TCP

EE 122, Fall 2013
Sylvia Ratnasamy
http://inst.eecs.berkeley.edu/~ee122/

Material thanks to Ion Stoica, Scott Shenker, Jennifer Rexford, Nick McKeown, and many other colleagues

What Did We Learn Last Time?

- Transport has two purposes:
  - Mux/Demux: done using *Ports* and *Sockets*
  - Optional: Reliable delivery
- Reliable delivery involves many mechanisms
  - Requires delicate design to get them to work together
  - TCP is the standard example of a reliable transport

Reminder: The TCP Abstraction

- TCP delivers a **reliable, in-order, bytestream**
- Reliable: TCP resends lost packets (recursively)
  - Until it gives up and shuts down connection
- In-order: TCP only hands consecutive chunks of data to application
- Bytestream: TCP assumes there is an incoming stream of data, and attempts to deliver it to app

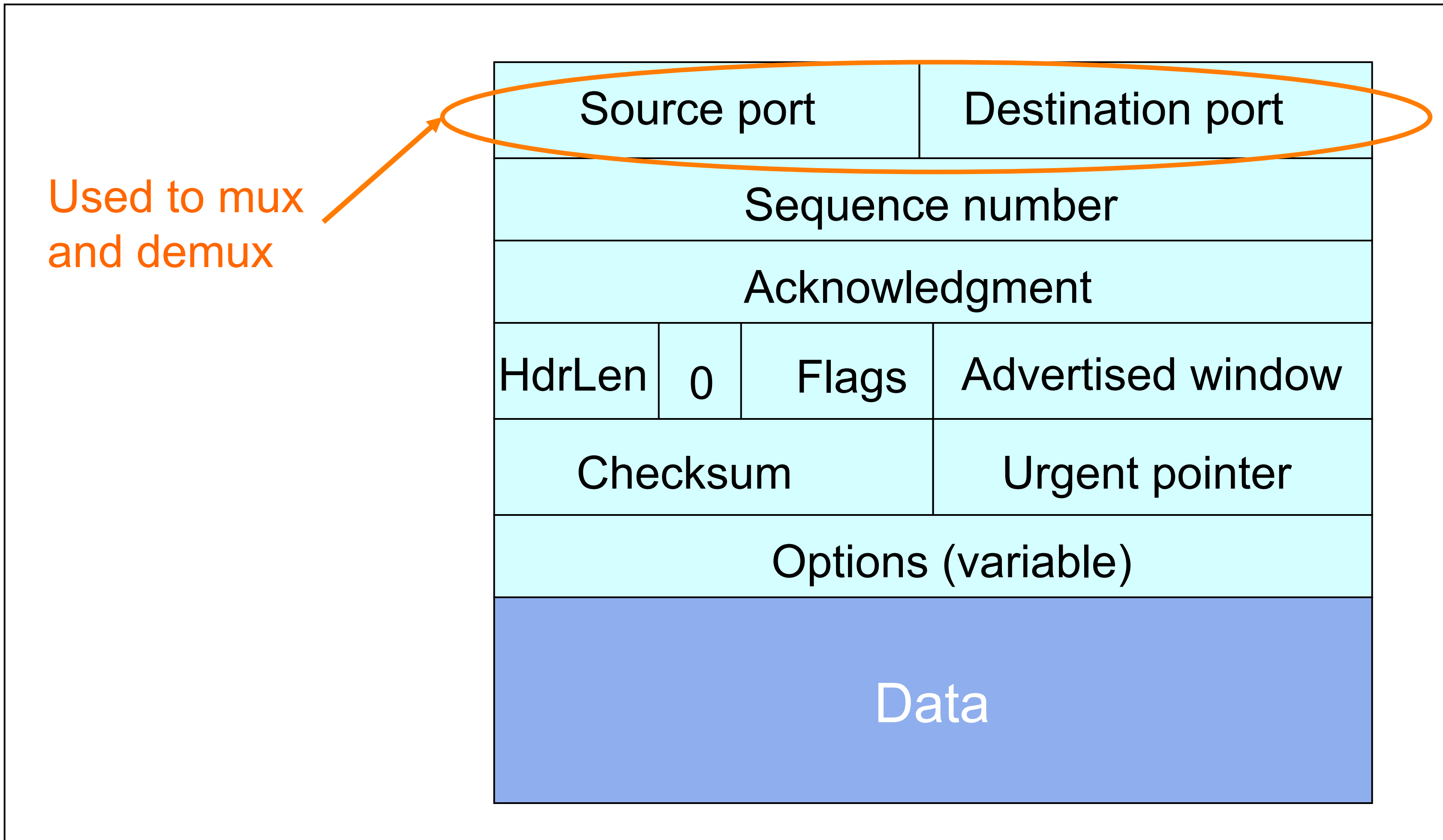
What Will We Cover Today?

- How TCP supports reliability

- TCP is not a perfect design
  - Probably wouldn’t make exactly same choices today

- But it is good enough
  - And is a great example of sweating the details…
  - ..and just happens to carry most of your traffic

TCP Header


Used to mux
and demux
Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

Last time: Components of a solution for reliable transport

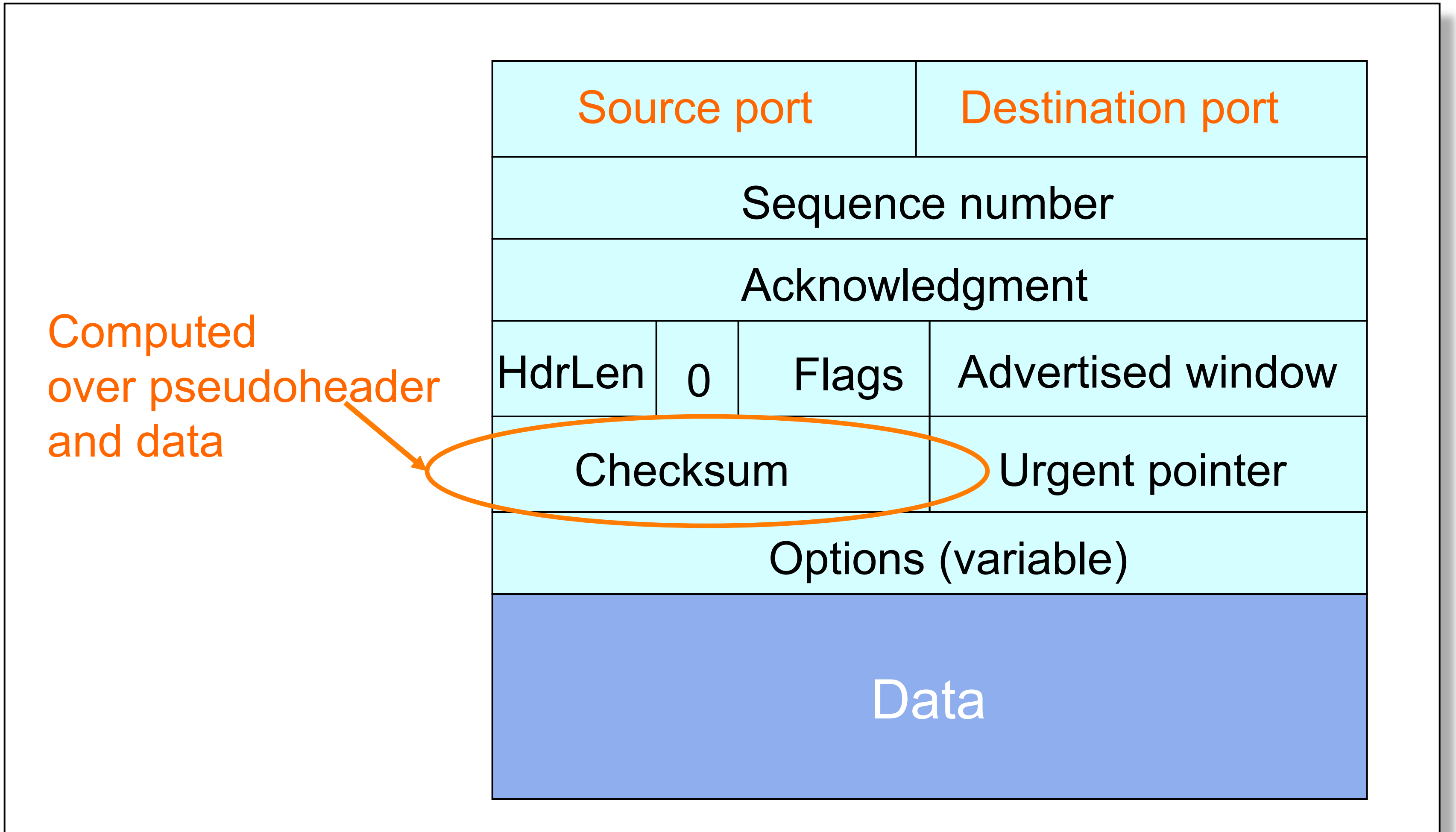
- Checksums (for error detection)
- Timers (for loss detection)
- Acknowledgments
  - cumulative
  - selective
- Sequence numbers (duplicates, windows)
- Sliding Windows (for efficiency)
- Retransmissions
  - Go-Back-N (GBN)
  - Selective Replay (SR)

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
Advertised window
Checksum
Urgent pointer
Options (variable)
Data
Computed
over pseudoheader
and data

What does TCP do?

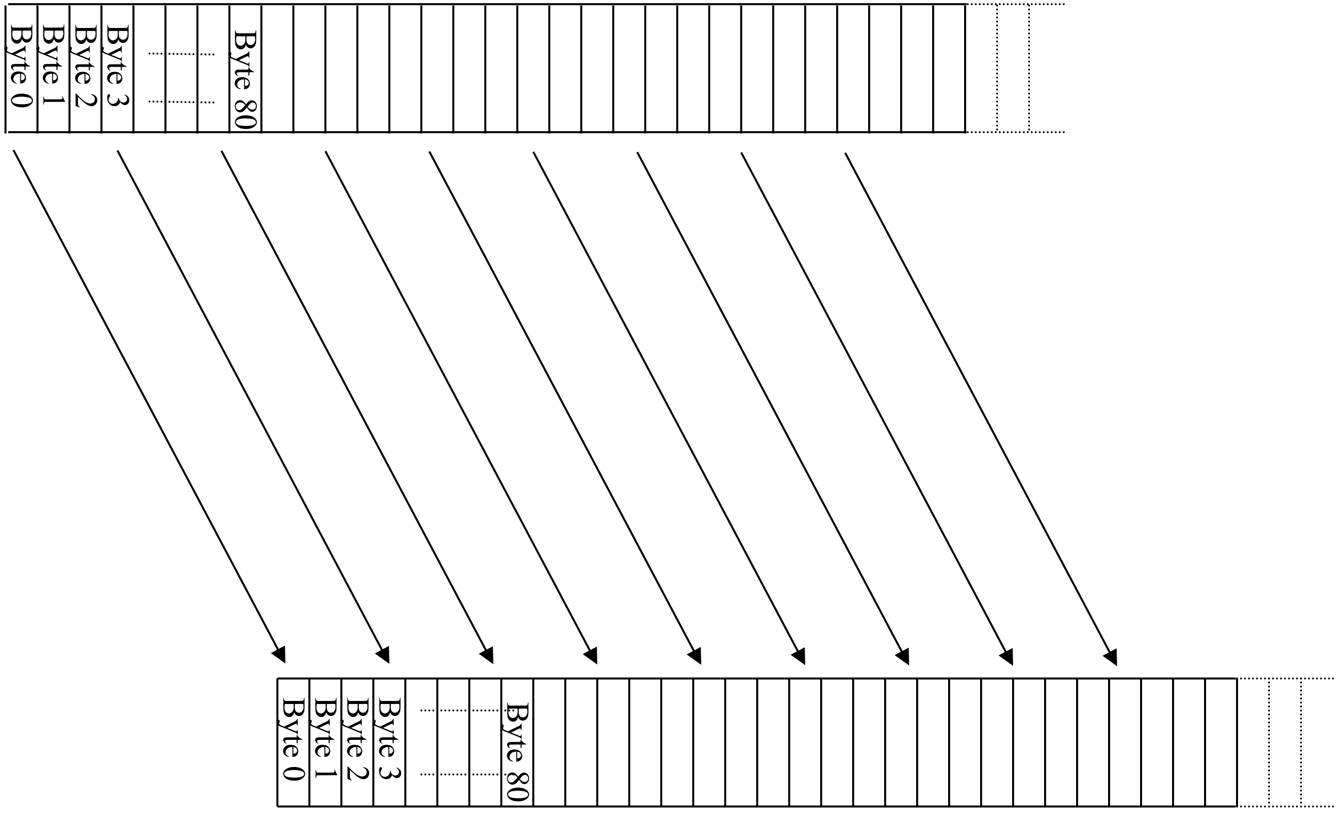
Many of our previous ideas, but some key differences

- Checksum
- **Sequence numbers are byte offsets**
  - And also includes notion of a “segment” and ISNs
  - Proof that networking is boring: 7 slides on sequence numbers!

TCP: Segments and Sequence Numbers

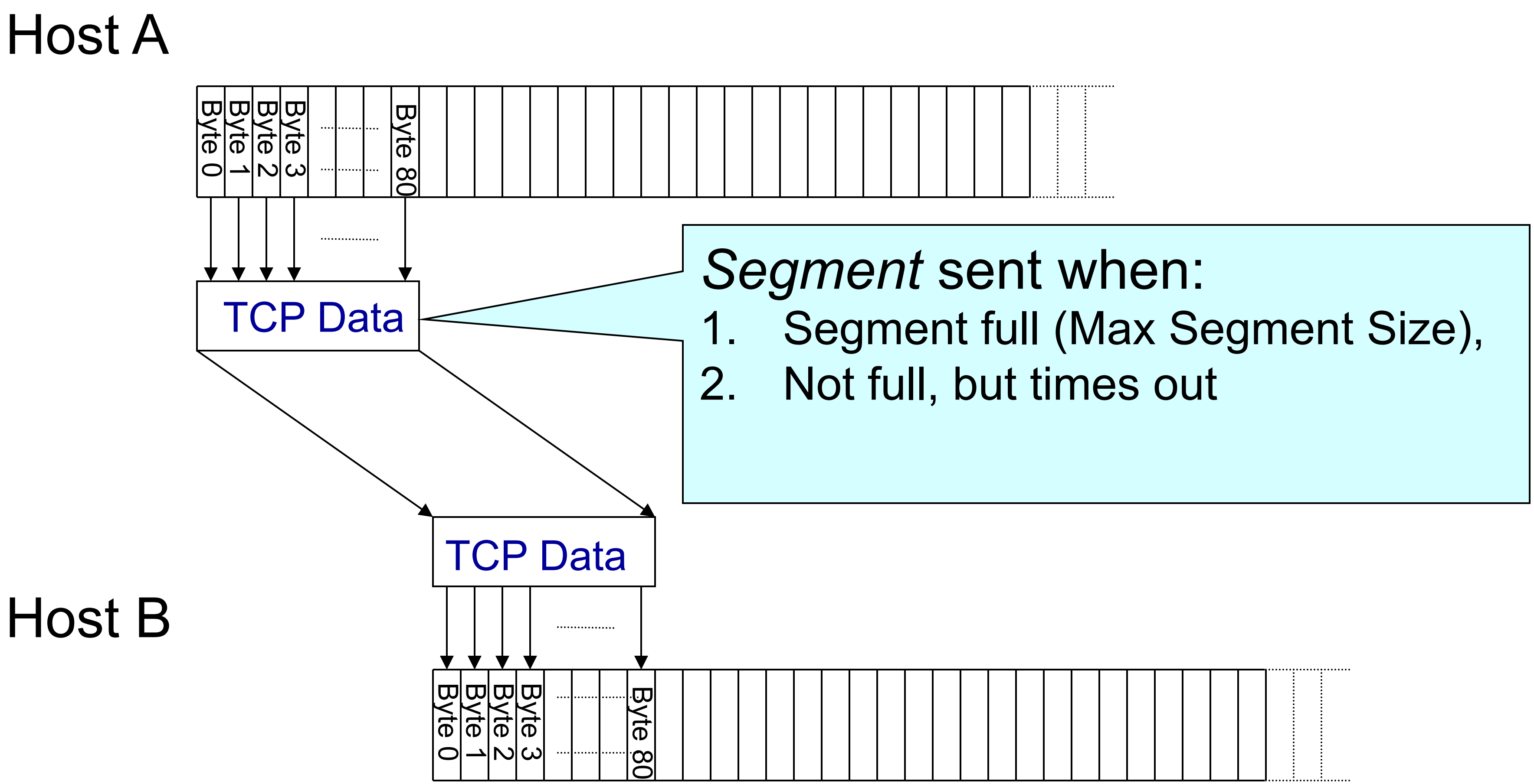
TCP “Stream of Bytes” Service…

Application @ Host A

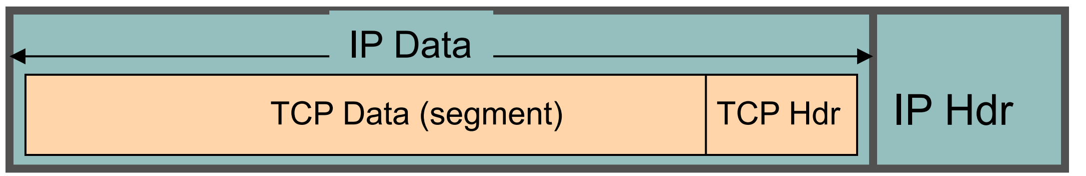


Application @ Host B

… Provided Using TCP "Segments"

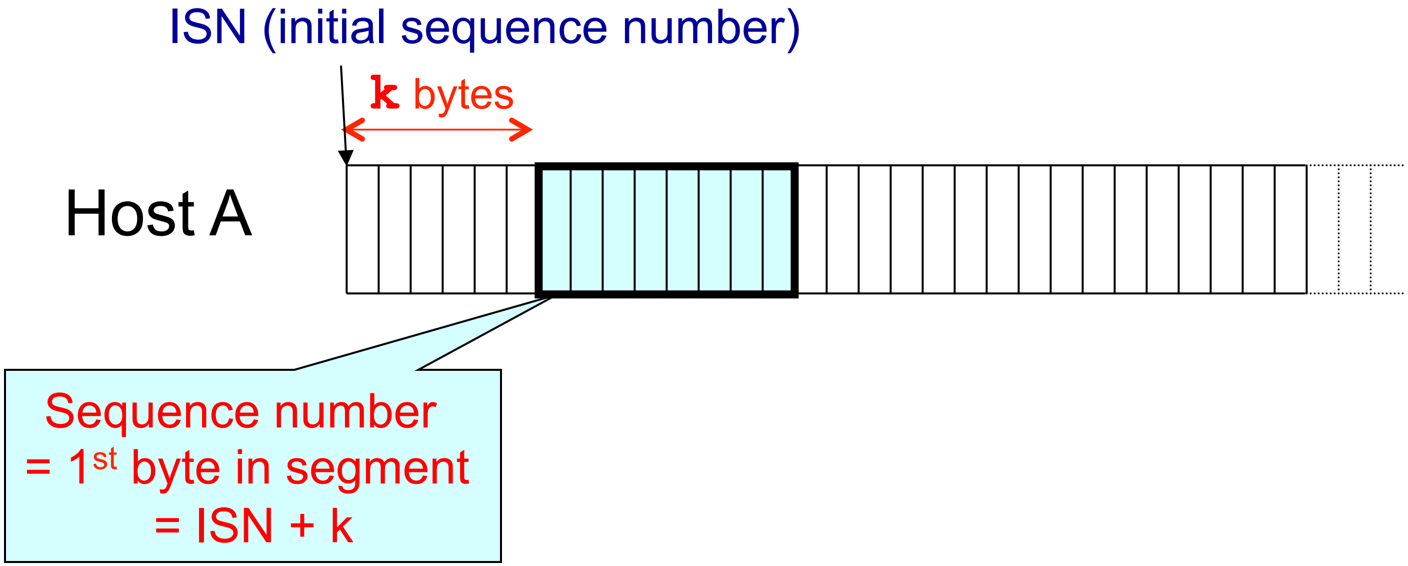


TCP Segment

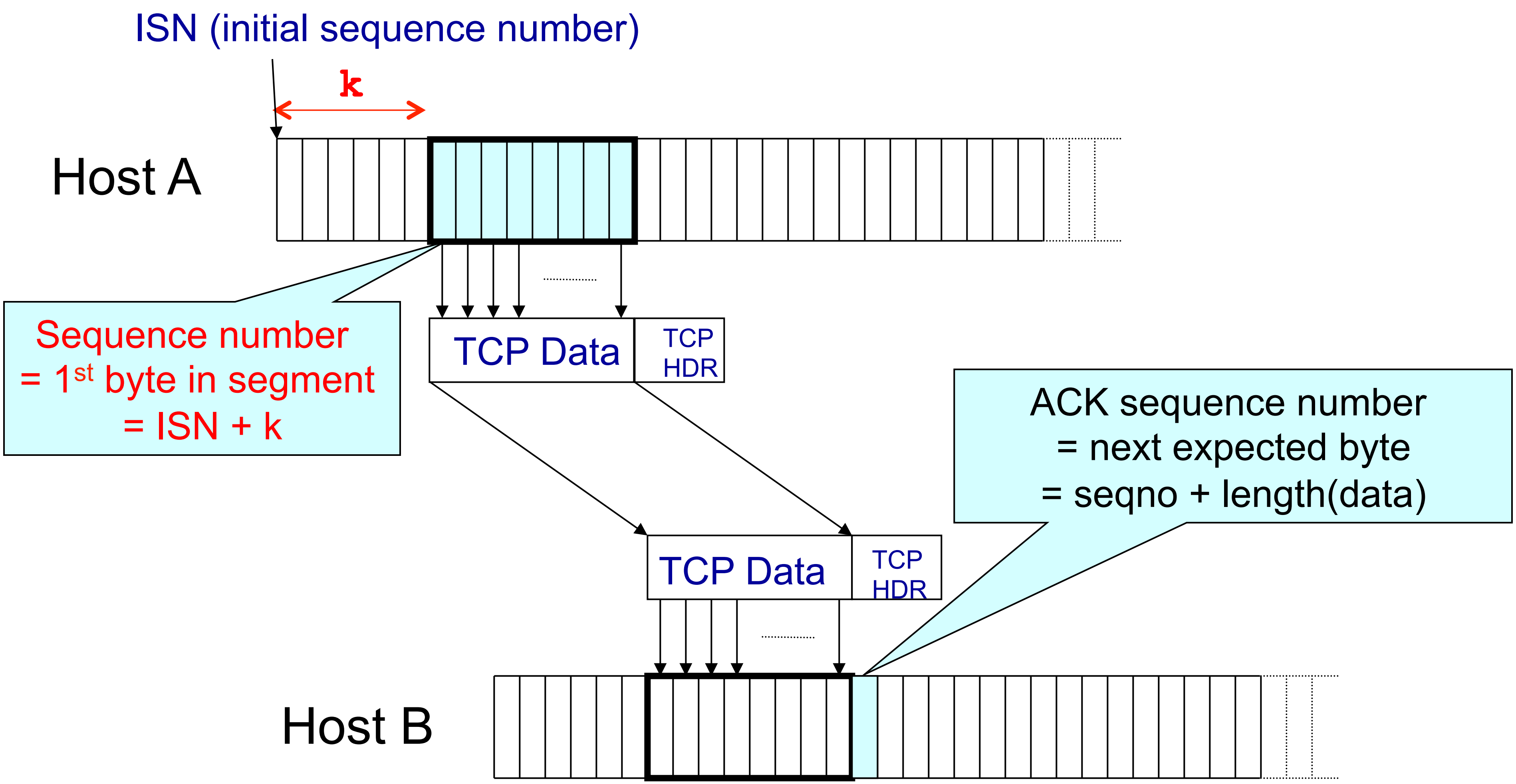


- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes with Ethernet
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header ≥ 20 bytes long
- TCP **segment**
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes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

Sequence Numbers

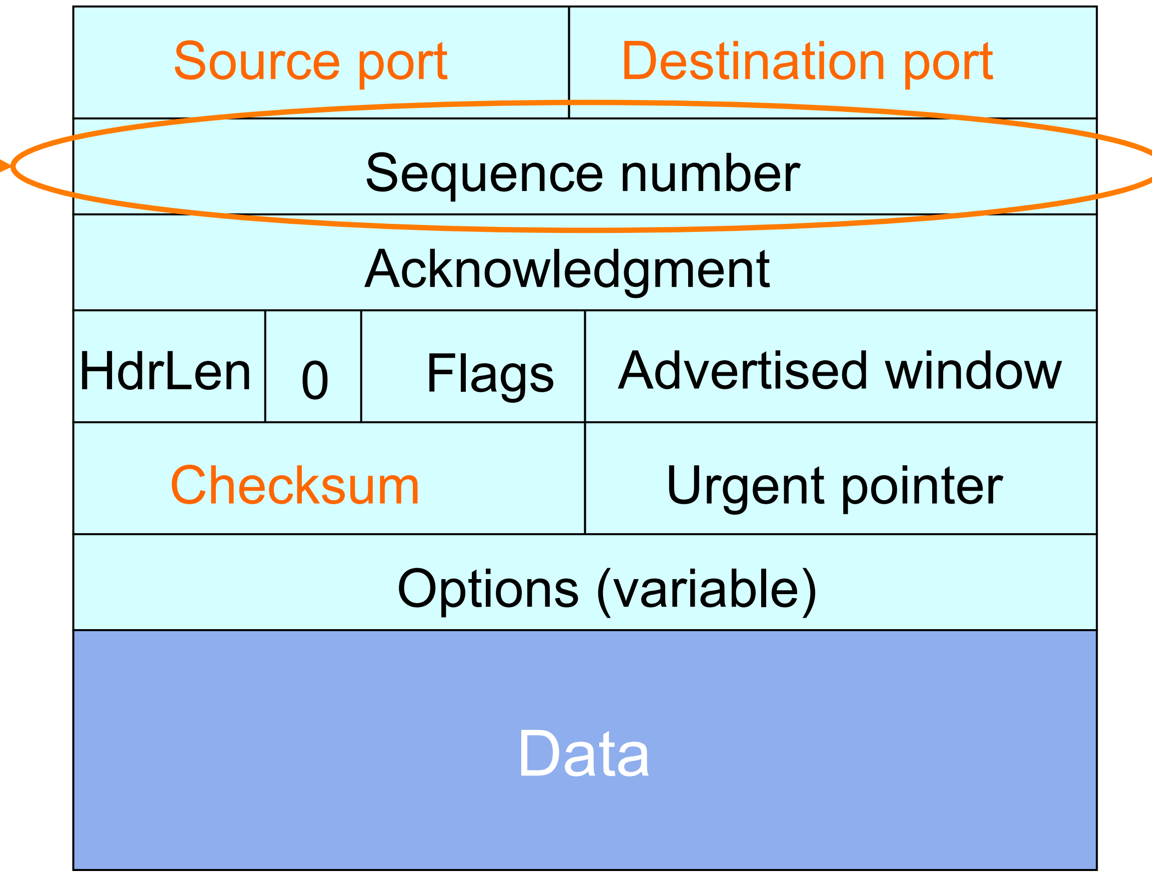


Sequence Numbers



TCP Header

Starting byte offset of data carried in this segment



What does TCP do?

Most of our previous tricks, but a few differences

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

ACKing and Sequence Numbers

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

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

Normal Pattern

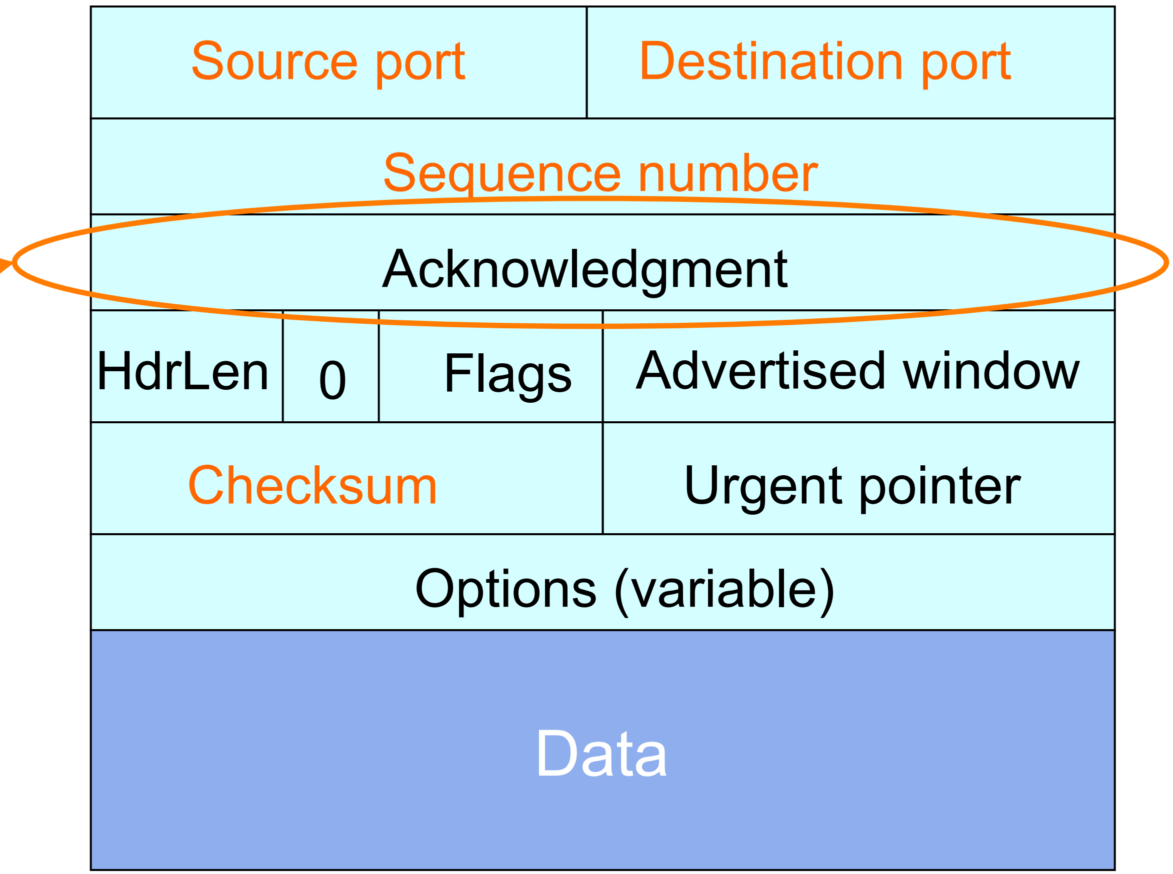
- Sender: seqno=X, length=B
- Receiver: ACK=X+B
- Sender: seqno=X+B, length=B
- Receiver: ACK=X+2B
- Sender: seqno=X+2B, length=B

- Seqno of next packet is same as last ACK field

TCP Header

Acknowledgment gives seqno just beyond highest seqno received **in order**

("What Byte is Next")



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 100B and seqnos.:
  - 100, 200, 300, 400, 500, 600, 700, 800, 900, …
- Assume the fifth packet (seqno 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 may not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission

Loss with cumulative ACKs

- "Duplicate ACKs" are a sign of an <u>isolated</u> 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
- But response to loss is trickier….

Loss with cumulative ACKs

- Two choices:
  - Send missing packet and move sliding window by the number of dup ACKs
    - Speeds up transmission, but might be wrong
  - Send missing packet, and wait for ACK to move sliding window
    - Is slowed down by single dropped packets

- Which should TCP do?

What does TCP do?

Most of our previous tricks, but a few differences

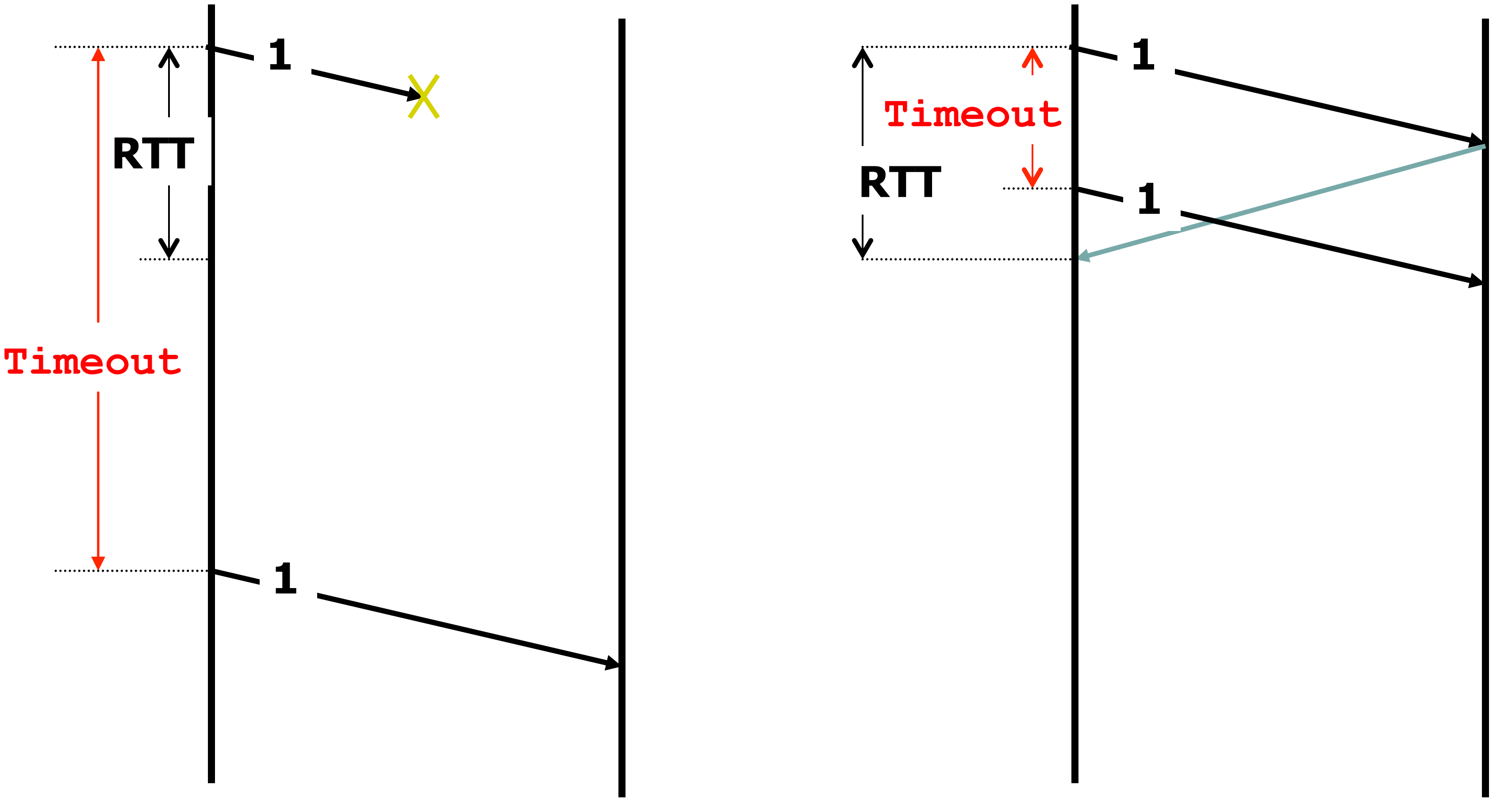
- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission
- Sender maintains a single retransmission timer (like GBN) and retransmits on timeout

Retransmission Timeout

- If the sender hasn’t received an ACK by timeout, retransmit the first packet in the window

- How do we pick a timeout value?

Timing Illustration



Timeout too long → inefficient

Timeout too short → duplicate packets

Retransmission Timeout

- If haven't received ack by timeout, retransmit the first packet in the window
- How to set timeout?
  - Too long: connection has low throughput
  - Too short: retransmit packet that was just delayed
- Solution: make timeout proportional to RTT
- But how do we measure RTT?

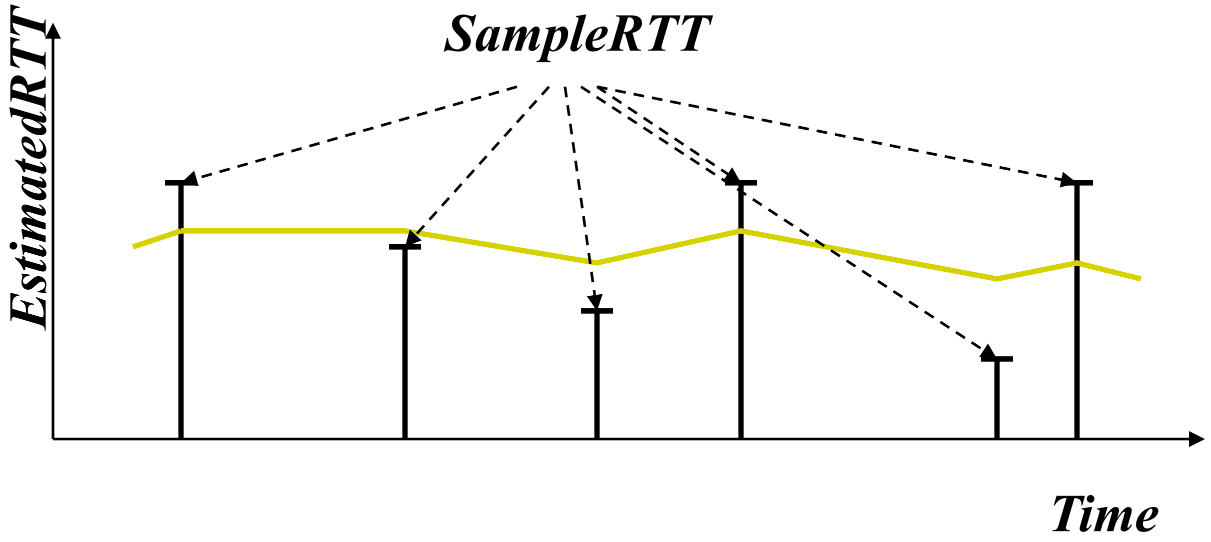
RTT Estimation

- Use exponential averaging of RTT samples

$$SampleRTT = AckRcvdTime - SendPacketTime$$

$$EstimatedRTT = \alpha \times EstimatedRTT + (1 - \alpha) \times SampleRTT$$

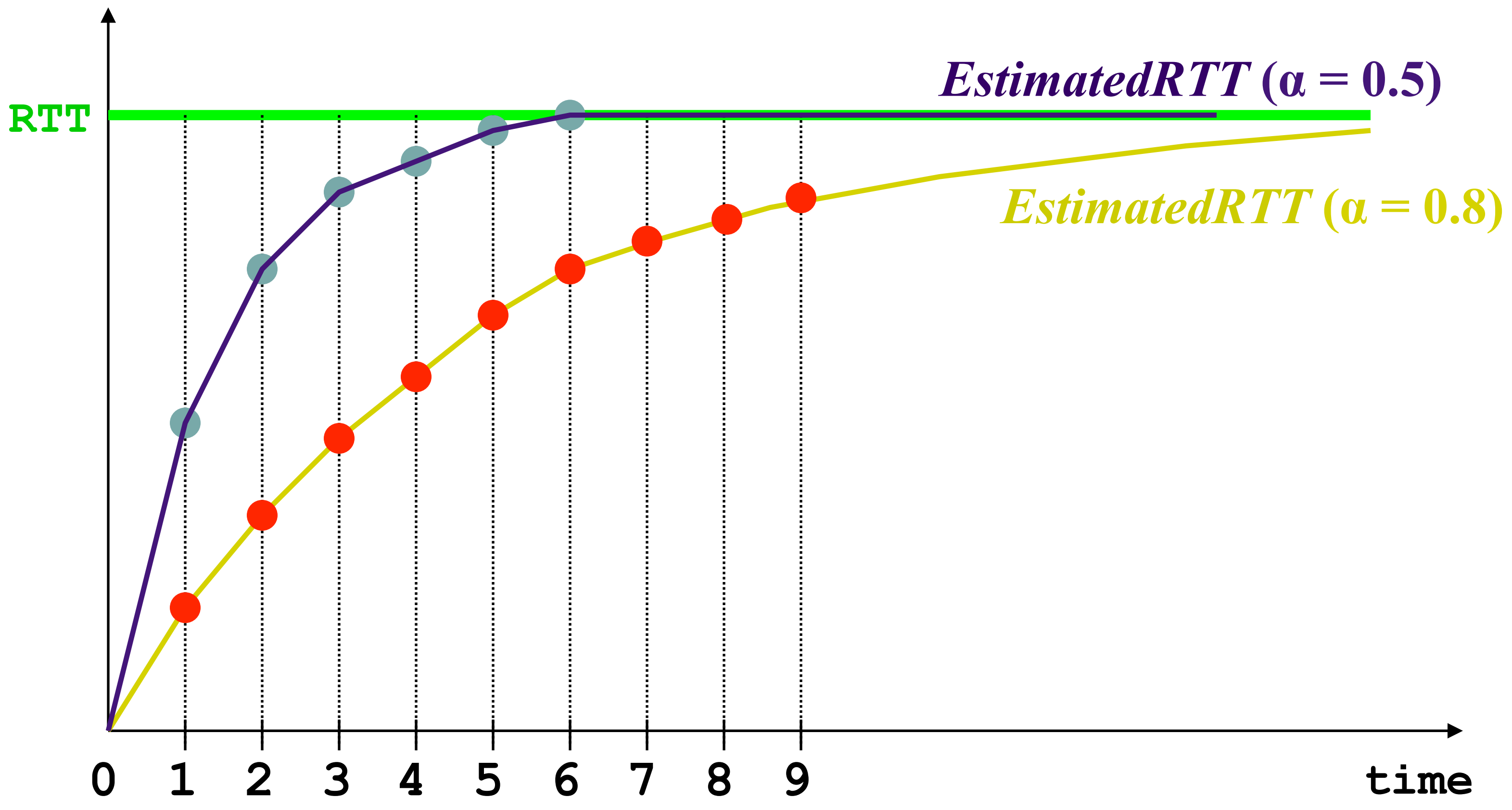
$$0 < \alpha \leq 1$$



Exponential Averaging Example

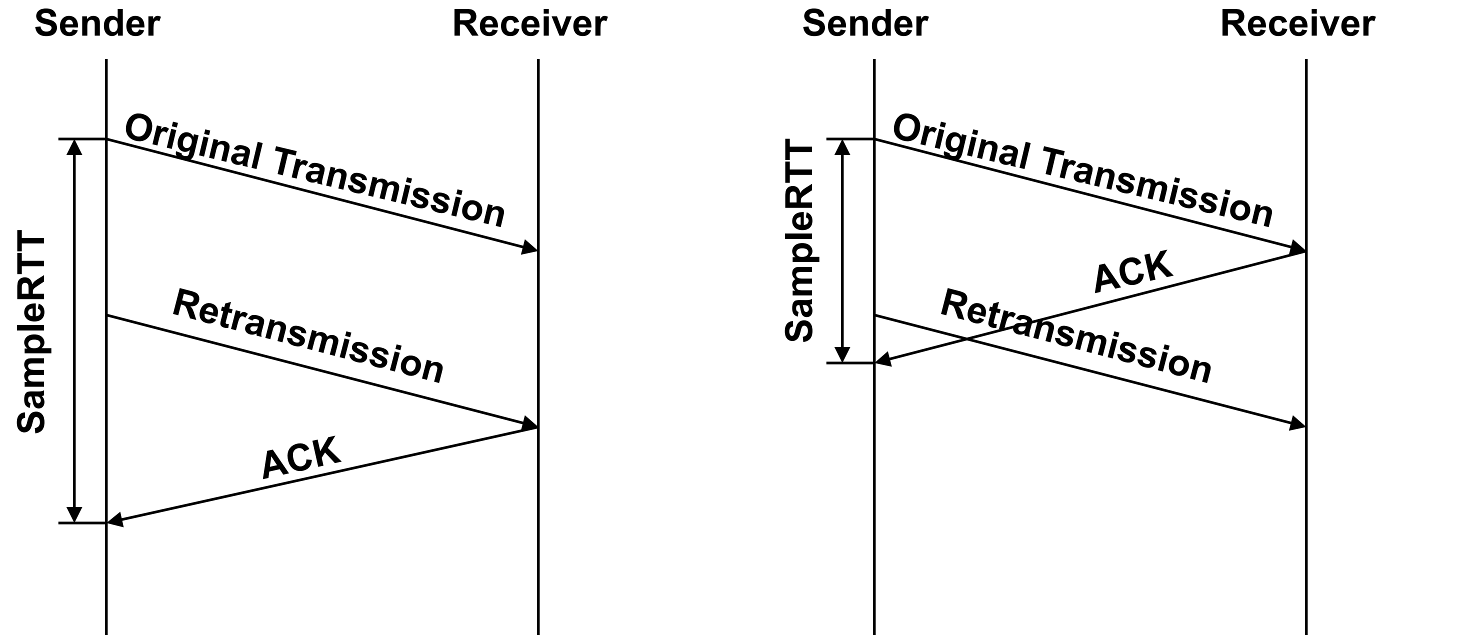
EstimatedRTT = α**EstimatedRTT* + (1 – α)**SampleRTT*

Assume RTT is constant → *SampleRTT* = RTT



Problem: Ambiguous Measurements

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



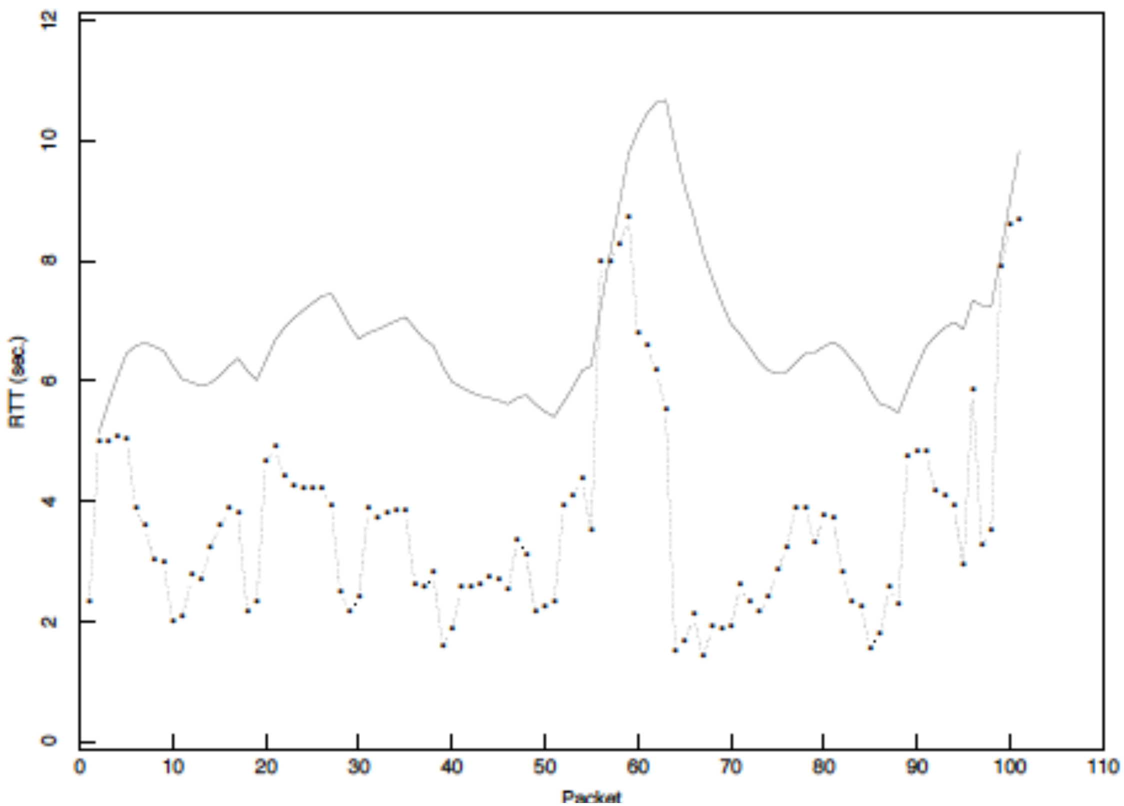
Karn/Partridge Algorithm

- Measure *SampleRTT* only for original transmissions
  - Once a segment has been retransmitted, do not use it for any further measurements
- Computes EstimatedRTT using $\alpha = 0.875$

- Timeout value (RTO) = 2 × EstimatedRTT
- Employs exponential backoff
  - Every time RTO timer expires, set RTO ← 2·RTO
  - (Up to maximum ≥ 60 sec)
  - Every time new measurement comes in (= successful original transmission), collapse RTO back to 2 × EstimatedRTT

Karn/Partridge in action

Figure 5: Performance of an RFC793 retransmit timer



from Jacobson and Karels, SIGCOMM 1988

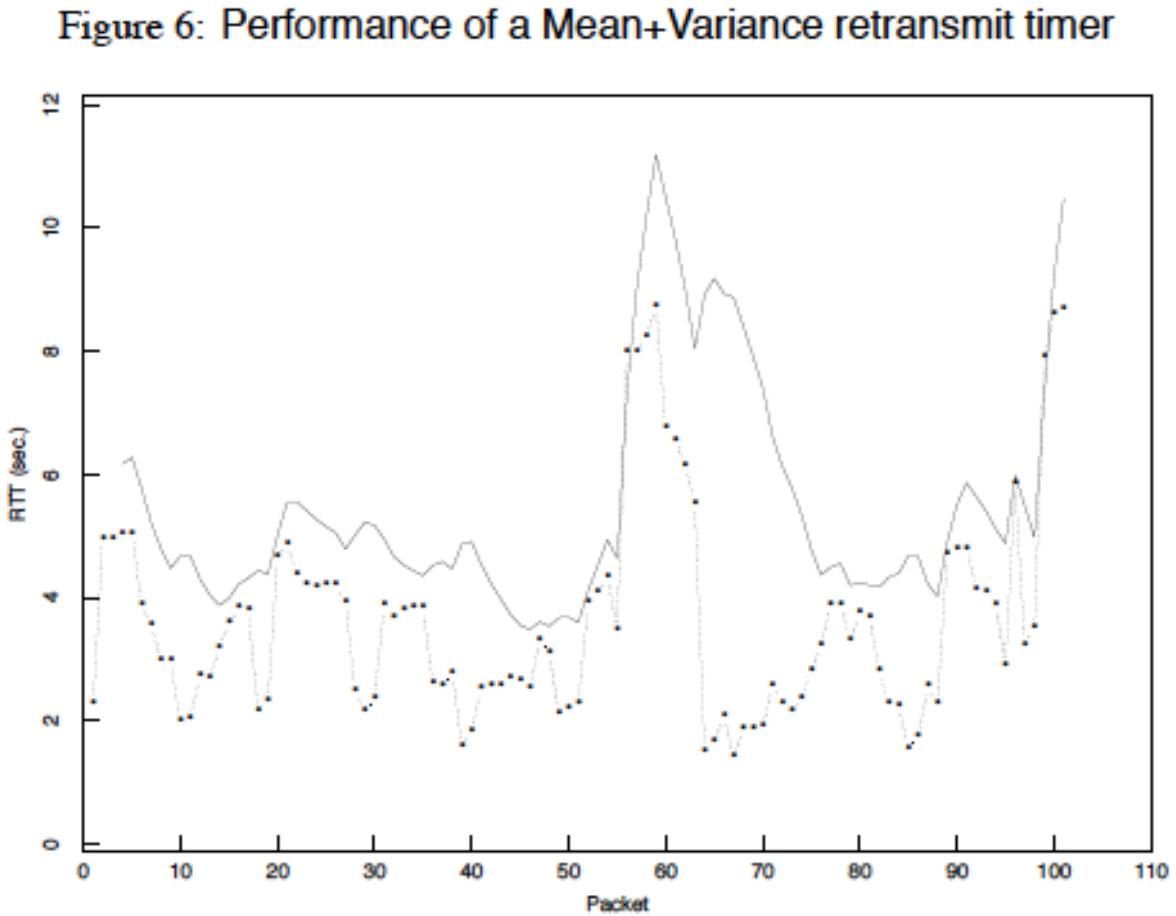
Jacobson/Karels Algorithm

- Problem: need to better capture variability in RTT
  - Directly measure deviation

- Deviation = | SampleRTT – EstimatedRTT |
- EstimatedDeviation: exponential average of Deviation

- RTO = EstimatedRTT + 4 x EstimatedDeviation

With Jacobson/Karels

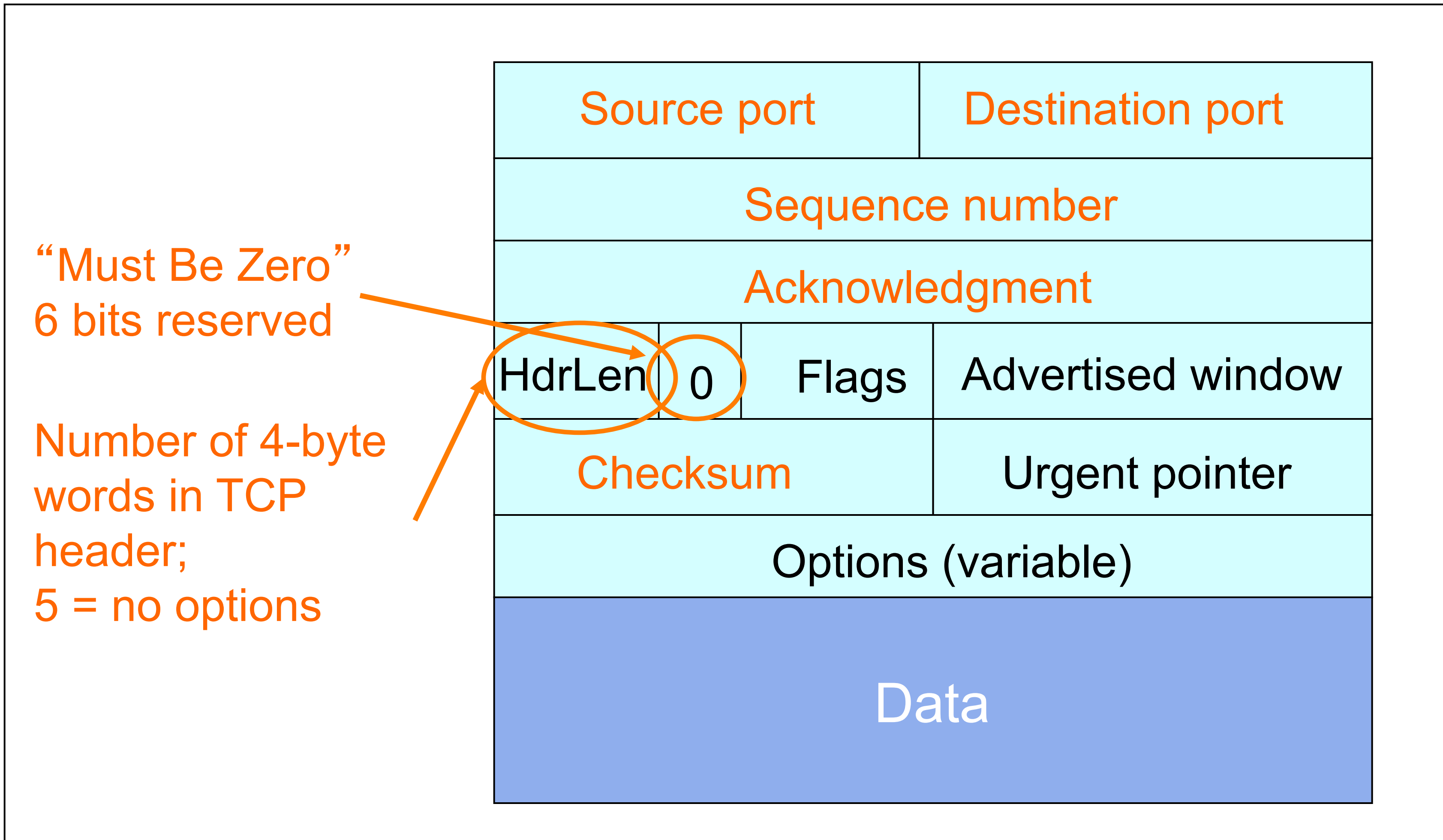


What does TCP do?

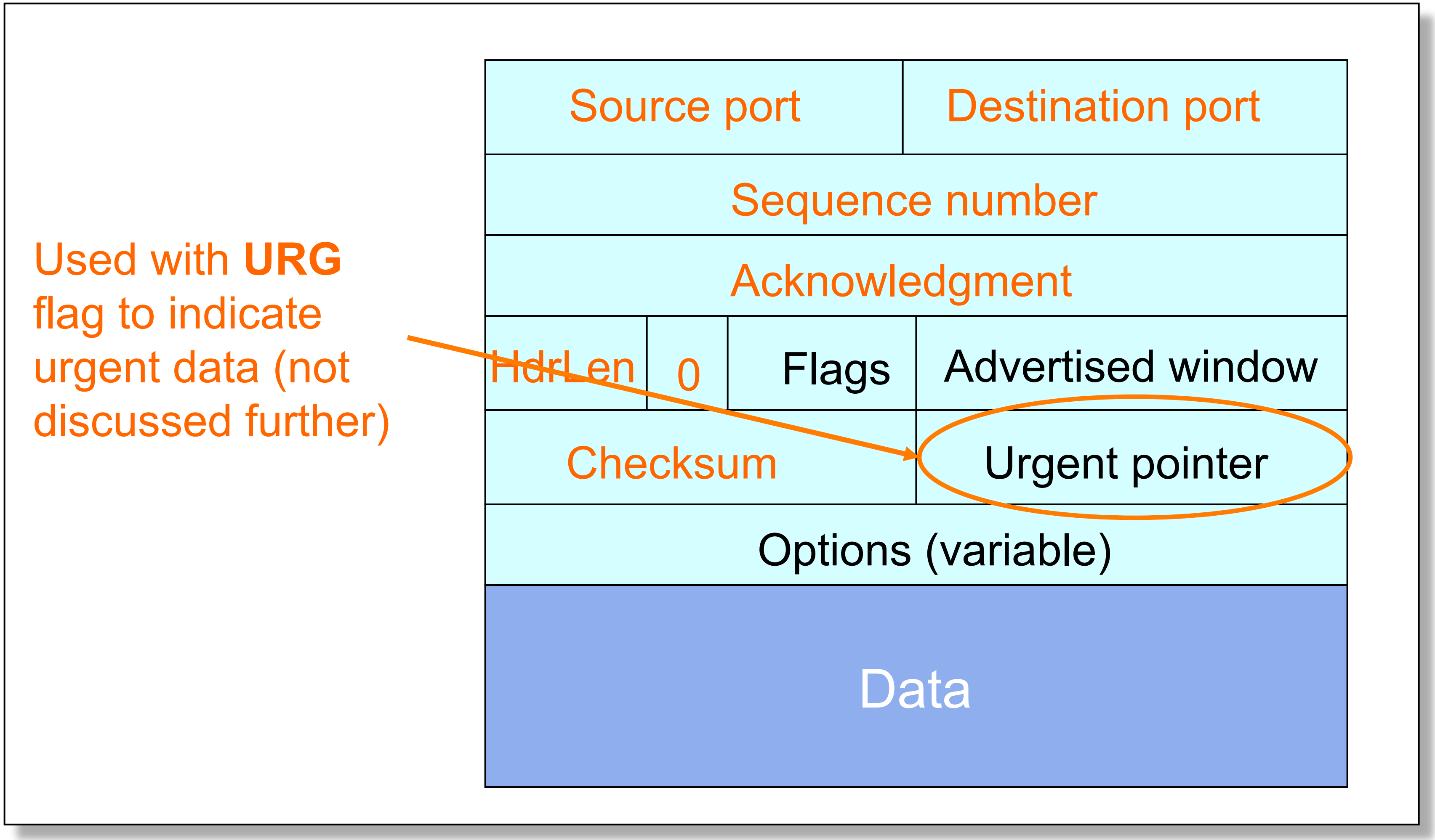
Most of our previous ideas, but some key differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission
- Sender maintains a single retransmission timer (like GBN) and retransmits on timeout

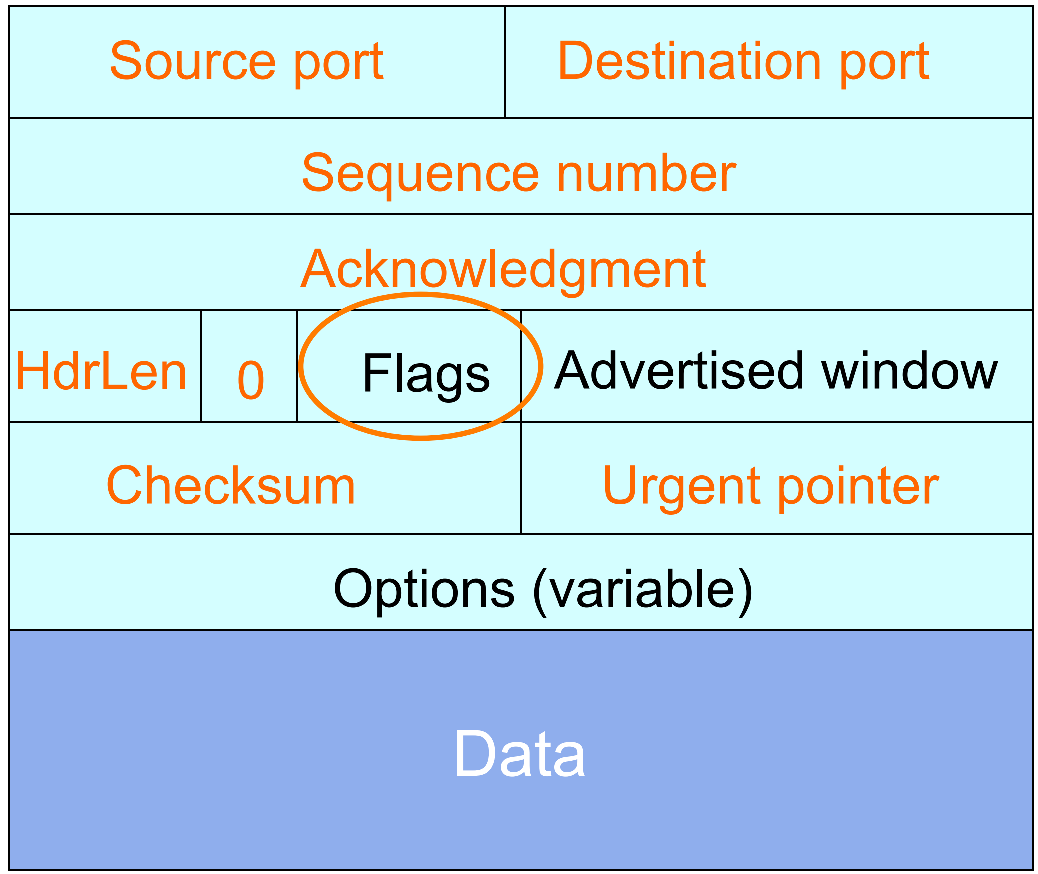
TCP Header: What's left?



TCP Header: What's left?



TCP Header: What's left?

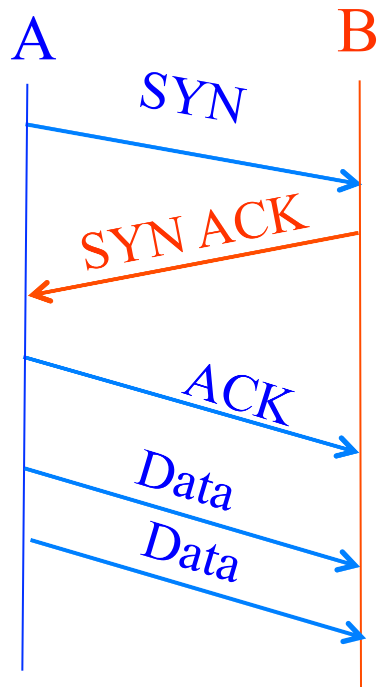


TCP Connection Establishment and Initial Sequence Numbers

Initial Sequence Number (ISN)

- Sequence number for the very first byte
- 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
  - Also, others might try to spoof your connection
  - ***Why does using ISN help?***
- TCP therefore requires changing ISN
- Hosts exchange ISNs when they establish a connection

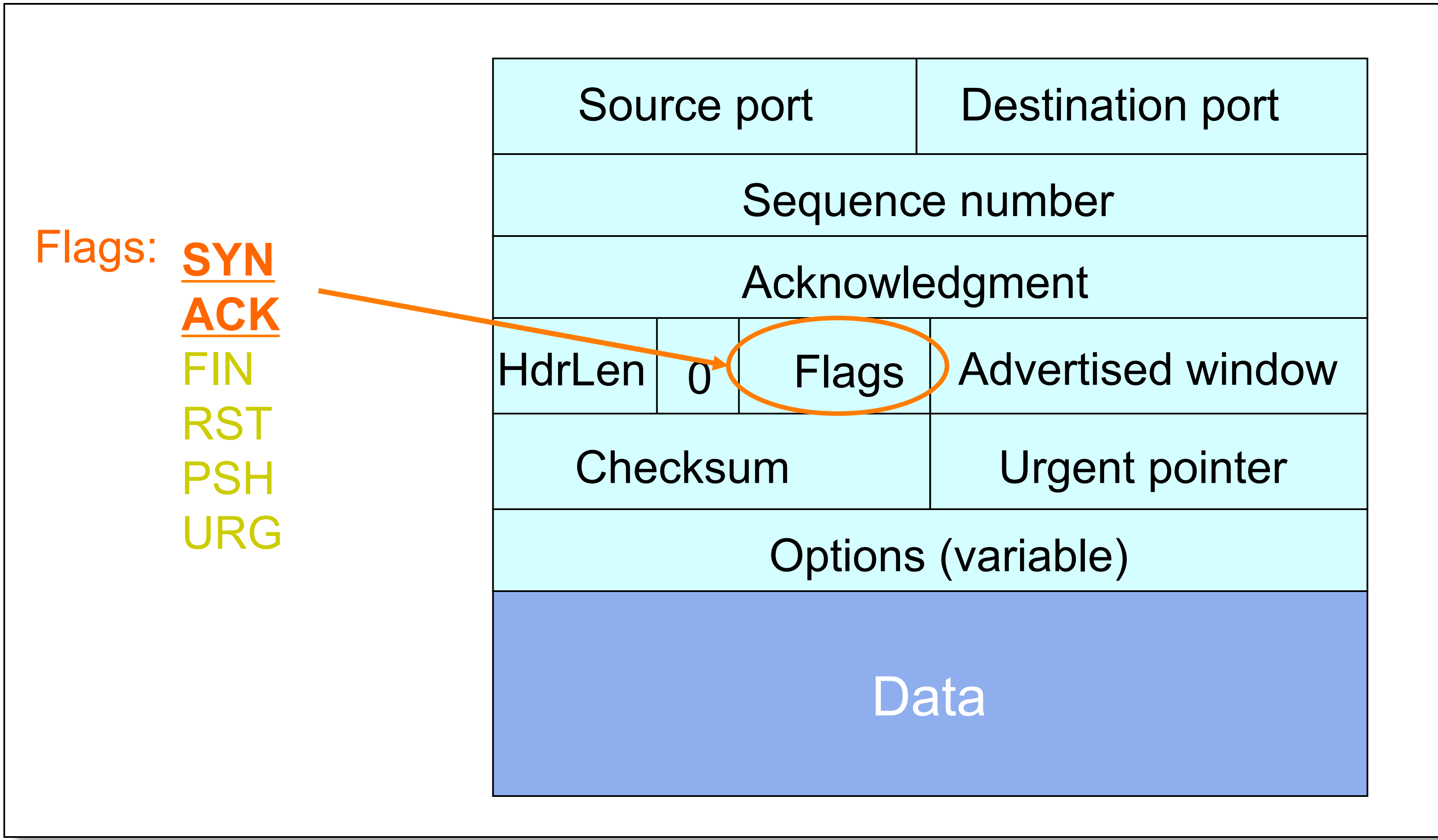
Establishing a TCP Connection



Each host tells its ISN to the other host.

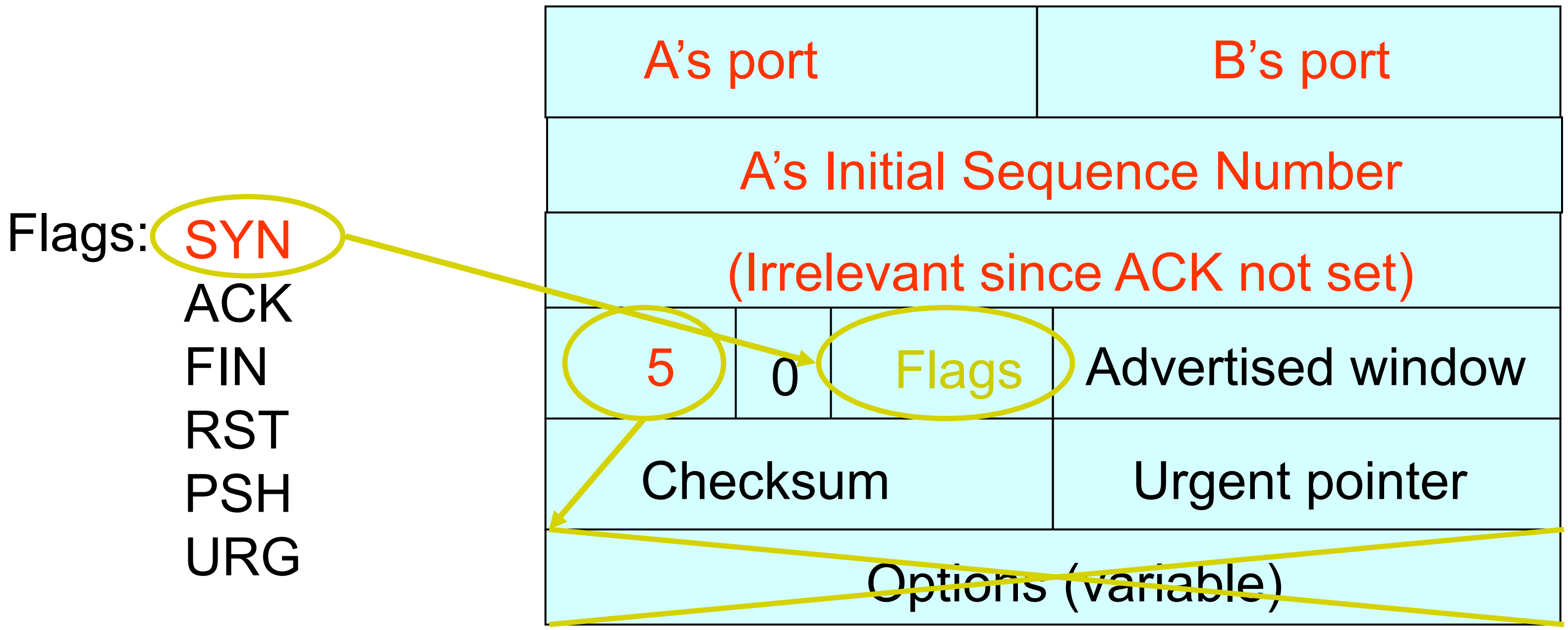
- Three-way handshake to establish connection
  - Host A sends a **SYN** (open; “synchronize sequence numbers”) to host B
  - Host B returns a SYN acknowledgment (**SYN ACK**)
  - Host A sends an **ACK** to acknowledge the SYN ACK

TCP Header



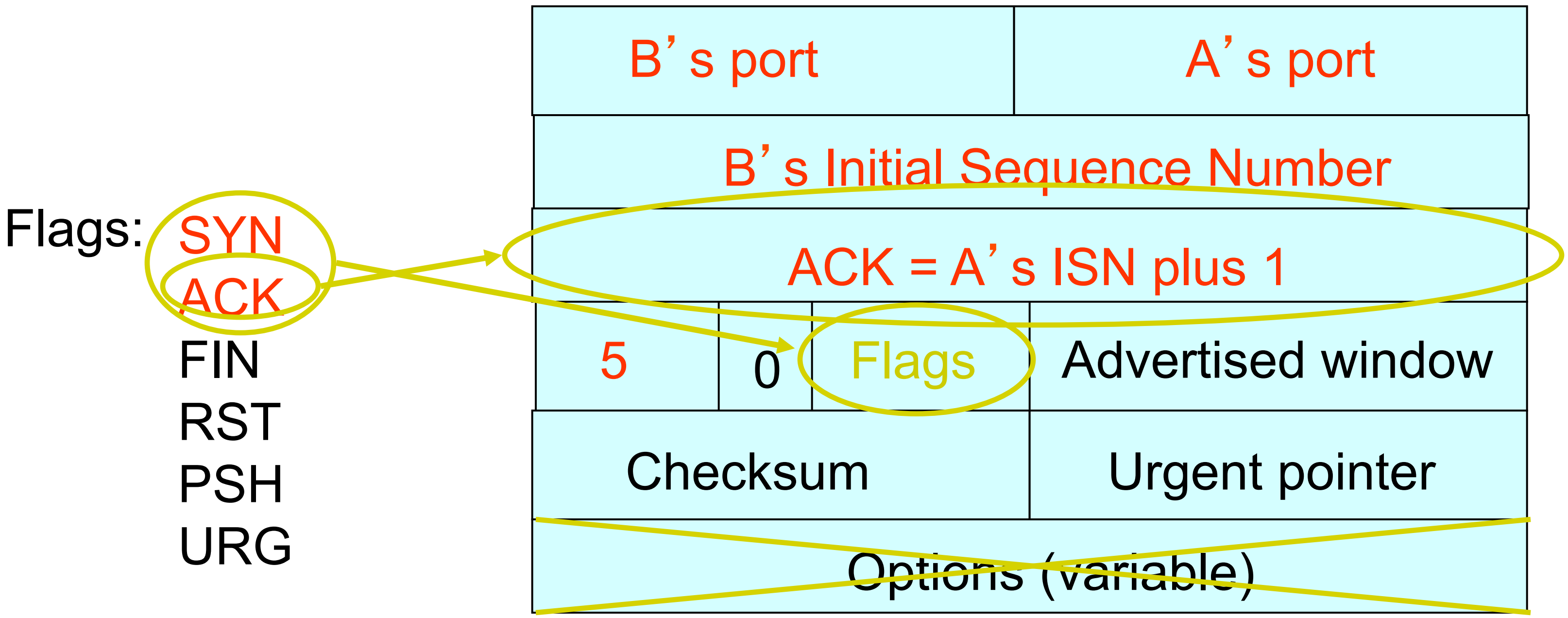
Flags:
SYN
ACK
FIN
RST
PSH
URG
Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

Step 1: A's Initial SYN Packet



A tells B it wants to open a connection…

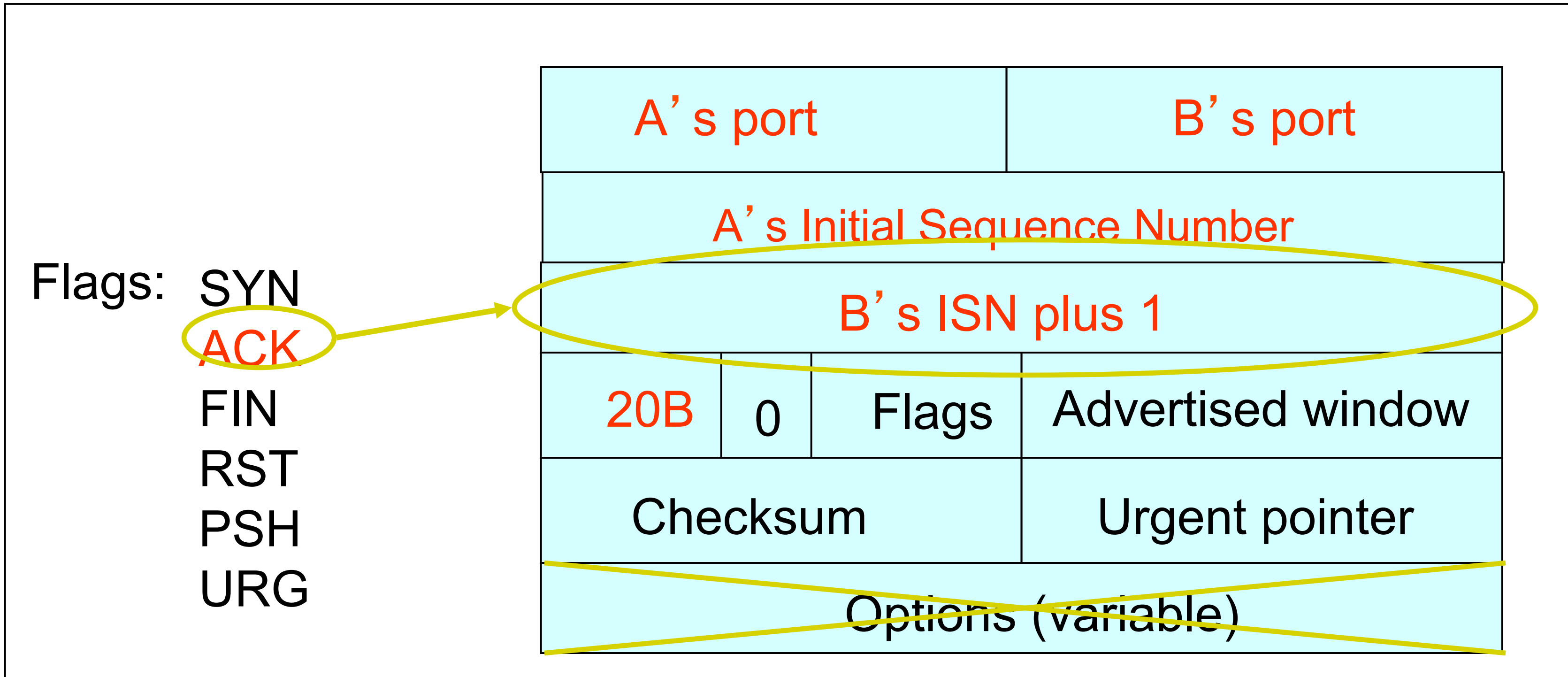
Step 2: B’s SYN-ACK Packet



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

… upon receiving this packet, A can start sending data

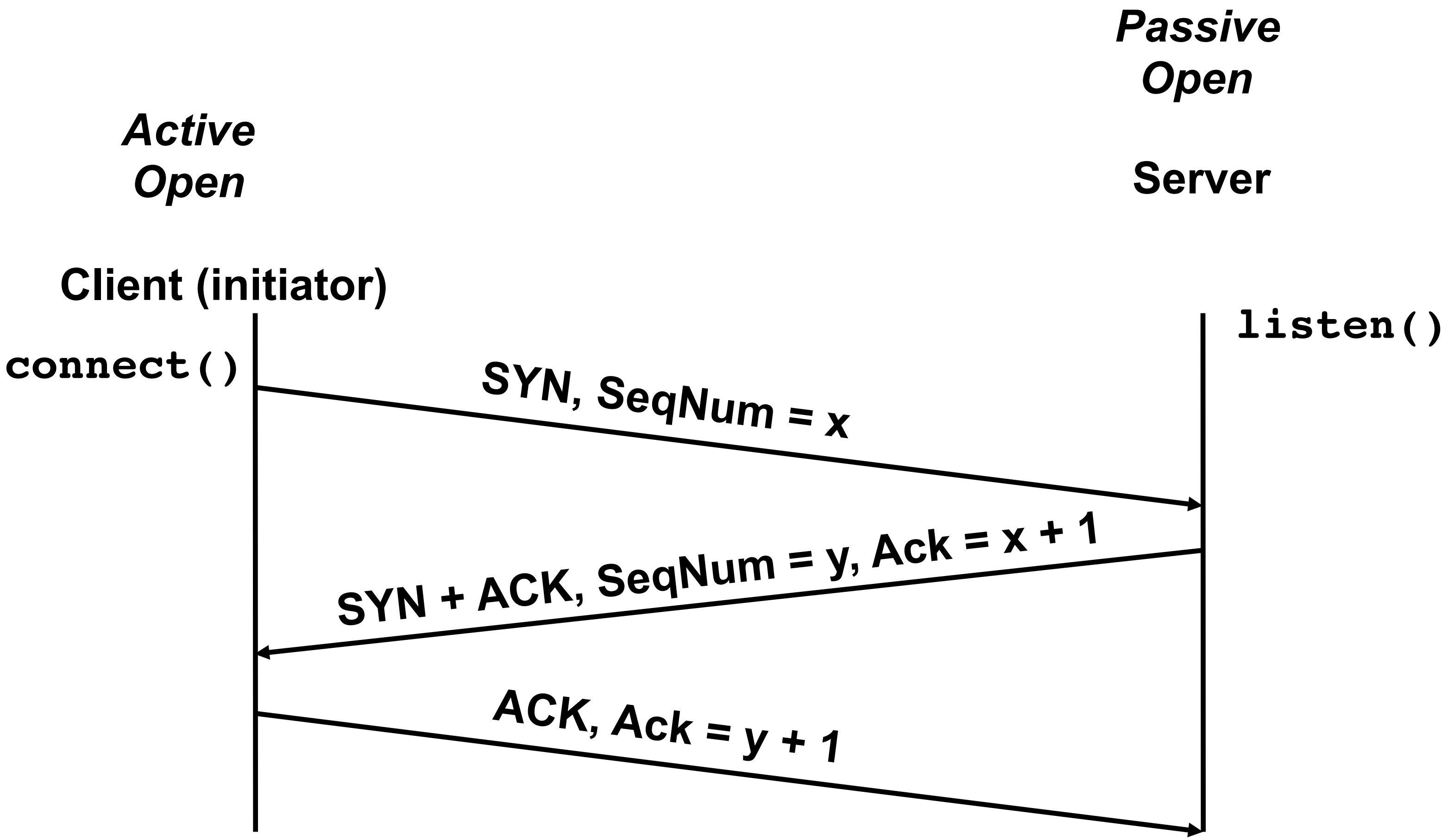
Step 3: A’s ACK of the SYN-ACK



A tells B it’s likewise okay to start sending

… upon receiving this packet, B can start sending data

Timing Diagram: 3-Way Handshaking



What if the SYN Packet Gets Lost?

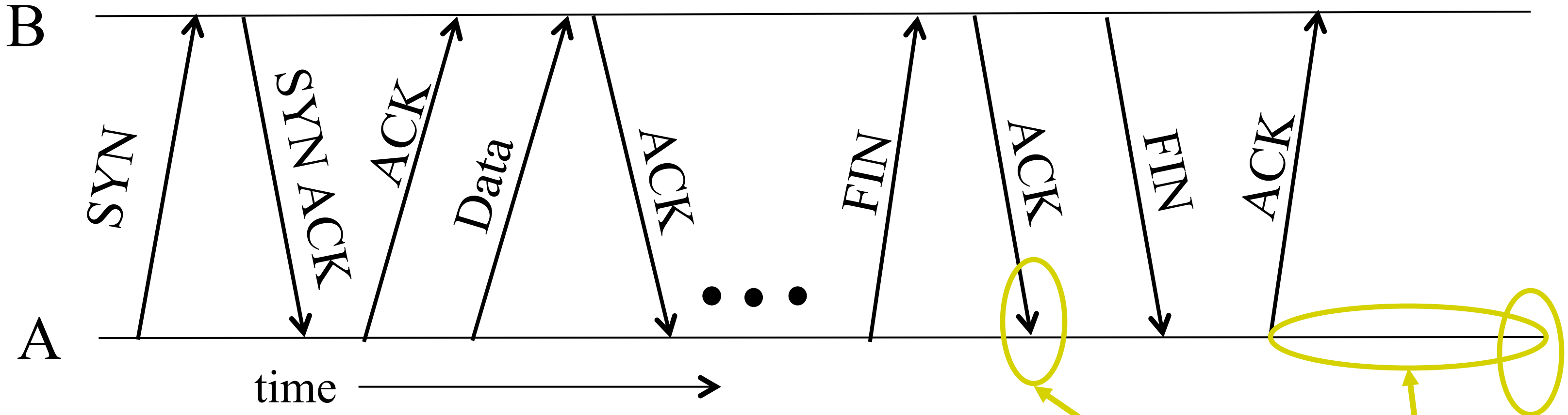
- Suppose the SYN packet gets lost
  - Packet is lost inside the network, or:
  - Server discards the packet (e.g., it's too busy)
- Eventually, no SYN-ACK arrives
  - Sender sets a timer and waits for the SYN-ACK
  - … and retransmits the SYN if needed
- How should the TCP sender set the timer?
  - 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 seconds
    - Some implementations instead use 6 seconds

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…
  - 3-6 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

Tearing Down the Connection

Normal Termination, One Side At A Time



- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space
- Other host acks the byte to confirm
- Closes A's side of the connection, but not B's
  - Until B likewise sends a **FIN**
  - Which A then acks

Connection now **closed**

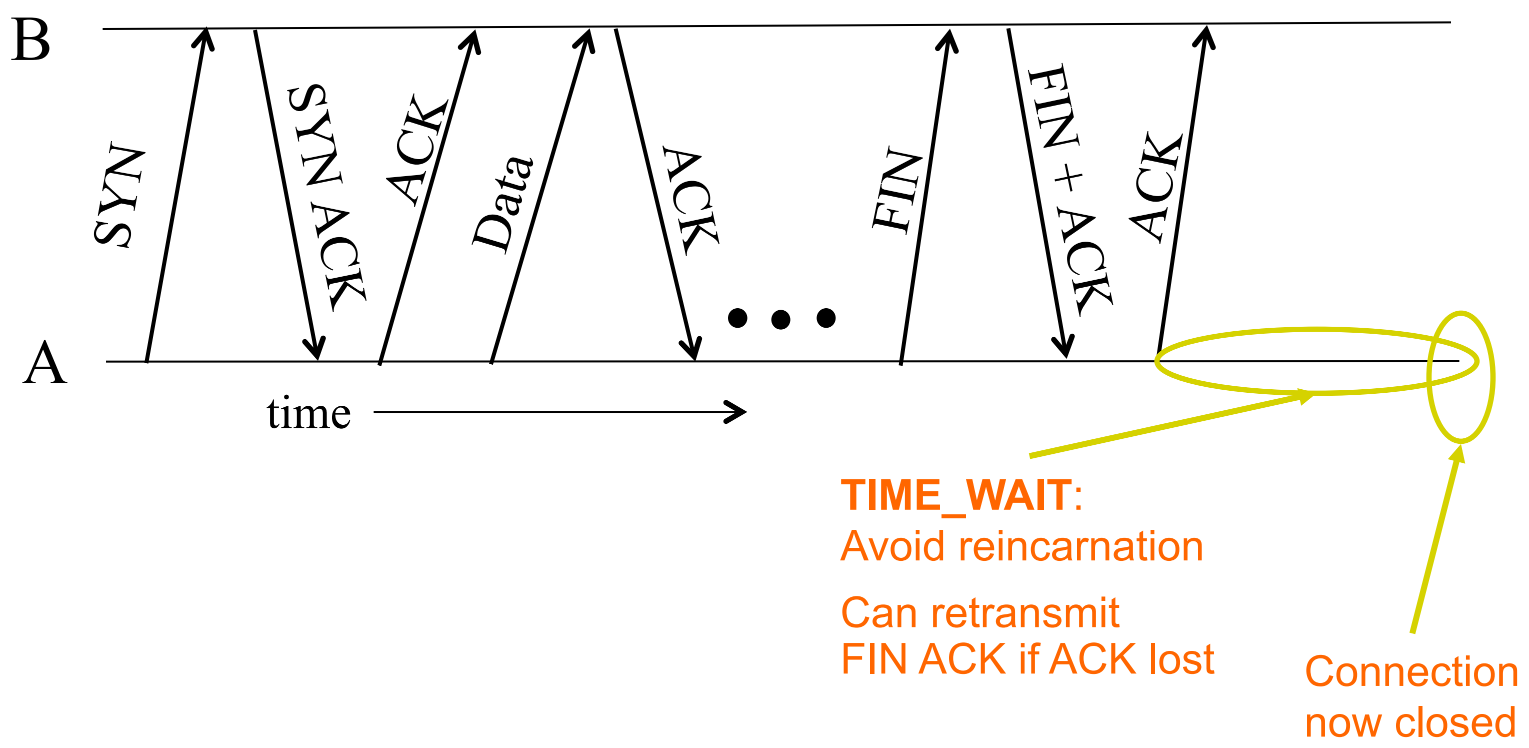
Connection now **half-closed**

TIME_WAIT:

Avoid reincarnation

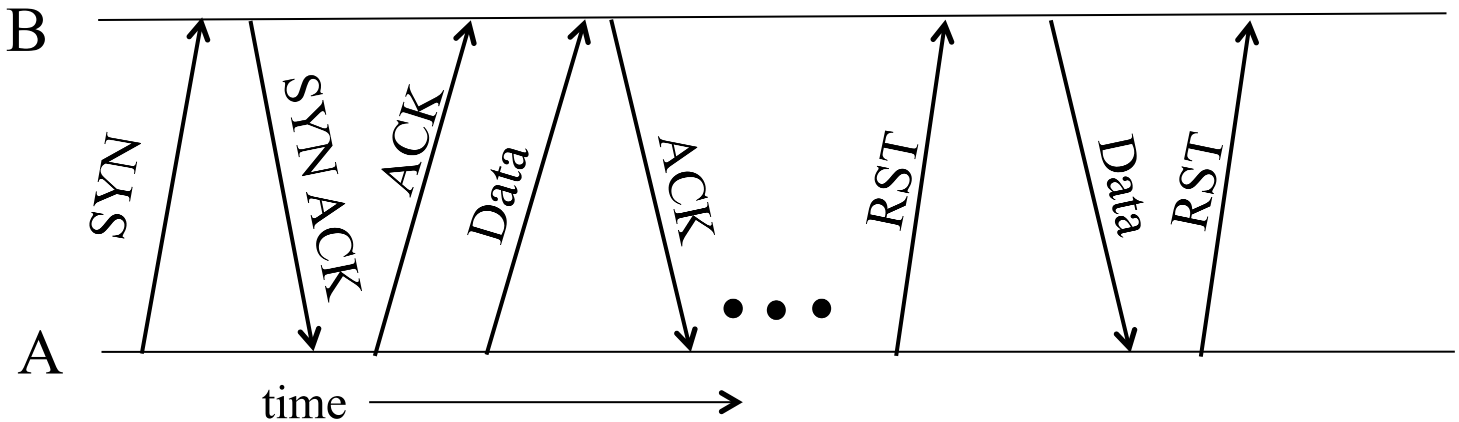
B will retransmit FIN if ACK is lost

Normal Termination, Both Together



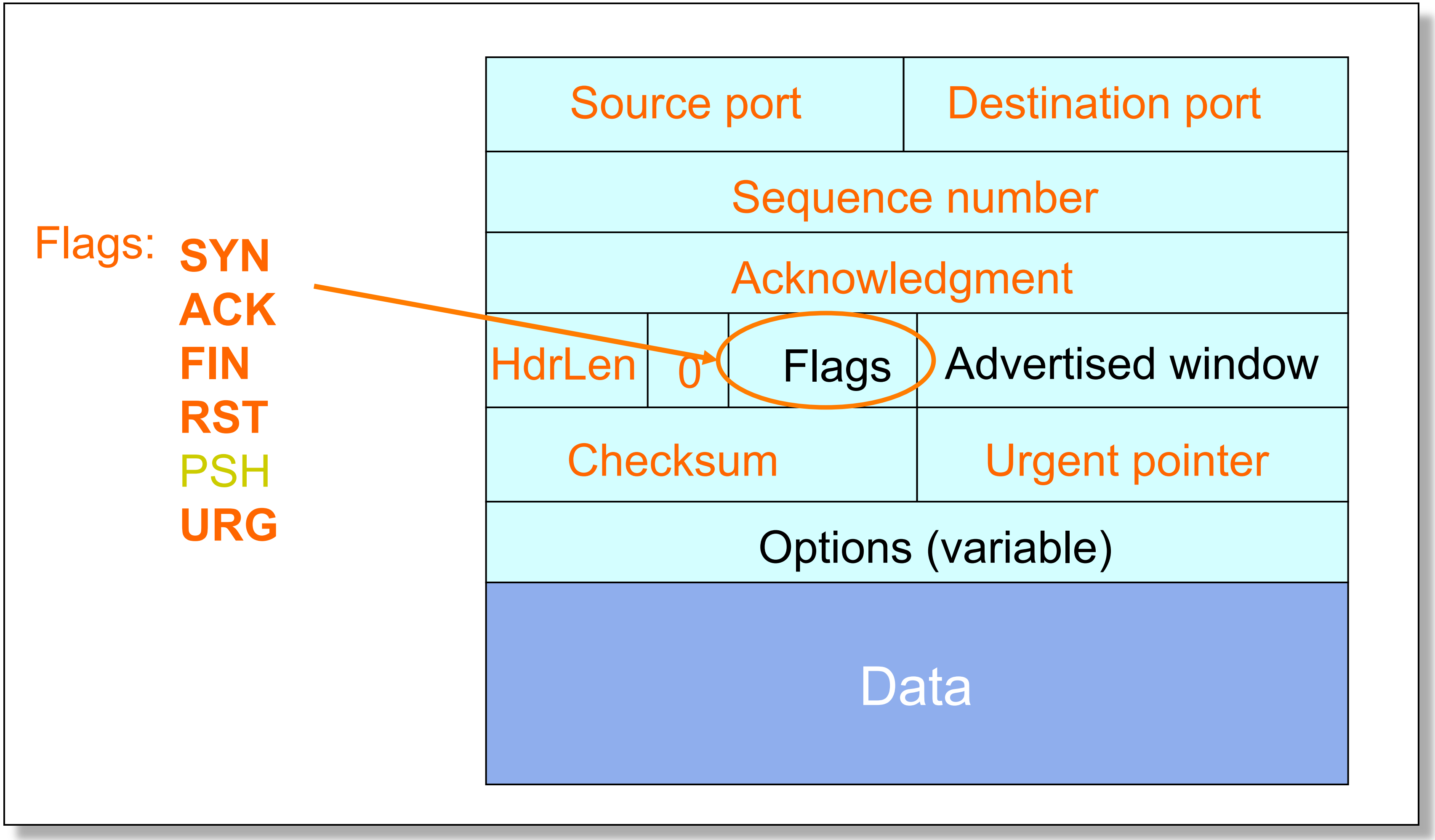
- Same as before, but B sets **FIN** with their ack of A's **FIN**

Abrupt Termination

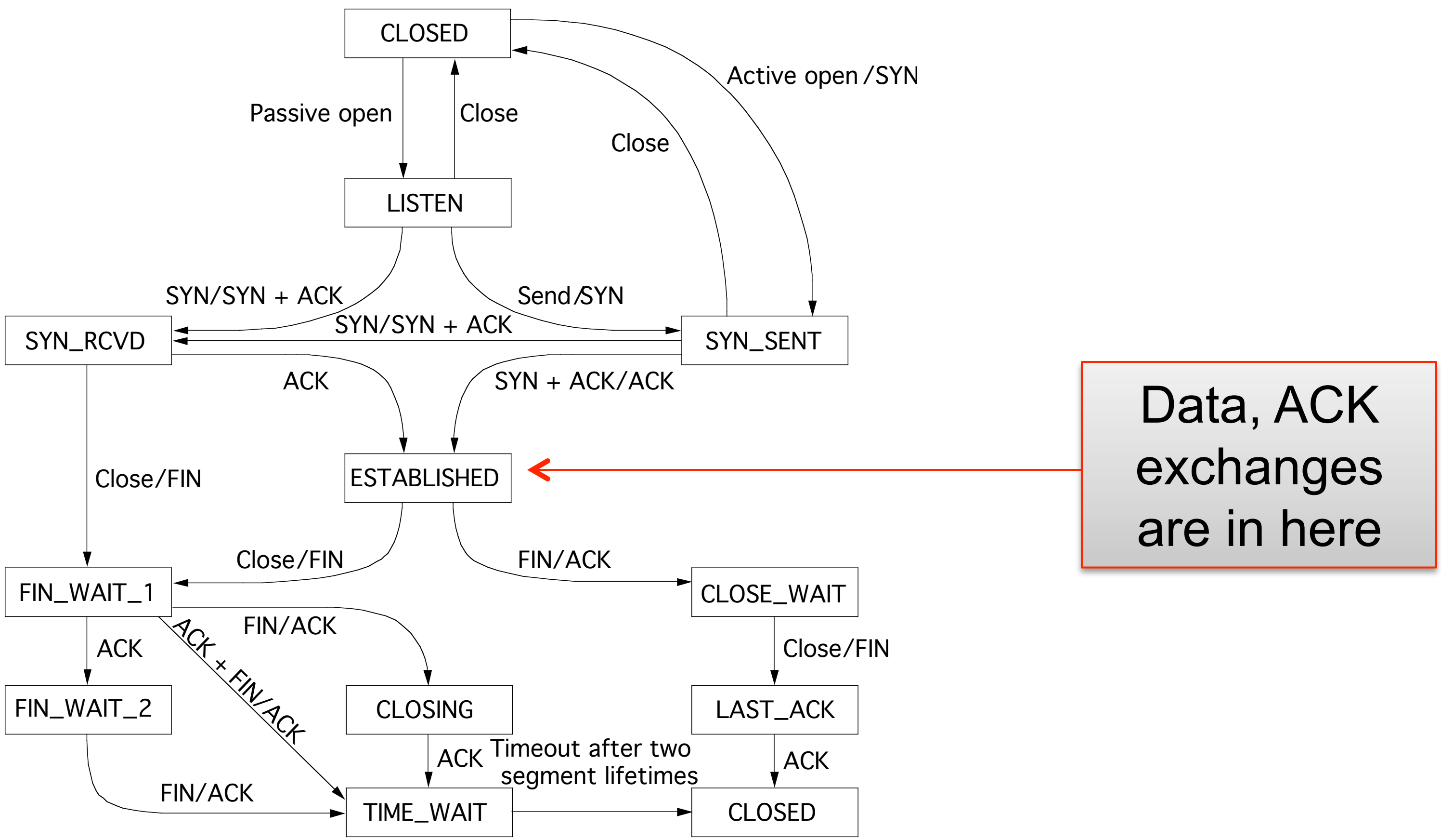


- A sends a RESET (**RST**) to B
  - E.g., because application process on A crashed
- That's it
  - B does not ack the **RST**
  - Thus, **RST** is not delivered reliably
  - And: any data in flight is lost
  - But: if B sends anything more, will elicit another **RST**

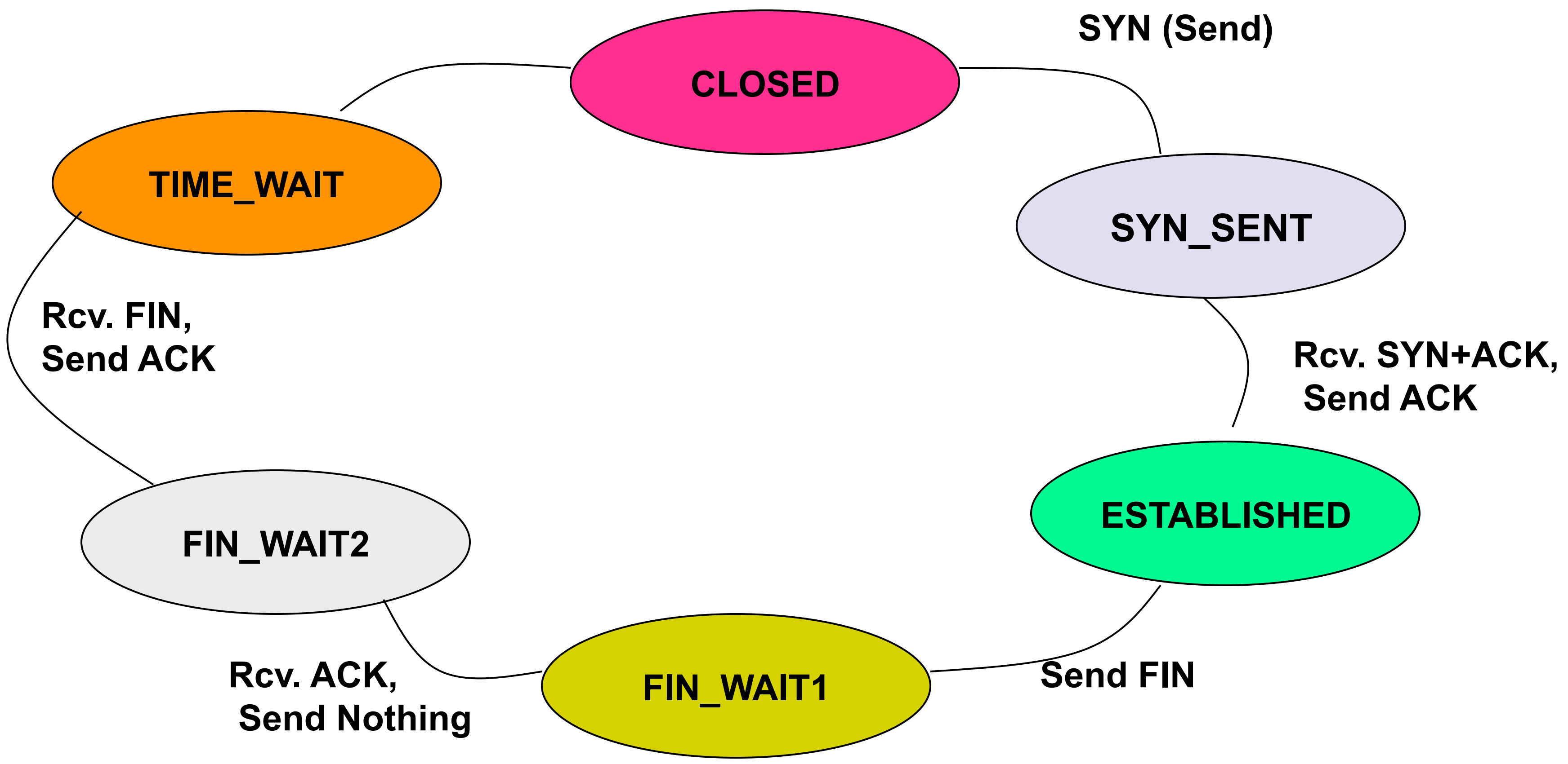
TCP Header


Flags:
SYN
ACK
FIN
RST
PSH
URG
Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

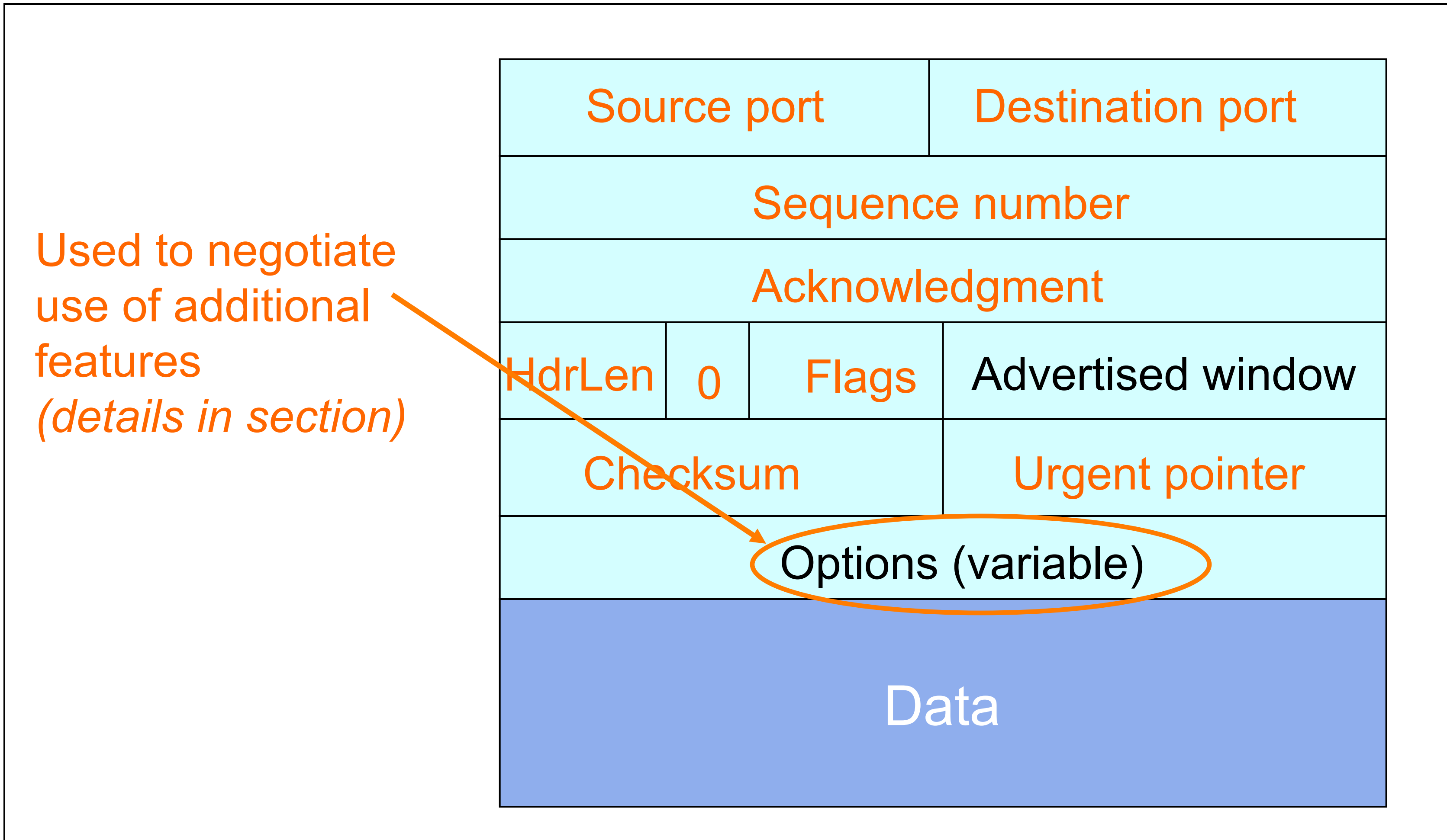
TCP State Transitions



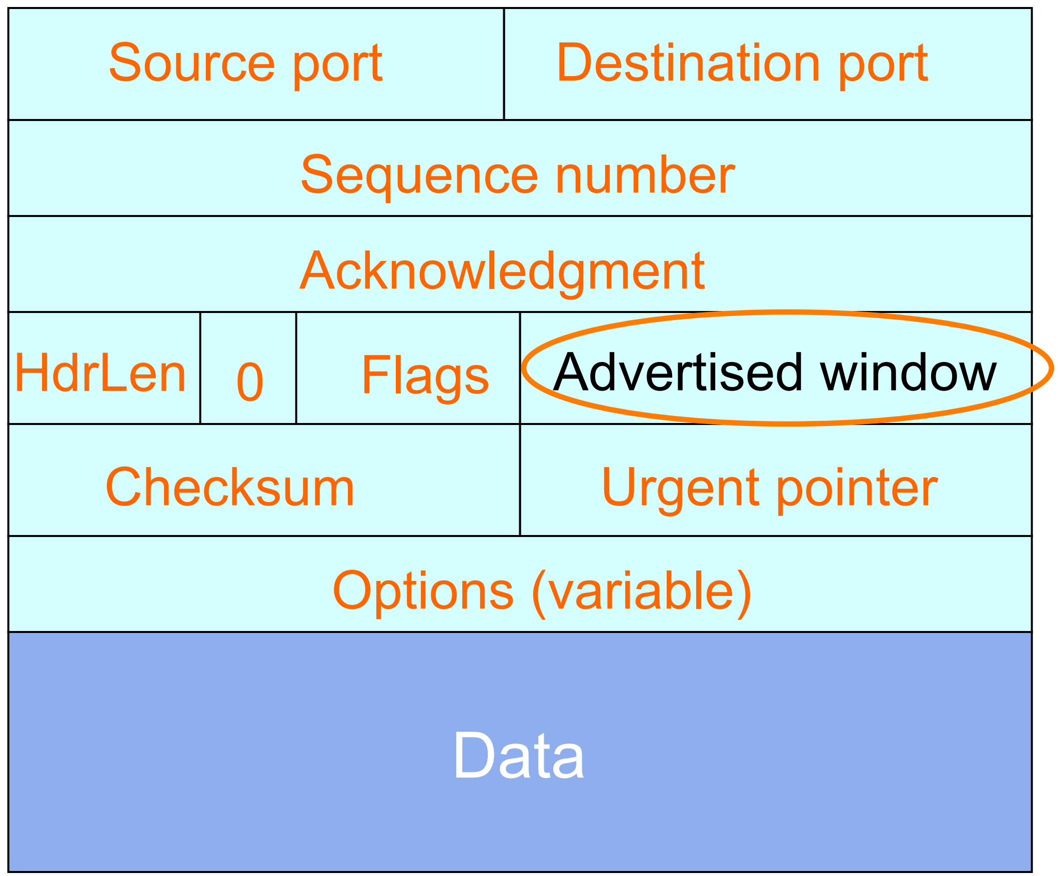
An Simpler View of the Client Side



TCP Header


Used to negotiate use of additional features
(details in section)
Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

TCP Header


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

Last time: Sliding Window

- Both sender & receiver maintain a **window**

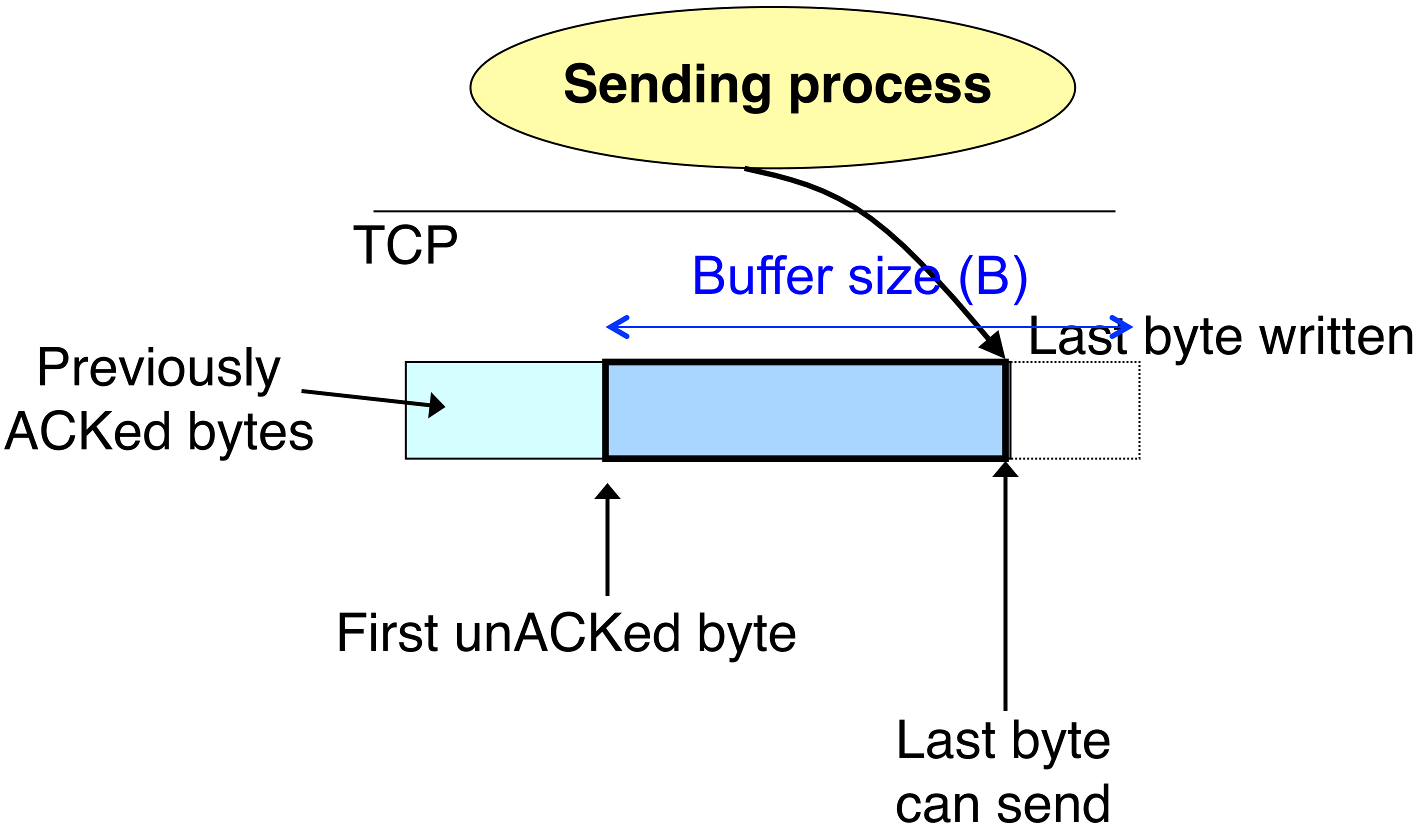
- Left edge of window:
  - Sender: beginning of unacknowledged data
  - Receiver: beginning of expected data
    - First “hole” in received data
    - When sender gets ack, knows that receiver’s window has moved

- Right edge: Left edge + *constant*
  - constant only limited by buffer size in the transport layer

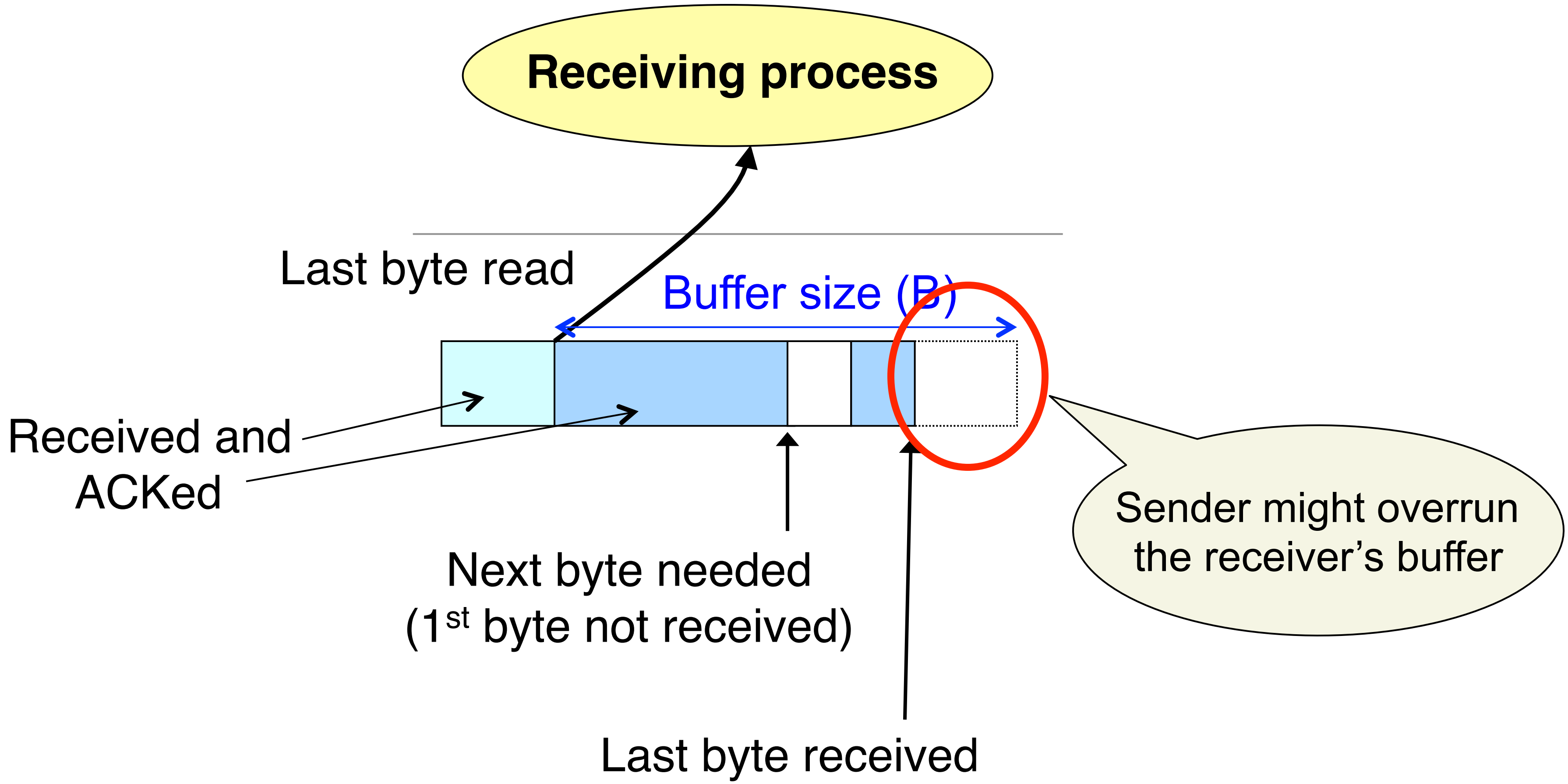
TCP: Sliding Window (so far)

- Both sender & receiver maintain a **window**
- Left edge of window:
  - Sender: beginning of unacknowledged data
  - Receiver: beginning of **undelivered** data
- Right edge: Left edge + *constant*
  - constant only limited by buffer size in the transport layer

Sliding Window at Sender (so far)



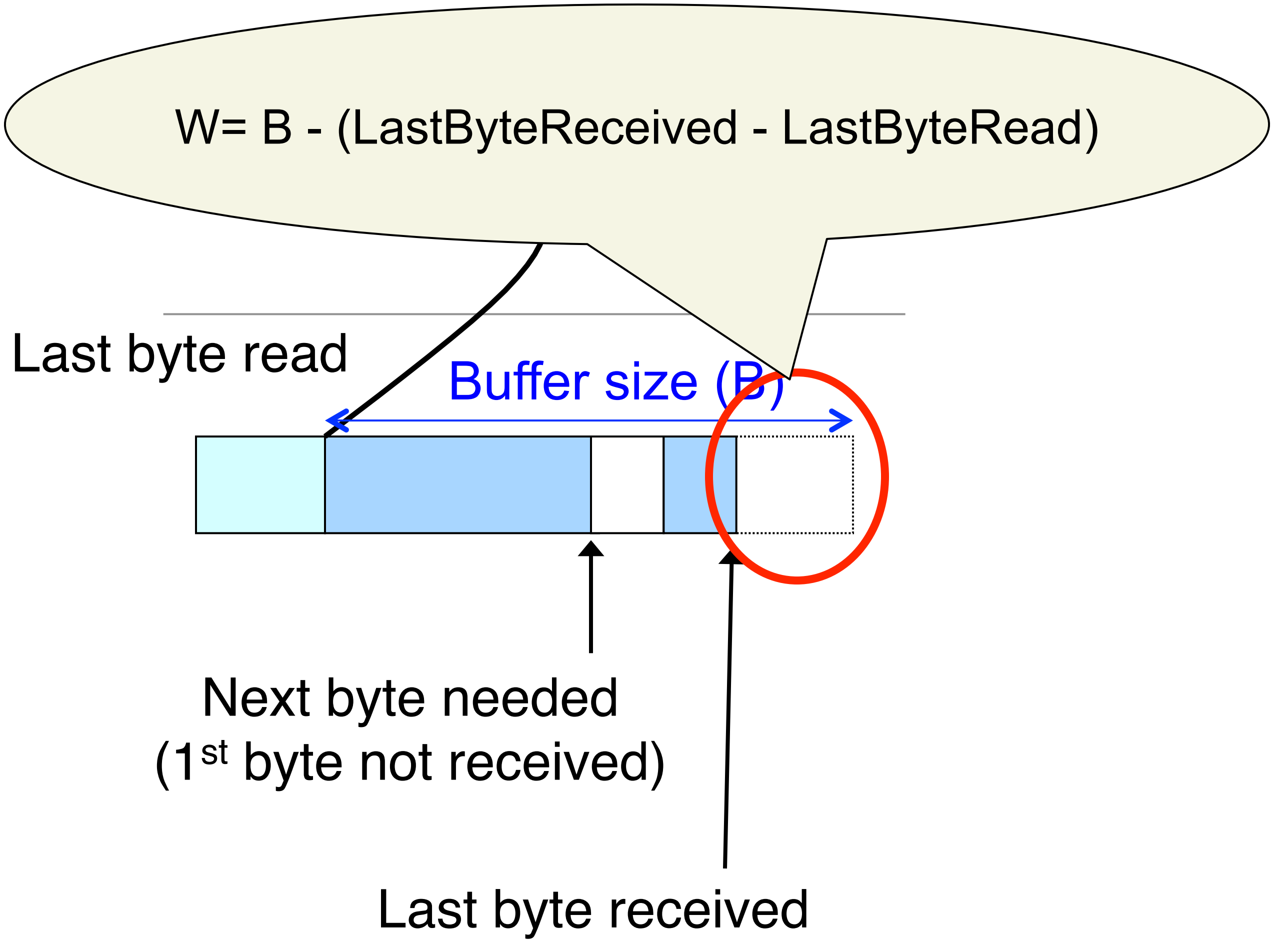
Sliding Window at Receiver (so far)



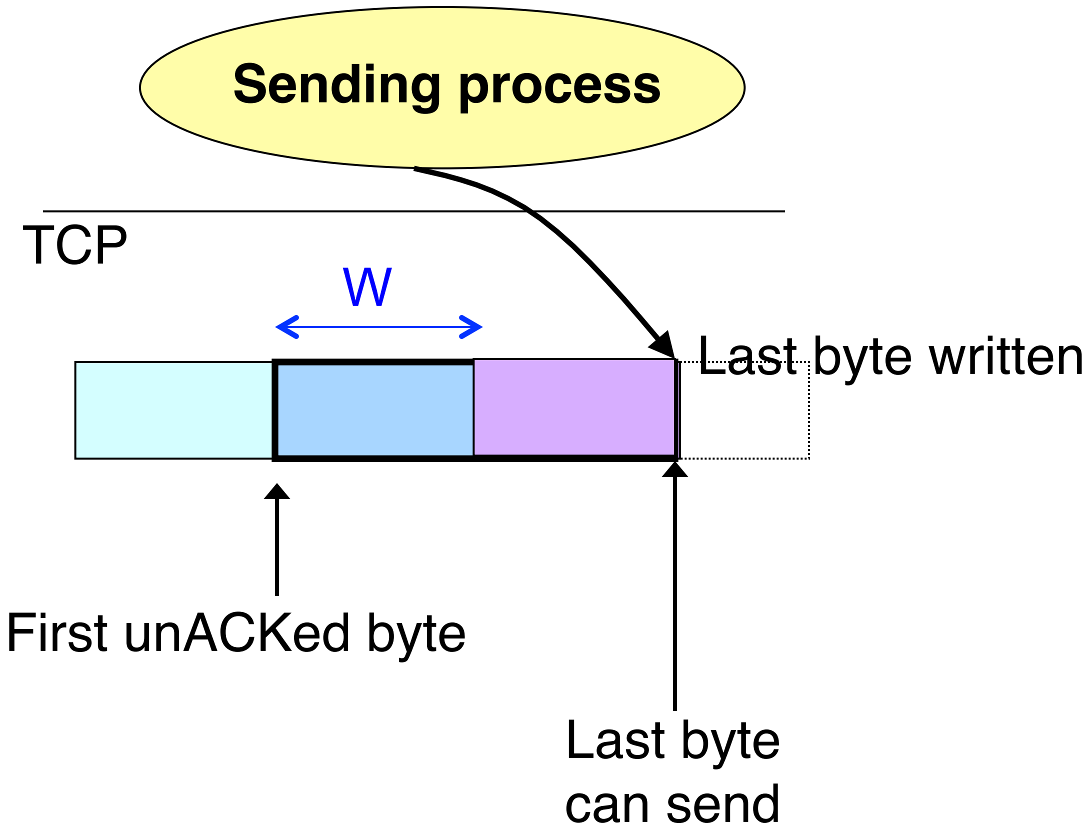
Solution: Advertised Window (Flow Control)

- Receiver uses an “Advertised Window” (W) to prevent sender from overflowing its window
  - Receiver indicates value of W in ACKs
  - Sender limits number of bytes it can have in flight <= W

Sliding Window at Receiver



Sliding Window at Sender (so far)



Sliding Window w/ Flow Control

- Sender: window advances when new data ack'd
- Receiver: window advances as receiving process consumes data
- Receiver advertises to the sender where the receiver window currently ends ("righthand edge")
  - Sender agrees not to exceed this amount

Advertised Window Limits Rate

- Sender can send no faster than W/RTT bytes/sec
- Receiver only advertises more space when it has consumed old arriving data
- In original TCP design, that was the **sole** protocol mechanism controlling sender's rate
- What's missing?

Taking Stock (1)

- The concepts underlying TCP are simple
  - acknowledgments (feedback)
  - timers
  - sliding windows
  - buffer management
  - sequence numbers

Taking Stock (1)

- The concepts underlying TCP are simple
- But tricky in the details
  - How do we set timers?
  - What is the seqno for an ACK-only packet?
  - What happens if advertised window = 0?
  - What if the advertised window is ½ an MSS?
  - Should receiver acknowledge packets right away?
  - What if the application generates data in units of 0.1 MSS?
  - What happens if I get a duplicate SYN? Or a RST while I'm in FIN_WAIT, *etc., etc., etc.*

Taking Stock (1)

- The concepts underlying TCP are simple
- But tricky in the details
- Do the details matter?

Sizing Windows for Congestion Control

- What are the problems?
- How might we address them?