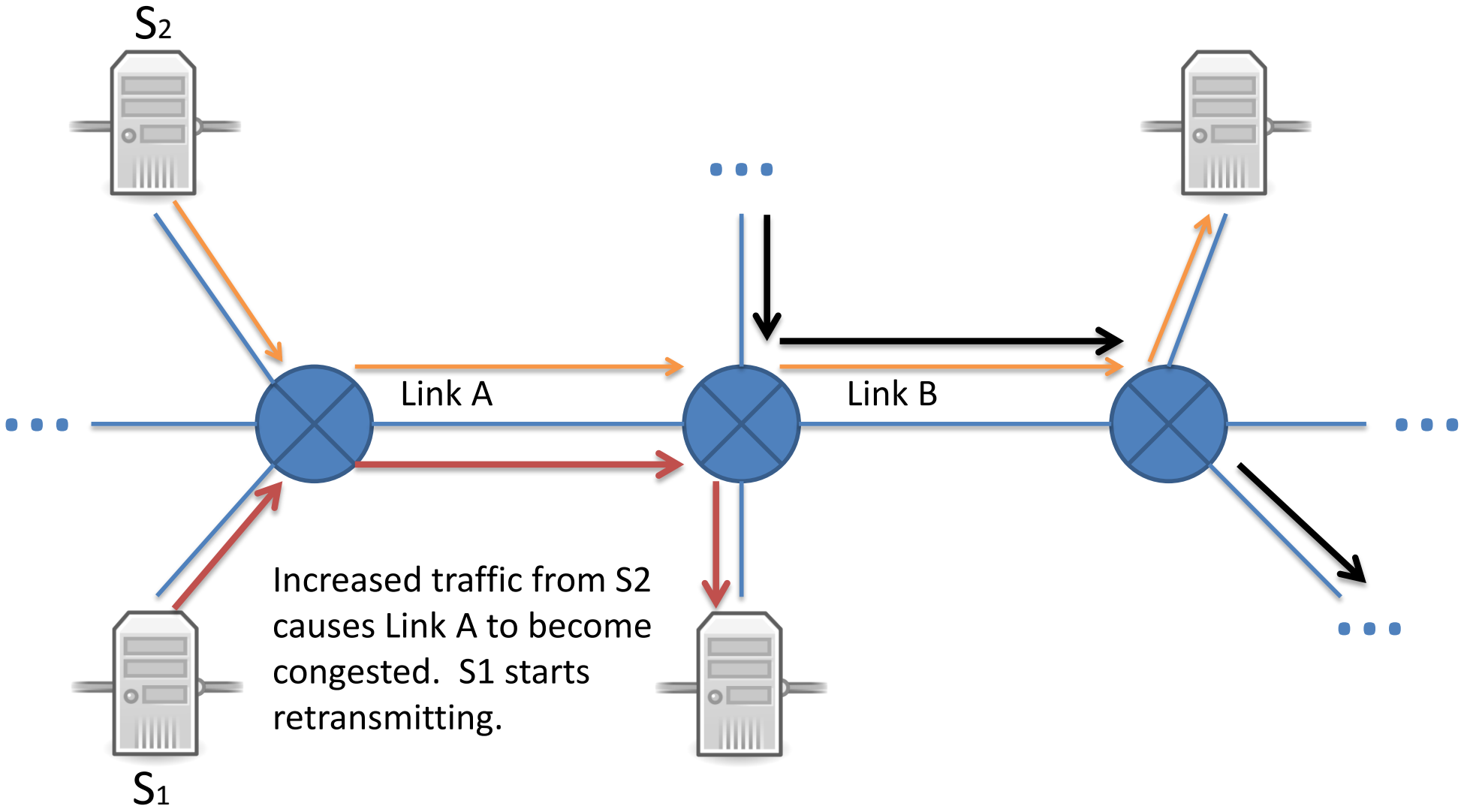
Congestion Collapse

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

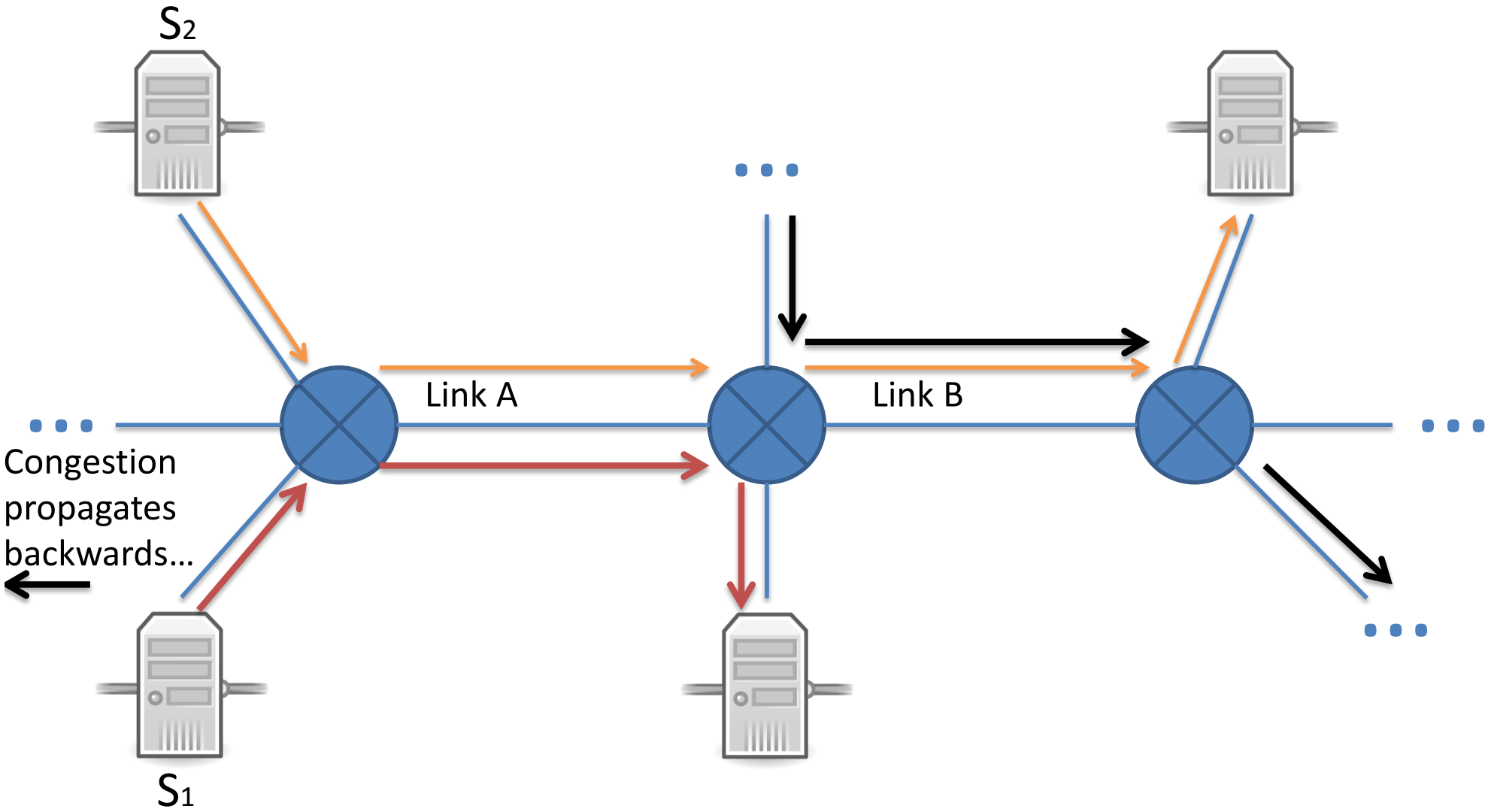


(This is very bad. S2 is now sending lots of traffic over link A that has no hope of crossing link B.)

Congestion Collapse



Congestion Collapse



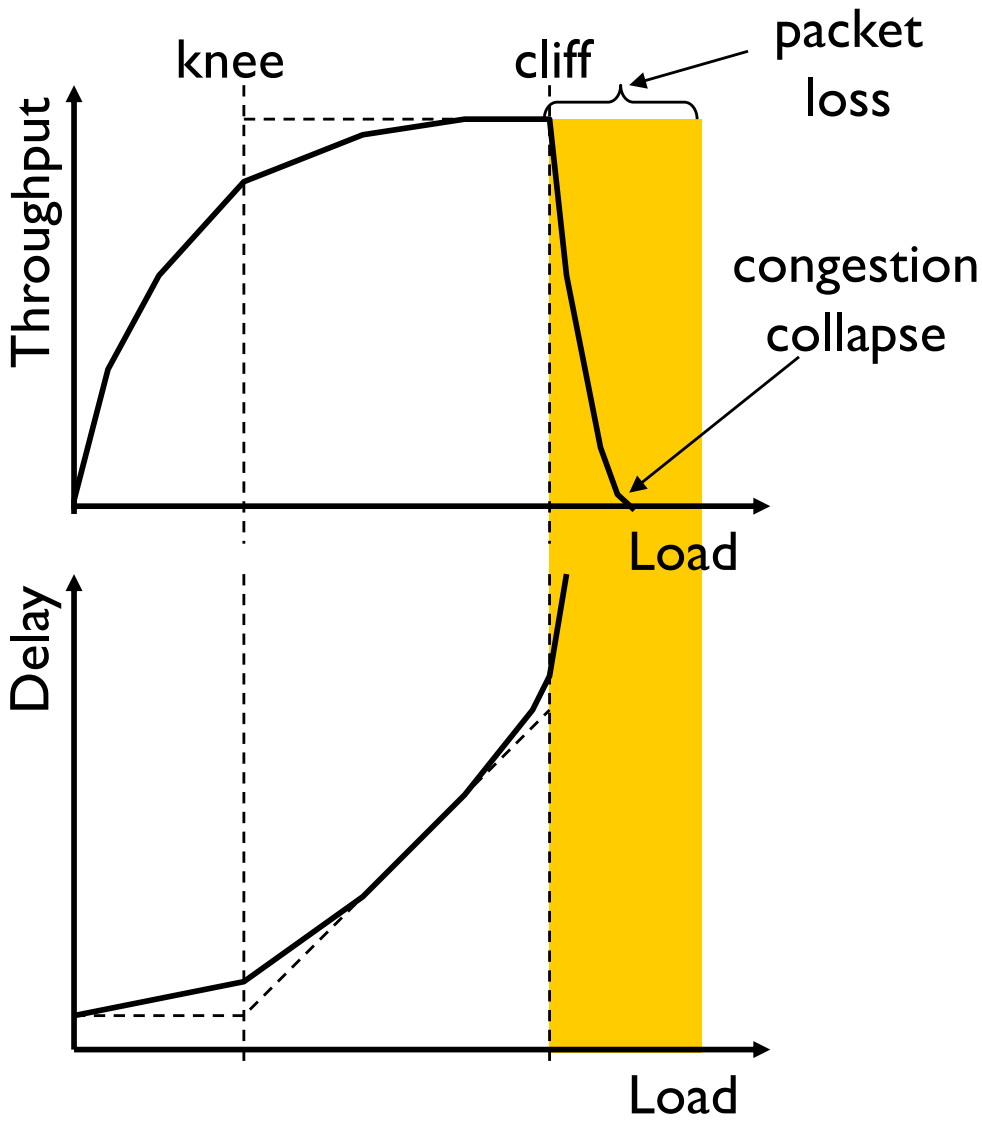
Without congestion control

congestion:

- Increases delivery latency
  - Variable 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



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

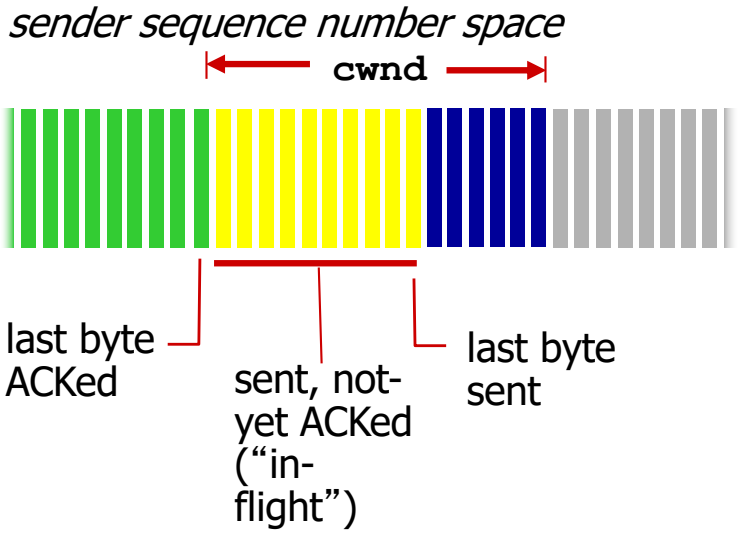
3.7 TCP congestion control

TCP's Approach in a Nutshell

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

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

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



- 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 lecture 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?

Quiz: What is a "congestion event"

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

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?)

www.zeetings.com/salil

Quiz: How should we set CWND?

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

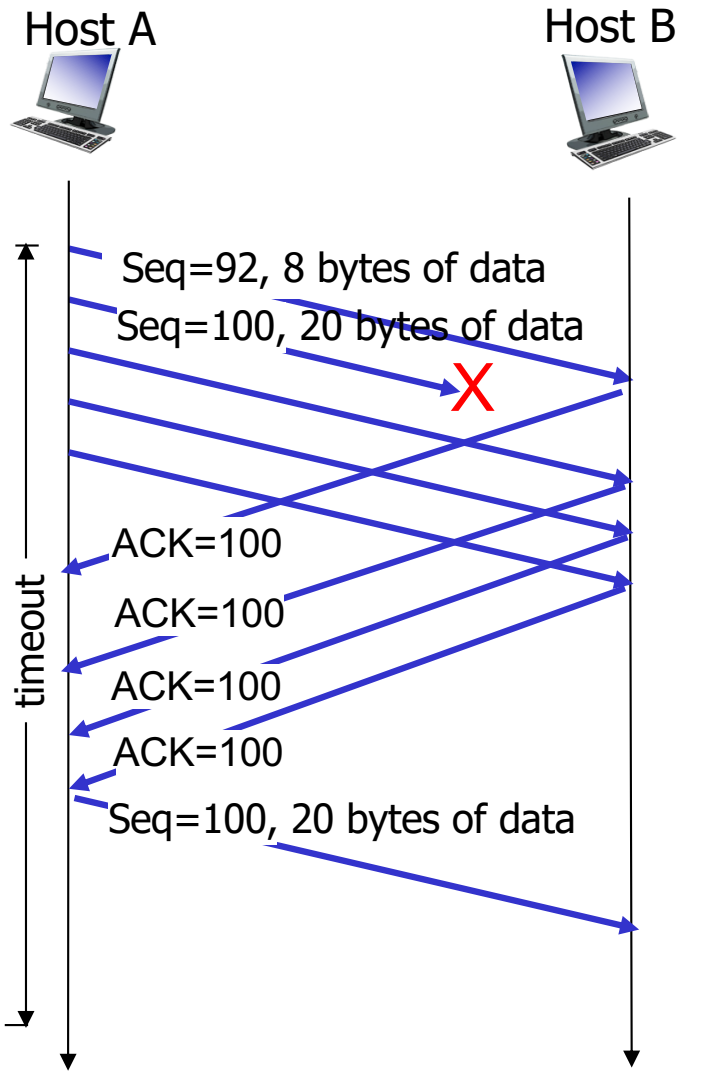
D: We should sent as fast as possible at all times

www.zeetings.com/salil

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

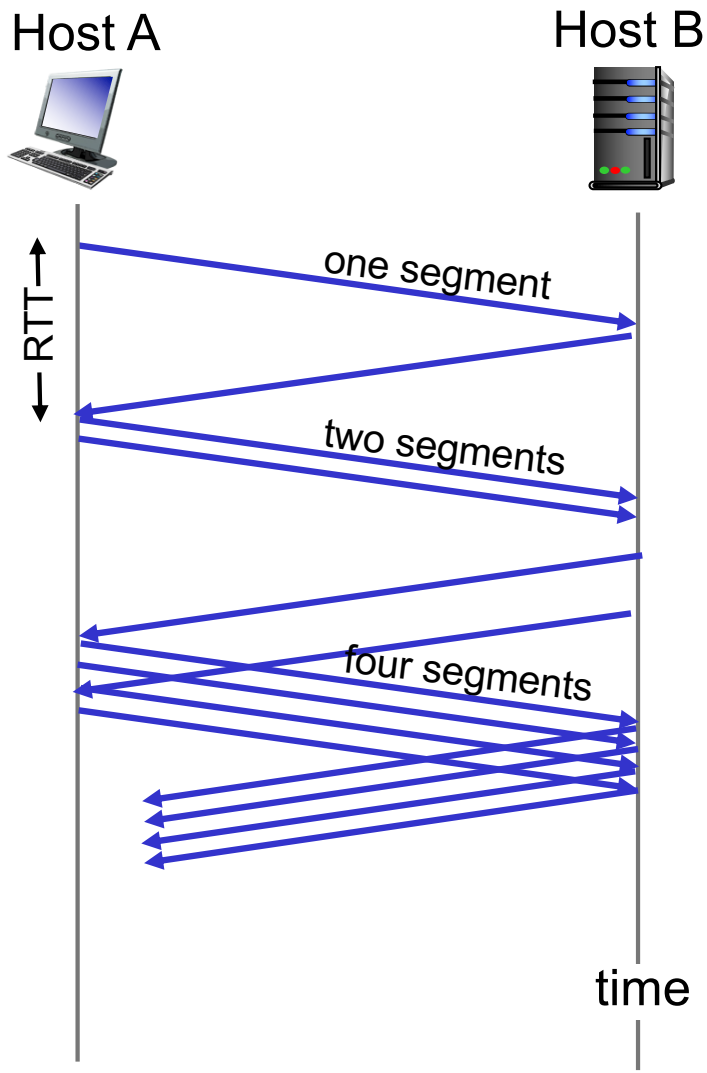
Rate Adjustment

- 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



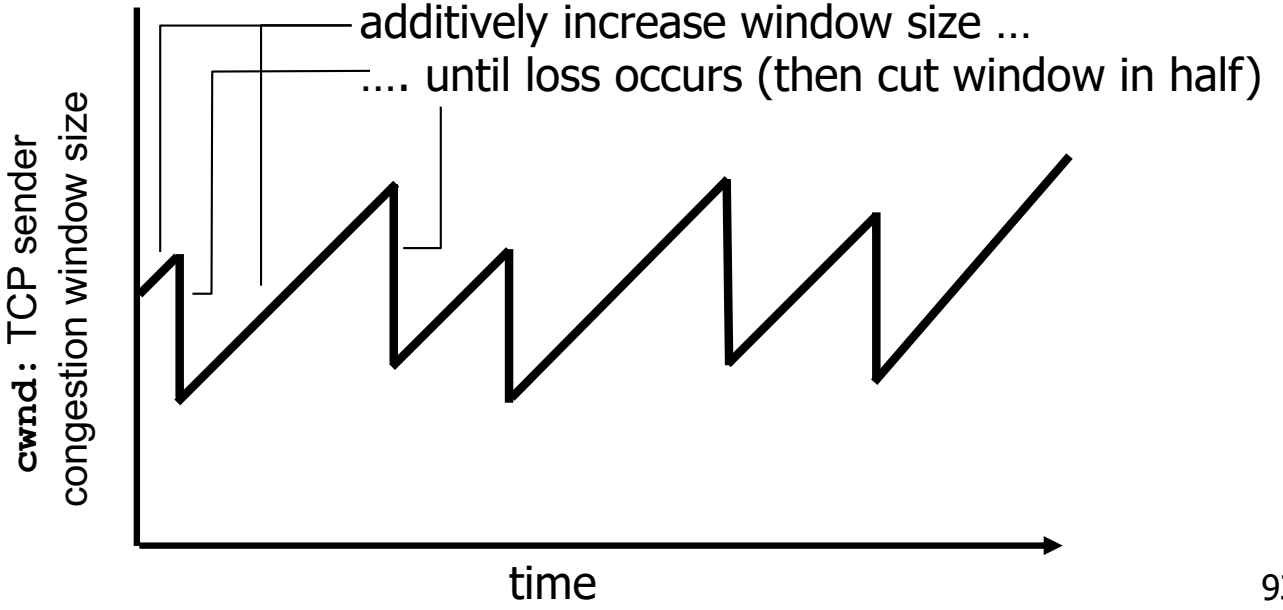
Adjusting to Varying Bandwidth

- 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…

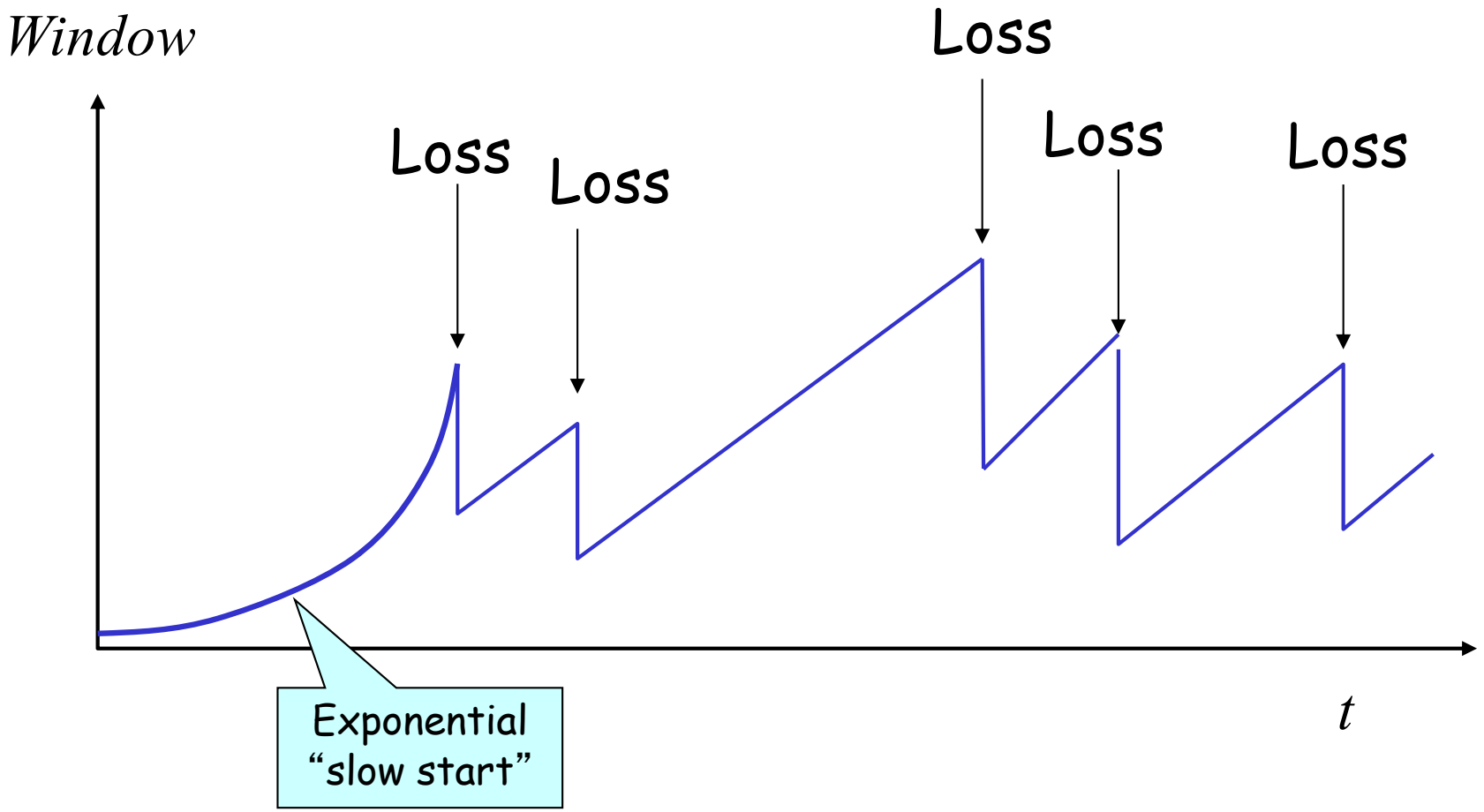
AIMD

- *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

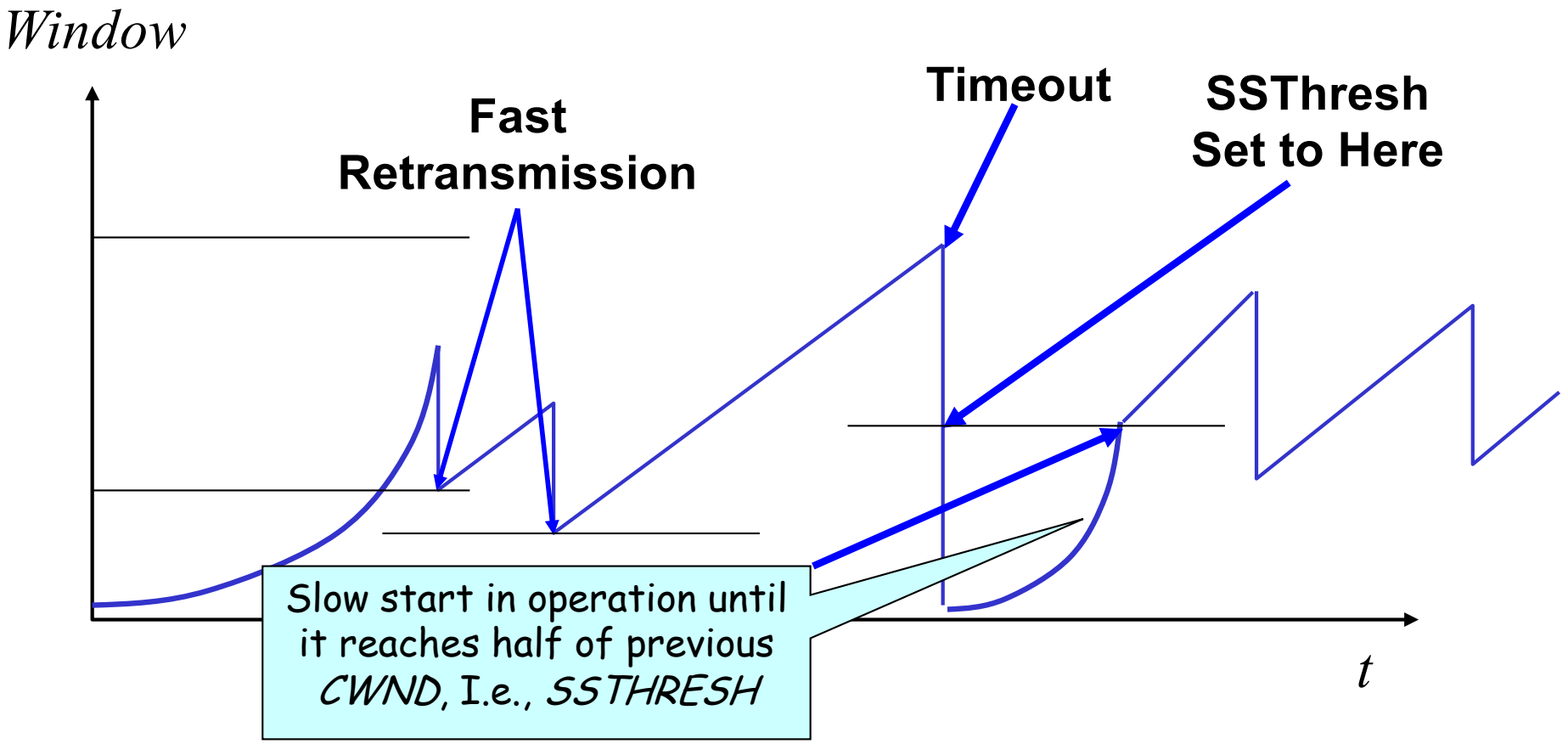
Event: dupACK

- 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

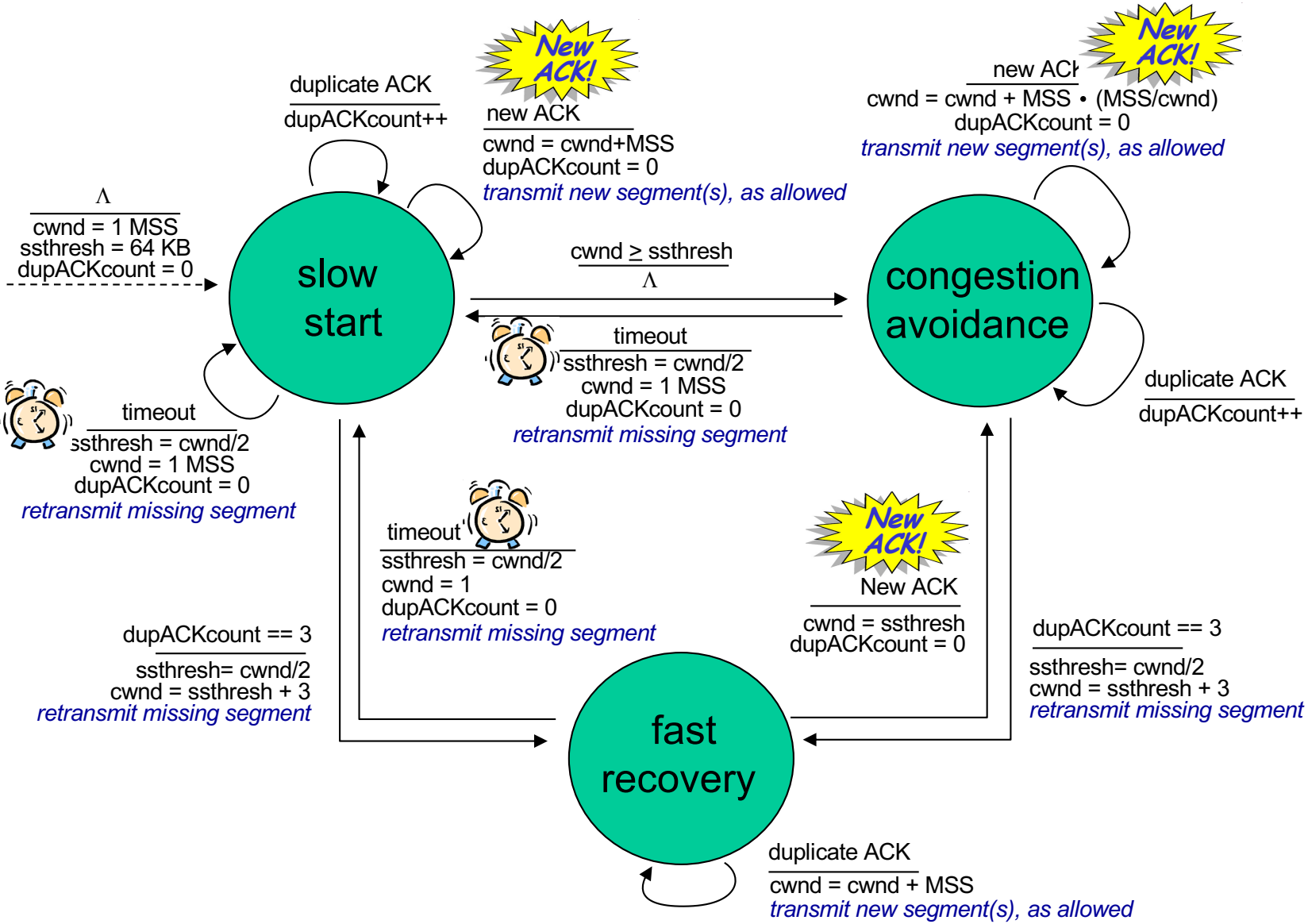
- Consider a TCP connection with:
  - CWND=10 packets (of size MSS = 100 bytes)
  - Last ACK was for byte # 101
    - i.e., receiver expecting next packet to have seq. no. 101

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

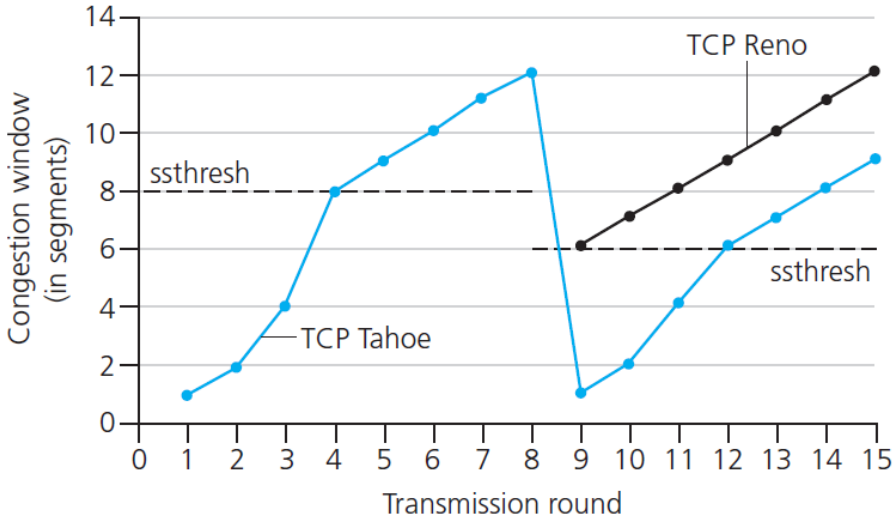
Timeline

- 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



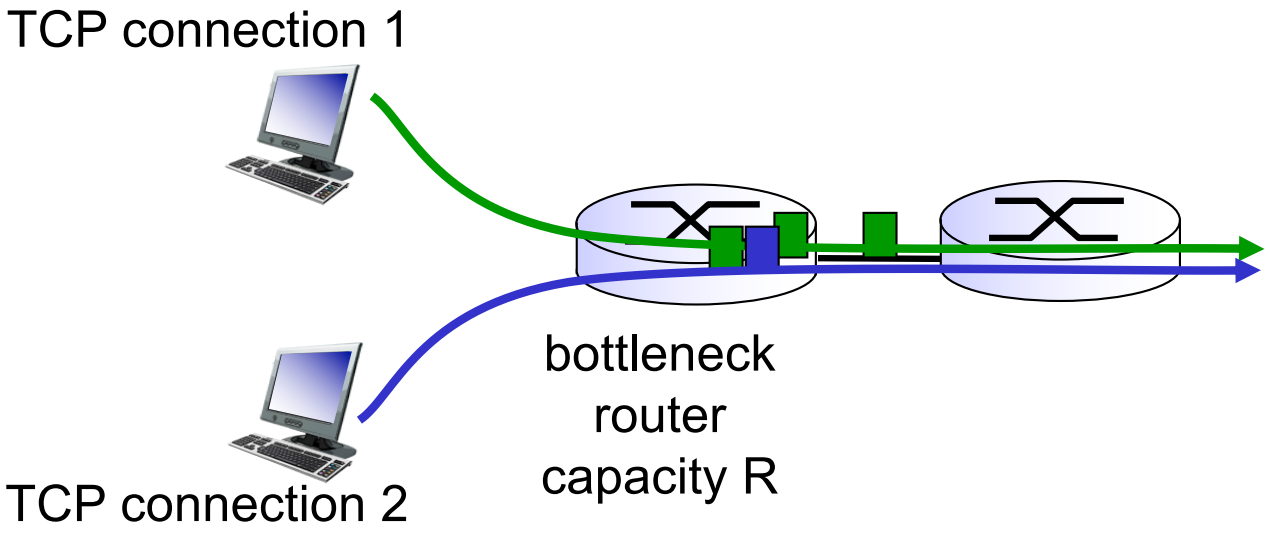
TCP Flavours



- 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



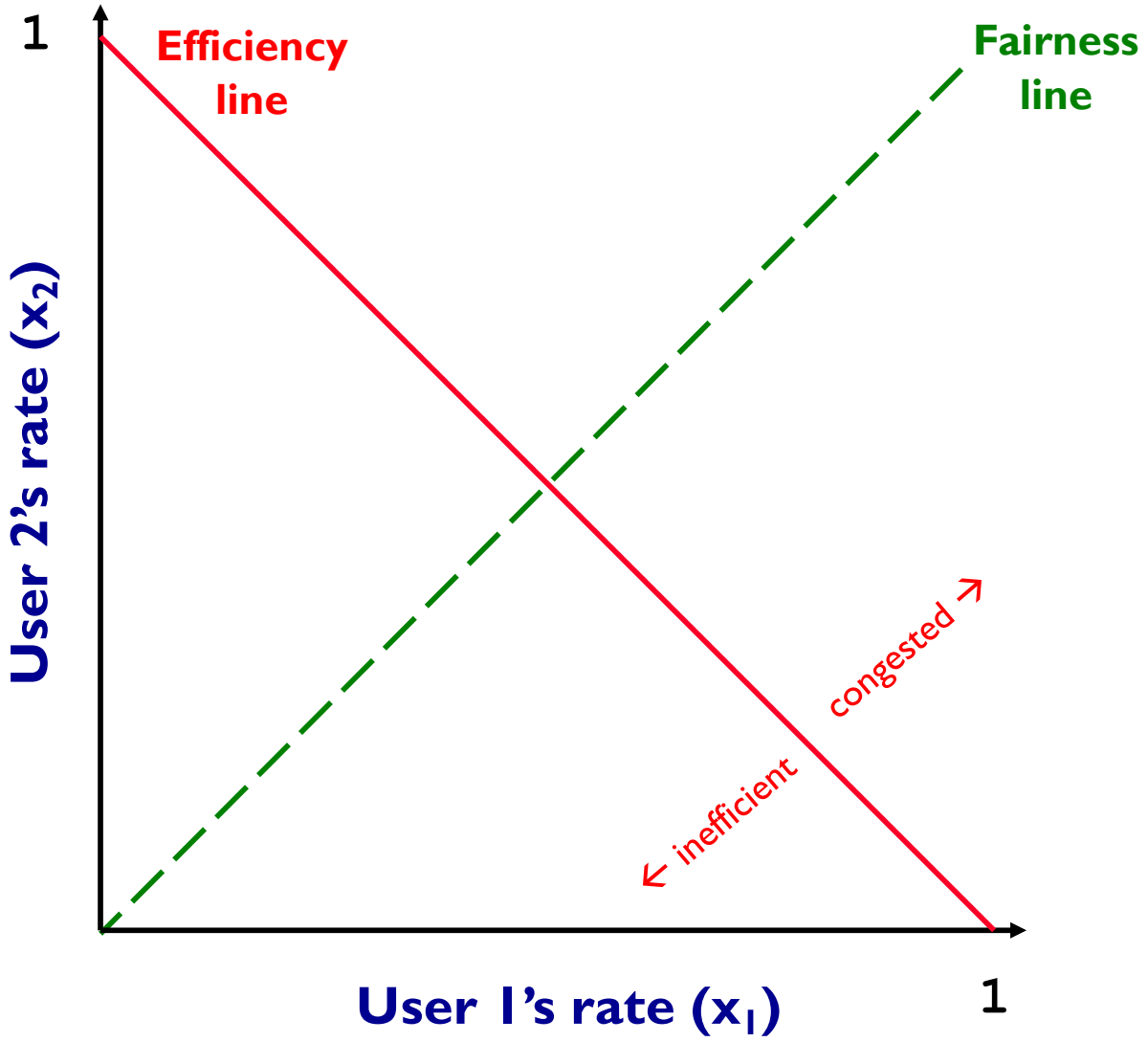
Why AIMD?

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

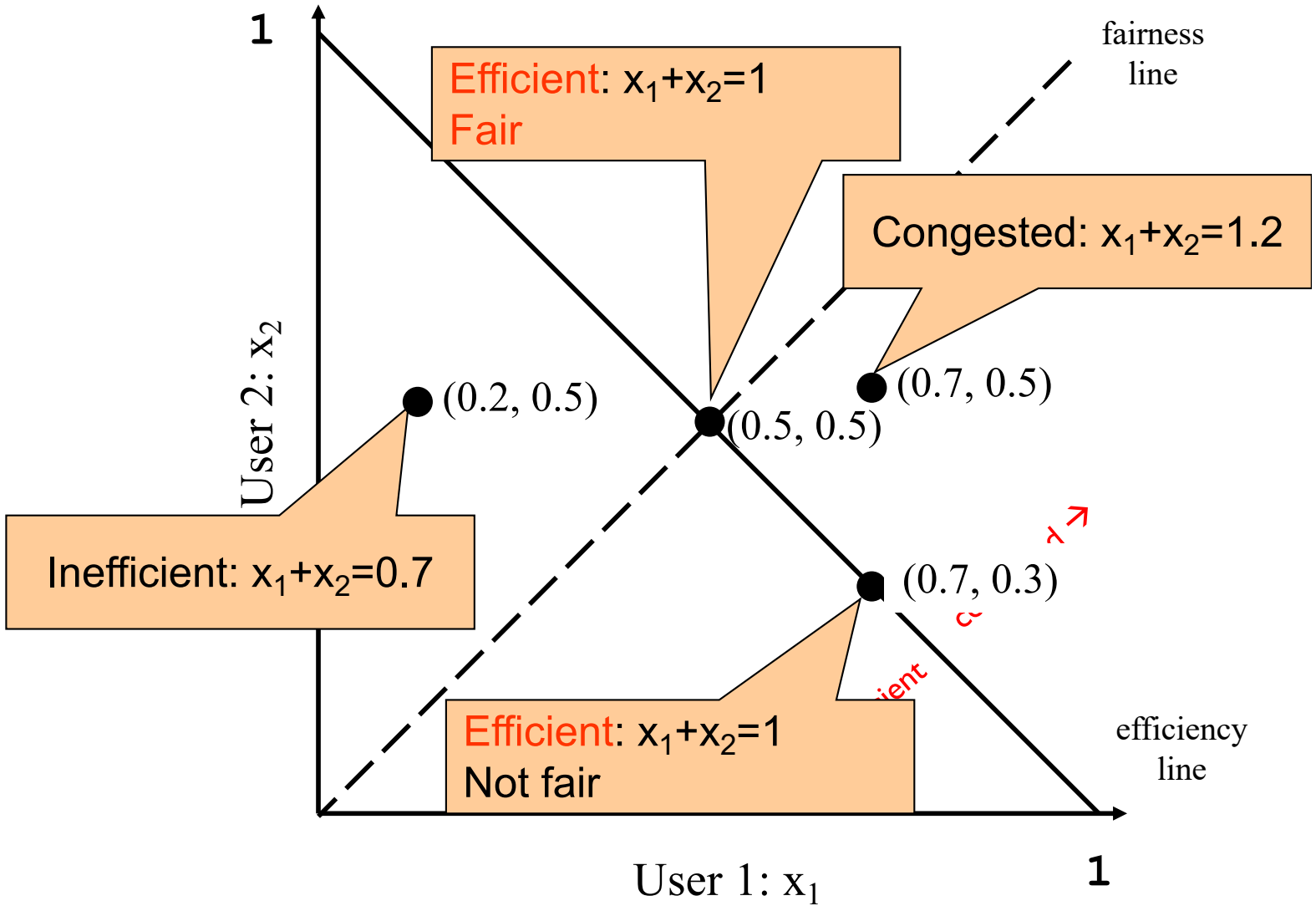
- Four alternatives:
  - AIAD: gentle increase, gentle decrease
  - AIMD: gentle increase, drastic decrease
  - MIAD: drastic increase, gentle decrease
  - MIMD: drastic increase and decrease

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 $x_1 = x_2$

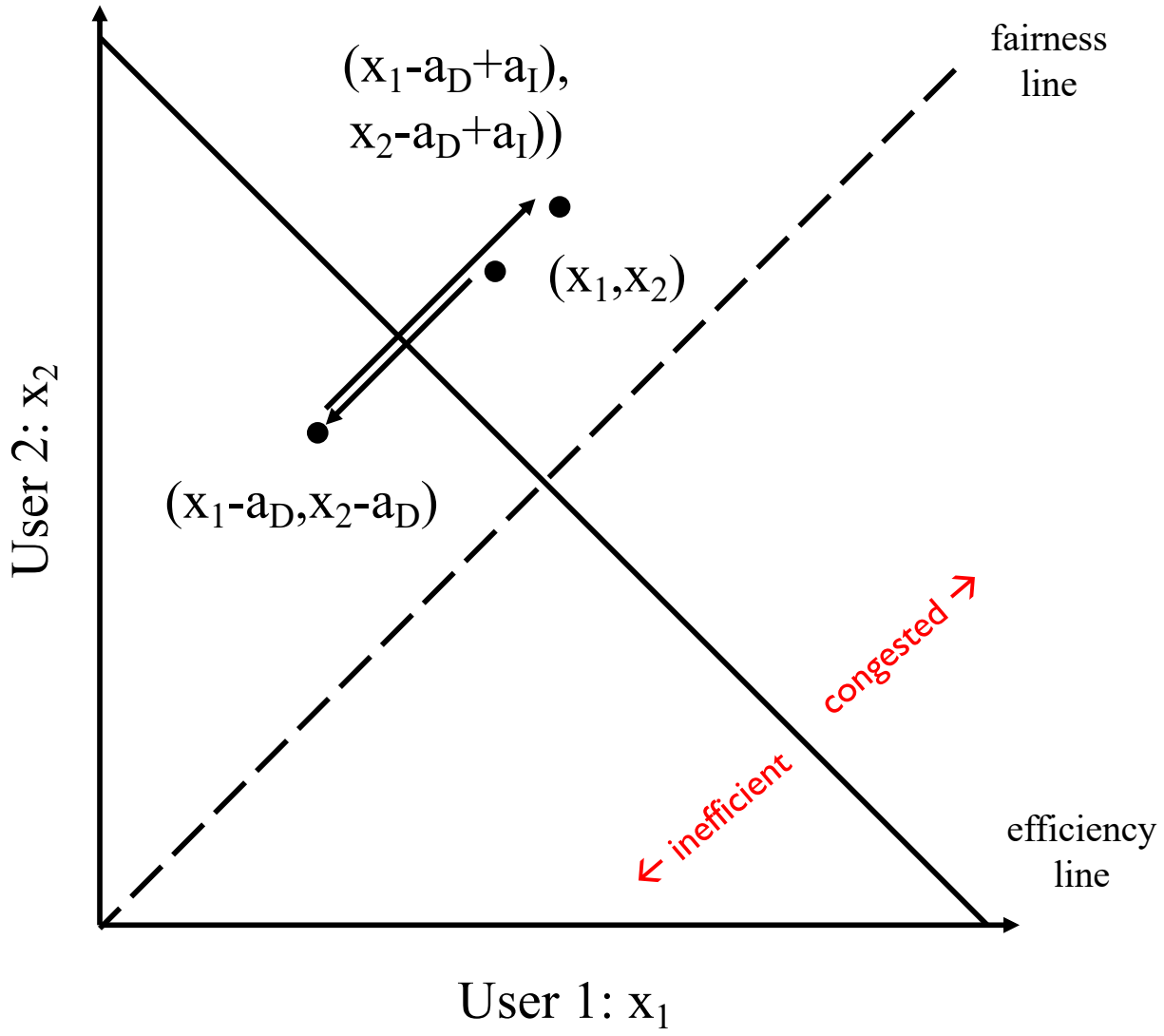


Example

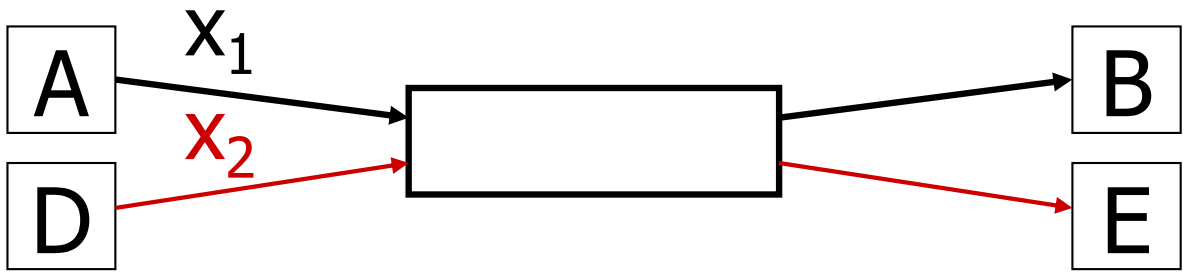

1
fairness line
Efficient: $x_1+x_2=1$
Fair
Congested: $x_1+x_2=1.2$
User 2: x_2
(0.2, 0.5)
(0.5, 0.5)
(0.7, 0.5)
Inefficient: $x_1+x_2=0.7$
(0.7, 0.3)
Efficient: $x_1+x_2=1$
Not fair
efficiency line
User 1: x_1
1

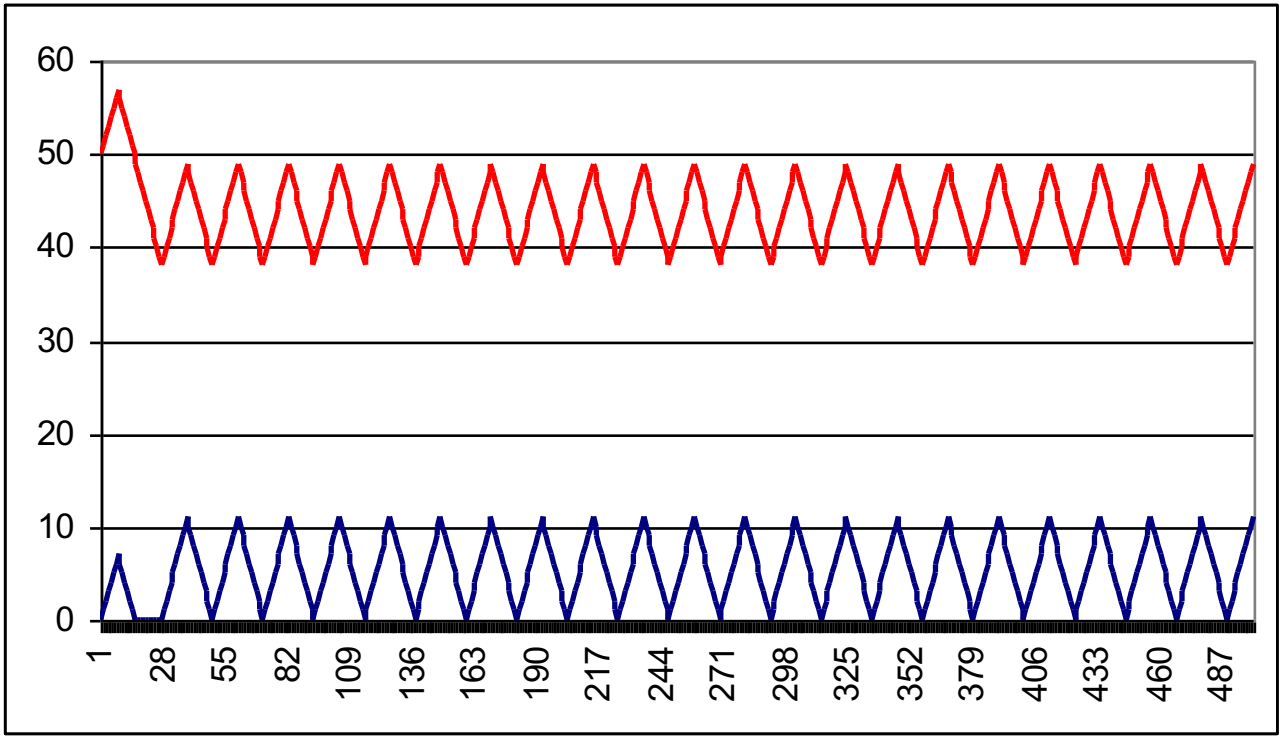
AIAD

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



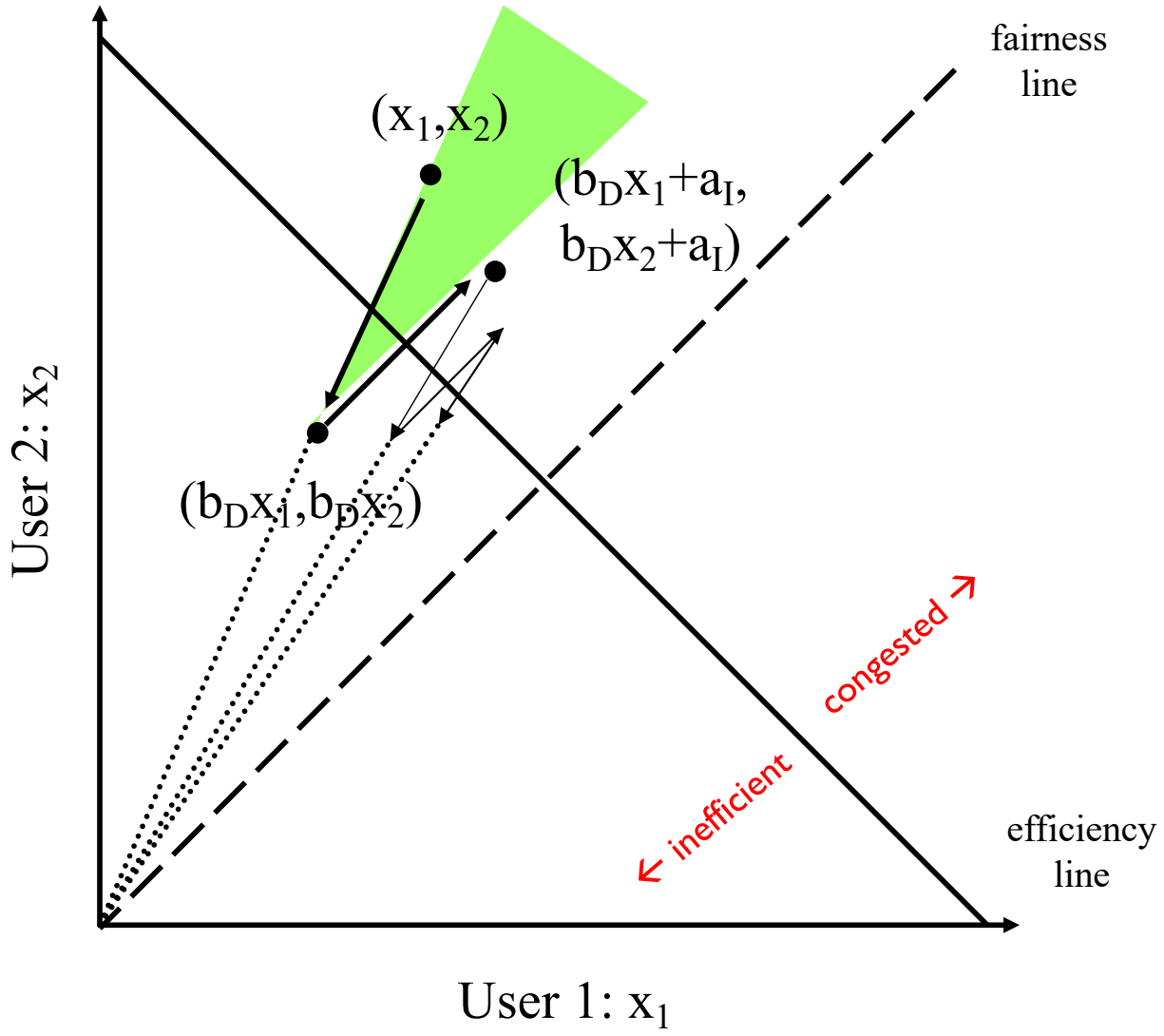
AIAD Sharing Dynamics



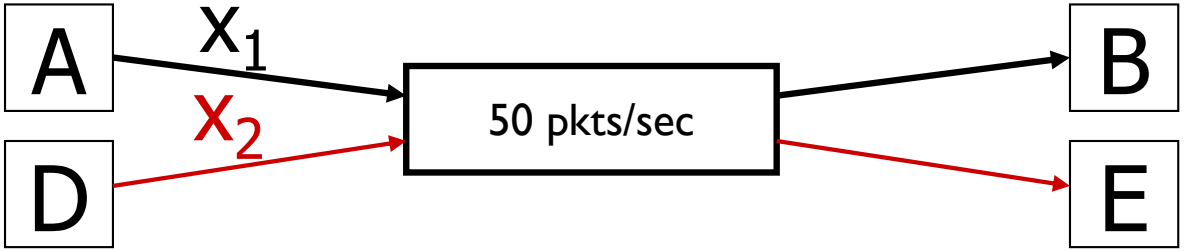


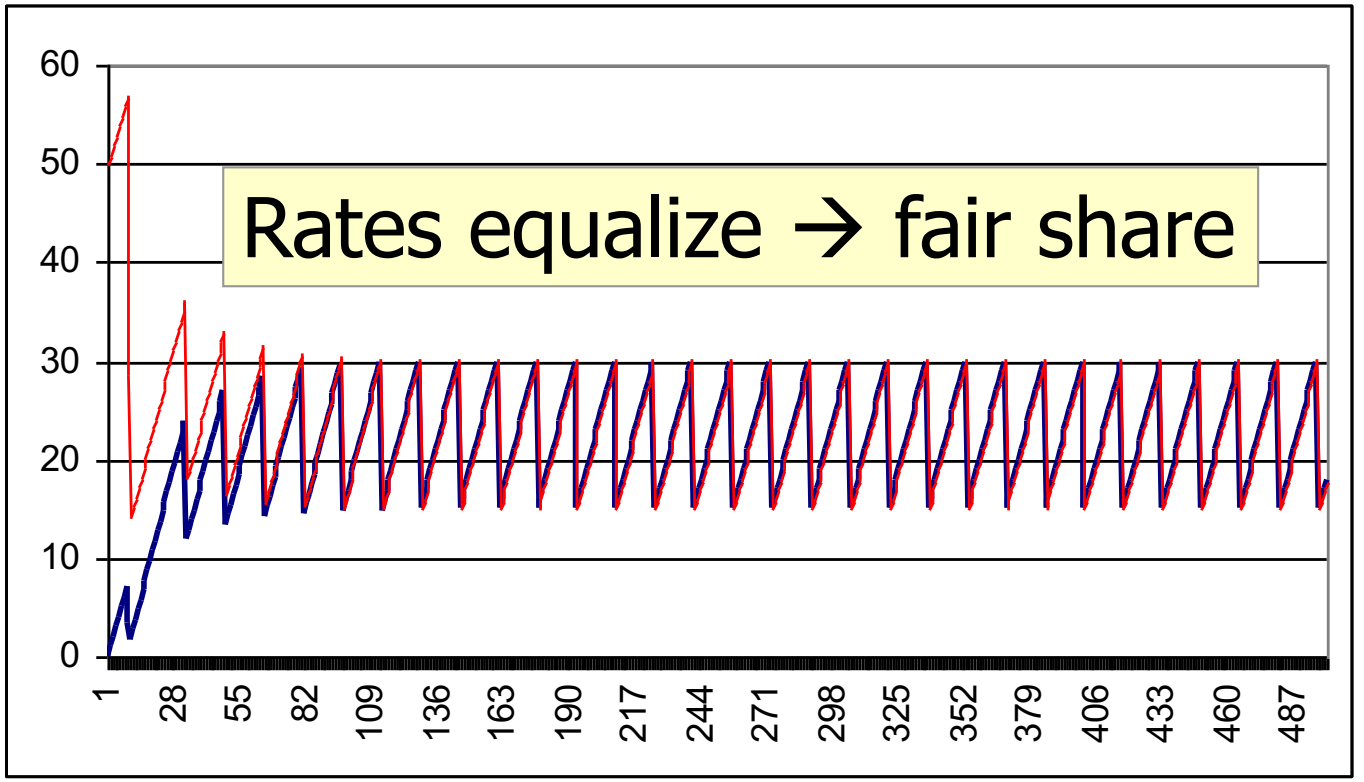
AIMD

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

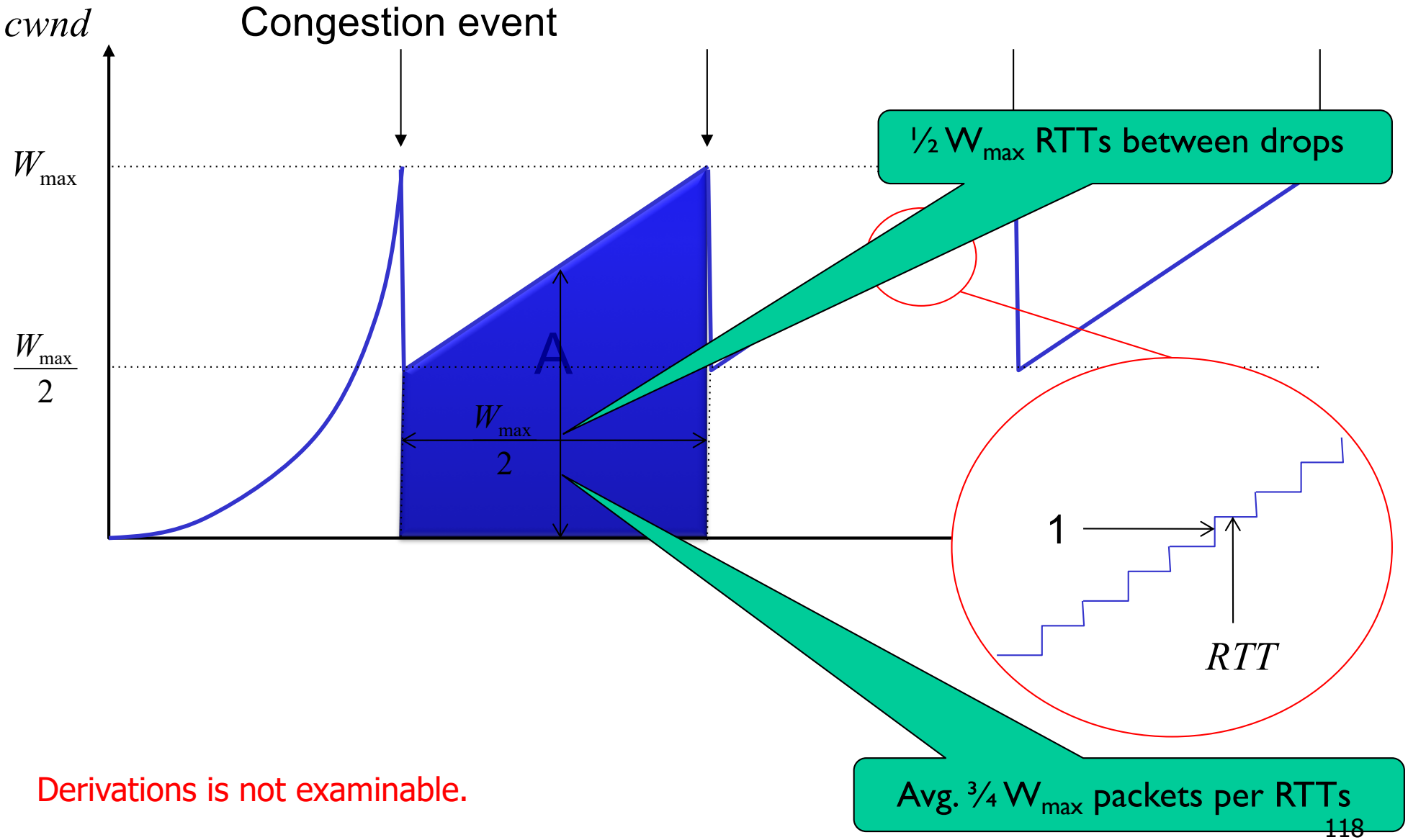


AIMD Sharing Dynamics



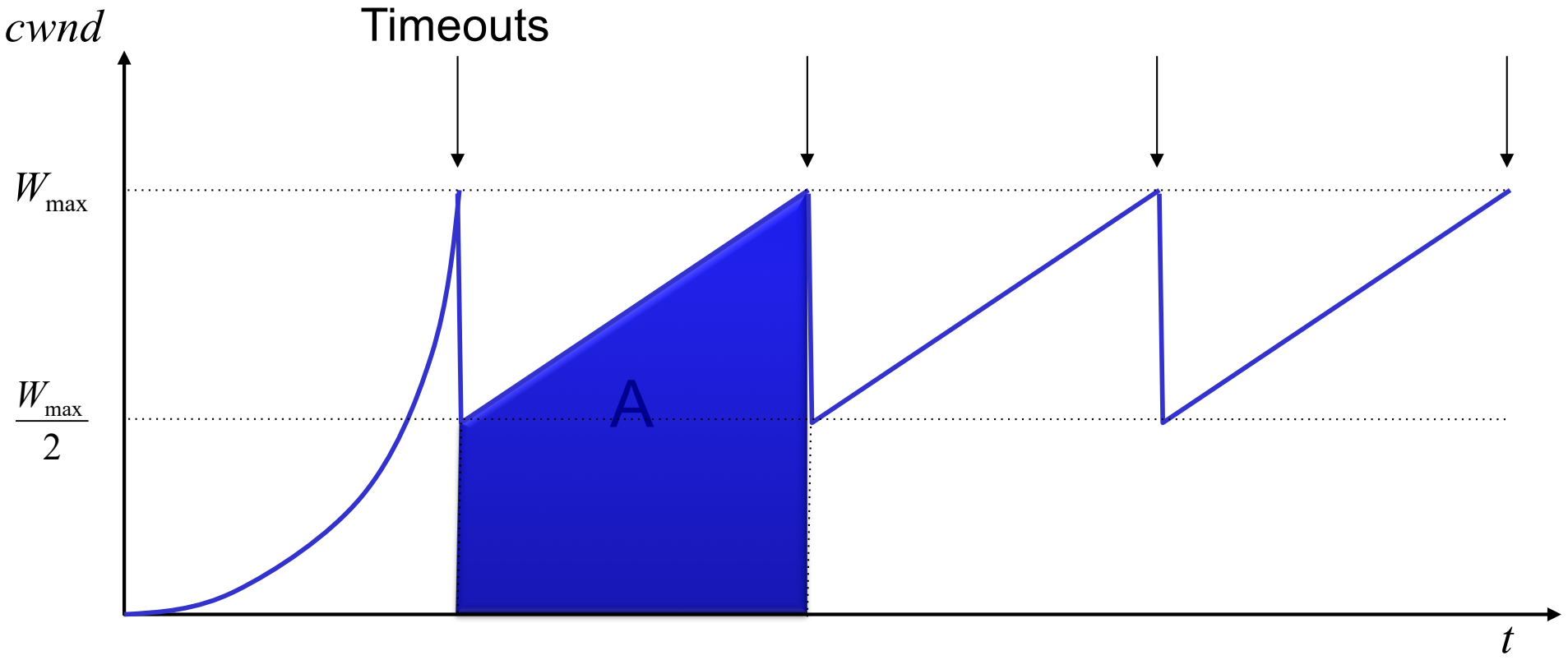


A Simple Model for TCP Throughput



Derivations is not examinable.

A Simple Model for TCP Throughput



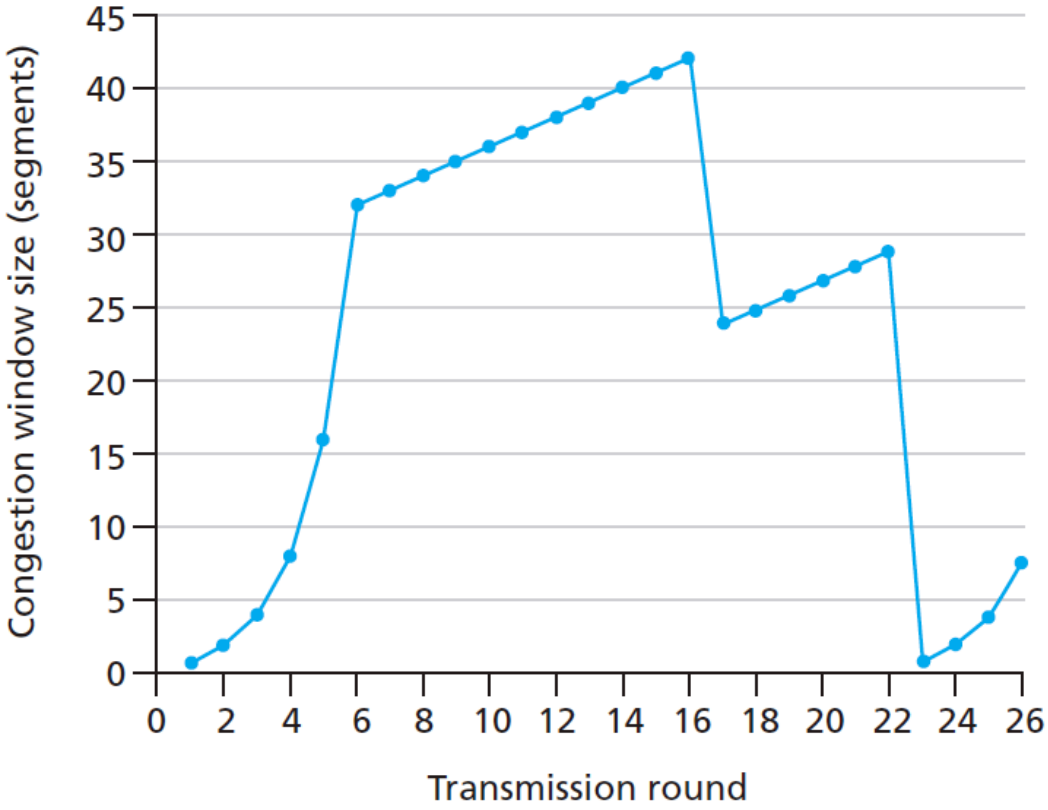
Packet drop rate, $p = 1/A$, where $A = \frac{3}{8} W_{max}^2$

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

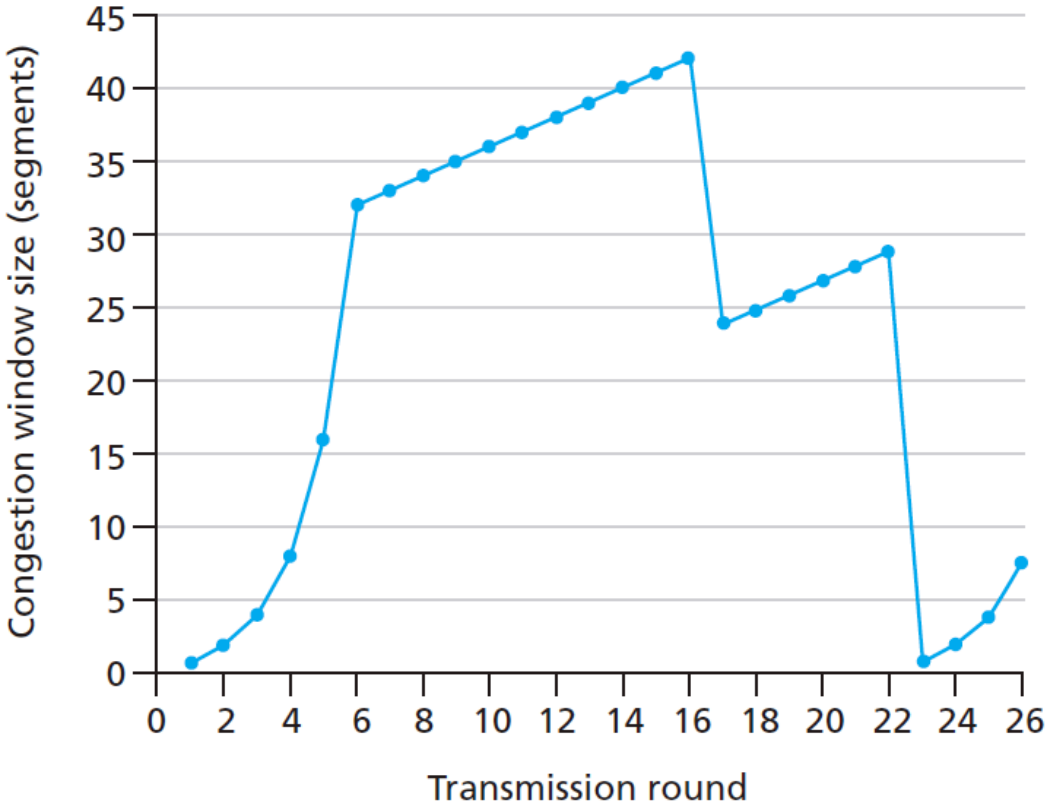


www.zeetings.com/salil

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

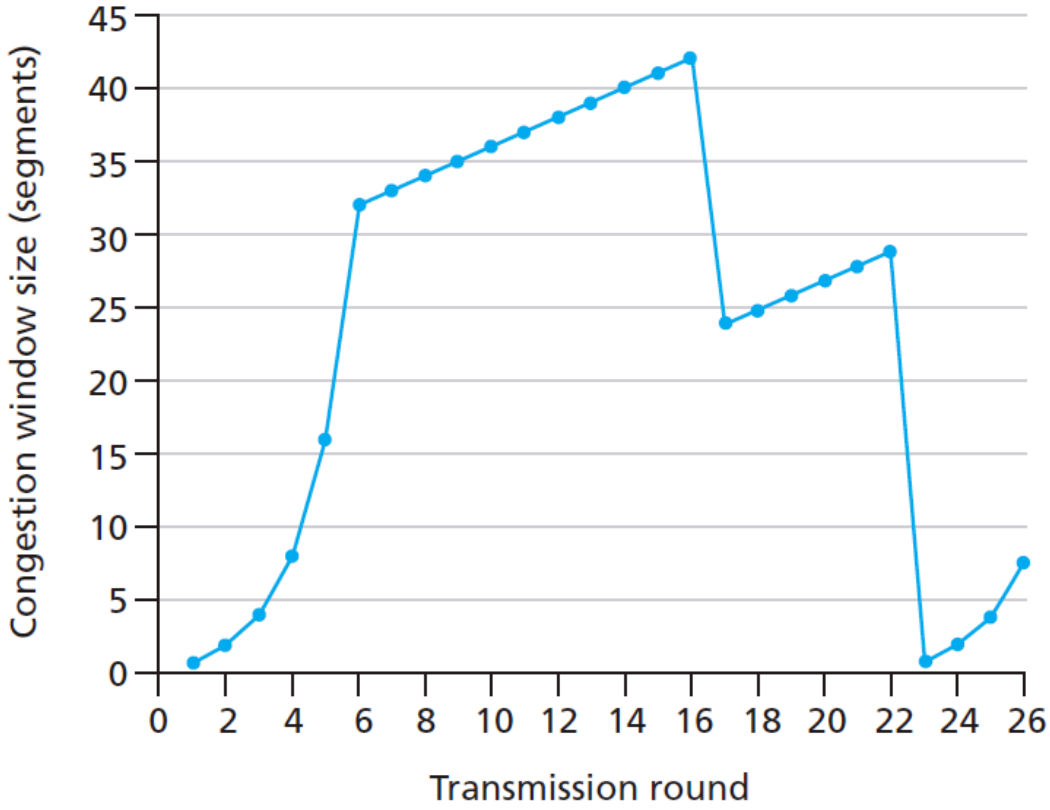


www.zeetings.com/salil

Quiz: TCP Congestion Control?

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

A. Triple Dup Ack
B. Timeout

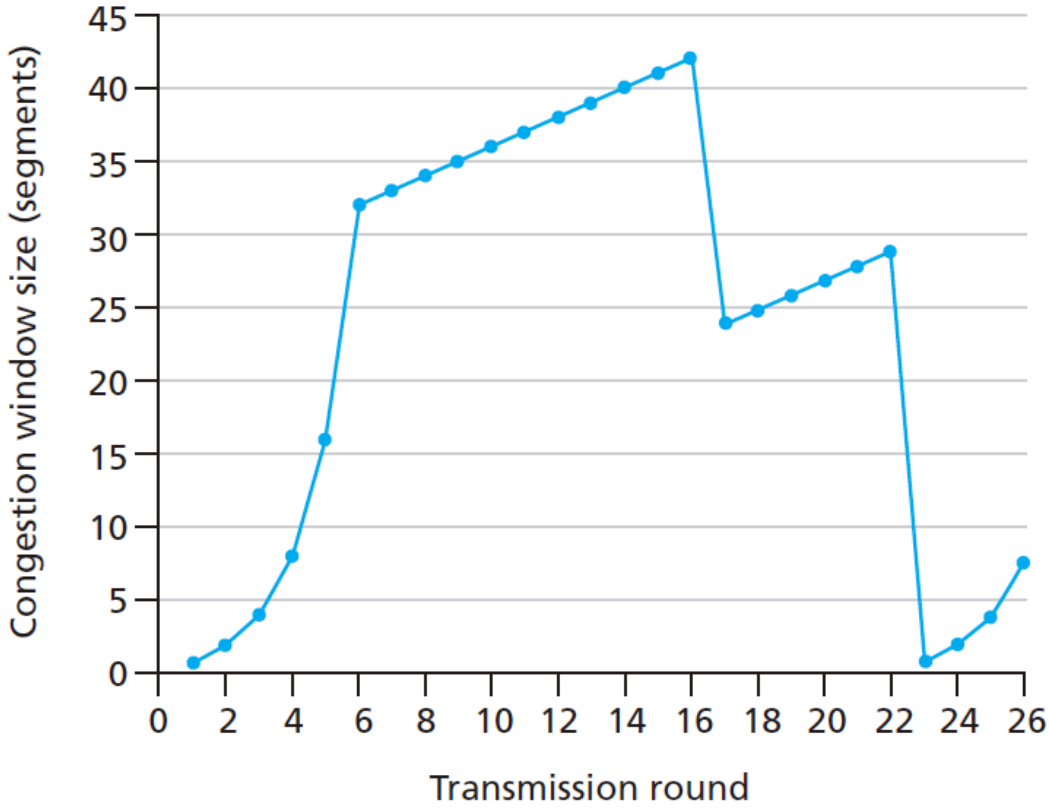


www.zeetings.com/salil

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

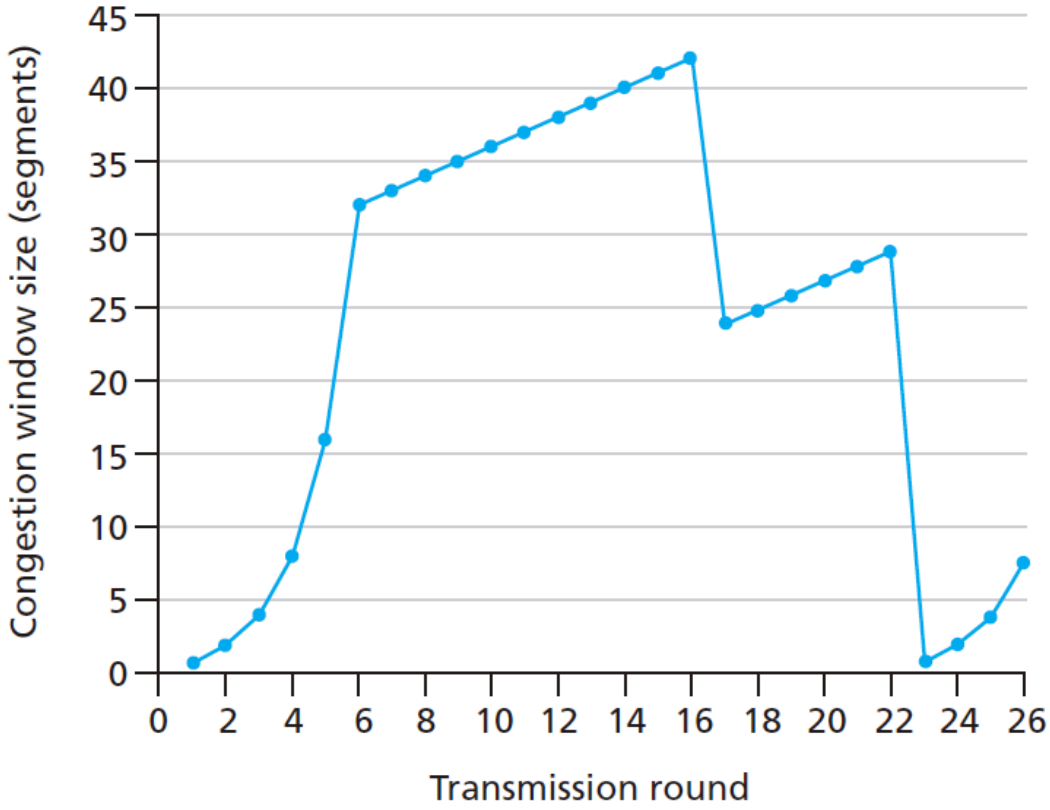


www.zeetings.com/salil

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
E. 20



www.zeetings.com/salil

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"