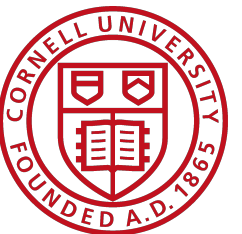


CS4450

Computer Networks: Architecture and Protocols

Lecture 21 TCP and Congestion Control

Rachit Agarwal



Announcements

- Problem set 6 has been posted (last one)
- I am going through a bit of a busy period
 - When the system load is high, queueing delays increase :-)
 - I am significantly behind in my emails
 - I am hoping to catch up this coming weekend
 - Please email cs4450questions@gmail.com

Recap: Four Goals for Reliable Transfer

- **Correctness**
 - As defined
- **“Fairness”**
 - Every flow must get a fair share of network resources
- **Flow Performance**
 - Latency, jitter, etc.
- **Utilization**
 - Would like to maximize bandwidth utilization
 - If network has bandwidth available, flows should be able to use it!

Recap: Complete Correctness Condition

A transport mechanism is “reliable” if and only if

- (a) It resends all dropped or corrupted packets
- (b) It attempts to make progress

Recap: WHYs behind TCP design

- Started from first principles
 - Correctness condition for reliable transport
- ... to understanding **why feedback from receiver is necessary** (sol-v1)
- ... to understanding **why timers may be needed** (sol-v2)
- ... to understanding **why window-based design may be needed** (sol-v3)
 - to understanding the “right” window size for a single-flow case
- ... to understanding **why cumulative ACKs may be a good idea**
 - Very close to modern TCP

Recap: Various acknowledgment mechanisms

- Per-packet ACKs
- Negative ACKs (NACKs)
- Full-information feedback (also referred to as SACK)
- Cumulative ACKs
- Each of these vary in terms of:
 - Nature of feedback
 - Loss detection
 - Response to loss

Any Questions?

Lets learn TCP

Transport layer

- Transport layer offer a “pipe” abstraction to applications
- Data goes in one end of the pipe and emerges from other
- **Pipes are between processes, not hosts**
- There are two basic pipe abstractions

Two Pipe Abstractions

- **Unreliable packet** delivery (UDP)
 - Unreliable (application responsible for resending)
 - Messages limited to single packet
- **Reliable byte stream** delivery
 - Bytes inserted into pipe by sender
 - They emerge, in order at receiver (to the app)
- What features must transport protocol implement to support these abstractions?

UDP (Datagram Messaging Service)

- Sources send packets
- **Destinations do nothing**, but receive packets
- If packets delayed/reordered/lost:
 - Meh!
 - Let application handle packet loss (or be oblivious to drops)
 - If application needs reliable delivery, it must use reliable transport
- Discarding corrupted packets (optional)
- Nothing else!
- A minimal extension of IP

TCP (Reliable, In Order Delivery)

- Source send **segments**
- Destinations send ACKs
- Source retransmits lost and/or corrupted **segments**
- Sources perform **Flow control** (to not overflow receiver)
- Sources perform **Congestion control** (to not overload network)
- Source and destination participate in “Connection” set-up and tear-down

Connections (Or Sessions)

- Reliability requires keeping state
 - Sender: packets sent but not yet ACKed, and related timers
 - Receiver: packets that arrived out-of-order
- Each byte stream is called a **connection** or **session**
 - Each with their own connection state
 - State is in hosts, not network

Transmission Control Protocol (TCP)

- Reliable, in-order delivery
 - Ensures byte stream (eventually) arrives intact
 - In the presence of corruption, delays, reordering, loss
- Connection oriented
 - Explicit set-up and tear-down of TCP session
- Full duplex stream of **byte service**
 - **Sends and receives stream of bytes, not messages**
- **Flow control**
 - Ensures the sender does not overwhelm the receiver
- **Congestion control**
 - Dynamic adaptation to network path's capacity

From design to implementation: major notation change

- Previously we focused on packets
 - Packets had numbers
 - ACKs referred to those numbers
 - Window sizes expressed in terms of # of packets
- **TCP focuses on bytes, thus**
 - Packets identified by the bytes they carry
 - **ACKs refer to the bytes received**
 - Window size expressed in terms of # of bytes

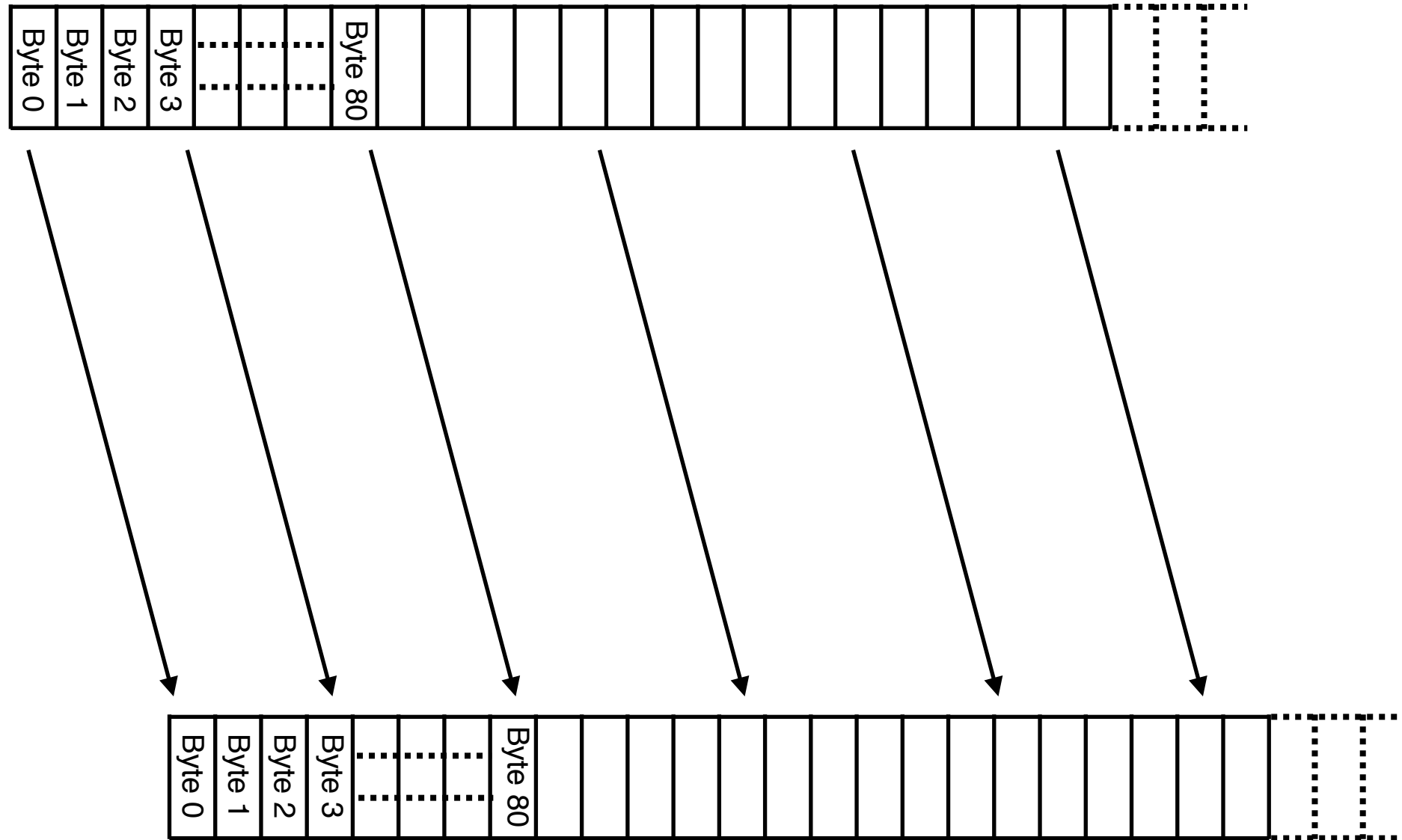
Basic Components of TCP

- **Segments, Sequence numbers, ACKs**
 - TCP uses byte sequence numbers to identify payloads
 - ACKs referred to sequence numbers
 - Window sizes expressed in terms of # of bytes
- **Retransmissions**
 - Can't be correct without retransmitting lost/corrupted data
 - TCP retransmits based on timeouts and duplicate ACKs
 - Timeouts based on estimate of RTT
- **Flow Control**
- **Congestion Control**

Segments, Sequence Numbers and ACKs

TCP "Stream of Bytes" Service

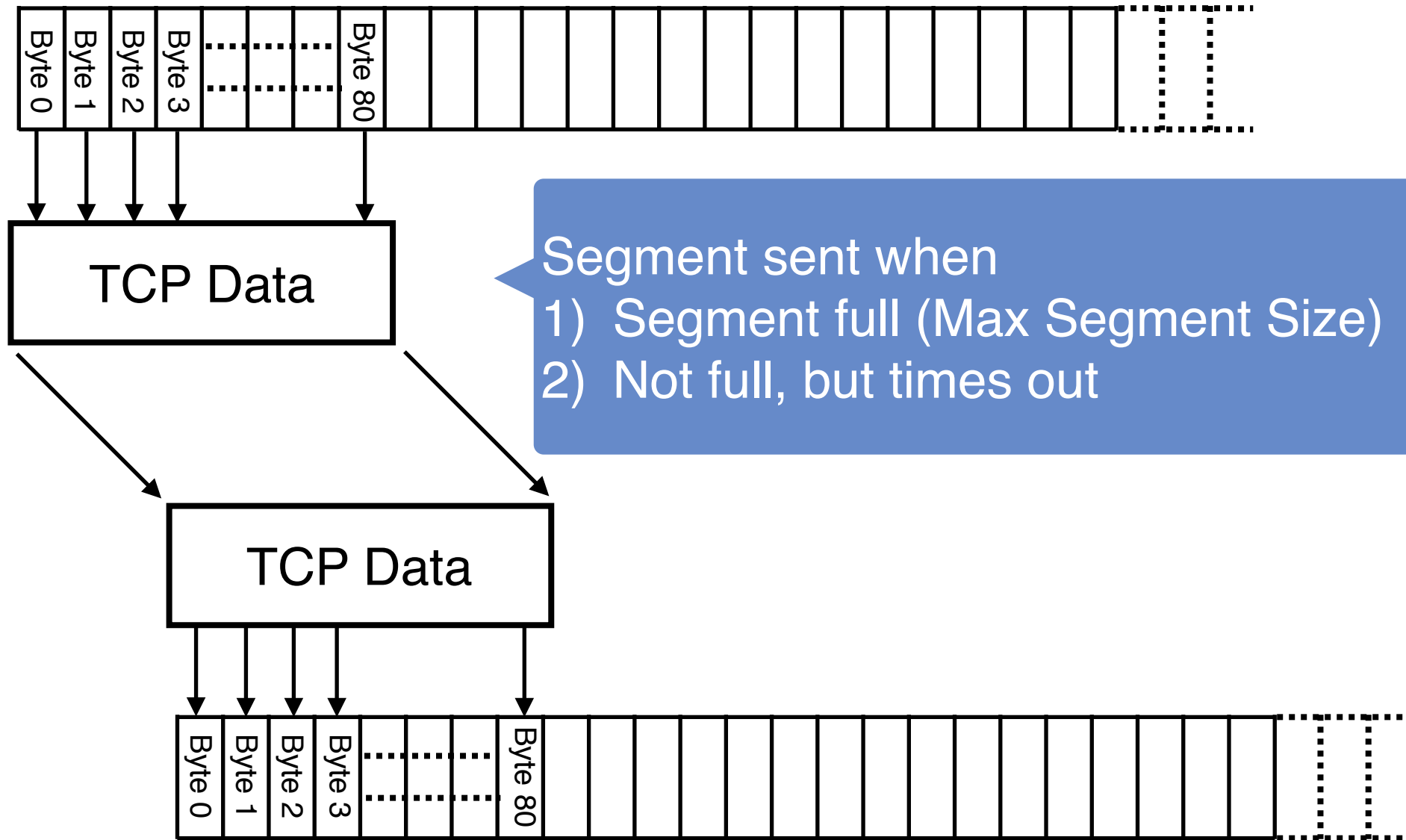
Application @ Host A



Application @ Host B

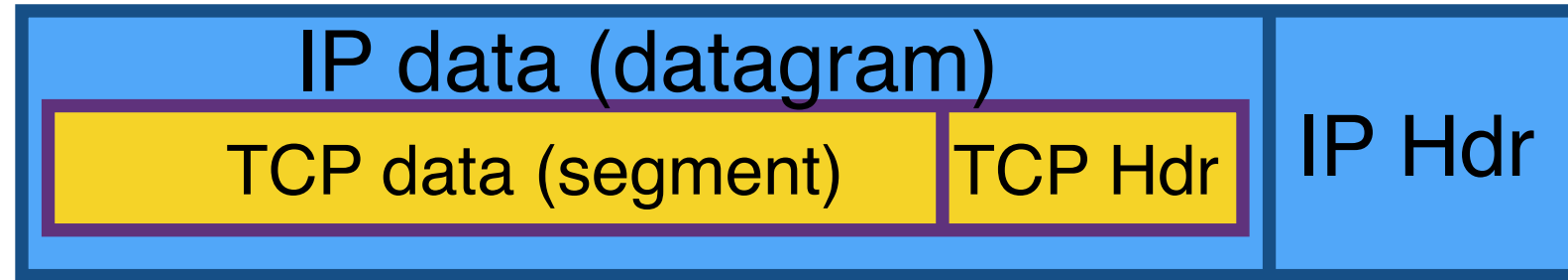
TCP "Stream of Bytes" Service

Application @ Host A



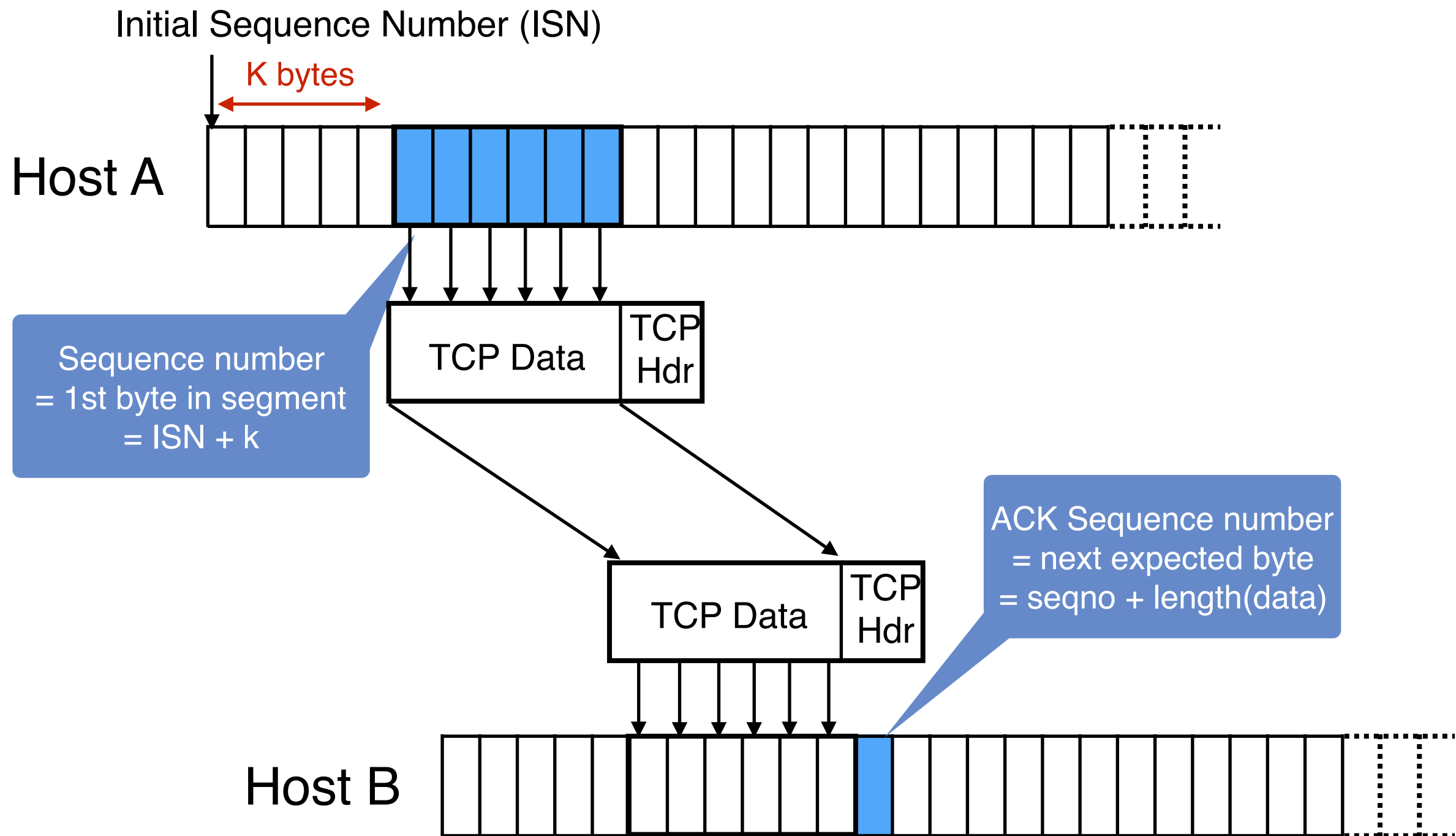
Application @ Host B

TCP Segment



- 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 \geq 20 bytes long
- TCP Segment
 - No more than MSS (Maximum Segment Size) bytes
 - E.g., up to 1460 consecutive bytes from the stream
 - $MSS = MTU - IP\ header - TCP\ header$

Sequence Numbers

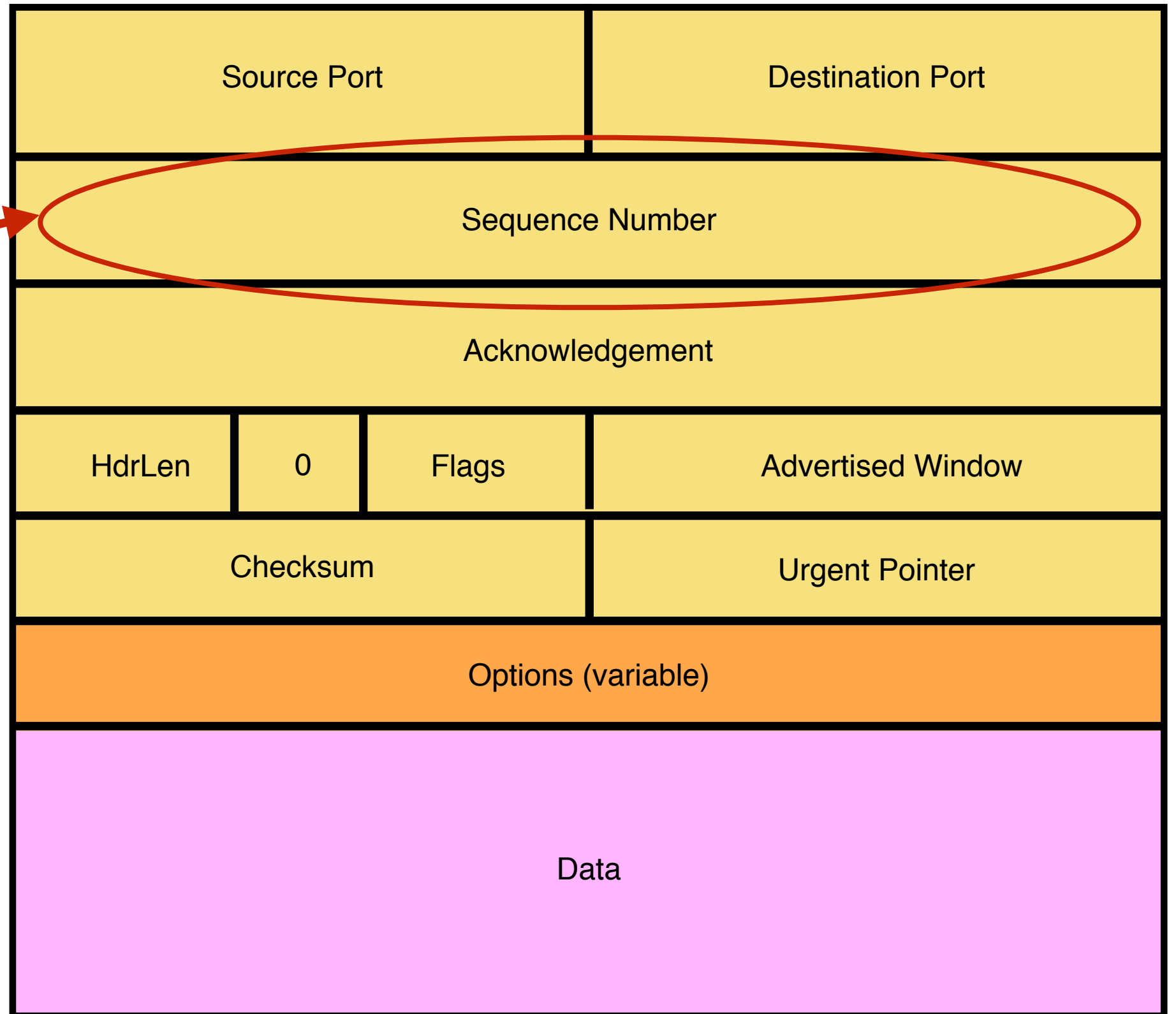


ACKing and Sequence Numbers

- Sender sends segments (byte stream)
 - Data starts with Initial Sequence Number (ISN): X
 - Packet contains B bytes
 - $X, X+1, X+2, \dots, X+B-1$
- Upon receipt of a segment, receiver sends an ACK
 - If all data prior to X already received:
 - ACK acknowledges $X+B$ (because that is next expected byte)
 - If highest contiguous byte received is smaller value Y
 - ACK acknowledges $Y+1$
 - Even if this has been ACKed before

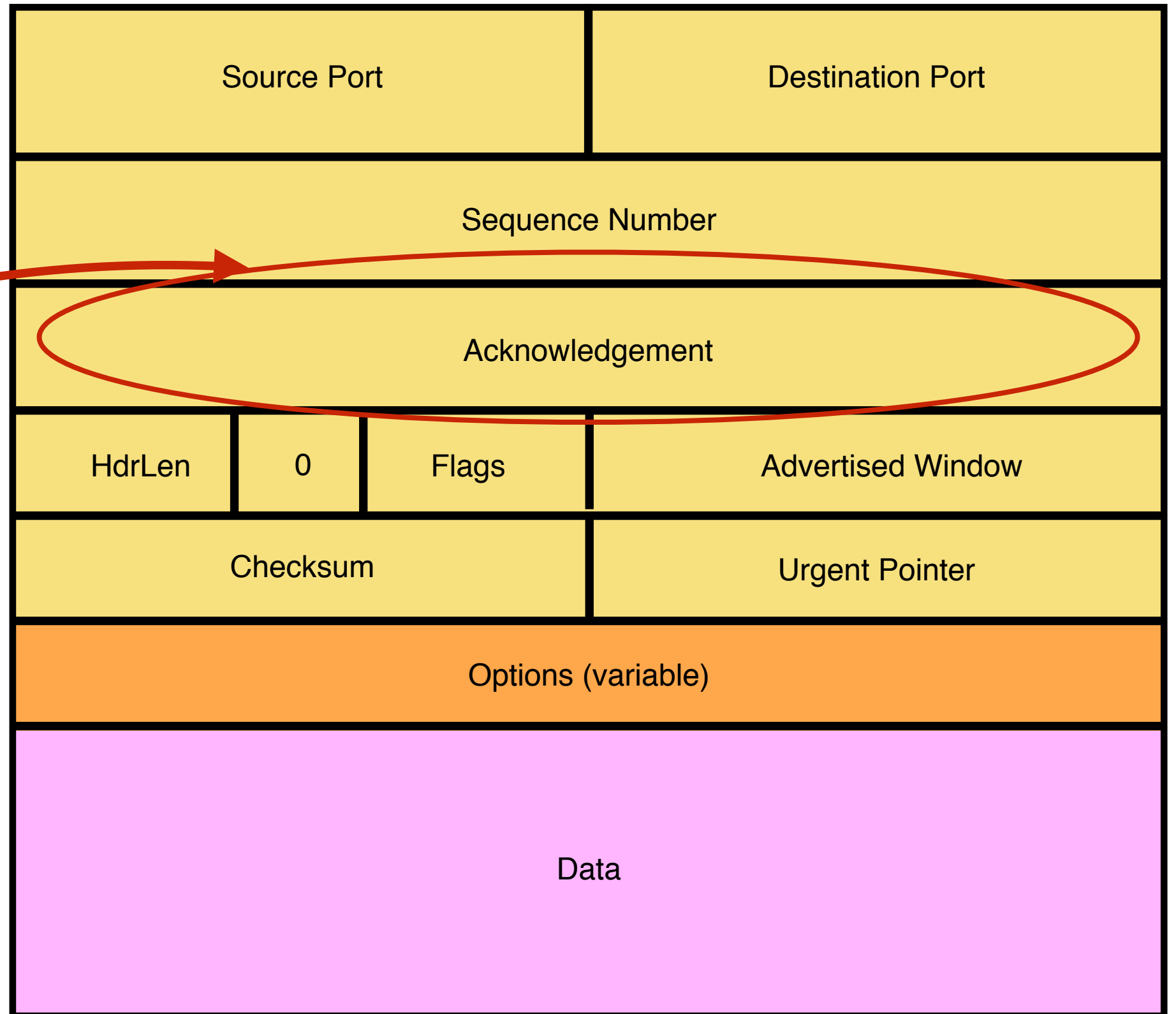
TCP Header

Starting byte offset
of data carried in
this segment



TCP Header

Acknowledgement gives sequence number just beyond highest sequence number received in order (“What byte is next”)



Any Questions?

TCP Retransmission

Two Mechanisms for Retransmissions

- Duplicate ACKs
- Timeouts

Loss with Cumulative ACKs

- Sender sends packets with 100B and seqnos
 - 100, 200, 300, 400, 500, 600, 700, 800, 900
- Assume 5th packet (seqno 500) is lost, but no others
- Stream of ACKs will be
 - 200, 300, 400, 500, 500, 500, 500, 500

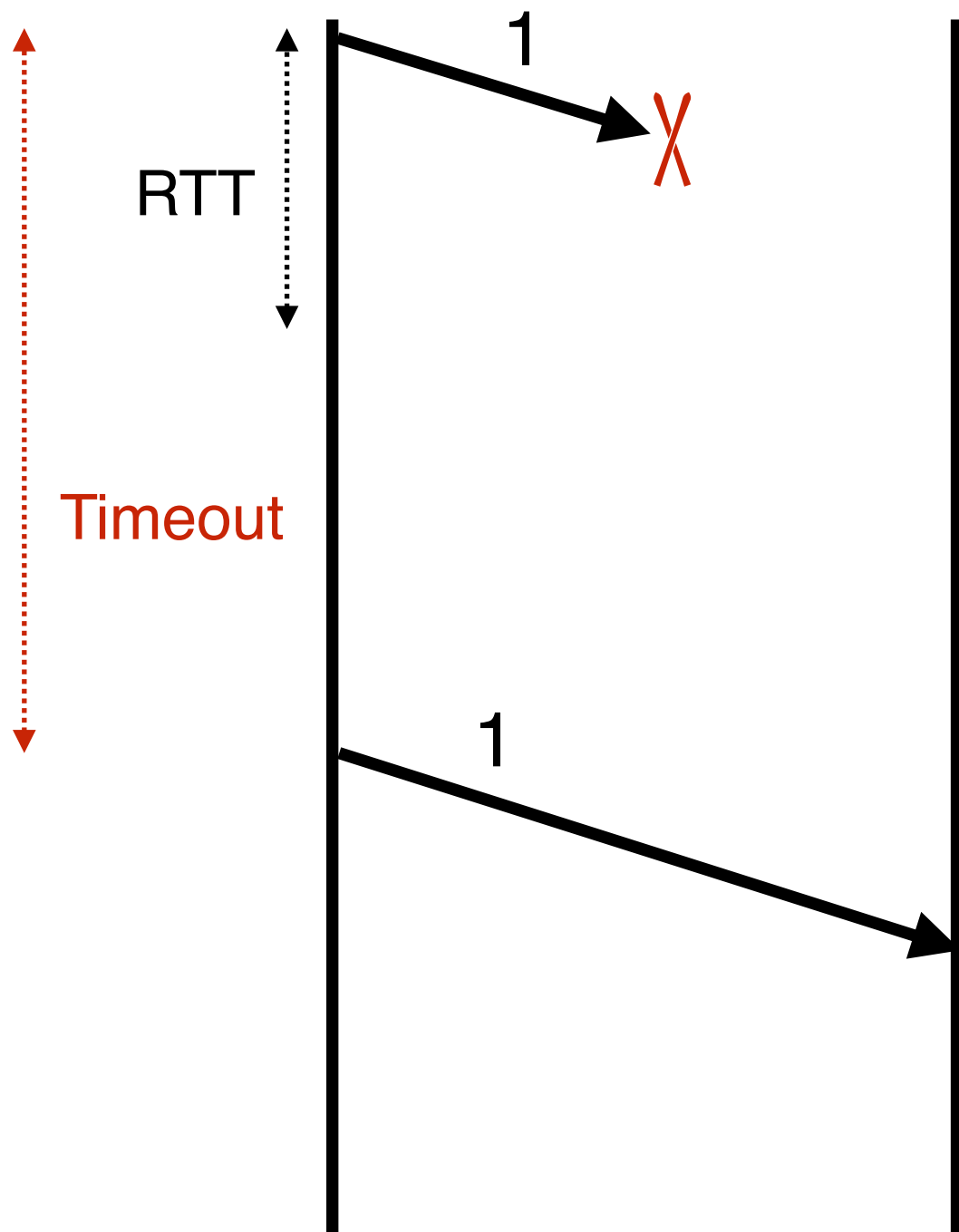
Loss with Cumulative ACKs

- Duplicate ACKs are a sign of an isolated loss
 - The lack of ACK progress means 500 hasn't been delivered
 - Stream of ACKs means some packets are being delivered
- Therefore, could trigger resend upon receiving k duplicate ACKs
 - TCP uses $k = 3$
- We will revisit this in congestion control

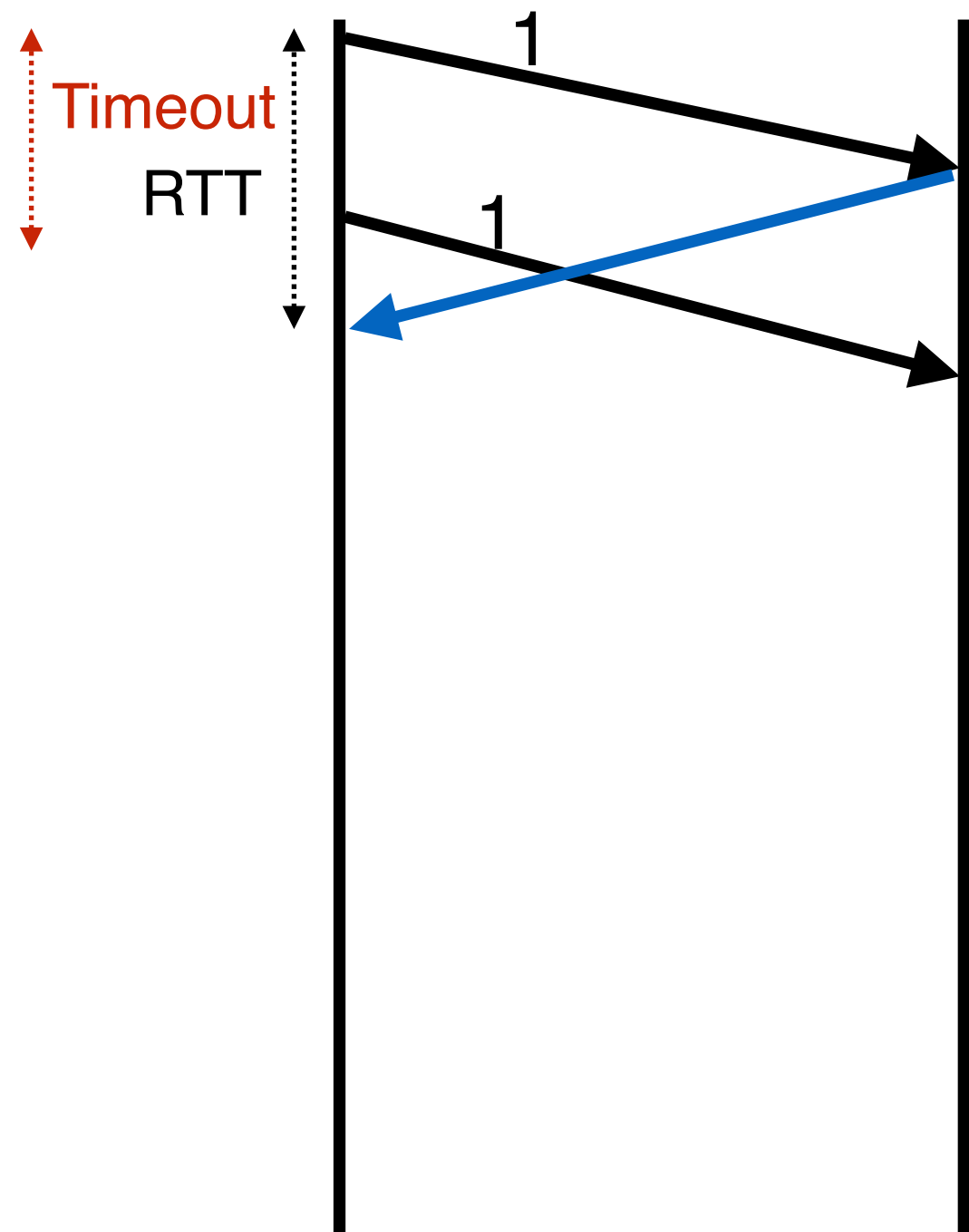
Timeouts and Retransmissions

- Reliability requires retransmitting lost data
- Involves setting timers and retransmitting on timeouts
- TCP only has a single timer
- TCP resets timer whenever new data is ACKed
- Retx packet containing “next byte” when timer expires
- RTO (Retransmit Time Out) is the basic timeout value

Setting the Timeout Value (RTO)



Timeout too long -> inefficient



Timeout too short -> duplicate packets

Setting RTO value

- Many ideas
 - See backup slides for some examples (not needed for exams)
- Implementations often use a coarser-grained timer
 - 500 msec is typical
- **Incurring a timeout is expensive**
- So we rely on duplicate ACKs

TCP Flow Control

Flow Control (Sliding Window)

- Advertised Window: W
 - Can send W bytes beyond the next expected byte
- Receiver uses W to prevent sender from overflowing buffer
- Limits number of bytes sender can have in flight

Implementing Sliding Window

- Sender maintains a window
 - Data that has been sent out but not yet ACK'ed
- Left edge of window:
 - Beginning of unacknowledged data
 - Moves when data is ACKed
- Window size = maximum amount of data in flight
- Receiver sets this amount, based on its available buffer space
 - If it has not yet sent data up to the app, this might be small

Advertised Window Limits Rate

- Sender can send no faster than W/RTT bytes/sec
- In original TCP, that was the sole protocol mechanism controlling sender's rate
- What's missing?
- **Congestion control** about how to adjust W to avoid network congestion

Any Questions?

TCP Congestion Control

TCP congestion control: high-level idea

- End hosts adjust sending rate
- Based on implicit feedback from the network
 - Implicit: router drops packets because its buffer overflows, not because it's trying to send message
- Hosts probe network to test level of congestion
 - Speed up when no congestion (i.e., no packet drops)
 - Slow down when when congestion (i.e., packet drops)
- How to do this efficiently?
 - Extend TCP's existing window-based protocol...
 - Adapt the window size based in response to congestion

All These Windows...

- **Flow control window:** Advertised Window (RWND)
 - How many bytes can be sent without overflowing receivers buffers
 - Determined by the receiver and reported to the sender
- **Congestion Window (CWND)**
 - How many bytes can be sent without overflowing routers
 - Computed by the sender using congestion control algorithm
- **Sender-side window** = $\text{minimum}\{\text{CWND}, \text{RWND}\}$
 - Assume for this lecture that $\text{RWND} \gg \text{CWND}$

Note

- 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

Basics of TCP Congestion

- Congestion Window (CWND)
 - Maximum # of unacknowledged bytes to have in flight
 - Rate \sim CWND/RTT
- Adapting the congestion window
 - Increase upon lack of congestion: optimistic exploration
 - Decrease upon detecting congestion
- But how do you detect congestion?

Not All Losses the Same

- **Duplicate ACKs: isolated loss**
 - Still getting ACKs
- **Timeout: possible disaster**
 - Not enough duplicate ACKs
 - Must have suffered several losses

How to Adjust CWND?

- Consequences of over-sized window much worse than having an under-sized window
 - Over-sized window: packets dropped and retransmitted
 - Under-sized window: somewhat lower throughput
- Approach
 - Gentle increase when un-congested (exploration)
 - Rapid decrease when congested

Additive Increase, Multiplicative Decrease (AIMD)

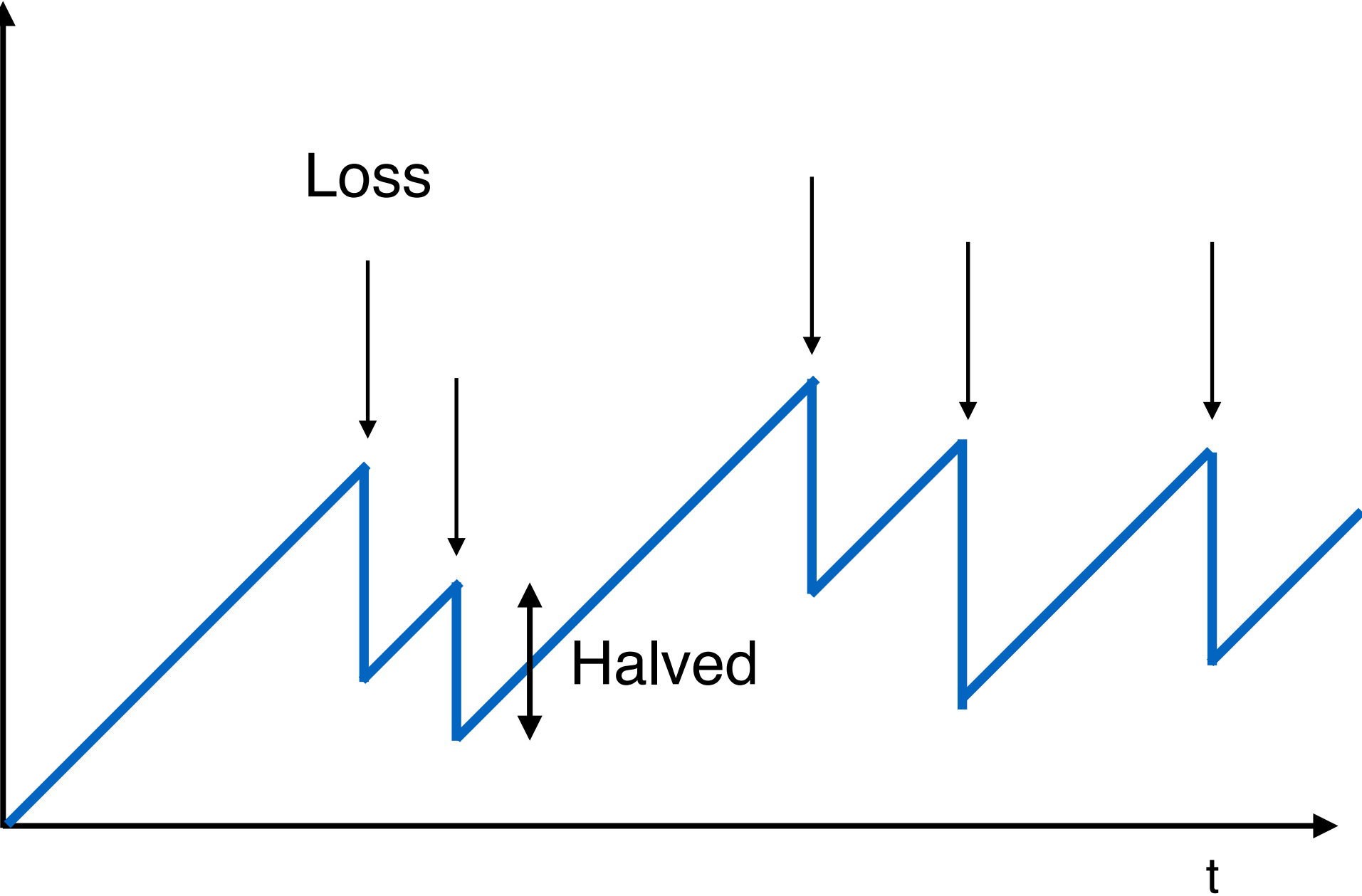
- Additive increase
 - On success of last window of data, increase by one MSS
 - If W packets in a row have been ACKed, increase W by one
 - i.e., $+1/W$ per ACK
- Multiplicative decrease
 - On loss of packets by DupACKs, divide congestion window by half
 - Special case: when timeout, reduce congestion window to one MSS

AIMD

- ACK: $CWND \rightarrow CWND + 1/CWND$
 - When CWND is measured in MSS
 - Note: after a full window, CWND increase by 1 MSS
 - Thus, **CWND increases by 1 MSS per RTT**
- 3rd DupACK: $CWND \rightarrow CWND/2$
- Special case of timeout: $CWND \rightarrow 1 \text{ MSS}$

Leads to the TCP Sawtooth

Window



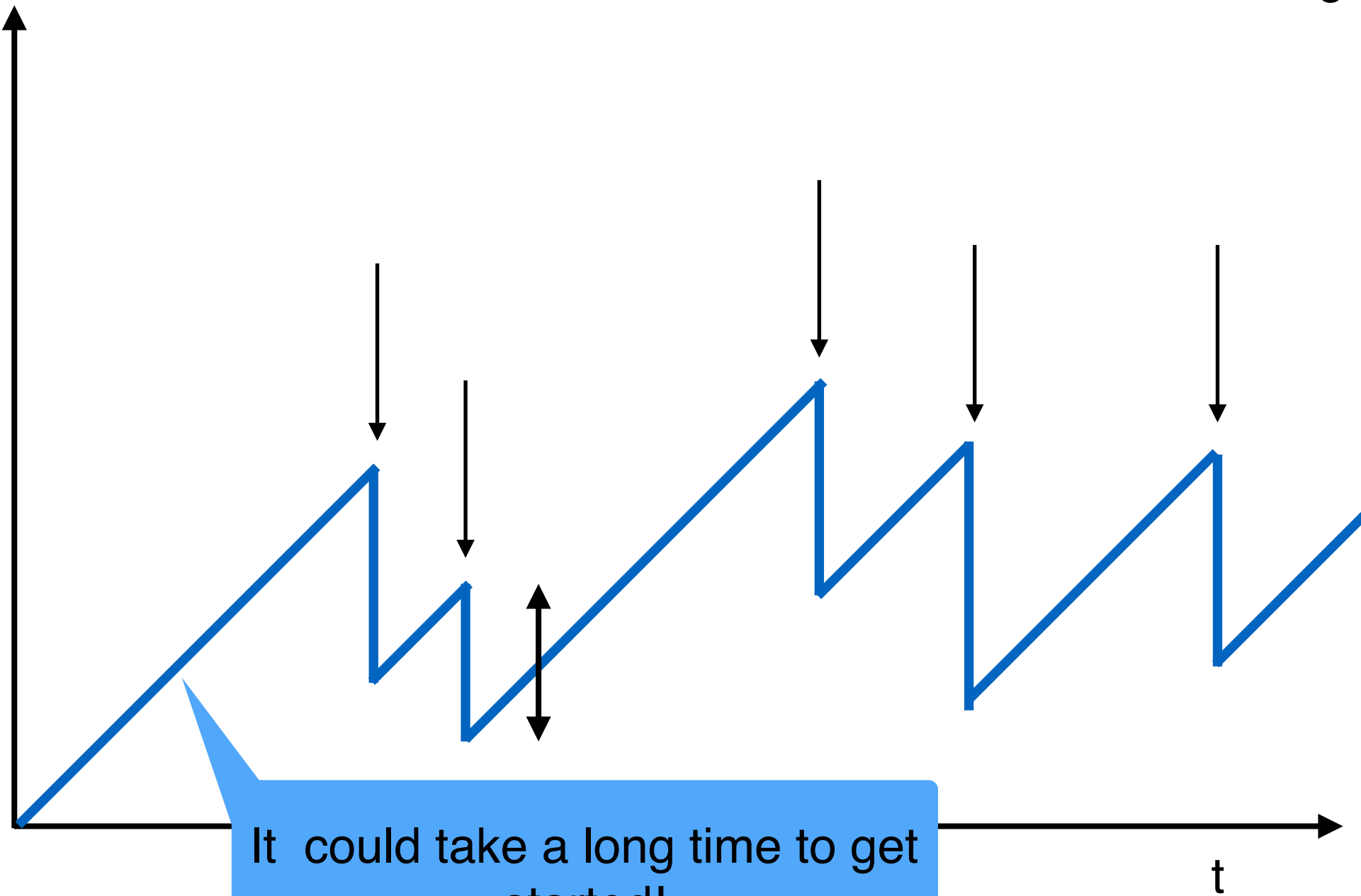
Any Questions?

Slow Start

AIMD Starts Too Slowly

Window

Need to start with a small CWND to avoid overloading the network



It could take a long time to get started!

Bandwidth Discovery with Slow Start

- Goal: estimate available bandwidth
 - Start slow (for safety)
 - But ramp up quickly (for efficiency)
- Consider
 - $RTT = 100\text{ms}$, $MSS=1000\text{bytes}$
 - Window size to fill 1Mbps of BW = 12.5 MSS
 - Window size to fill 1 Gbps = 12,500 MSS
 - With just AIMD, it takes about 12500 RTTs to get to this window size!
 - ~21 mins

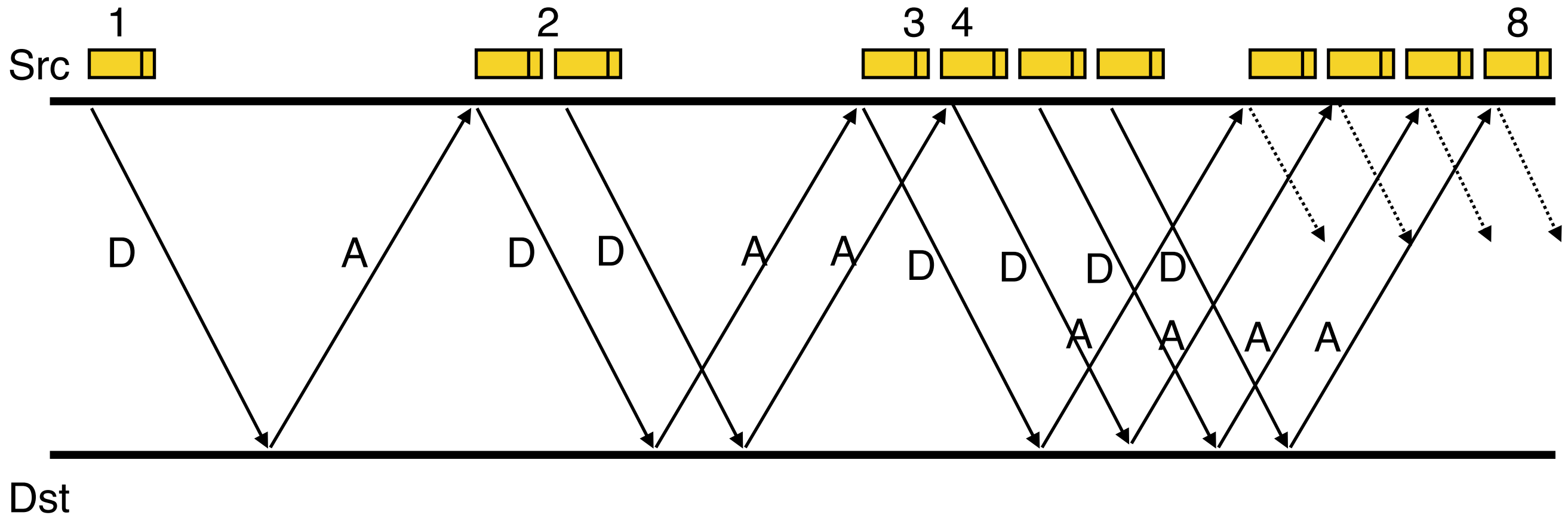
“Slow Start” Phase

- Start with a small congestion window
 - Initially, CWND is 1 MSS
 - So, initial sending rate is MSS/RTT
- That could be pretty wasteful
 - Might be much less than the actual bandwidth
 - Linear increase takes a long time to accelerate
- Slow-start phase (**actually “fast start”**)
 - Sender starts at a slow rate (hence the name)
 - ... but increases exponentially until first loss

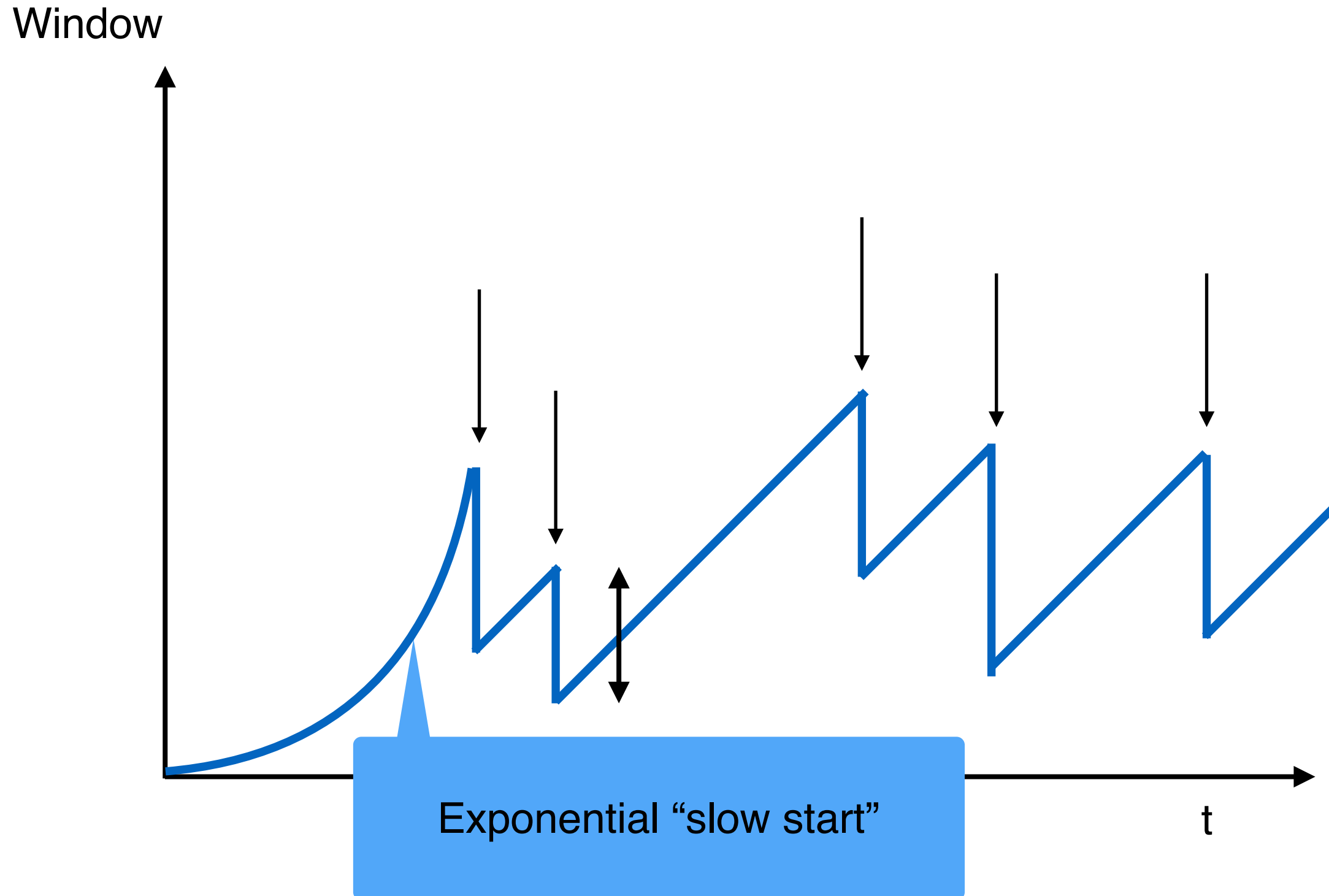
Slow Start in Action

Double CWND per round-trip time

Simple implementation: on each ACK, $CWND += MSS$



Slow Start and the TCP Sawtooth



Why is it called slow-start? Because TCP originally had no congestion control mechanism. The source would just start by sending a whole window's worth of data.

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
 - On timeout, **ssthresh** = **CWND/2**
- When $CWND > ssthresh$, sender switches from slow-start to AIMD-style increase

Timeouts

Loss Detected by Timeout

- Sender starts a timer that runs for RTO seconds
- **Restart timer whenever ACK for new data arrives**
- If timer expires
 - Set $SSTHRESH \leftarrow CWND/2$ (“Slow Start Threshold”)
 - Set $CWND \leftarrow 1$ (MSS)
 - Retransmit **first** lost packet
 - Execute Slow Start until $CWND > SSTHRESH$
 - After which switch to Additive Increase

Summary of Increase

- “Slow start”: increase CWND by 1 (MSS) for each ACK
 - A factor of 2 per RTT
- Leave slow-start regime when either:
 - $CWND > SSTHRESH$
 - Packet drop detected by dupacks
- Enter AIMD regime
 - Increase by 1 (MSS) for each window’s worth of ACKed data

Summary of Decrease

- Cut CWND half on loss detected by dupacks
 - **Fast retransmit to avoid overreacting**
- Cut CWND all the way to 1 (MSS) on **timeout**
 - Set ssthresh to $\text{CWND}/2$
- Never drop CWND below 1 (MSS)
 - Our correctness condition: always try to make progress

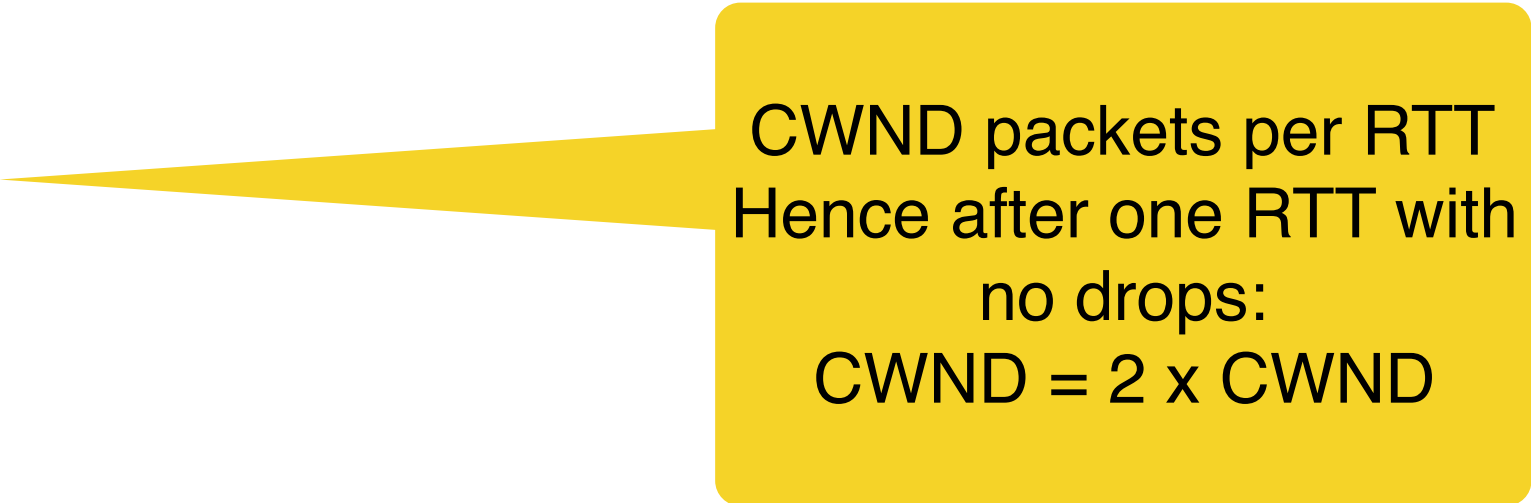
TCP Congestion Control Details

Implementation

- State at sender
 - CWND (initialized to a small constant)
 - ssthresh (initialized to a large constant)
 - dupACKcount
 - Timer, as before
- Events at sender
 - ACK (new data)
 - dupACK (duplicate ACK for old data)
 - Timeout
- What about receiver? Just send ACKs upon arrival

Event: ACK (new data)

- If in slow start
 - $CWND += 1$

A yellow callout box with a pointer pointing to the 'CWND += 1' bullet point. The text inside the box explains the consequence of this action: 'CWND packets per RTT Hence after one RTT with no drops: CWND = 2 x CWND'.

CWND packets per RTT
Hence after one RTT with
no drops:
 $CWND = 2 \times CWND$

Event: ACK (new data)

- If $CWND \leq ssthresh$
 - $CWND += 1$
- Else
 - $CWND = CWND + 1/CWND$

Slow Start Phase

Congestion Avoidance Phase
(additive increase)


CWND packets per RTT
Hence after one RTT with
no drops:
 $CWND = CWND + 1$

Event: Timeout

- On Timeout
 - $ssthresh \leftarrow CWND/2$
 - $CWND \leftarrow 1$

Event: dupACK

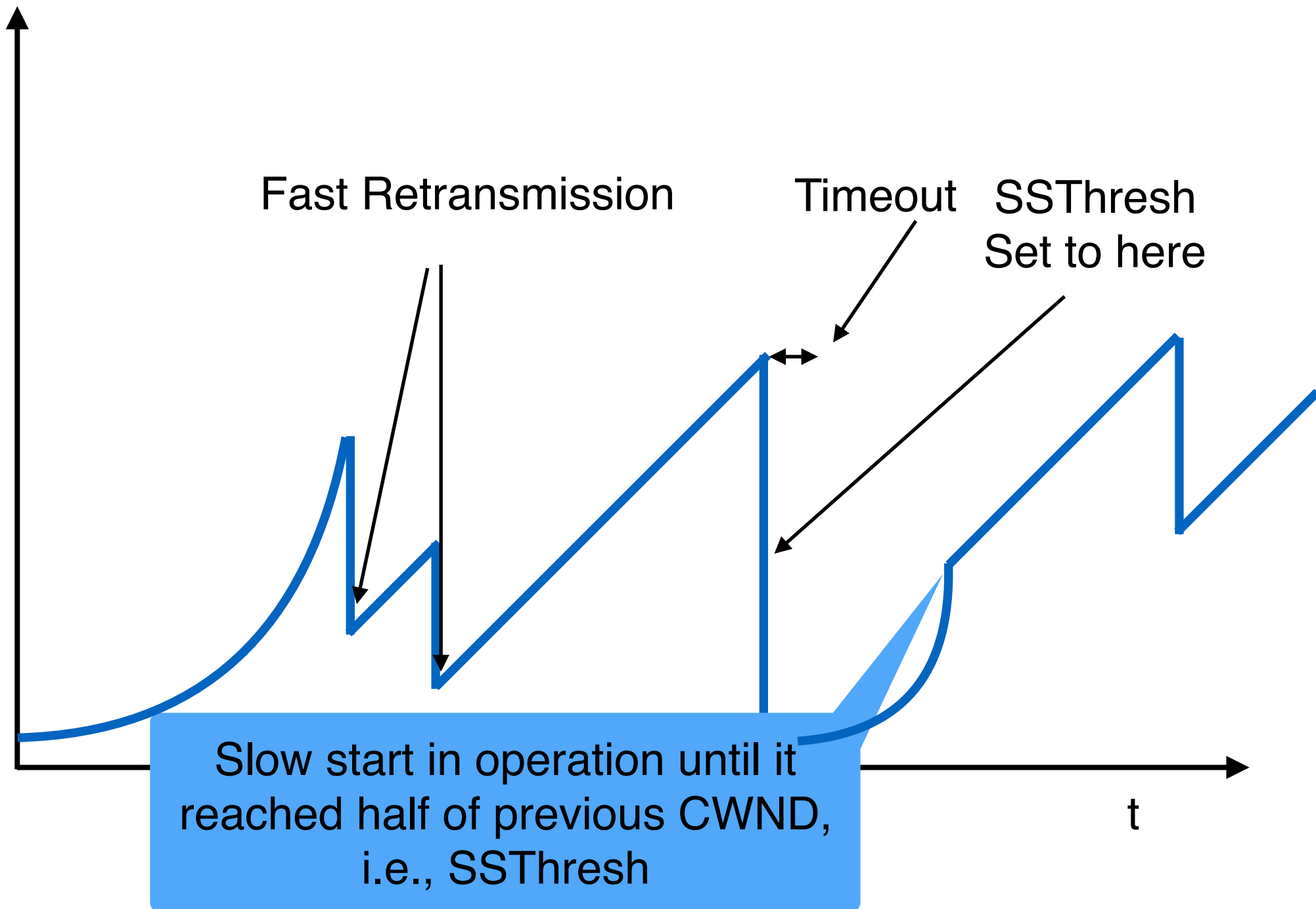
- dupACKcount++
- If dupACKcount = 3 /* fast retransmit */
 - ssthresh <- CWND/2
 - CWND <- CWND/2



Remains in congestion avoidance after fast retransmission

Time Diagram

Window



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

TCP Flavors

- TCP Tahoe
 - $CWND = 1$ on triple dupACK
- TCP Reno
 - $CWND = 1$ on timeout
 - $CWND = CWND/2$ on triple dupACK
- TCP-newReno
 - TCP-Reno + improved fast recovery
- TCP-SACK
 - Incorporates selective acknowledgements



Our default assumption

Done!

Next lecture: Critical Analysis of TCP

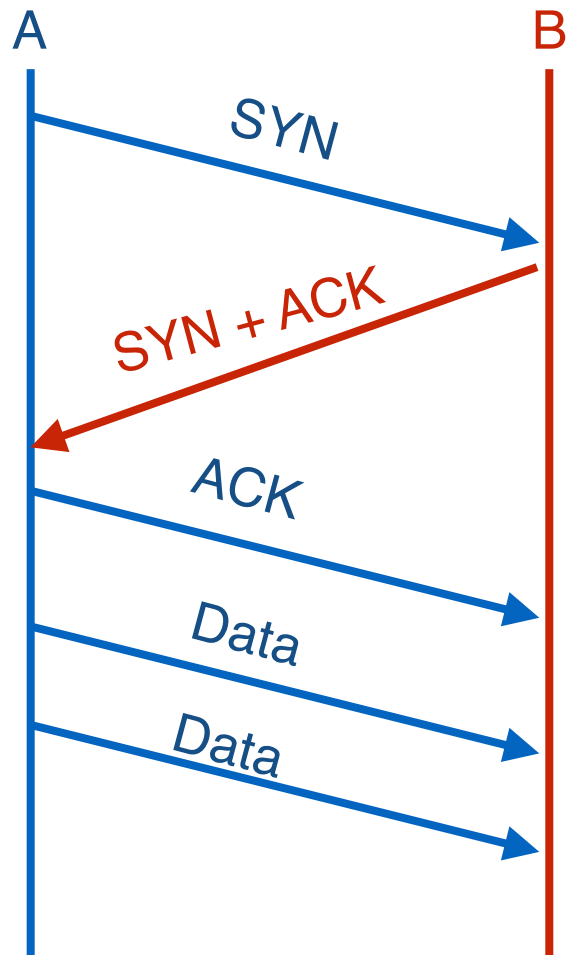
TCP Back up slides

TCP Connection Establishment and Initial Sequence Numbers

Initial Sequence Number (ISN)

- Sequence number for the very first byte
 - E.g., Why not just use ISN = 0?
- Practical issue
 - IP addresses and port #s uniquely identify a connection
 - Eventually, though, these port #s do get **used again**
 - ... small chance an old packet is **still in flight**
- TCP therefore requires changing ISN
 - Set from 32-bit clock that ticks every 4 microseconds
 - ... only wraps around once every 4.55 hours
- To establish a connection, hosts exchange ISNs
 - How does this help?

Establishing a TCP Connection

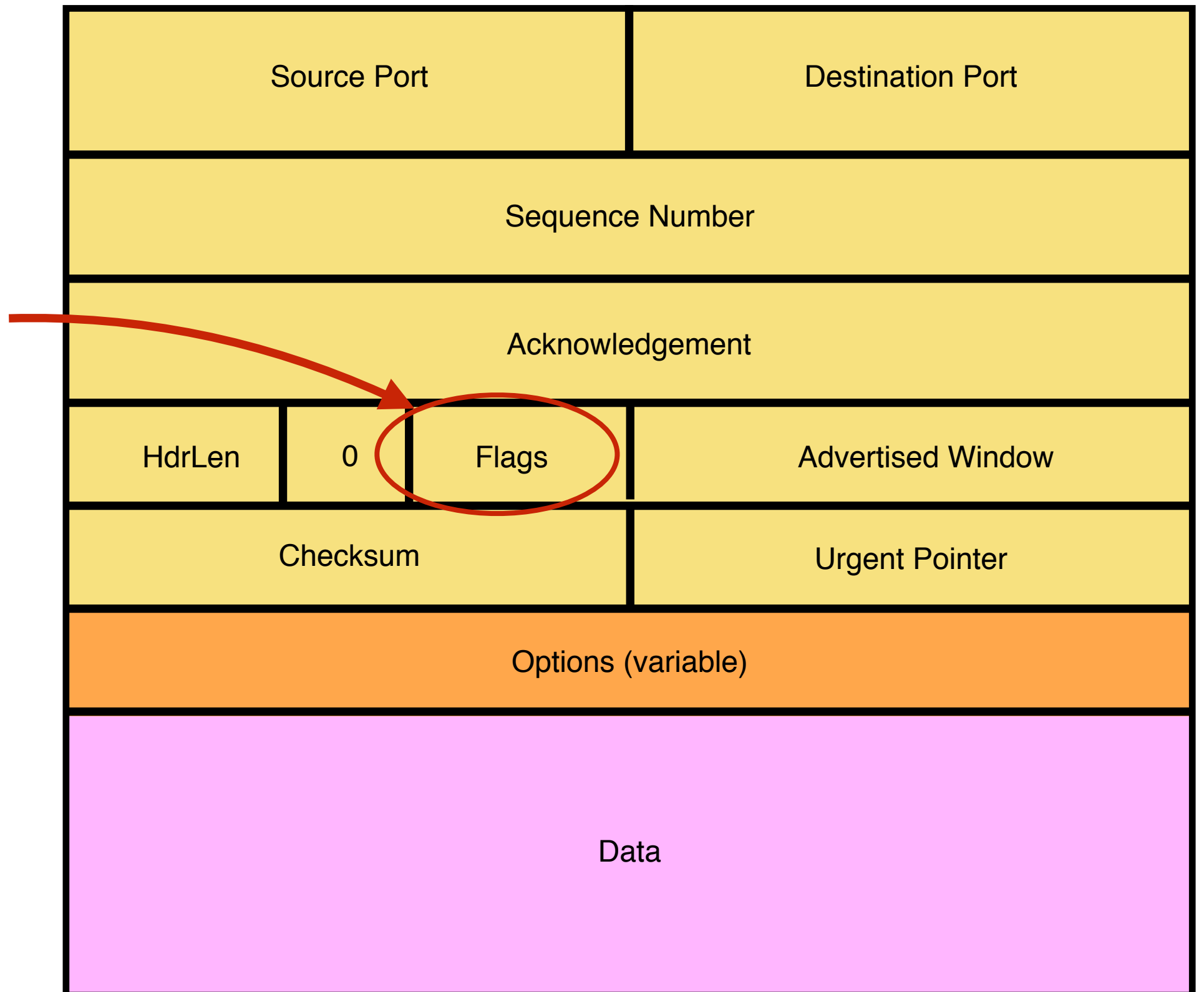


Each host tells its ISN to the other host.

- Three-way handshake to establish connection
 - Host A sends a **SYN** (open; “synchronize sequence numbers”) to host B
 - Host B returns a SYN acknowledgement (**SYN ACK**)
 - Host sends an **ACK** to acknowledge the SYN ACK

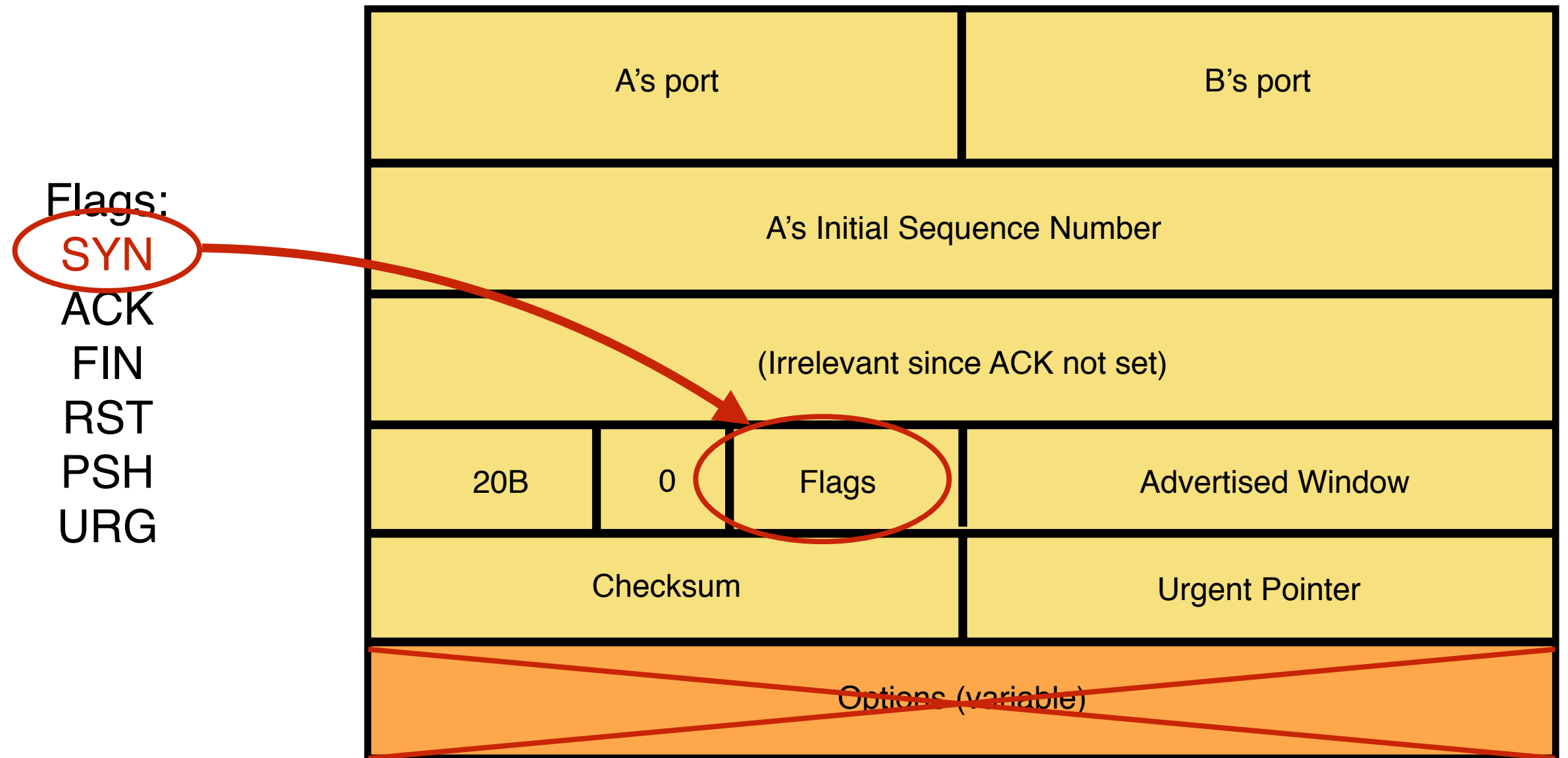
TCP Header

Flags:
SYN
ACK
FIN
RST
PSH
URG



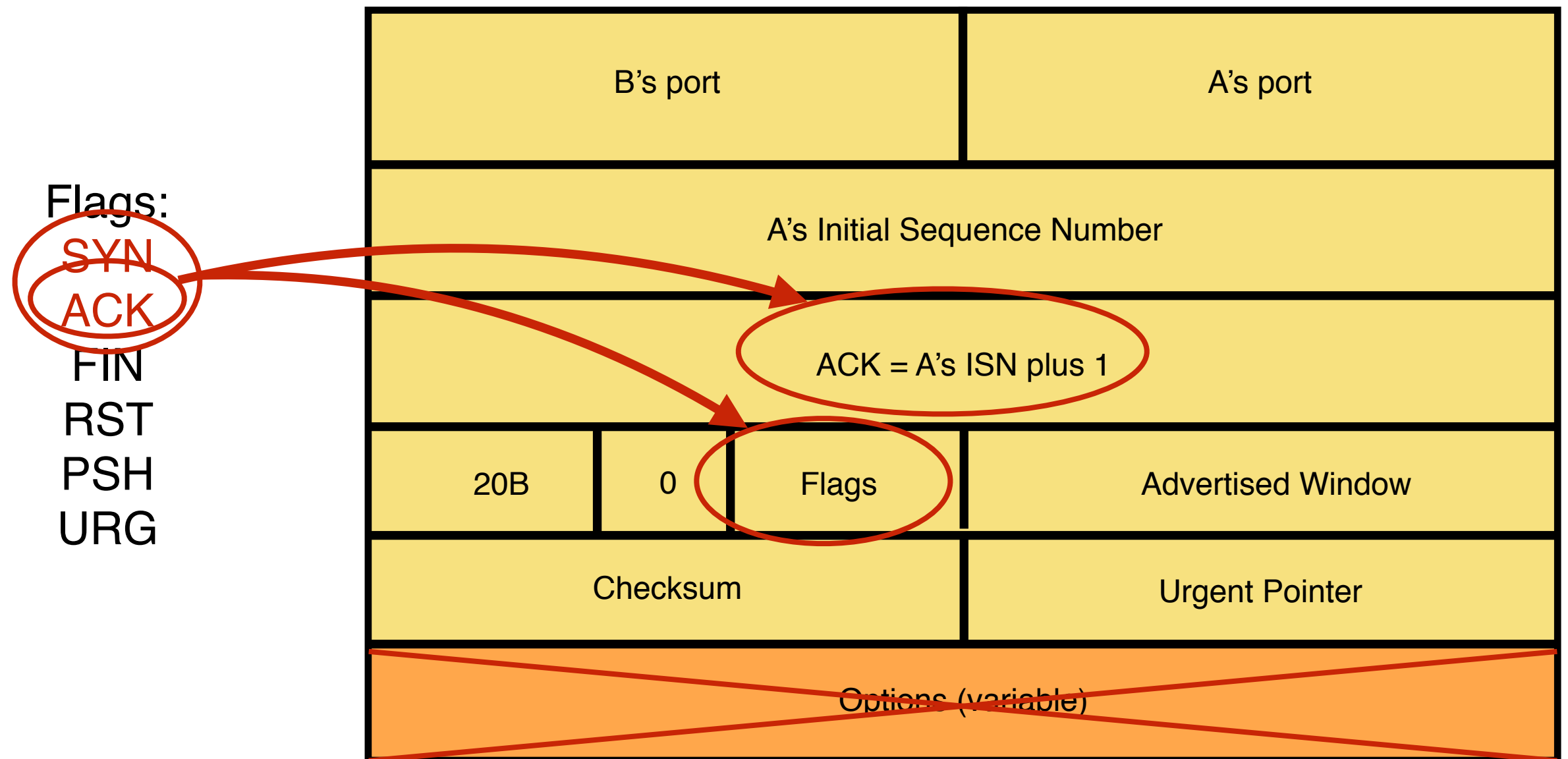
See `/usr/include/netinet/tcp.h` on Unix Systems

Step 1: A's Initial SYN Packet



A tells B it wants to open a connection...

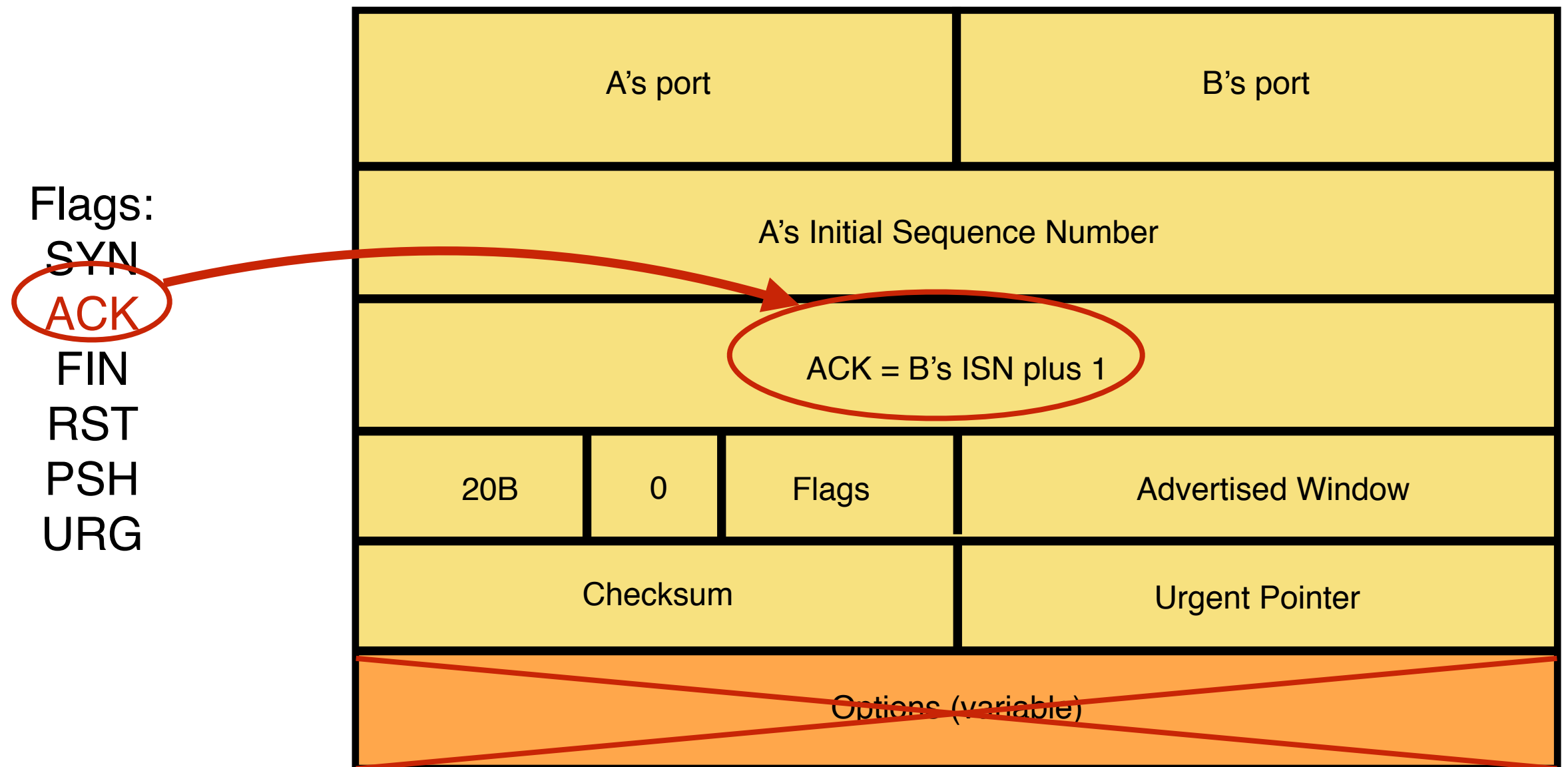
Step 2: B's SYN-ACK Packet



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

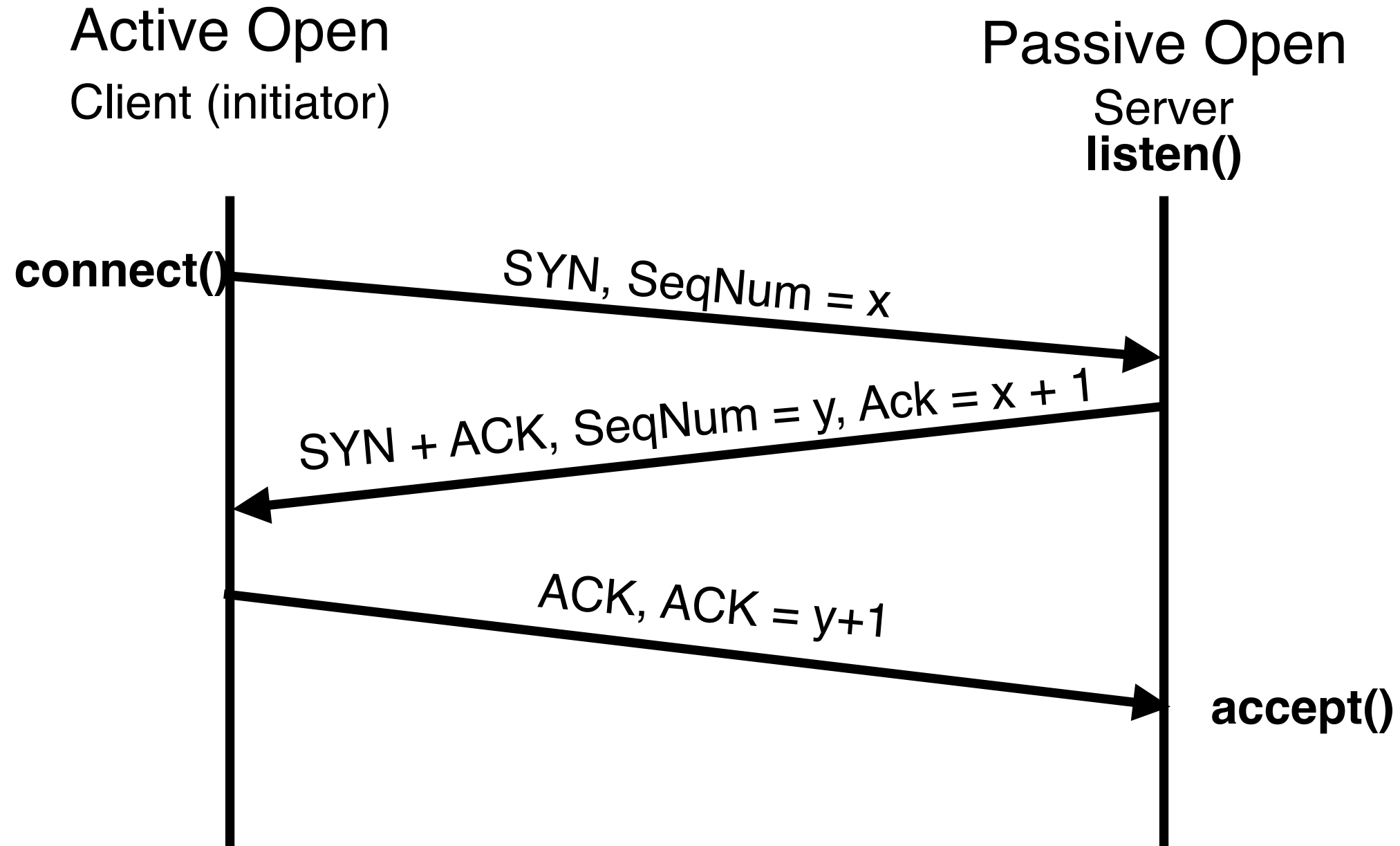
... upon receiving this packet, A can start sending data

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

Timing Diagram: 3-Way Handshaking



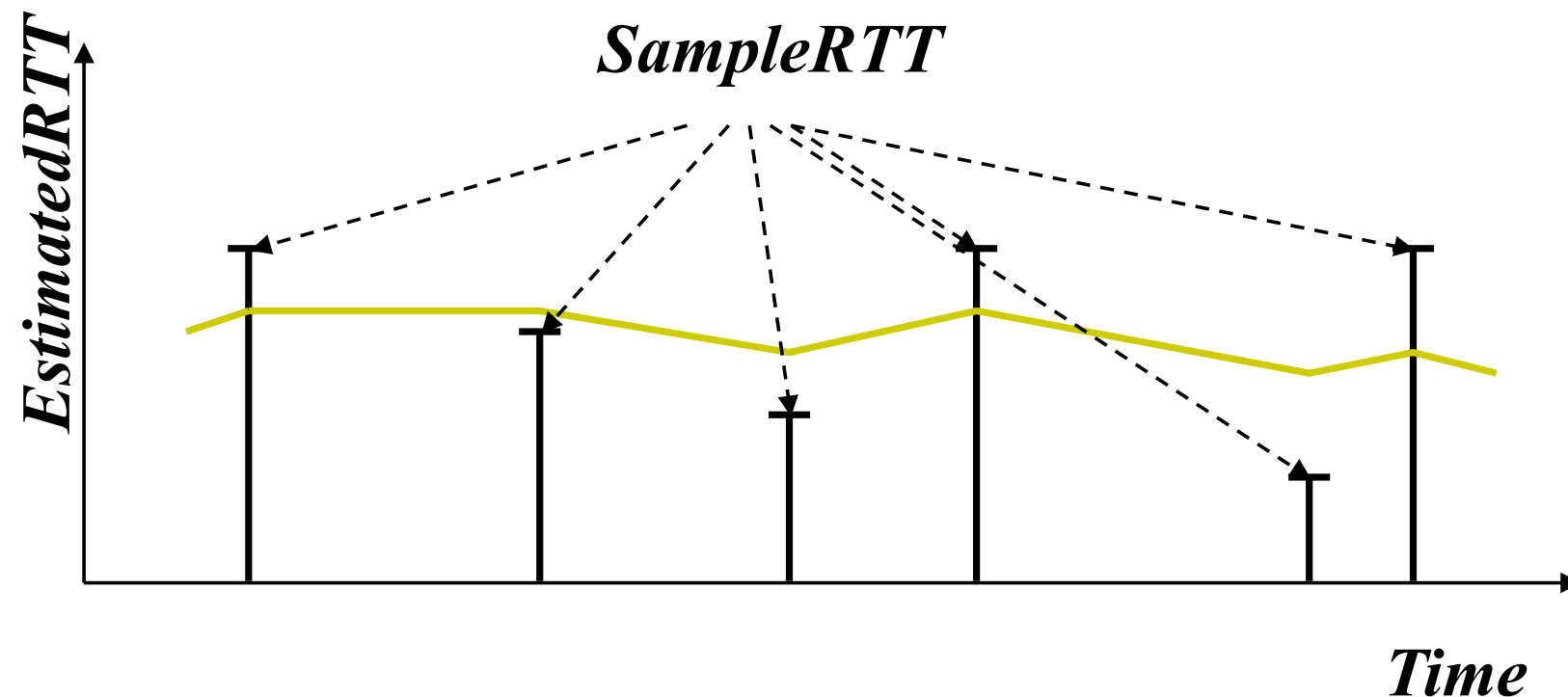
Could Base RTO on RTT Estimation

- Use exponential averaging if RTT samples

$\text{SampleRTT} = \text{AckRcvdTime} - \text{SendPktTime}$

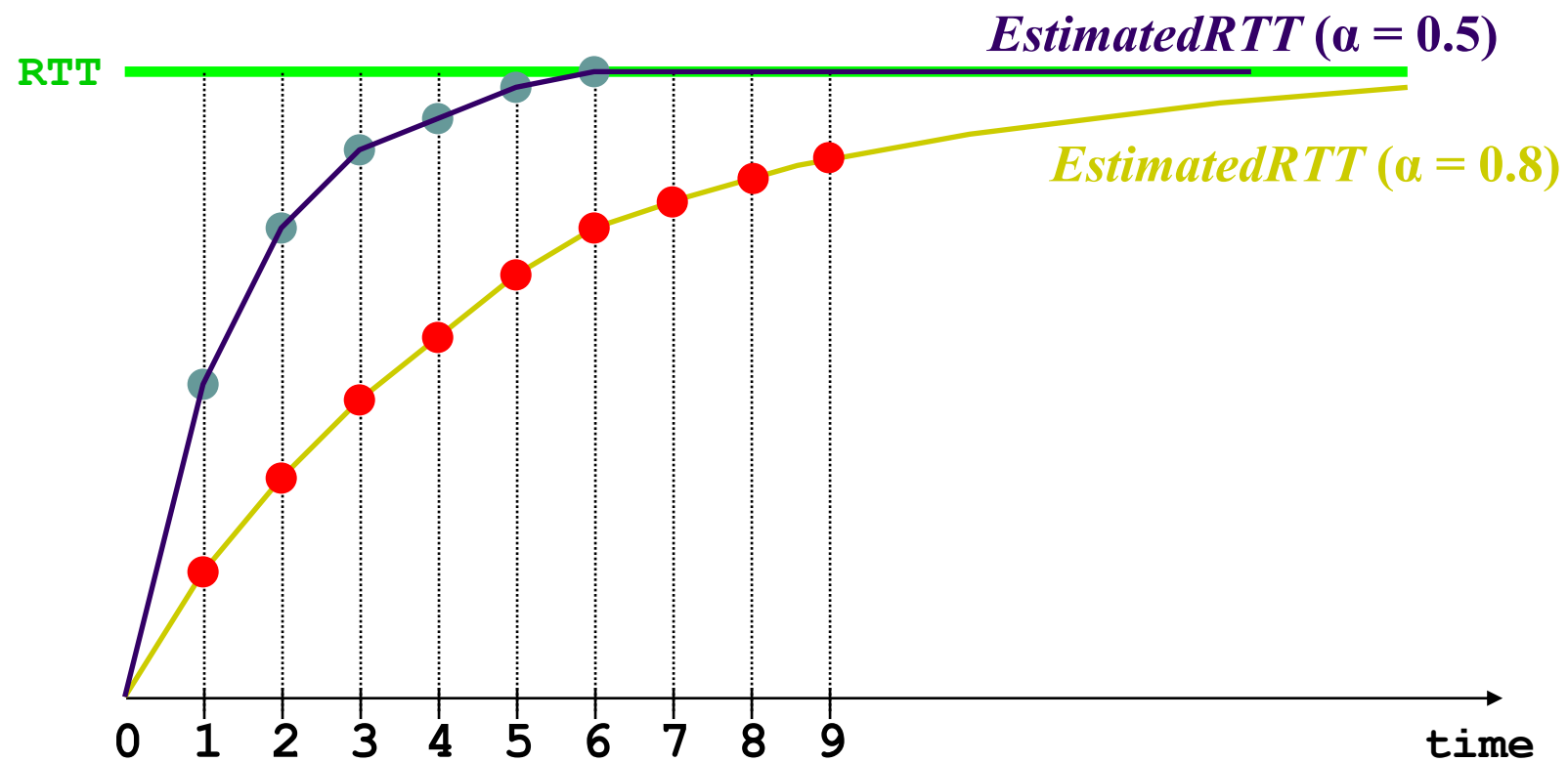
$\text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + (1-\alpha) \times \text{SampledRTT}$

$0 < \alpha \leq 1$



Exponential Averaging Example

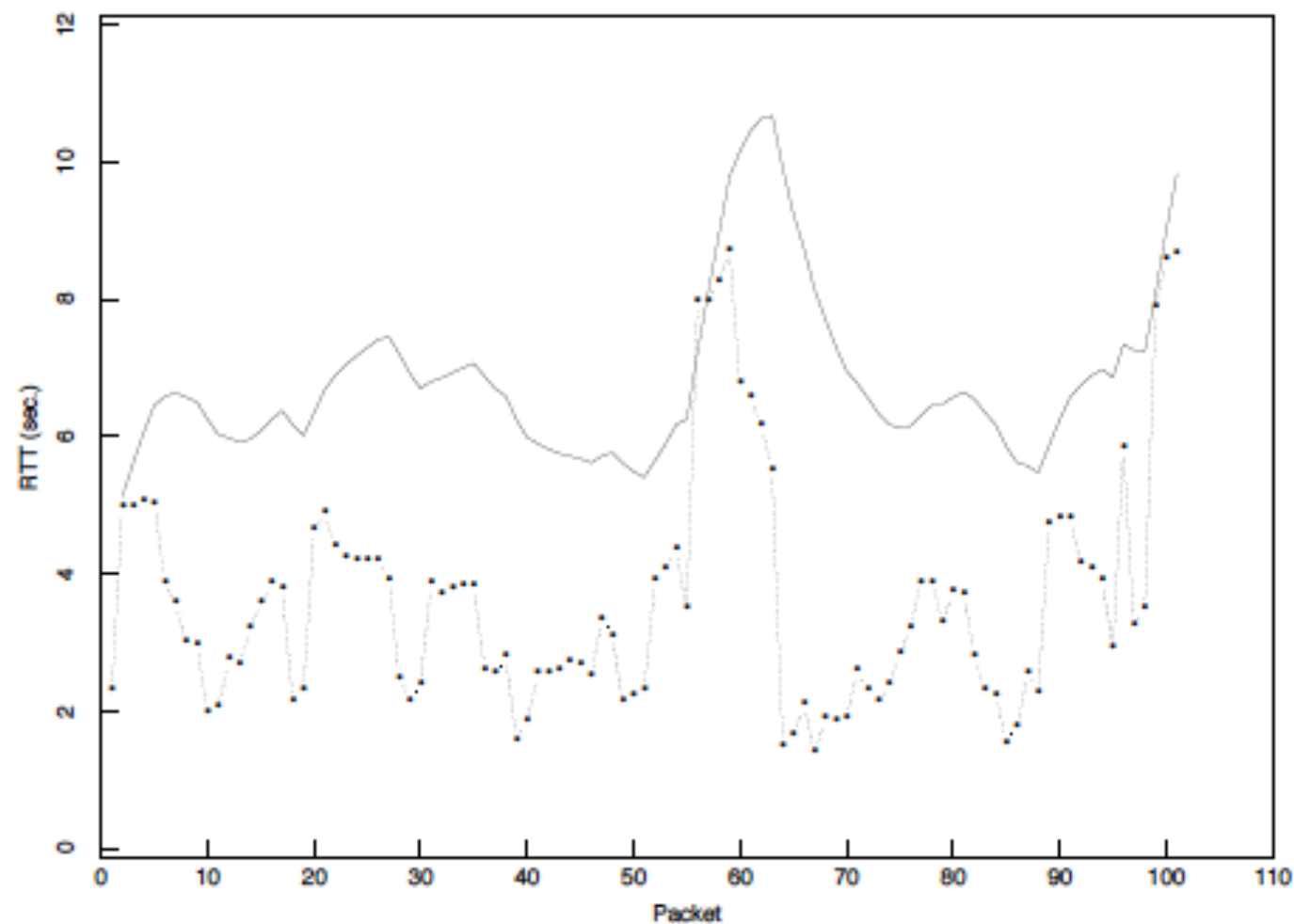
EstimatedRTT = α x EstimatedRTT + (1- α) x SampledRTT
(Assume RTT is constant => SampleRTT = RTT)



Exponential Averaging in Action

Set Timeout Estimate (ETO) = 2 x EstimatedRTT

Figure 5: Performance of an RFC793 retransmit timer



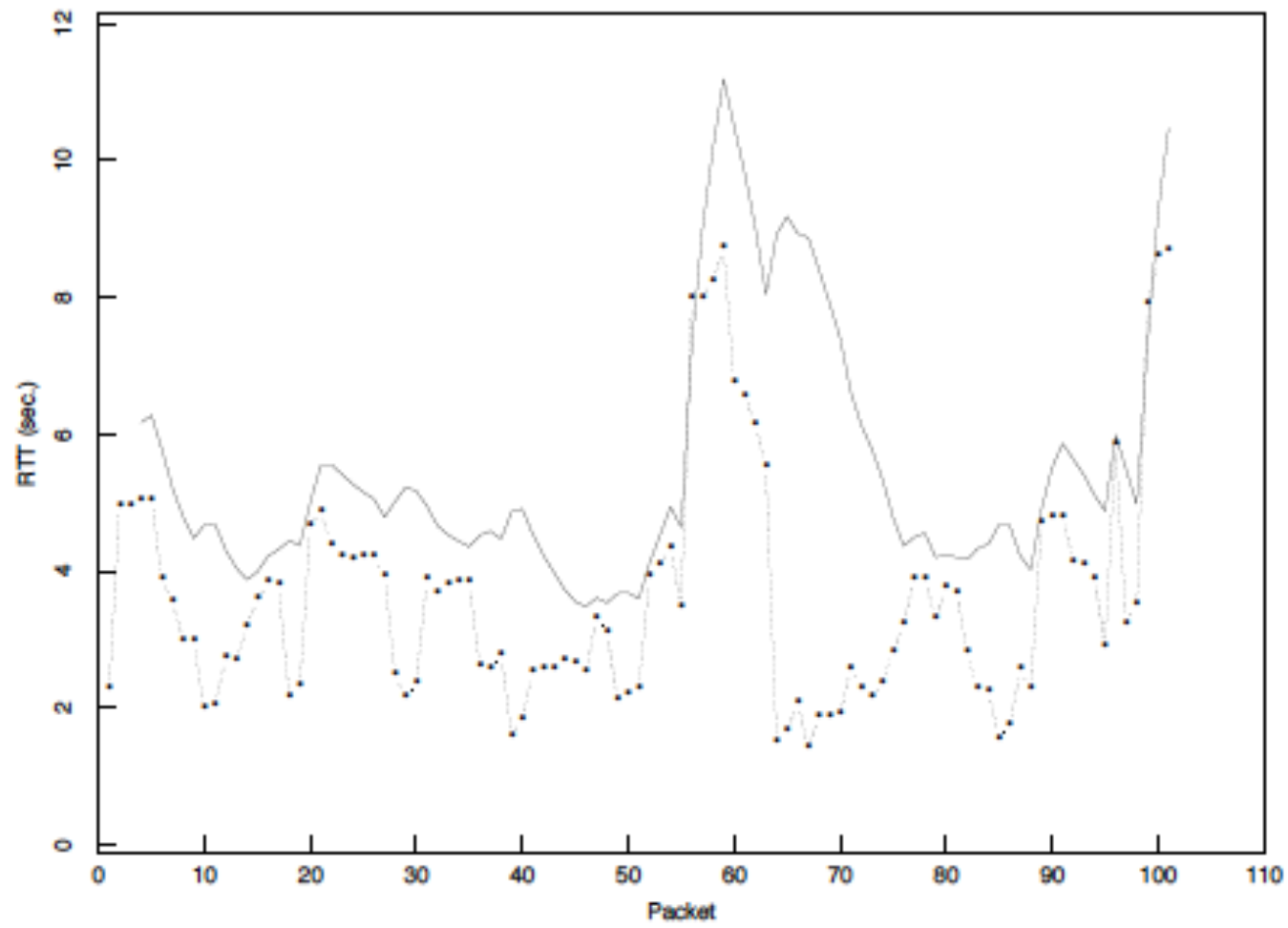
From Jacobson and Karels, SIGCOMM 1988

Jacobson/Karels Algorithm

- Problem: need to better capture variability in RTT
 - Directly measure deviation
- Deviation = $| \text{SampleRTT} - \text{EstimatedRTT} |$
- Estimated Deviation: exponential average of Deviation
- ETO = $\text{EstimatedRTT} + 4 \times \text{EstimatedDeviation}$

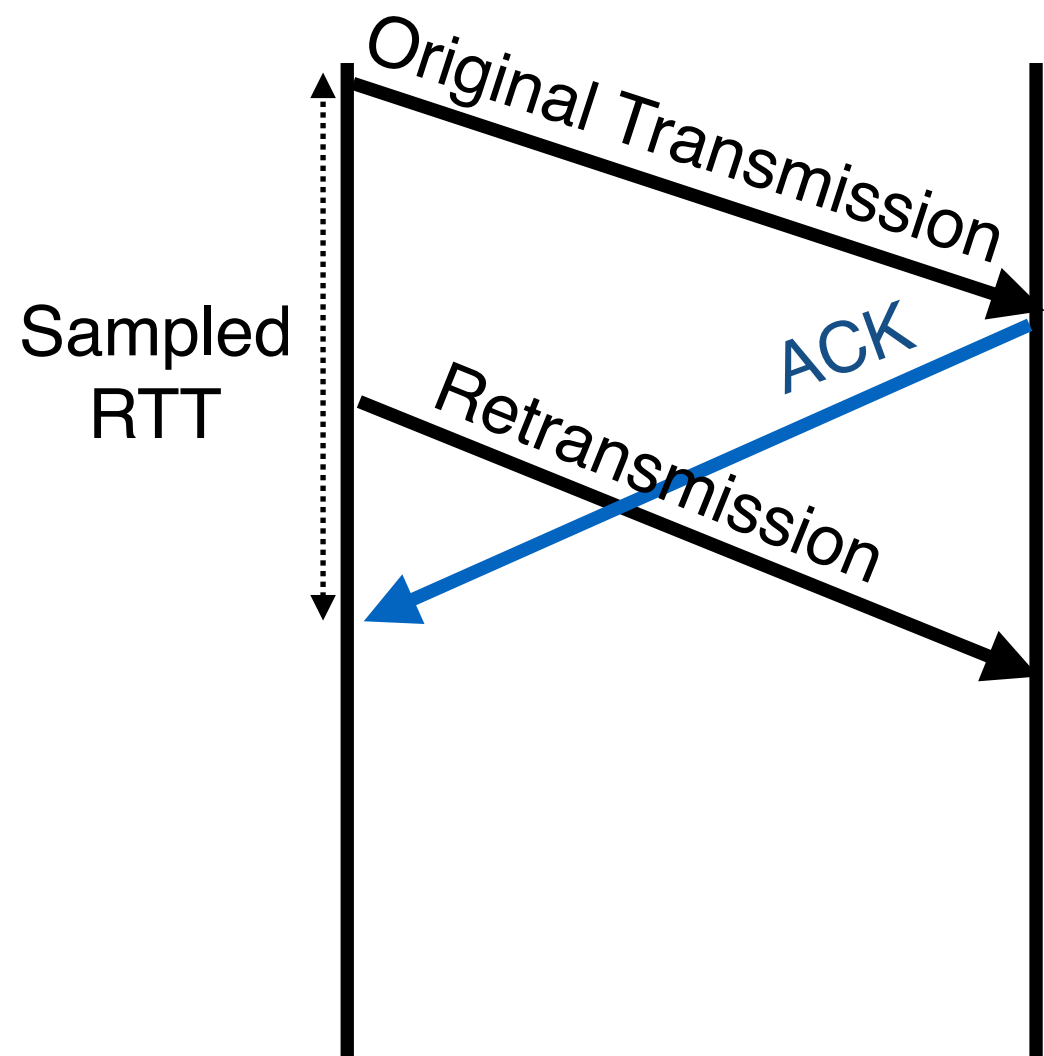
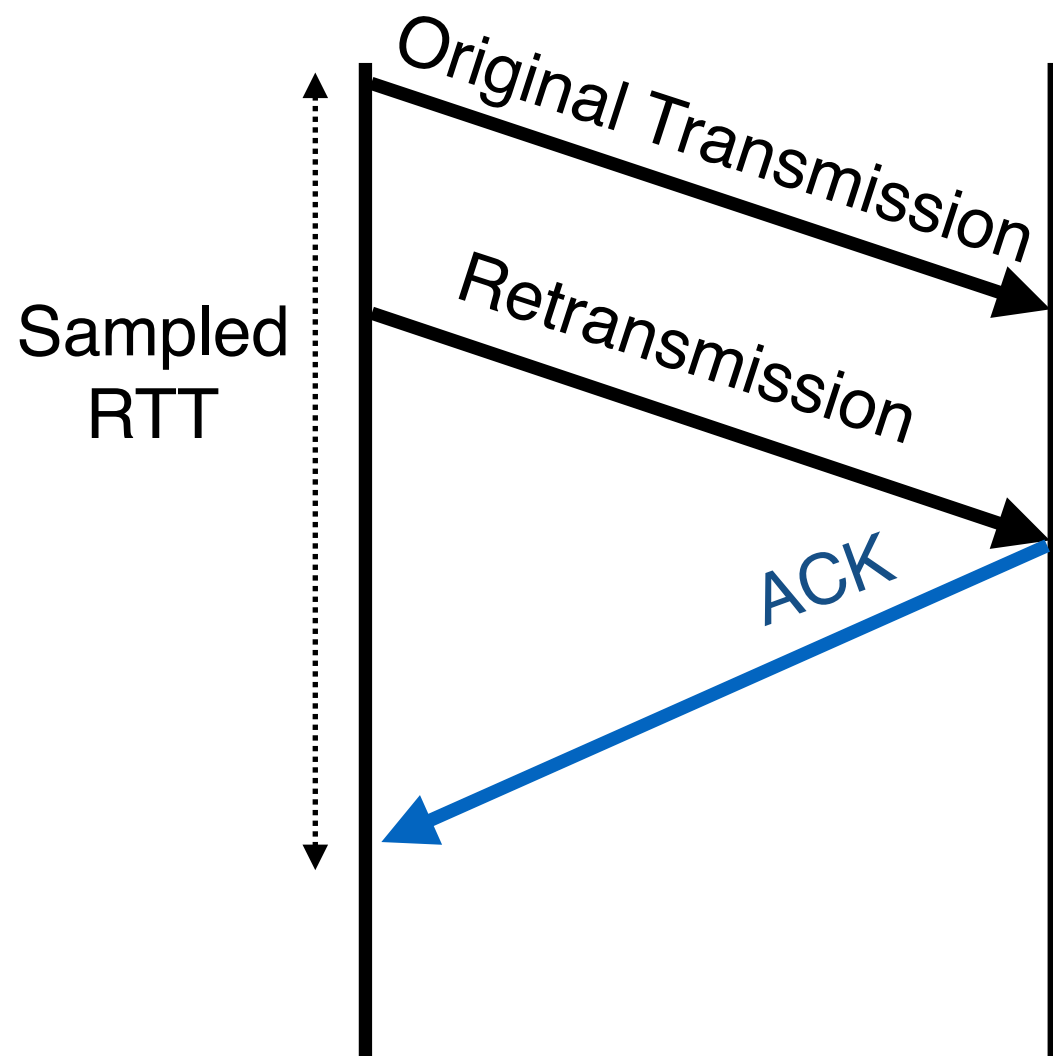
With Jacobson/Karels

Figure 6: Performance of a Mean+Variance retransmit timer



Problem: Ambiguous Measurements

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



TCP Timers

- Two important quantities
 - RTO: value you set timer to for timeouts
 - ETO: current estimate of appropriate “raw” timeout
- Use exponential averaging to estimate
 - RTT
 - Deviation = $| \text{Estimated RTT} - \text{Sample RTT} |$
- $\text{ETO} = \text{Estimated RTT} + 4 \times \text{Estimated Deviation}$

Use Only “Clean” Samples for ETO

- Only update ETO when you get a clean sample
- Where clean means ACK includes no retransmitted segments

Example

- Send 100, 200, 300
 - 100 means packet whose first byte is 100, last byte is 199
- Receive A200
 - A200 means bytes up to 199 rep'd, expecting 200 next
 - Clean sample
- 200 times out, resend 200, receive A300
 - No clean samples
- Send 400, 500, receive A600
 - Clean samples

Setting RTO

- Every time RTO timer expires, set $RTO \leftarrow 2 \cdot RTO$
 - Upto maximum ≥ 60 sec
- Every time clean sample arrives set RTO to ETO

Example

- First arriving ACK expects 100 (adv. window=500)
 - Initialize ETP; $RTO = ETO$
 - Restart timer for RTO seconds (new data ACK'ed)
 - Remember TCP only has one timer, not timer per packet
 - Send packets 100, 200, 300, 400 and 500
- Arriving ACK expects 300 (A300)
 - Update ETO; $RTO = ETO$
 - Restart timer for RTO seconds (new data ACKed)
 - Send packets 600, 700
- Arriving ACK expects 300 (A300)

Example (cont'd)

- Timer goes off
 - $RTO = 2 * RTO$ (back off timer)
 - Restart timer for RTO seconds (it had expired)
 - Resend packet 300
- Arriving ACK expects 800
 - Don't update ETO (ACK includes a retransmission)
 - Restart timer for RTO seconds (new data ACKed)
 - Send packets 800, 900, 1000, 1100, 1200

Example (cont'd)

- Arriving ACK expects 1000
 - Updates ETO; $RTO = ETO$
 - Restart timer for RTO seconds (new data ACKed)
 - Send packets 1300, 1400
- ... Connection continues...

Example

- Consider a TCP connection with:
 - CWND = 10 packets
 - Last ACK was for packet # 101
 - i.e., receiver expecting next packet to have seq no 101
- 10 packets [101, 102, 103, ..., 110] are in flight
 - Packet 101 is dropped
 - What ACKs do they generate?
 - And how does the sender respond?

Timeline

- ACK 101(due to 102) CWND = 10 dupACK #1 (no xmit)
- ACK 101(due to 103) CWND = 10 dupACK #2 (no xmit)
- ACK 101(due to 104) CWND = 10 dupACK #3 (no xmit)
- **RETRANSMIT 101 ssthresh = 5 CWND = 5**
- ACK 101 (due to 105) CWND=5 (no xmit)
- ACK 101 (due to 106) CWND=5 (no xmit)
- ACK 101 (due to 107) CWND=5 (no xmit)
- ACK 101 (due to 108) CWND=5 (no xmit)
- ACK 101 (due to 109) CWND=5 (no xmit)
- ACK 101 (due to 110) CWND=5 (no xmit)
- **ACK 111 (due to 101)<- only now can we transmit new packets**
- **Plus no packets in flight so no ACKs for another RTT**

Note that you do not restart dupACKcounter on same packet!

Solution: Fast Recovery

- Idea: Grant the sender temporary “credit” for each dupACK so as to keep packets in flight (each ACK due to arriving pkt)
- If dupACKcount = 3
 - ssthresh = CWND / 2
 - CWND = ssthresh + 3
- While in fast recovery
 - CWND = CWND + 1 for each additional duplicate packet
- Exit fast recovery after receiving new ACK
 - Set CWND = ssthresh (which had been set to CWND/2 after loss)

Example

- Consider a TCP connection with:
 - CWND = 10 packets
 - Last ACK was for packet # 101
 - i.e., receiver expecting next packet to have seq no 101
- 10 packets [101, 102, 103, ..., 110] are in flight
 - Packet 101 is dropped

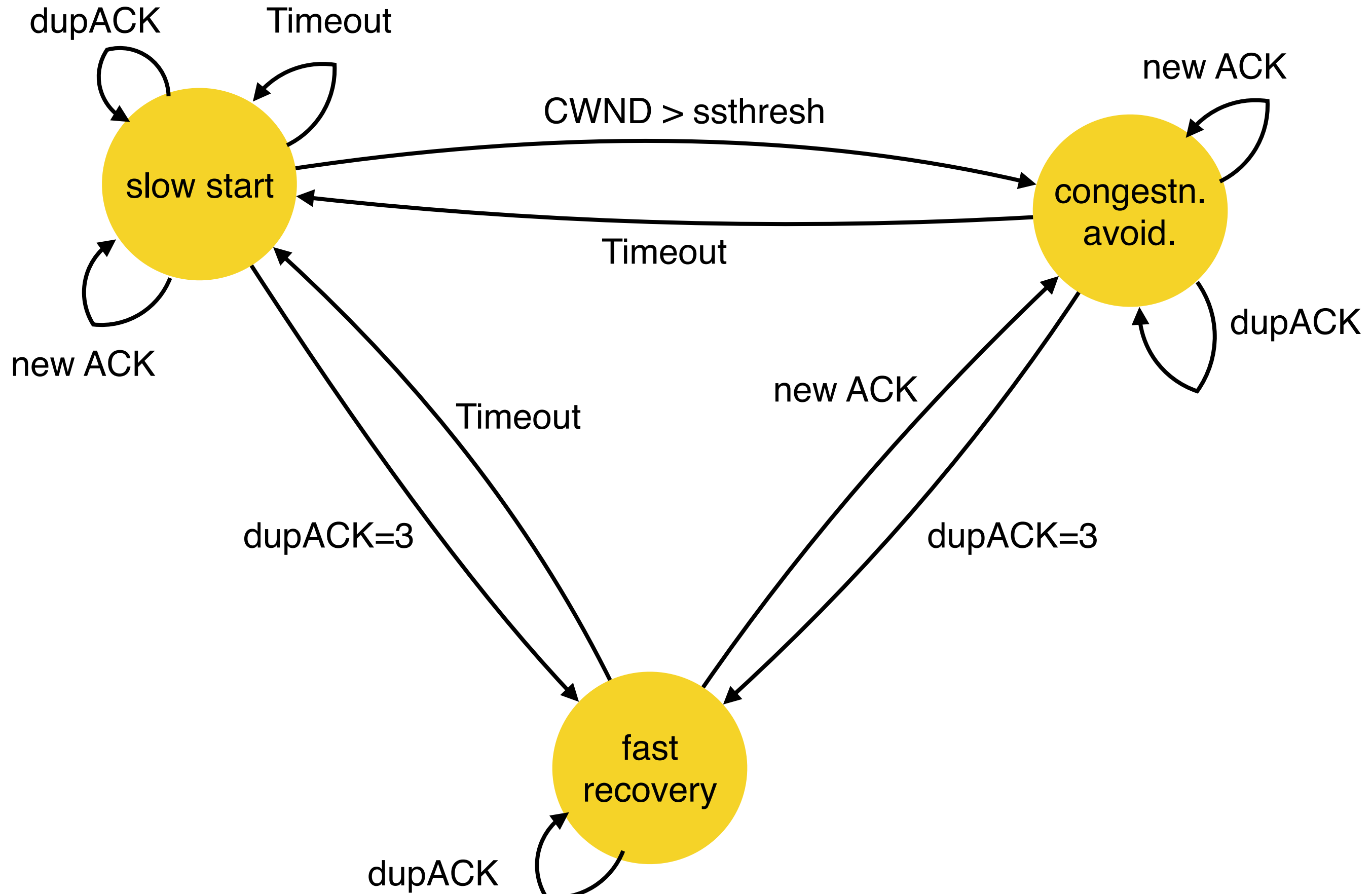
Timeline

- ACK 101(due to 102) CWND = 10 dupACK #1 (no xmit)
- ACK 101(due to 103) CWND = 10 dupACK #2 (no xmit)
- ACK 101(due to 104) CWND = 10 dupACK #3 (no xmit)
- RETRANSMIT 101 ssthresh = 5 CWND = 8 (5 + 3)
- ACK 101 (due to 105) CWND=9 (no xmit)
- ACK 101 (due to 106) CWND=10 (no xmit)
- ACK 101 (due to 107) CWND=11 (xmit 111)
- ACK 101 (due to 108) CWND=12 (xmit 112)
- ACK 101 (due to 109) CWND=13 (xmit 113)
- ACK 101 (due to 110) CWND=14 (xmit 114)
- ACK 111 (due to 101) CWND = 5 (xmit 115) <- exiting fast recovery
- Packets 111-114 already in flight (and not sending 115)
- ACK 112 (due to 111) CWND = 5 + 1/5 <- back to congestion avoidance

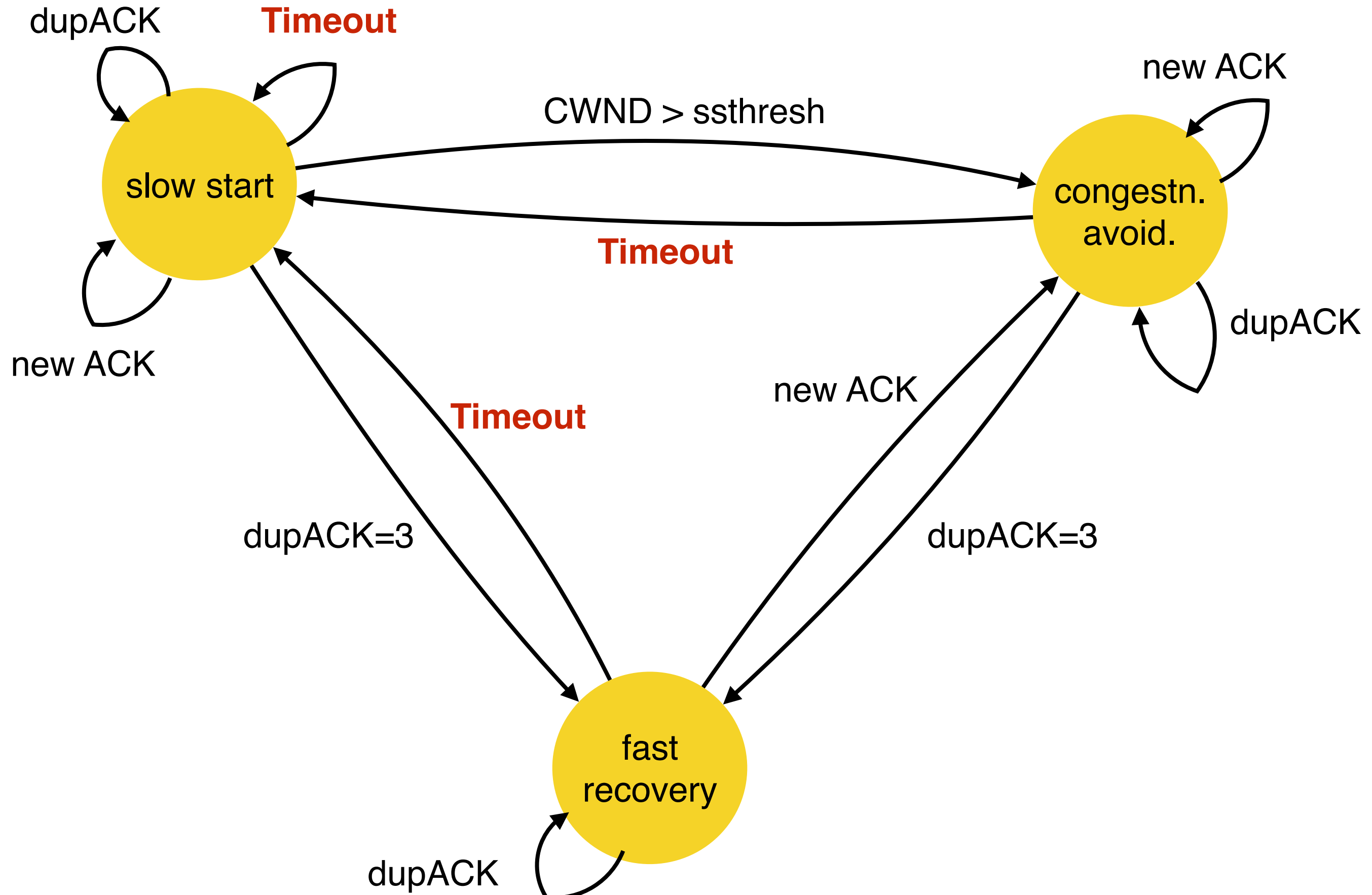
TCP “Phases”

- Slow-start
 - Enter on timeout
 - Leave when $CWND > ssthresh$ (to Cong. Avoid.)
 - The $>$ only applies here...
- Congestion Avoidance
 - Leave when timeout
- Fast recovery
 - Enter when $dupACK=3$
 - Leave when New ACK or Timeout

TCP State Machine



TCP State Machine



TCP State Machine

