

# **EECS 489**

# **Computer Networks**

**Fall 2019**

Mosharaf Chowdhury

*Material with thanks to Aditya Akella, Sugih Jamin, Philip Levis, Sylvia Ratnasamy, Peter Steenkiste, and many other colleagues.*

# Agenda

---

- From reliable data transfer to TCP
- TCP connection setup and teardown

# Recap: Designing a reliable transport protocol

---

- Stop and wait is correct but inefficient
- Sliding window uses pipelining to increase throughput

# Recap: Acknowledgements

---

- Cumulative
  - Acknowledge many packets at a time
- Selective
  - Acknowledge individual packets

# Recap: Sliding window protocols

---

- Resending packets: two canonical approaches
  - Go-Back-N: Resend all N packets
  - Selective Repeat: Resend only the missing packets
- Many variants that differ in implementation details

---

# **TCP: TRANSMISSION CONTROL PROTOCOL**

# The TCP Abstraction

---

- TCP delivers a reliable, in-order, byte stream
- **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
- **Byte stream**: TCP assumes there is an incoming stream of data, and attempts to deliver it to app

# What does TCP use from what we've seen so far?

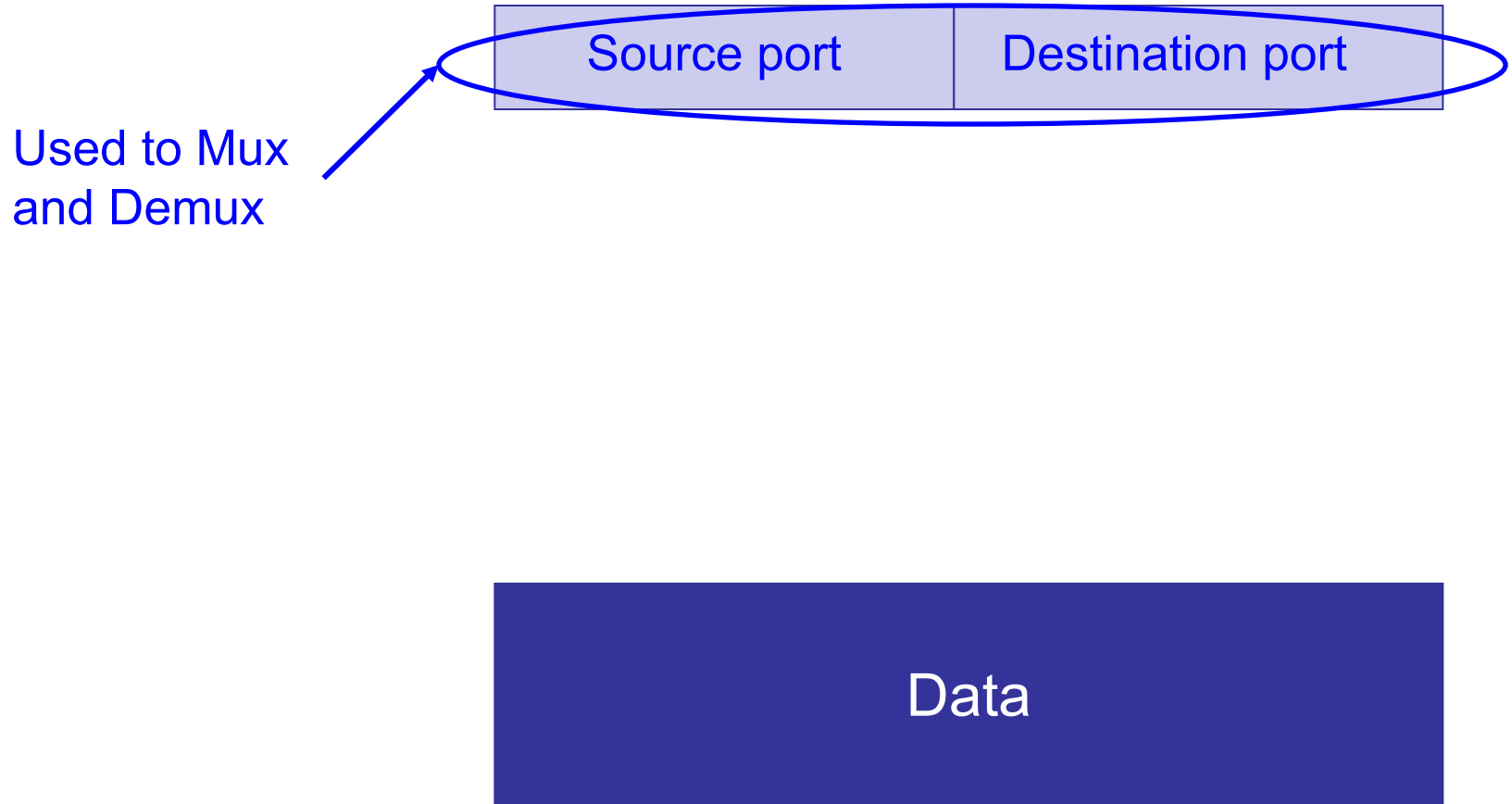
---

- Most of what we've seen
  - Checksums
  - Sequence numbers are byte offsets
  - Sender and receiver maintain a sliding window
  - Receiver sends cumulative acknowledgements (like GBN)
    - » Sender maintains a single retransmission timer
  - Receivers buffer out-of-sequence packets (like SR)
- Few more: fast retransmit, timeout estimation algorithms etc.



# TCP header

---

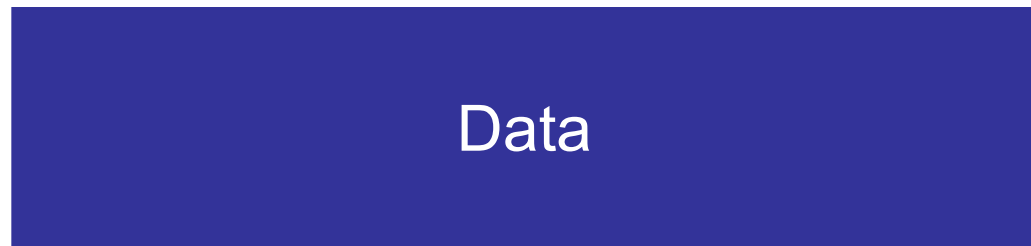


# TCP header

---



Computed  
over pseudo-header  
and data



# What does TCP do?

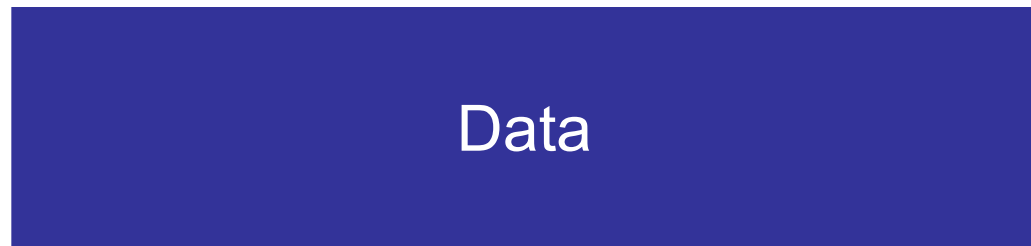
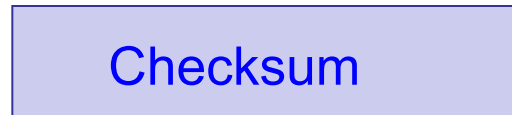
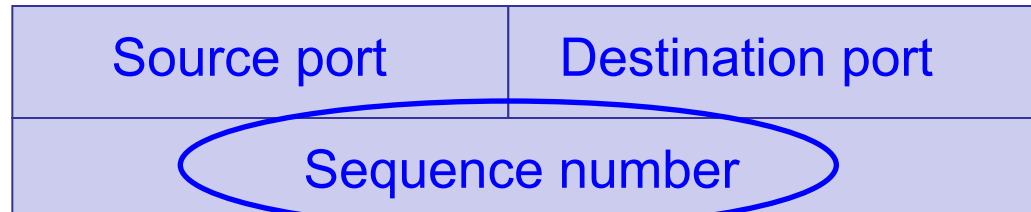
---

- Most of what we've seen
  - Checksum
  - Sequence numbers are byte offsets

# TCP header

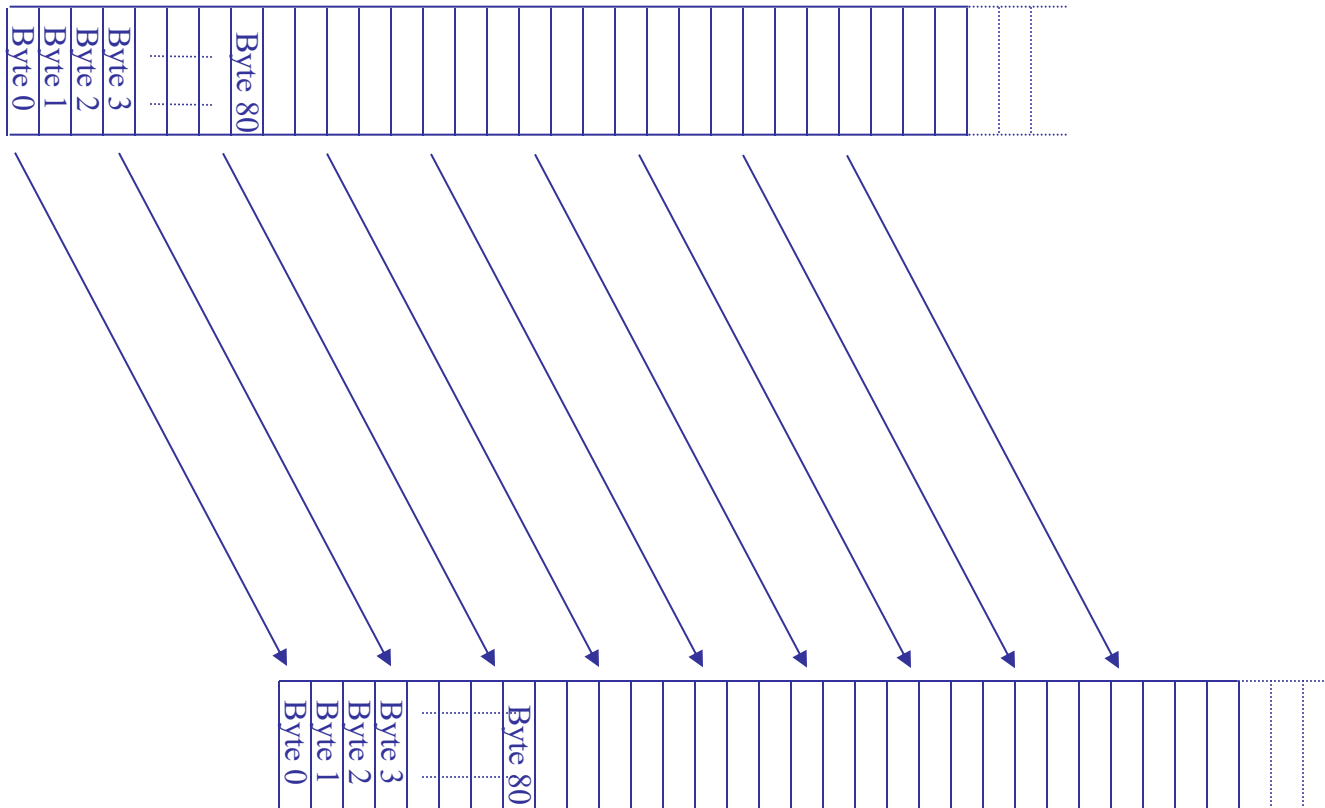
---

Byte offsets  
(NOT packet id),  
because TCP is a  
byte stream



# TCP “stream of bytes” service...

Application @ Host A

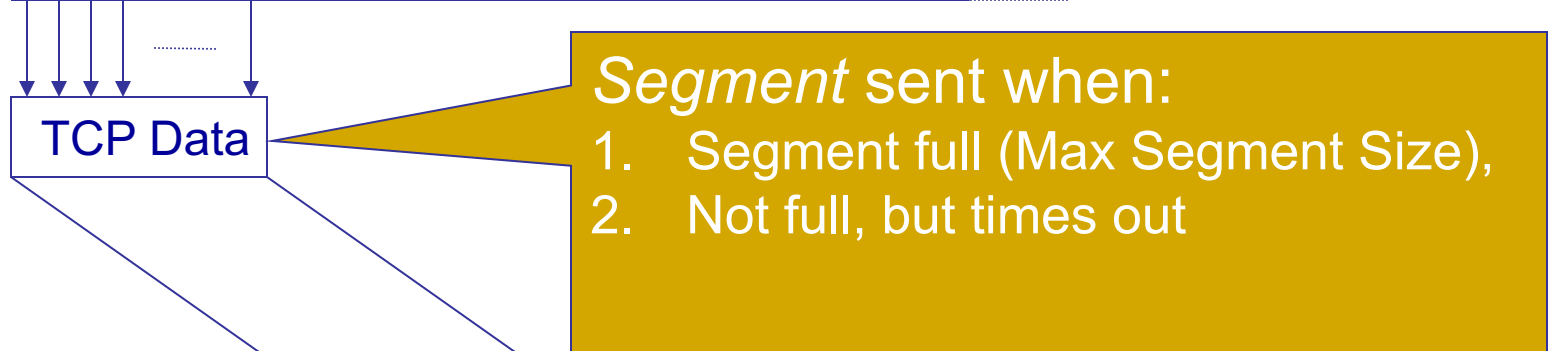
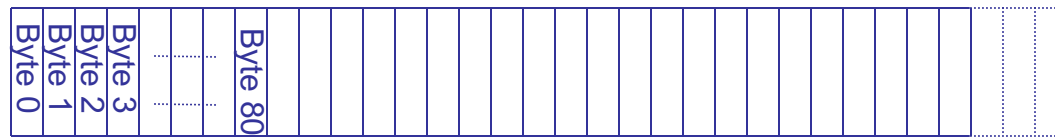


Application @ Host B

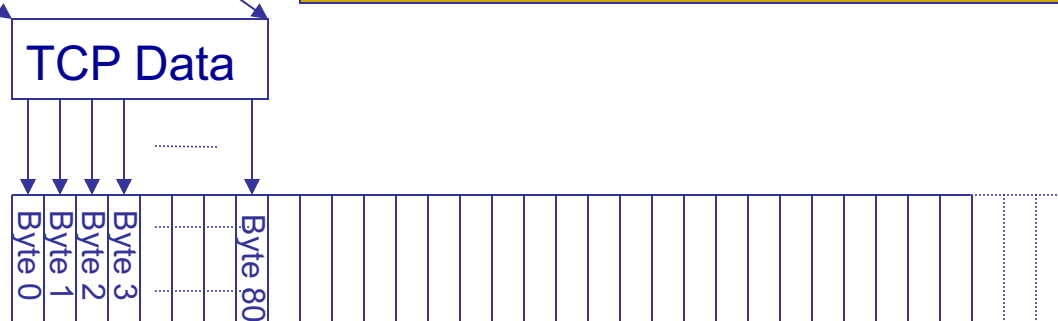
# ... provided using TCP

## “segments”

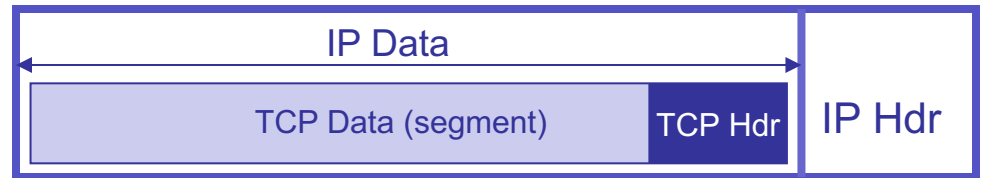
Host A



Host B

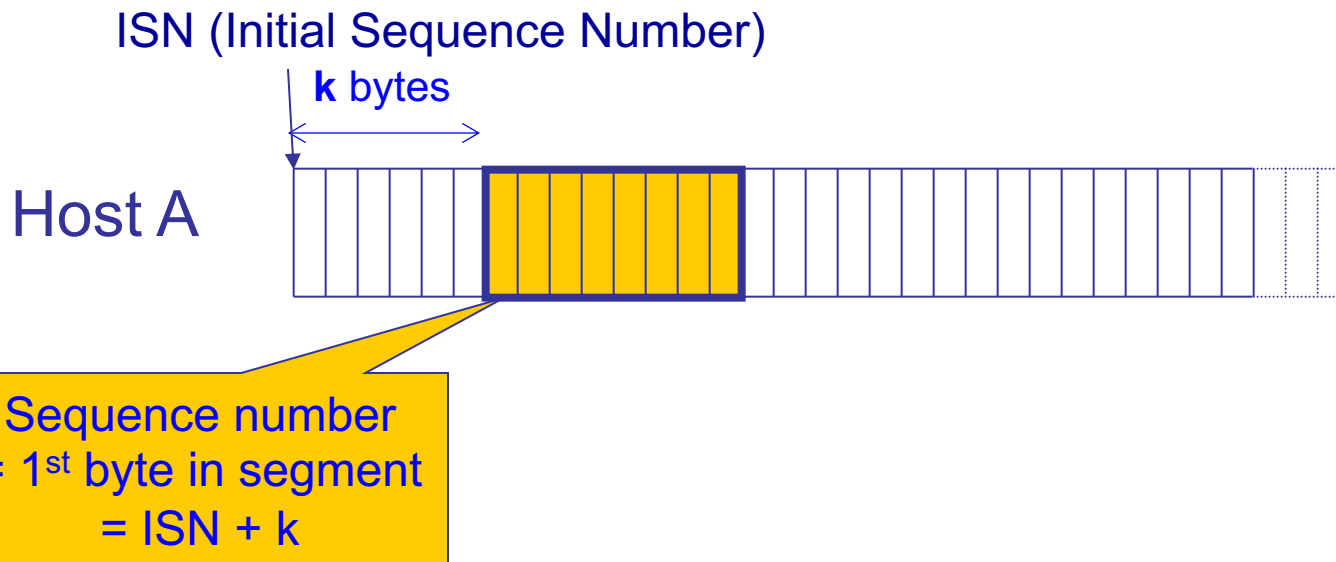


# TCP segment



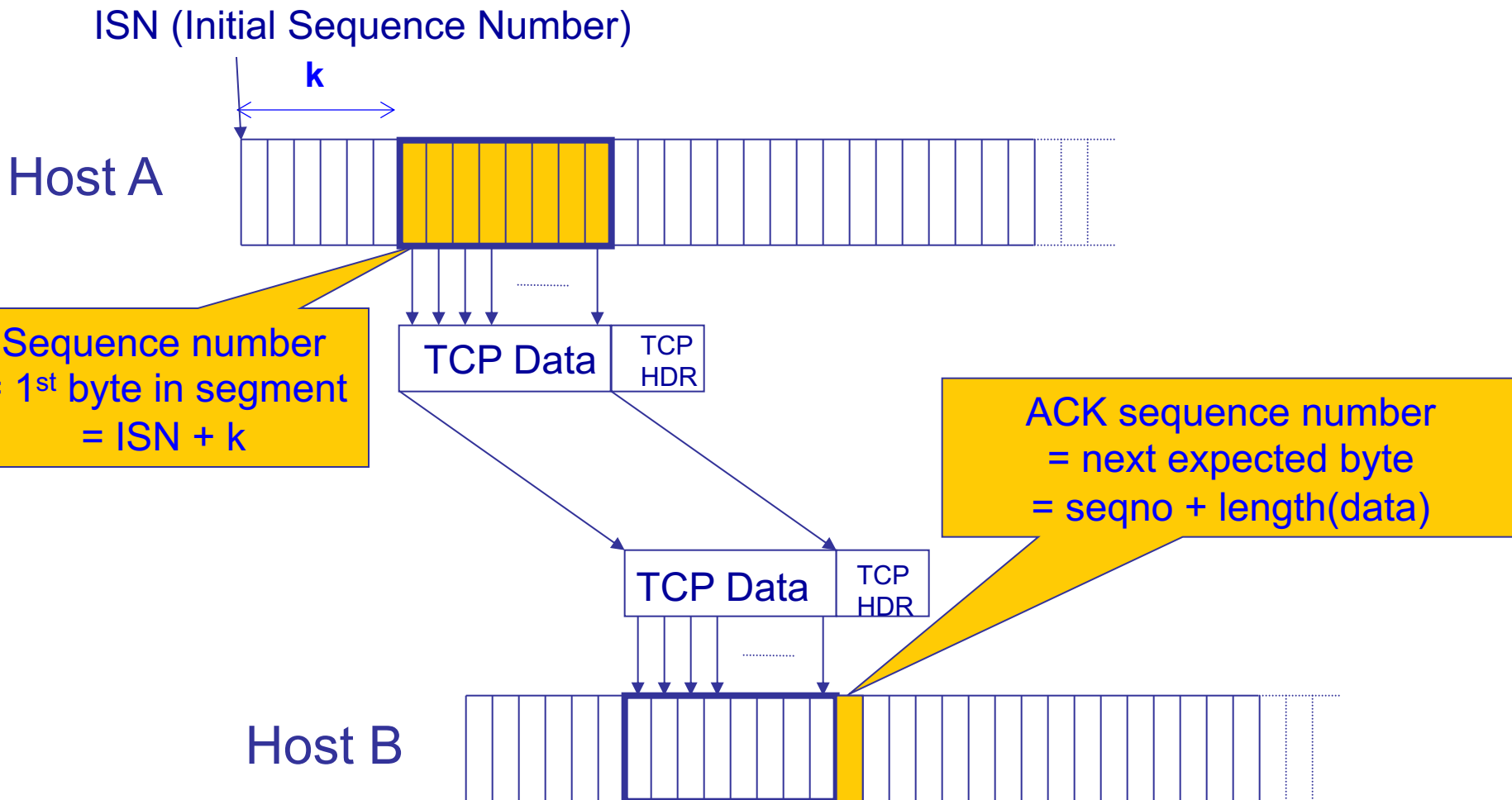
- IP packet
  - No bigger than **Maximum Transmission Unit (MTU)**
  - E.g., up to 1500 bytes with Ethernet
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header  $\geq 20$  bytes long
- TCP segment
  - No more than **Maximum Segment Size (MSS)** bytes
  - E.g., up to 1460 consecutive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

# Sequence numbers





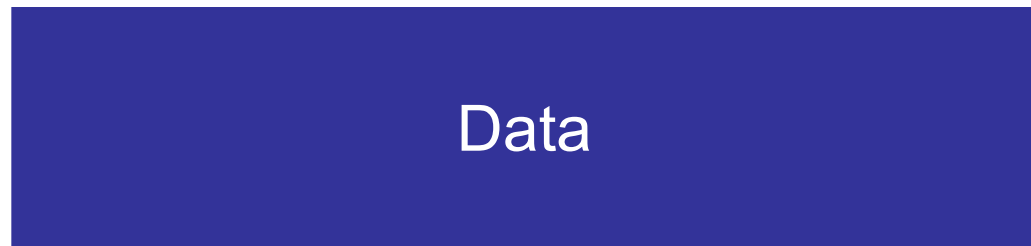
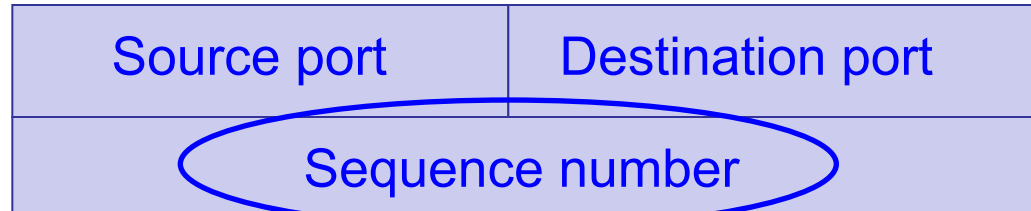
# Sequence numbers



# TCP header

---

Starting byte  
offset of data  
carried in this  
segment



# What does TCP do?

---

- Most of what we've seen
  - Checksum
  - Sequence numbers are byte offsets
  - Receiver sends **cumulative acknowledgements** (like GBN)

# ACKs and sequence numbers

---

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

# Typical operation

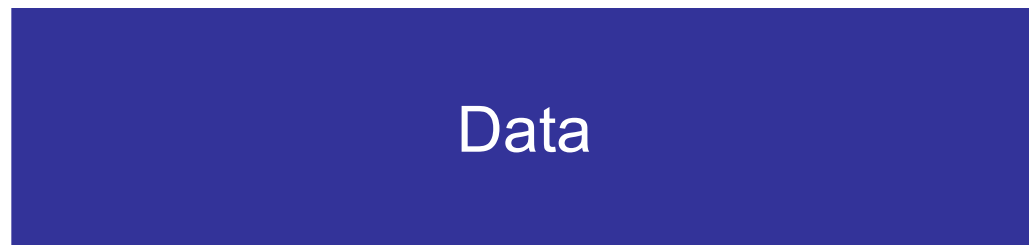
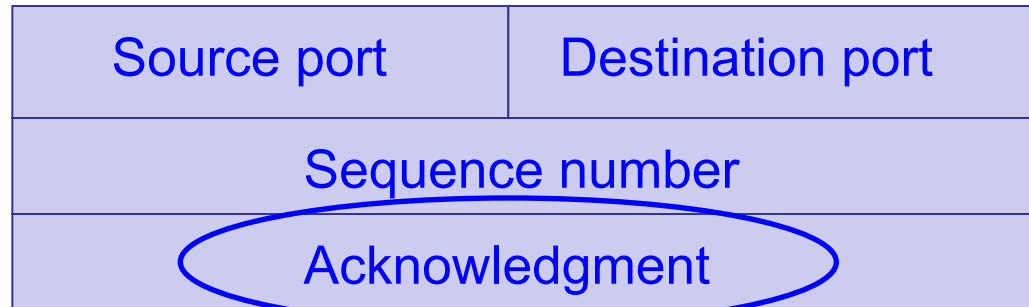
---

- Sender:  $\text{seqno} = X$ ,  $\text{length} = B$
- Receiver:  $\text{ACK} = X + B$
- Sender:  $\text{seqno} = X + B$ ,  $\text{length} = B$
- Receiver:  $\text{ACK} = X + 2B$
- Sender:  $\text{seqno} = X + 2B$ ,  $\text{length} = B$
  
- Seqno of next packet is same as last ACK field

# TCP header

---

Acknowledgment  
gives seqno just  
beyond highest  
seqno received  
in order



# What does TCP do?

---

- Most of what we've seen
  - 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, ...
- 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 introduce?

---

- Most of what we've seen
  - Checksum
  - Sequence numbers are byte offsets
  - Receiver sends cumulative acknowledgements (like GBN)
  - Receivers can buffer out-of-sequence packets (like SR)
- Introduces **fast retransmit**: duplicate ACKs trigger early retransmission

# 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
- Trigger retransmission upon receiving k duplicate ACKs
  - » TCP uses  $k=3$
  - » Faster than waiting for timeout

# Loss with cumulative ACKs

---

- Two choices after resending
  - 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?
  - Choose correctness

# What does TCP introduce?

---

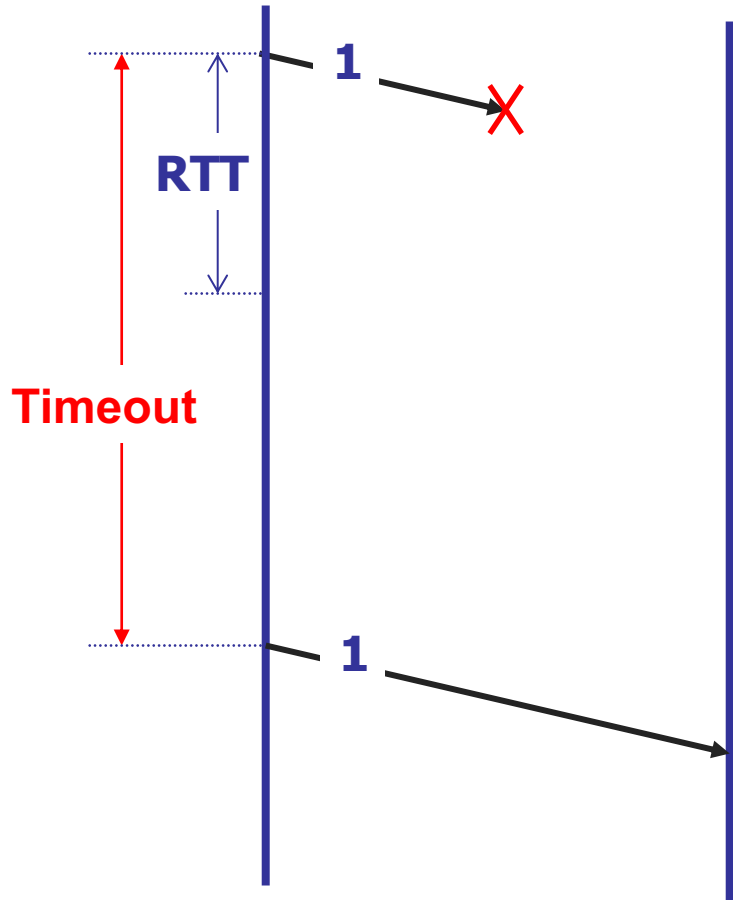
- Most of what we've seen
  - Checksum
  - Sequence numbers are byte offsets
  - Receiver sends cumulative acknowledgements (like GBN)
  - Receivers buffer out-of-sequence packets (like SR)
- Introduces fast retransmit: duplicate ACKs trigger early retransmission
- Sender maintains a **single retransmission timer** (like GBN) and retransmits on timeout

# Retransmission timeout

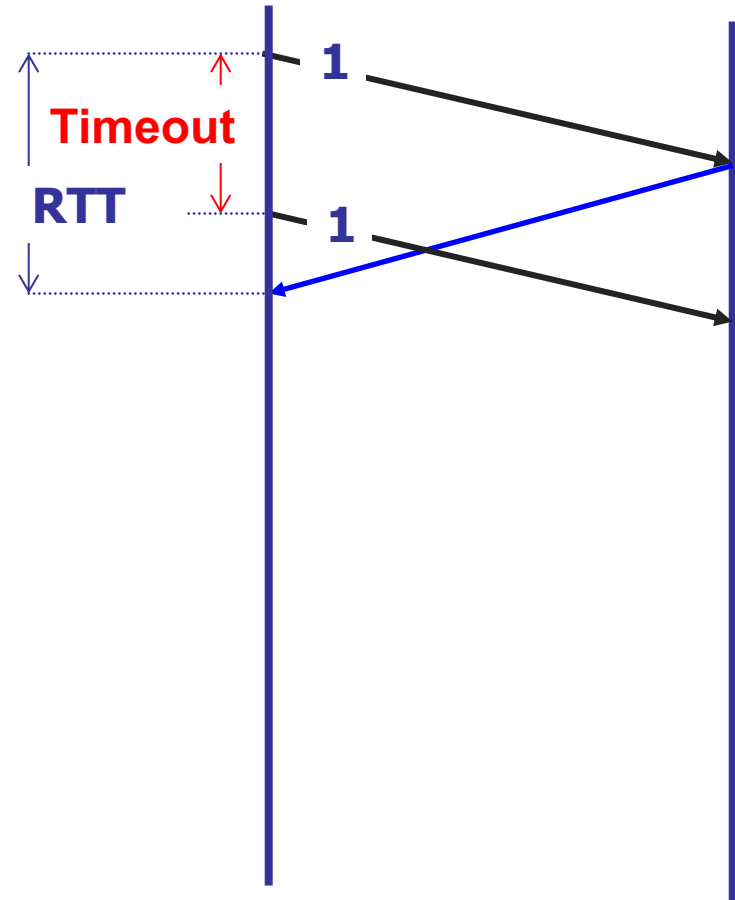
---

- 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

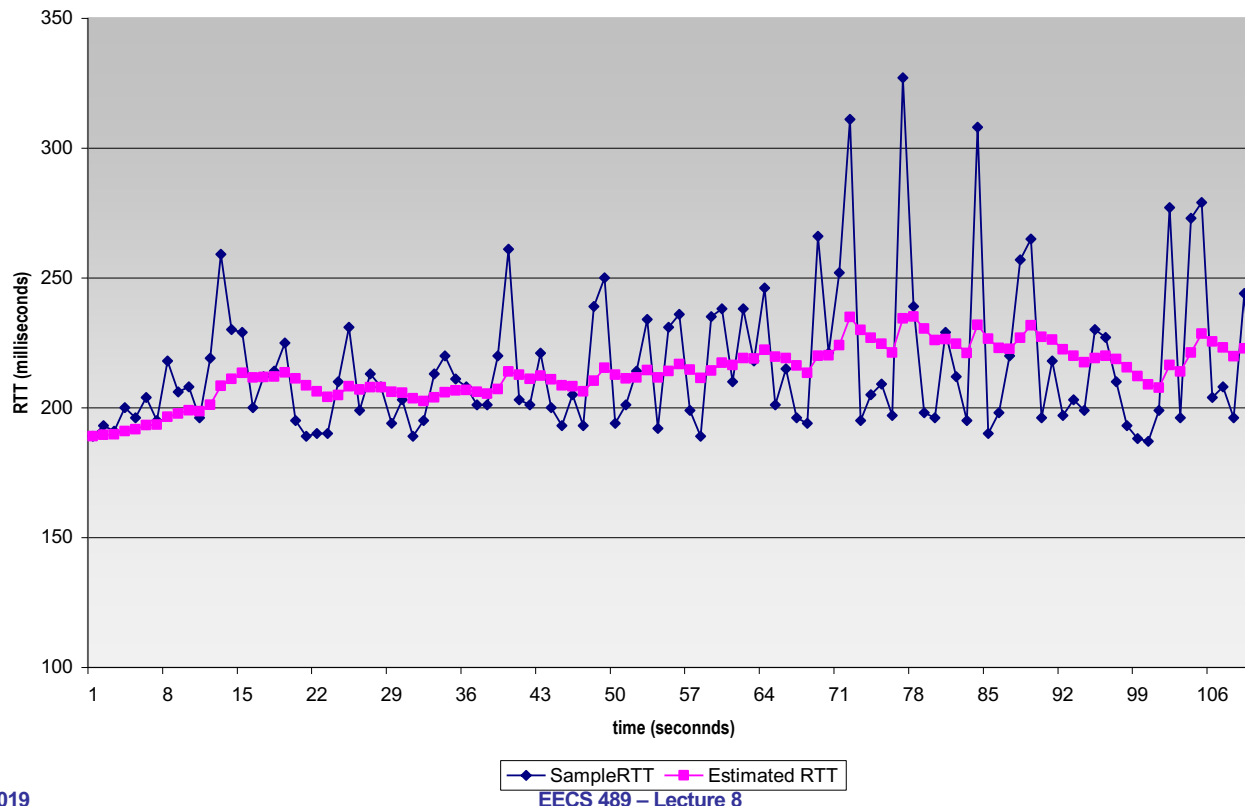
---

- If the sender hasn't received an 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

- Exponential weighted average of RTT samples

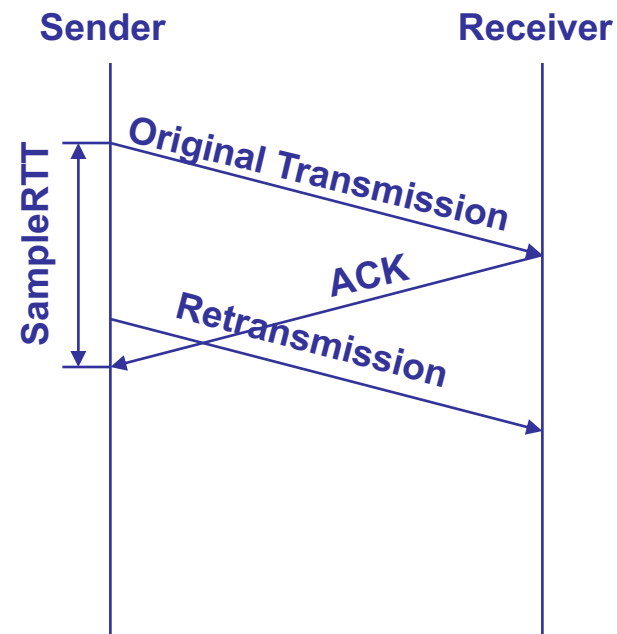
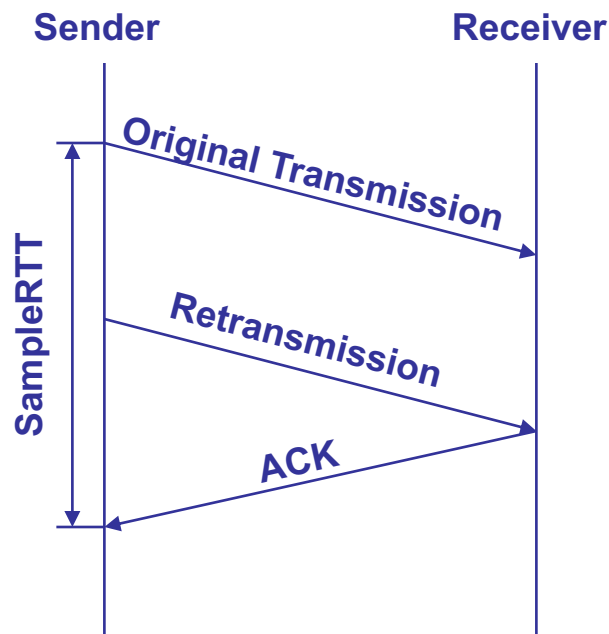
$$\text{EstimatedRTT} = (1 - \alpha) * \text{EstimatedRTT} + \alpha * \text{SampleRTT}$$





# Problem: Ambiguous measurements

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



# Karn/Partridge algorithm

---

- Don't use SampleRTT from retransmissions
  - Once retransmitted, ignore that segment in the future
- Computes EstimatedRTT using  $\alpha = 0.125$
- Timeout value (RTO) =  $2 \times \text{EstimatedRTT}$ 
  - Employs exponential backoff
    - » Every time RTO timer expires, set  $\text{RTO} \leftarrow 2 \cdot \text{RTO}$ 
      - (Up to maximum  $\geq 60$  sec)
    - » Every time new measurement comes in (= successful original transmission), collapse RTO back to  $2 \times \text{EstimatedRTT}$
- Sensitive to RTT variations

# Jacobson/Karels algorithm

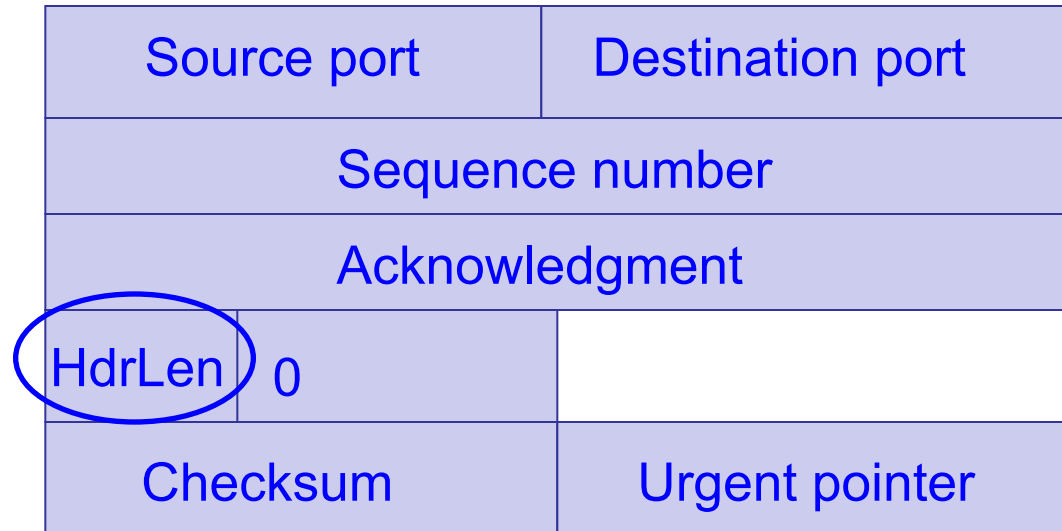
---

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

# TCP header

---

Number of 4-  
byte words in the  
header;  
5: No options



Data

---

# **5-MINUTE BREAK!**

# Announcements

---

- Assignment 2 is live!

---

# **TCP CONNECTION ESTABLISHMENT**

# Initial Sequence Number (ISN)

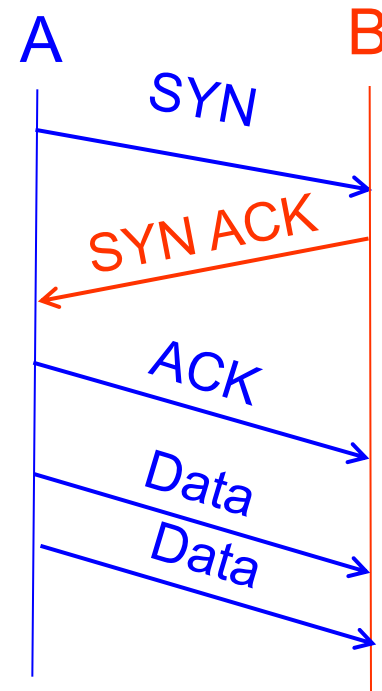
---

- 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?
- Hosts exchange ISNs when establishing connection



# Establishing a TCP connection

- 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

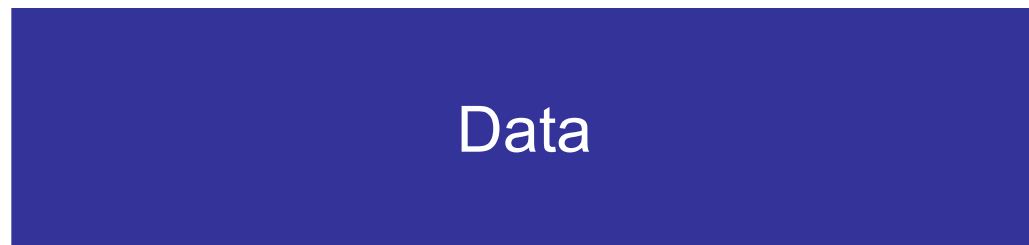
ACK

FIN

RST

PSH

URG



# Step 1: A's initial SYN packet

---

A tells B to open  
a connection

A's port			B's port
A's Initial Sequence Number			
N/A			
5	0	SYN	
Checksum			Urgent pointer

# Step 1: B's SYN-ACK packet

---

B tells it accepts  
and is ready to  
accept next  
packet

B's port		A's port	
B's Initial Sequence Number			
ACK=A's ISN+1			
5	0	SYN ACK	
Checksum		Urgent pointer	

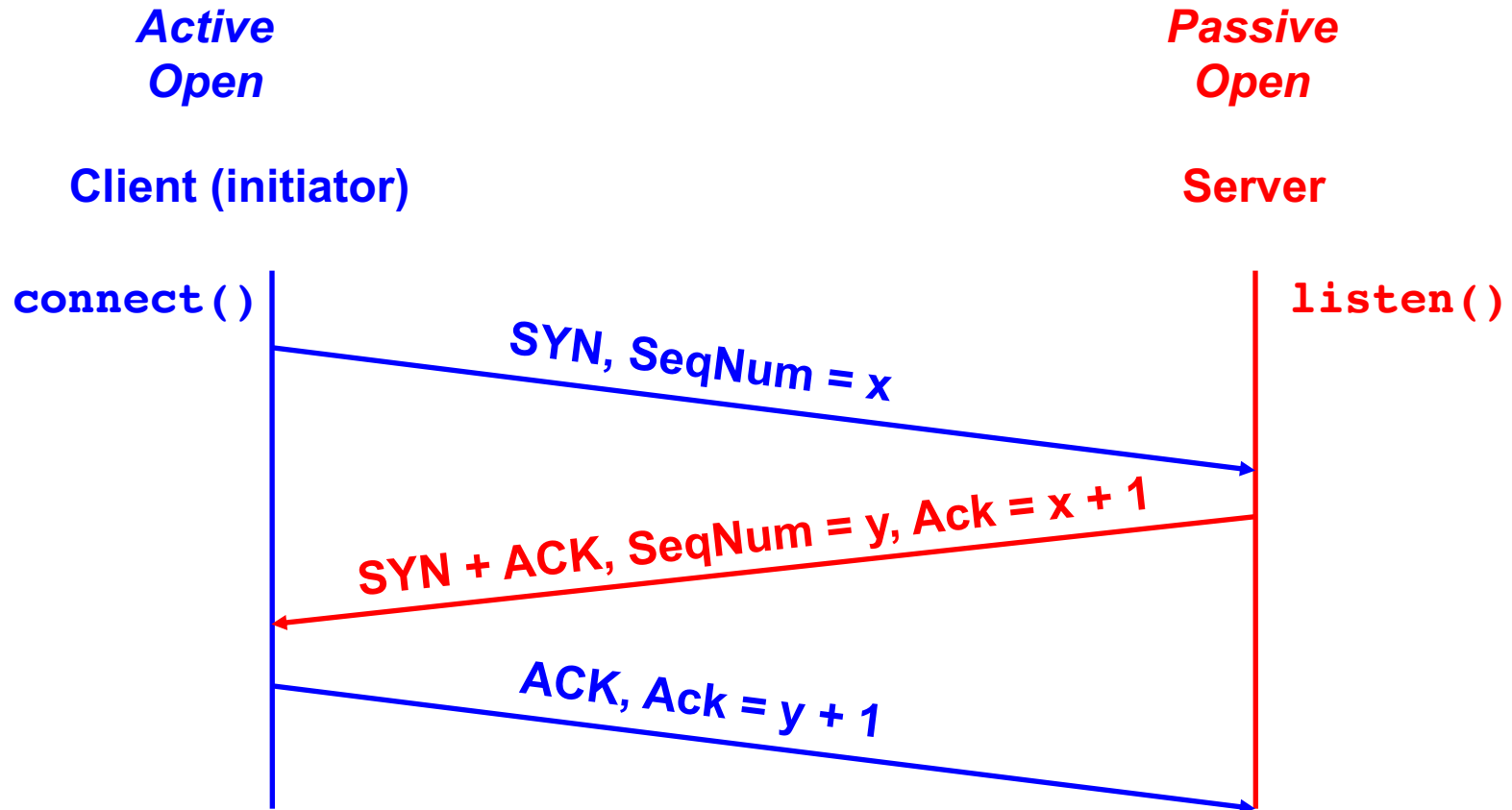
# Step 1: A's ACK to SYN-ACK

---

A tells B to open  
a connection

A's port			B's port
A's Initial Sequence Number			
ACK=B's ISN+1			
5	0	ACK	
Checksum			Urgent pointer

# TCP's 3-Way handshaking



# What if the SYN Packet Gets Lost?

---

- Suppose the SYN packet gets lost
  - Packet dropped by the network or server is busy
- Eventually, no SYN-ACK arrives
  - Sender retransmits the SYN on timeout
- 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

---

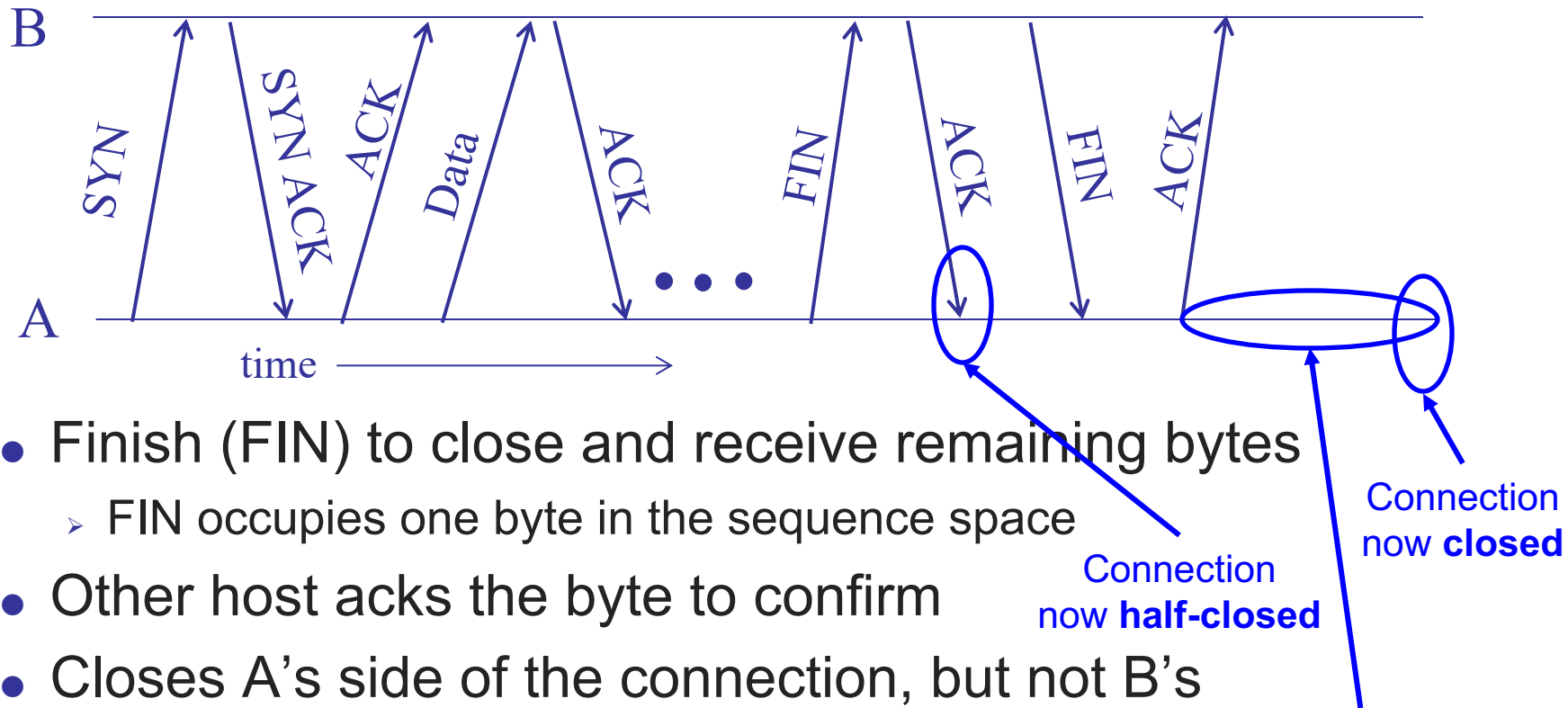
- User clicks on a hypertext link
  - Browser creates a socket and does a “connect”
  - The “connect” triggers the OS to transmit a SYN
- If the SYN is lost...
  - 3-6 seconds of delay: can be very long
  - User may become impatient and can retry
- User triggers an “abort” of the “connect”
  - Browser creates a new socket and another “connect”
  - Can be effective in some cases



---

# TCP CONNECTION TEARDOWN

# Normal termination, one side at a time



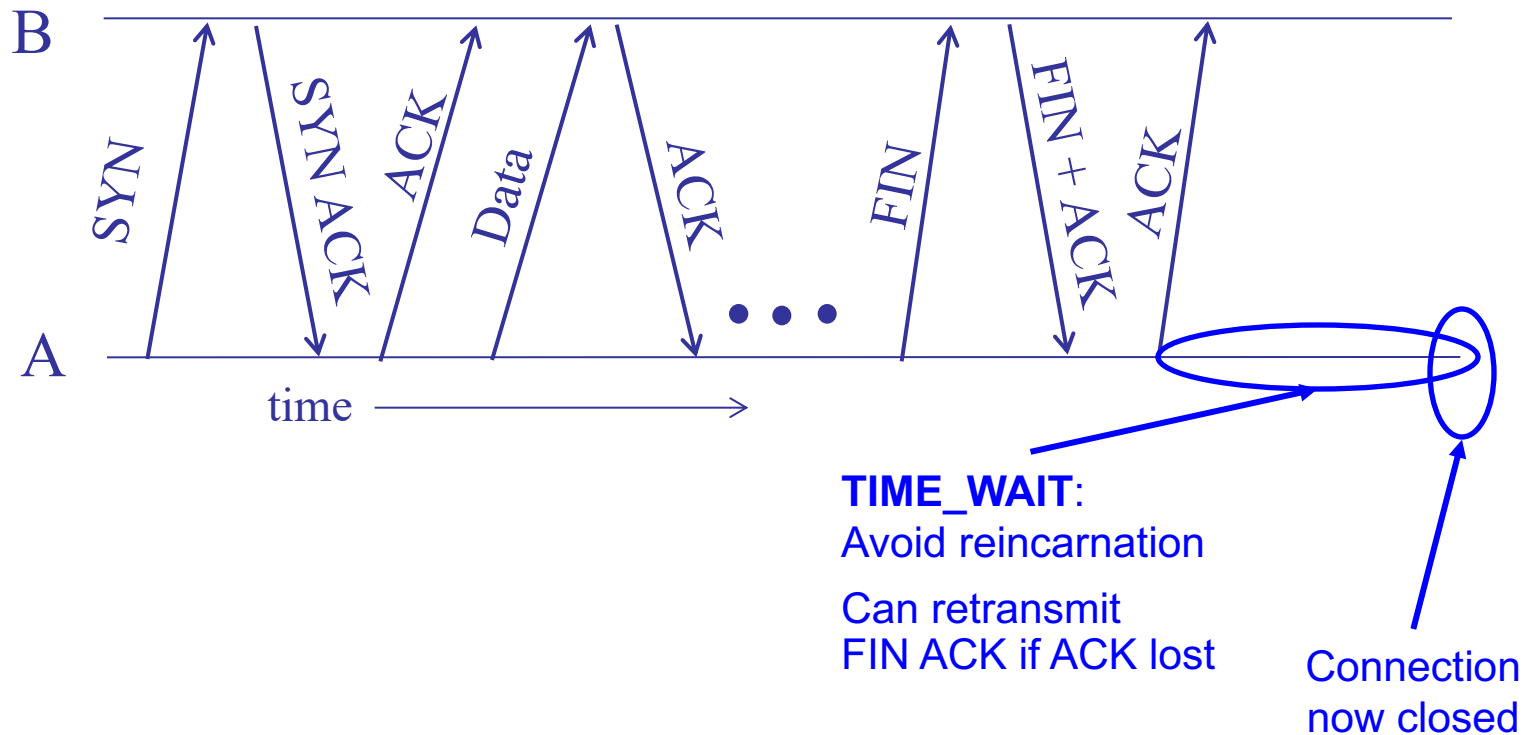
- Finish (FIN) to close and receive remaining bytes
  - FIN occupies one byte in the sequence space
- Other host acks the byte to confirm
- Closes A's side of the connection, but not B's
  - Until B likewise sends a FIN
  - Which A then acks

**TIME\_WAIT:**

Avoid reincarnation

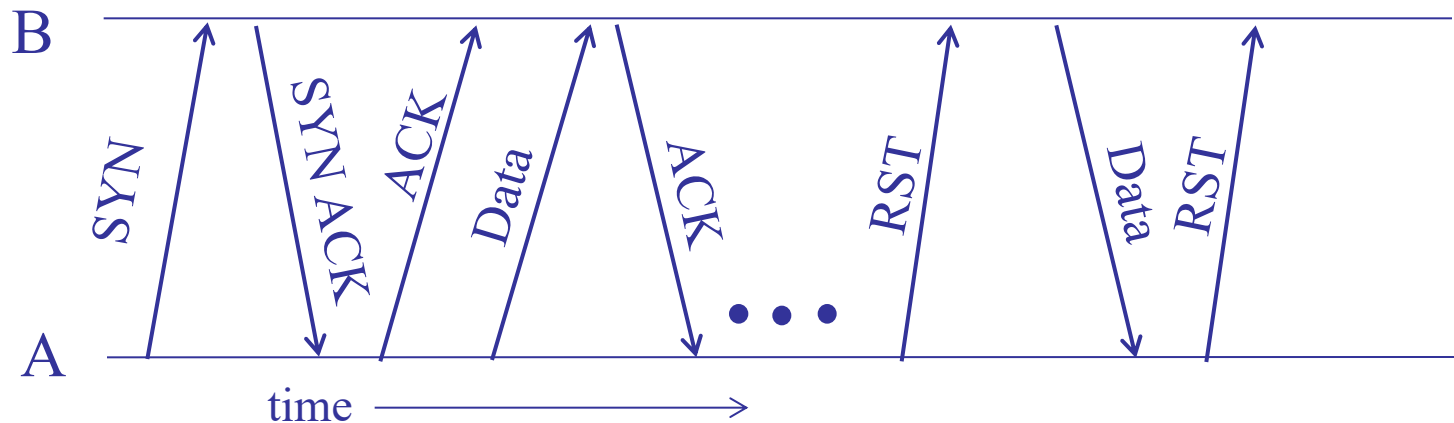
B will retransmit FIN  
if ACK is lost

# 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

# Summary

---

- Reliability is not easy!
- Next
  - Flow control
  - LOTS of congestion control