



Computer Networks

Wenzhong Li, Chen Tian

Nanjing University

Material with thanks to James F. Kurose, Mosharaf Chowdhury, and other colleagues.



Outline

- UDP: User Datagram Protocol
- TCP: Transmission Control Protocol
- TCP Connection Setup



User Datagram Protocol (UDP)



UDP: User Datagram Protocol

- Lightweight communication between processes
 - Avoid overhead and delays of order & reliability
- UDP described in RFC 768 – (1980!)
 - Destination IP address and port to support demultiplexing



UDP (cont'd)

- “Best effort” service, UDP segments may be:
 - lost
 - delivered out-of-order to app
- Connectionless:
 - no handshaking between UDP sender, receiver
 - each UDP segment handled independently of others
- UDP use:
 - streaming multimedia apps (loss tolerant, rate sensitive)
 - DNS
 - SNMP

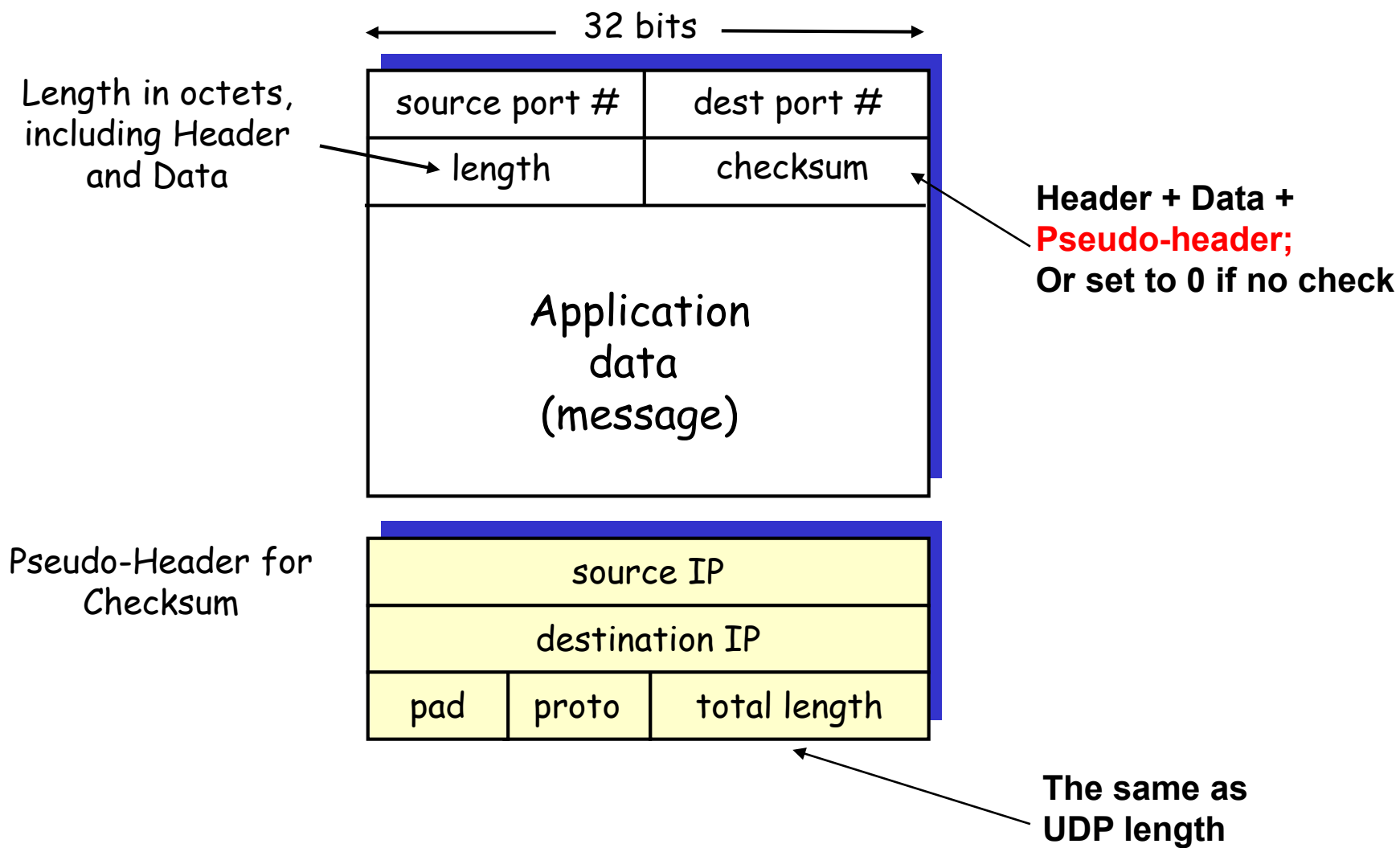


Why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control: UDP can blast away as fast as desired



UDP Segment Format





UDP checksum

Goal: detect “errors” (e.g., flipped bits) in transmitted segment

sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
 - NO - error detected
 - YES - no error detected. *But maybe errors nonetheless?*
More later



Internet checksum: example

example: add two 16-bit integers

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																	
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
<hr/>																	
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result



Optional error checking

- Optional error checking on the packet contents
 - (checksum field = 0 means “don’t verify checksum”)
 - See text on how checksums are calculated
- Source port is also optional
 - Useful to respond back to the sender in some cases



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



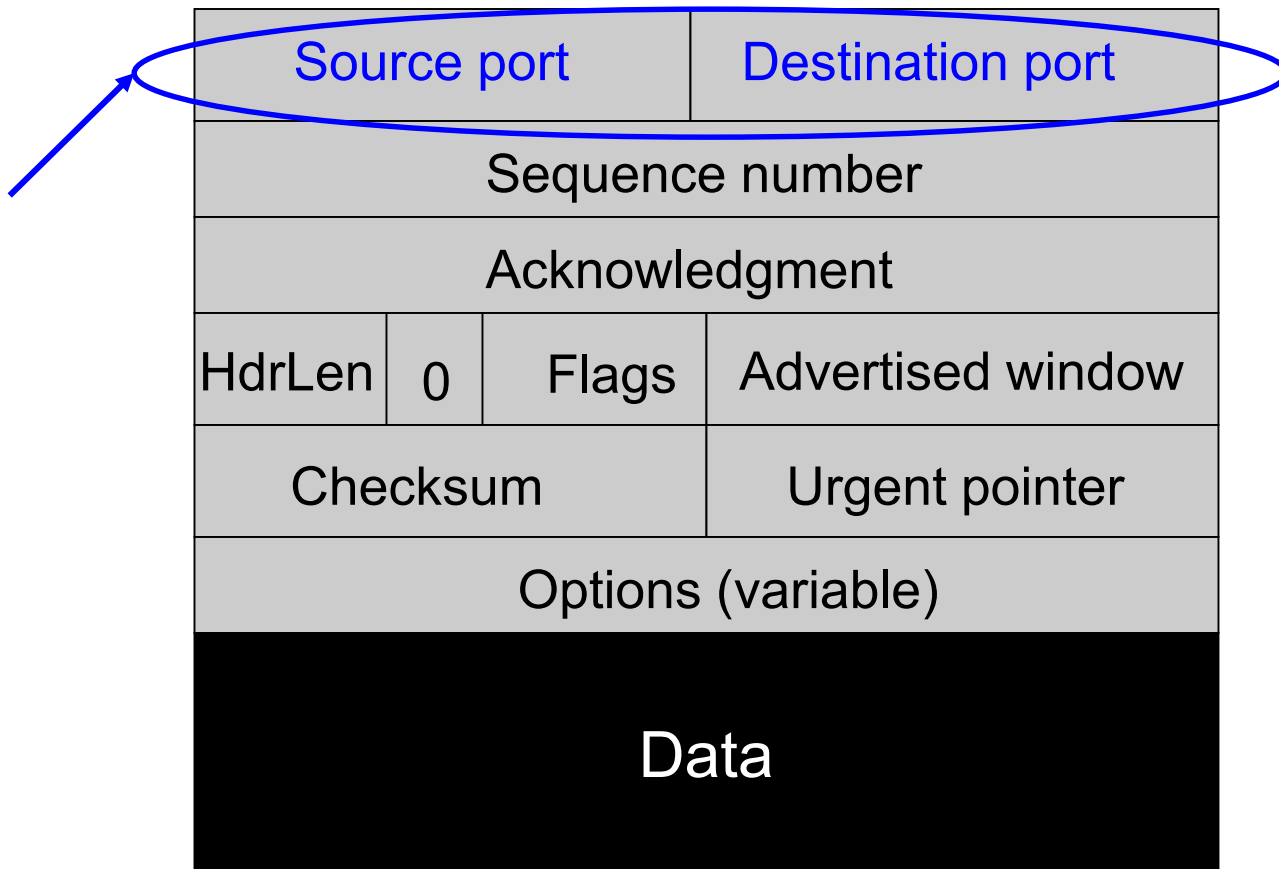
What does TCP use from what we've seen so far?

- 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

Used to Mux
and Demux





TCP header

Computed
over pseudo-header
and data

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



What does TCP do?

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



TCP header

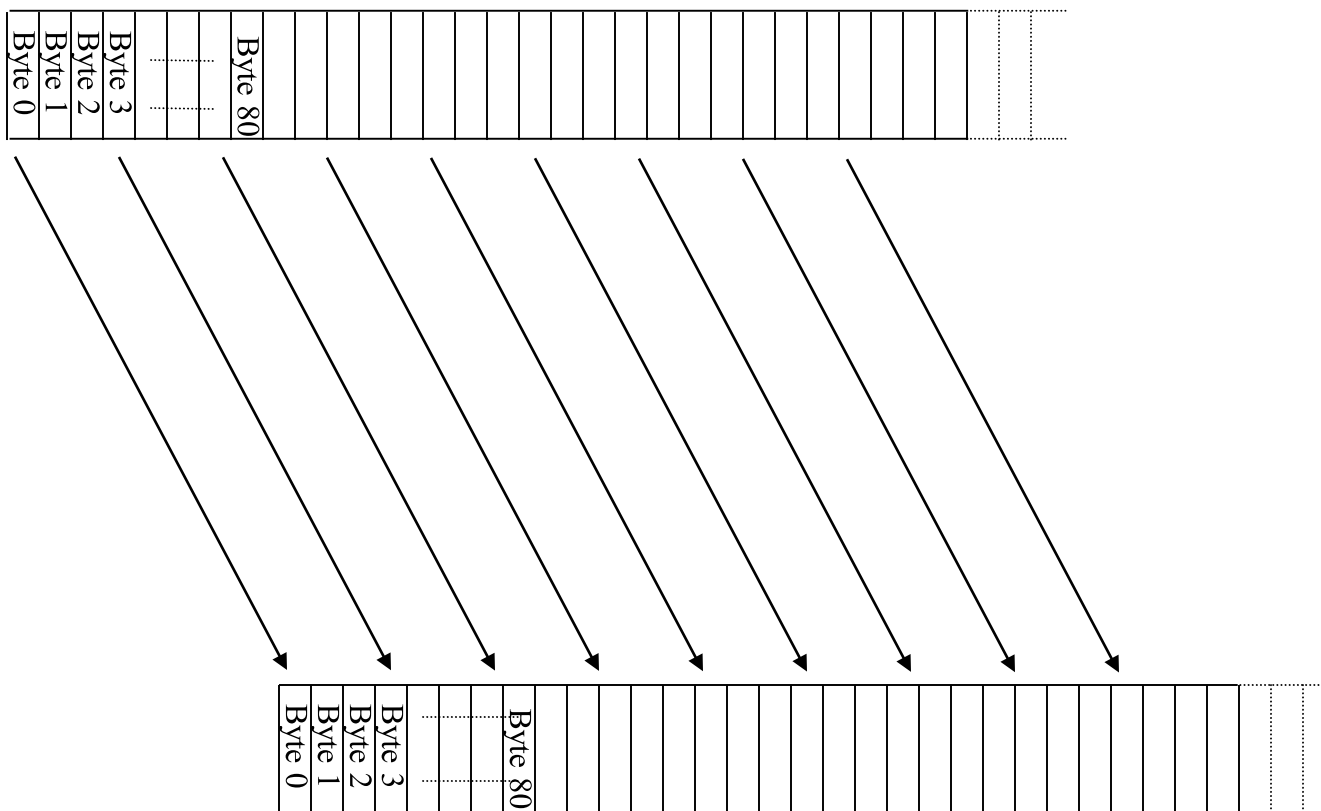
Byte offsets
(NOT packet id),
because TCP is a
byte stream

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



TCP “stream of bytes” service...

Application @ Host A

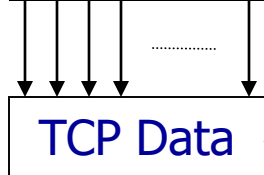
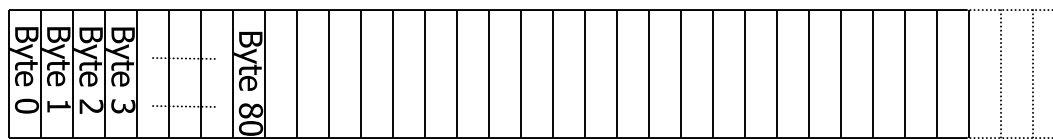


Application @ Host B



... provided using TCP “segments”

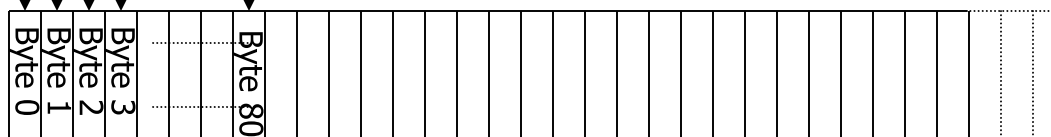
Host A



Segment sent when:

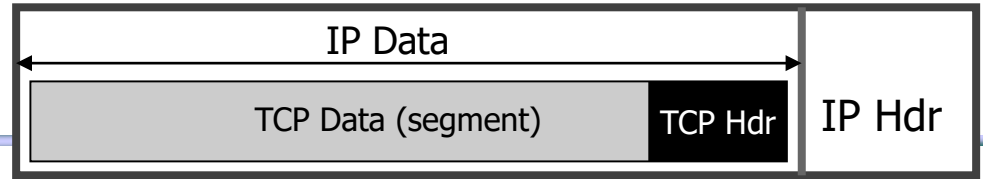
1. Segment full (Max Segment Size),
2. Not full, but times out

Host B





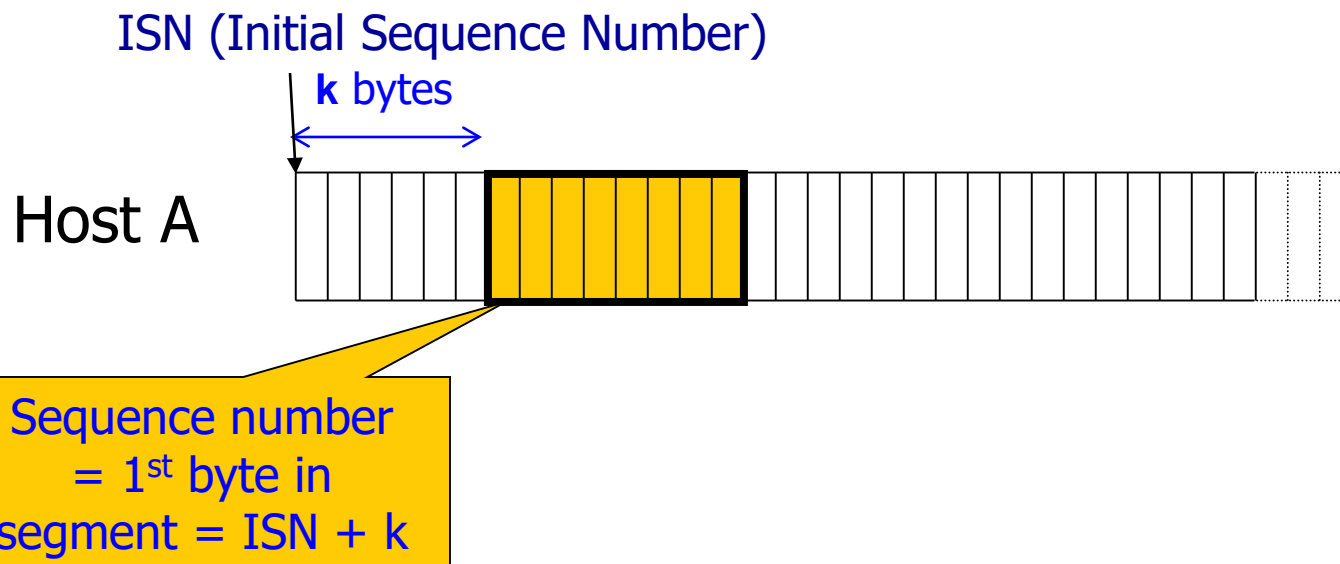
TCP segment



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



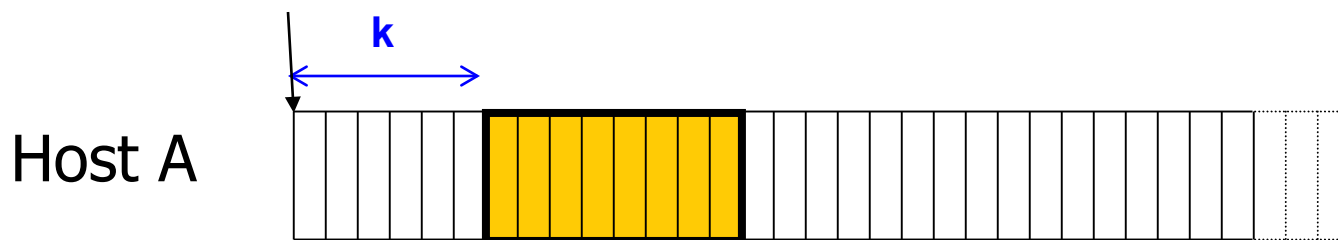
Sequence numbers





Sequence numbers

ISN (Initial Sequence Number)



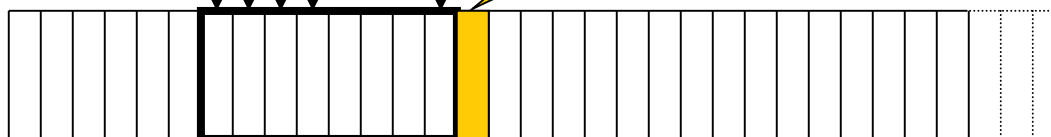
Sequence number
= 1st byte in
segment = $ISN + k$

TCP Data TCP HDR

ACK sequence number
= next expected byte
= $seqno + \text{length}(\text{data})$

TCP Data TCP HDR

Host B





TCP header

Starting byte
offset of data
carried in this
segment

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



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

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



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, 900, ...
- Assume the fifth packet (seqno 500) is lost, but no others
- Stream of ACKs will be:
 - 200, 300, 400, 500 (seqno:600), 500 (seqno:700), 500 (seqno:800), 500 (seqno:900),...



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?



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

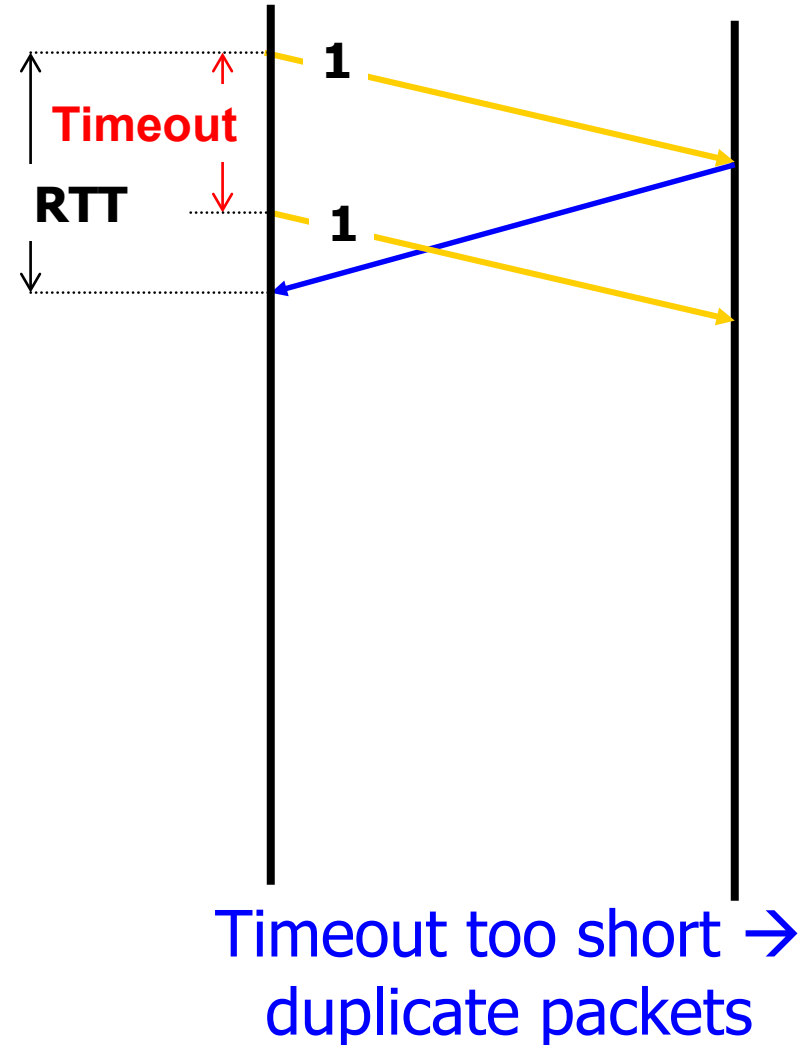
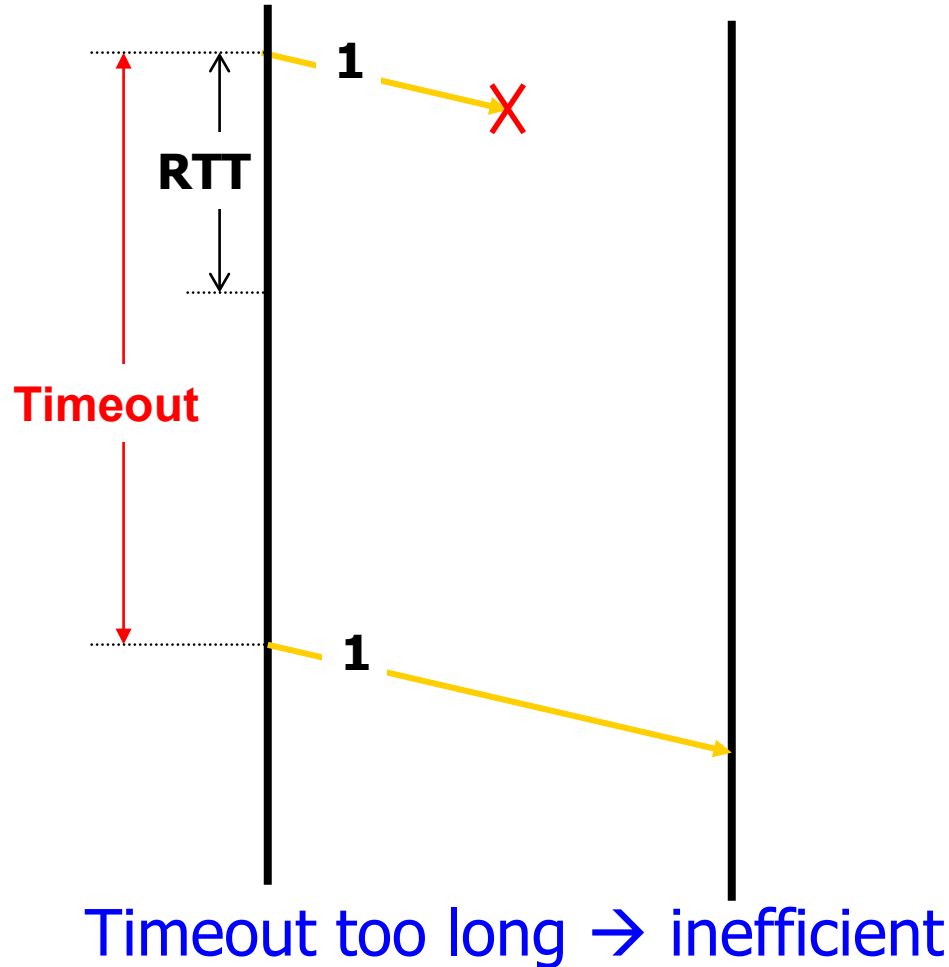


Retransmission timeout

- 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





Retransmission timeout

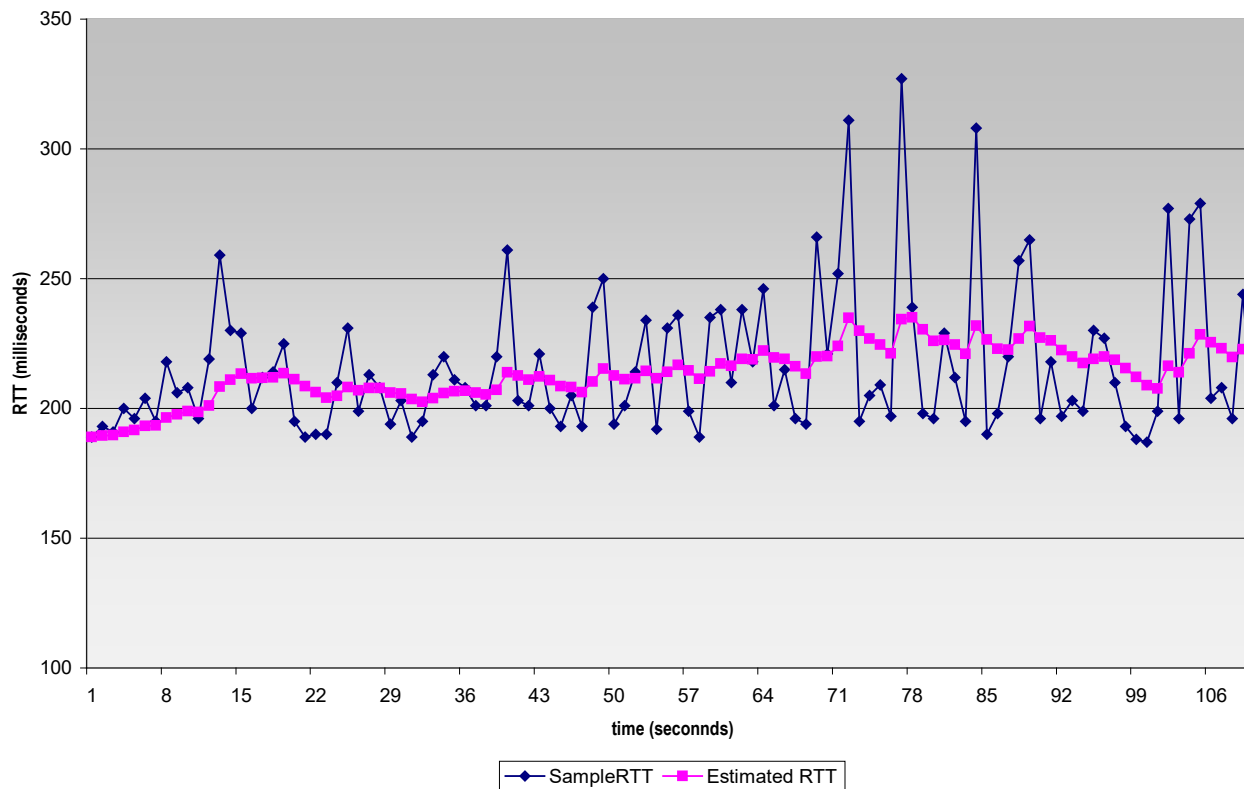
- If the sender hasn't received an ACK by timeout, retransmit the first packet in the window
- How to set timeout?
 - Too long: connection has low throughput
 - Too short: retransmit packet that was just delayed
- Solution: **make timeout proportional to RTT**
 - But how do we measure RTT?



RTT estimation

- Exponential weighted average of RTT samples

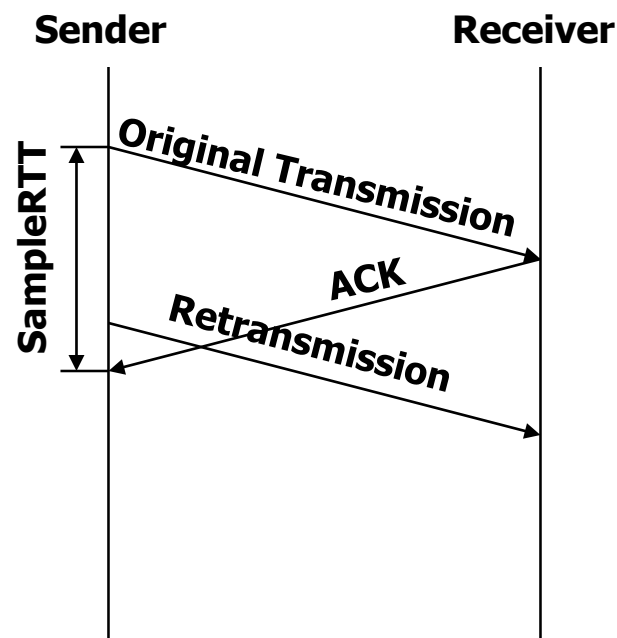
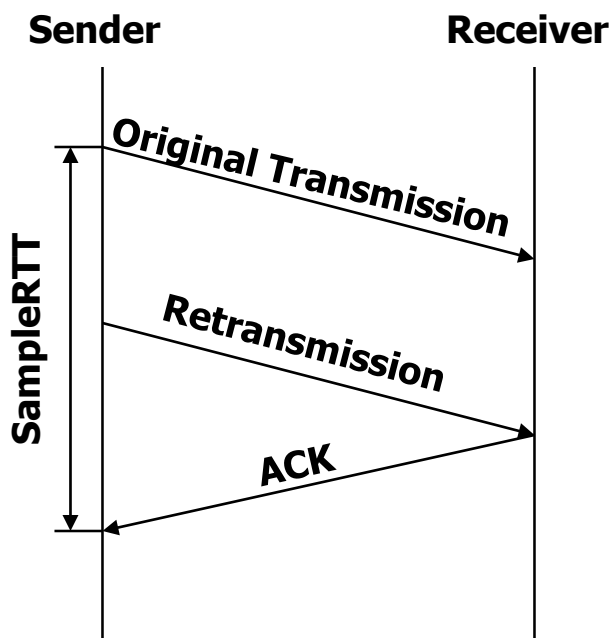
$$\text{EstimatedRTT} = (1 - \alpha) * \text{EstimatedRTT} + \alpha * \text{SampleRTT}$$





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 ≥ 60 sec)
 - Every time new measurement comes in (= successful original transmission), collapse RTO back to $2 \times \text{EstimatedRTT}$
- Insensitive to RTT variations



Jacobson/Karels algorithm

- **Problem**: need to better capture variability in RTT
 - Directly measure deviation
- Deviation = | SampleRTT – EstimatedRTT |
- DevRTT: exponential average of Deviation
- **RTO = EstimatedRTT + 4 x DevRTT**

$$SRTT(k+1) = (1-g) \times SRTT(k) + g \times RTT(k+1)$$

$$SERR(k+1) = RTT(k+1) - SRTT(k)$$

$$SDEV(k+1) = (1-h) \times SDEV(k) + h \times |SERR(k+1)|$$

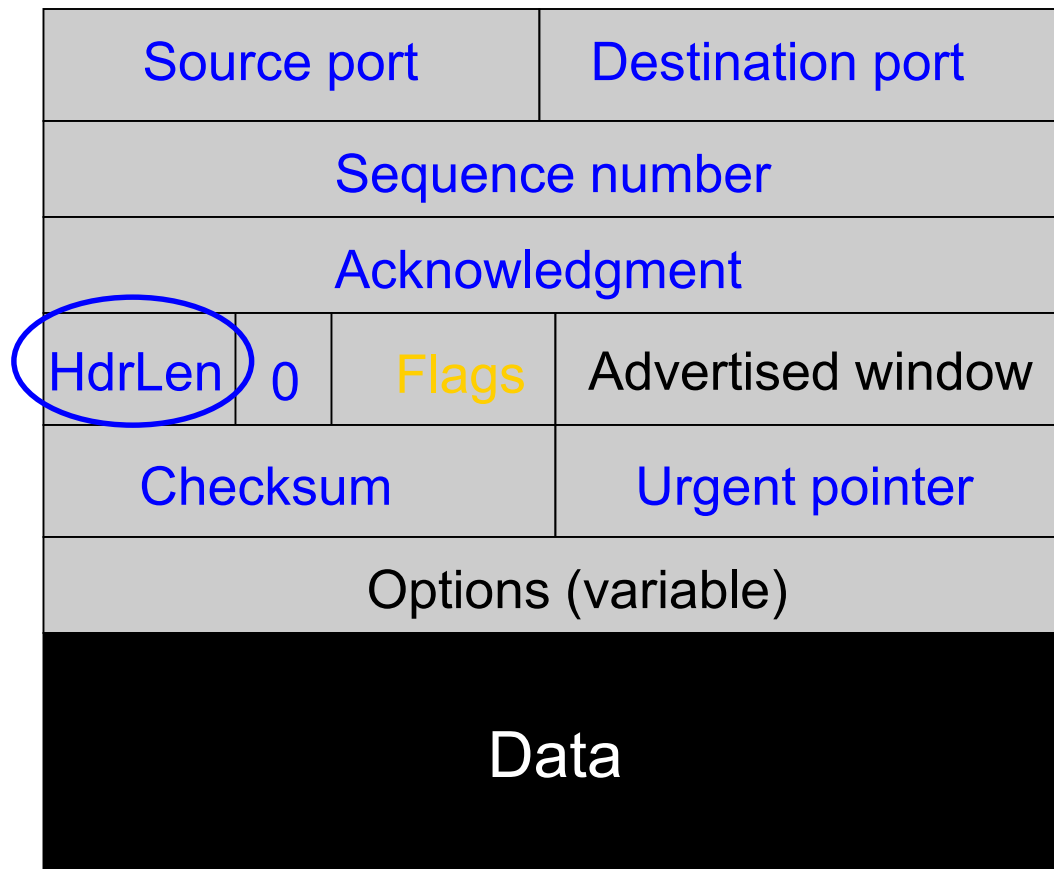
$$RTO(k+1) = SRTT(k+1) + f \times SDEV(k+1)$$

$$g = \frac{1}{8} = 0.125 \quad h = \frac{1}{4} = 0.25 \quad f = 2 \text{ or } 4$$



TCP header

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





TCP Connection Establishment



Connection Establishment

■ 2-way handshake

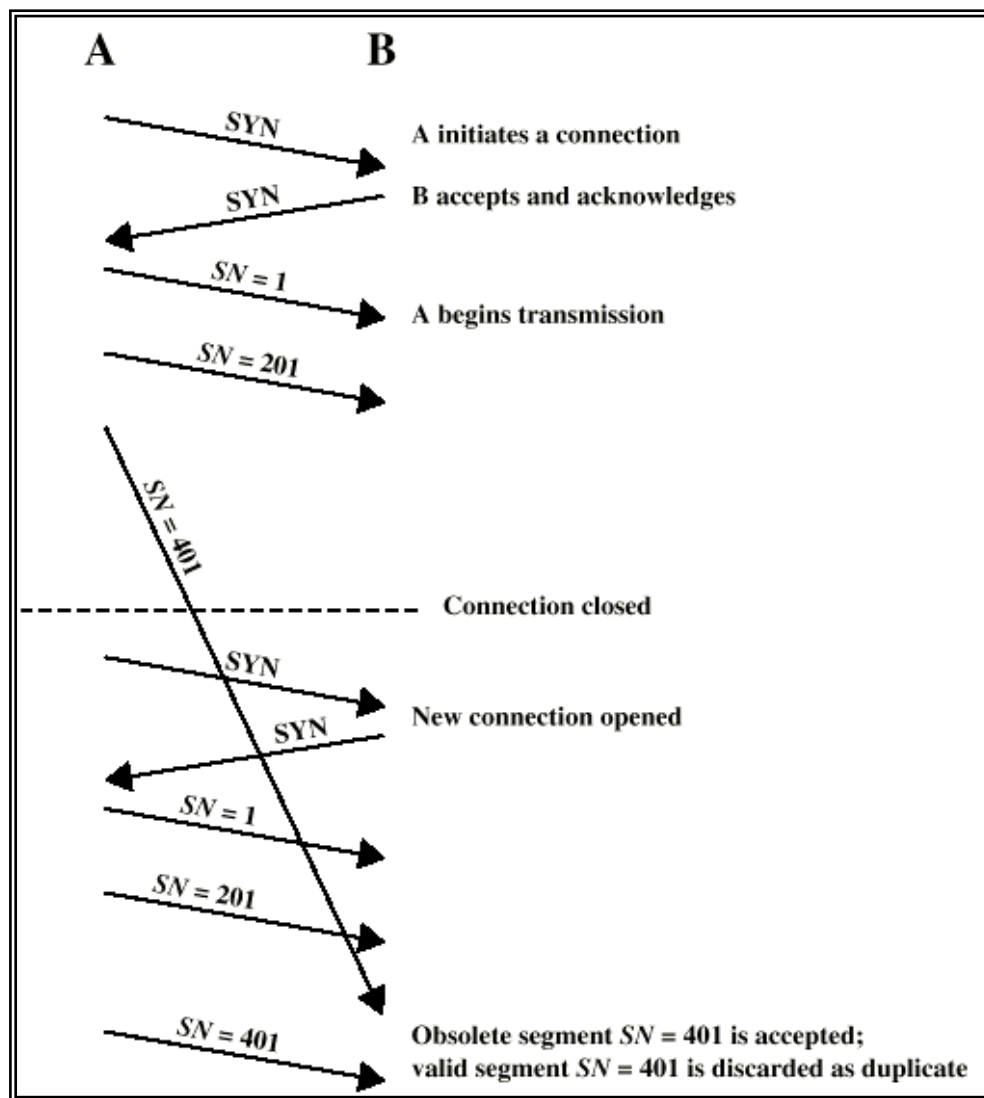
- A sends SYN, B replies with SYN
- Lost SYNs handled by re-transmission
- Ignore duplicate SYNs once connected

■ Problem

- How to recognize **slipped segments from old connection**
- How to recognize duplicated **obsolete SYN**



2-Way Handshake: Slipped Data Segment



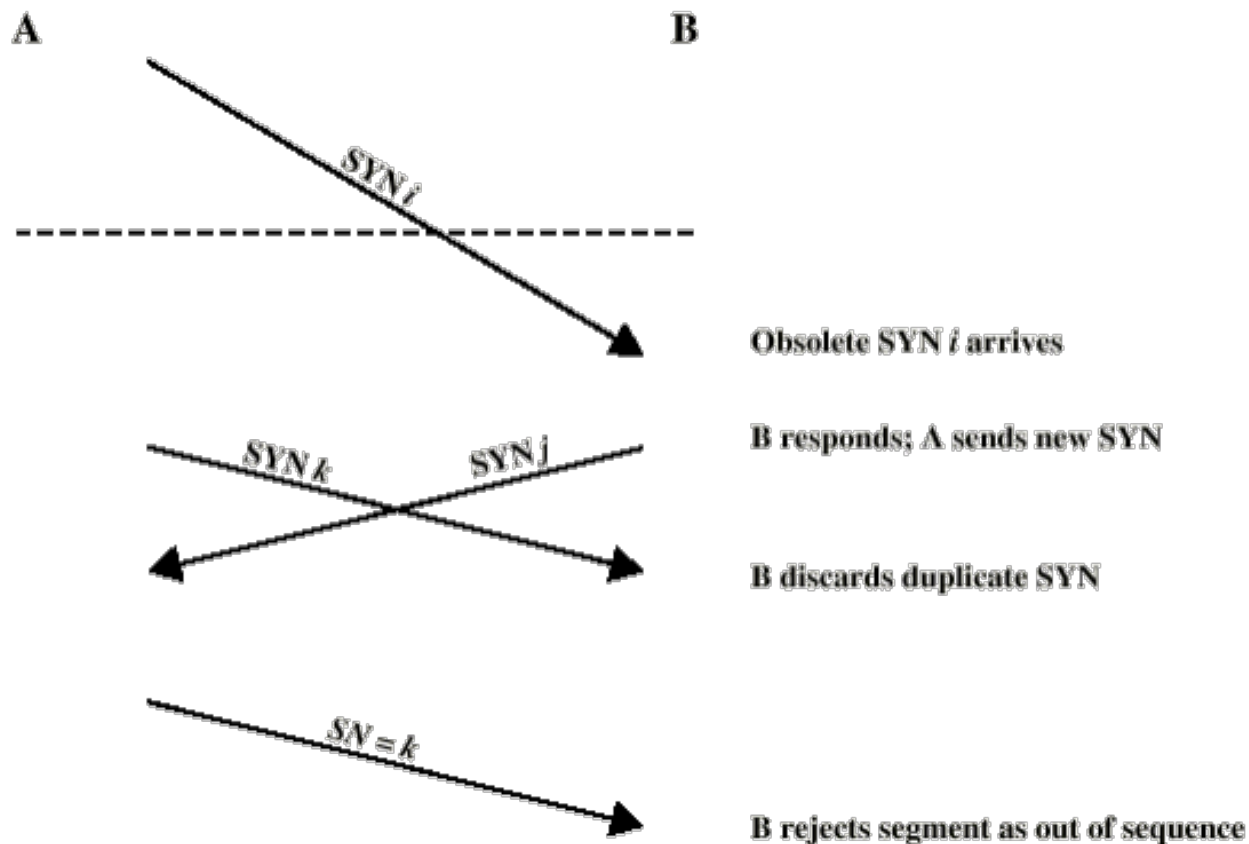


Initial Sequence Number (ISN)

- Handle
 - Start each new connection with a different **initial sequence number (ISN)** far from previous connection
 - The connection request is of the form SYN $i+1$, where i is the sequence number of the first data segment that will be sent on this connection.
- However:



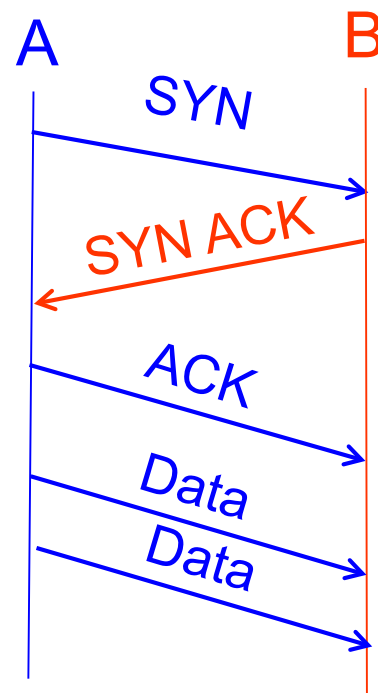
2-Way Handshake: Obsolete SYN





Solution: three-way handshake

- 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



三方握手：确认对方的**SYN**和序号



TCP header

Flags:

SYN

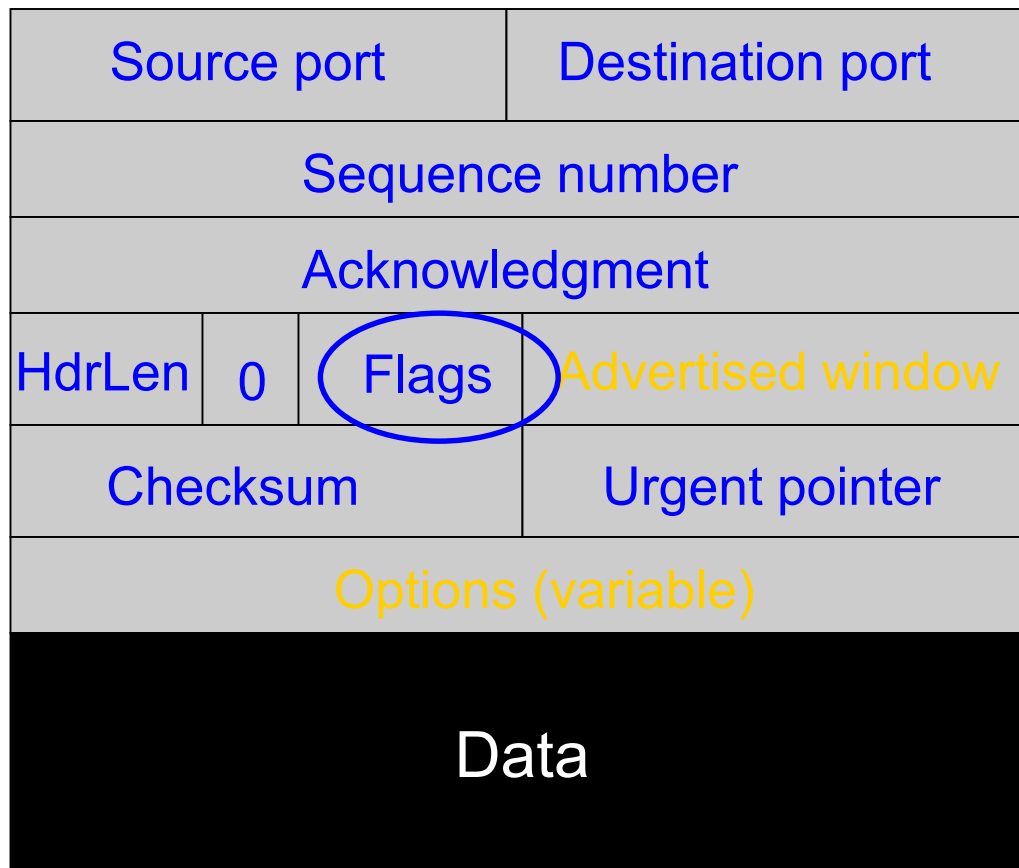
ACK

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	Advertised window
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	Advertised window
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	Advertised window
Checksum		Urgent pointer	



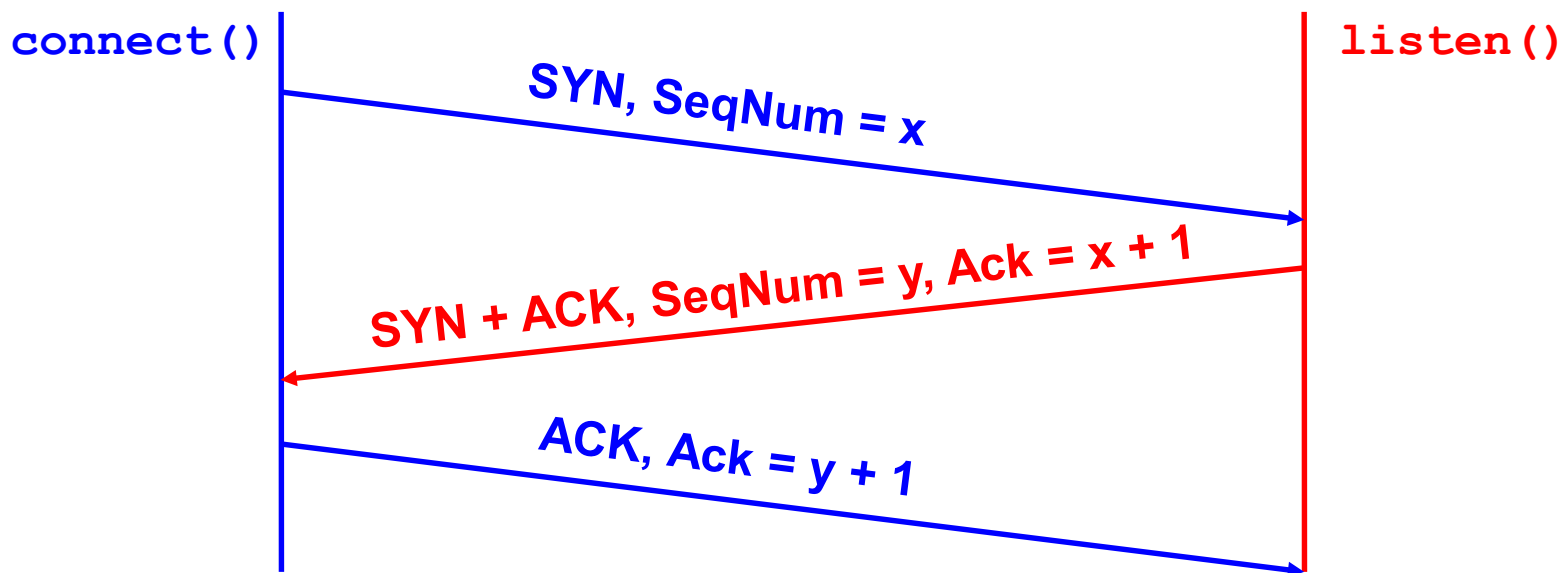
TCP's 3-Way handshaking

*Active
Open*

Client (initiator)

*Passive
Open*

Server





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

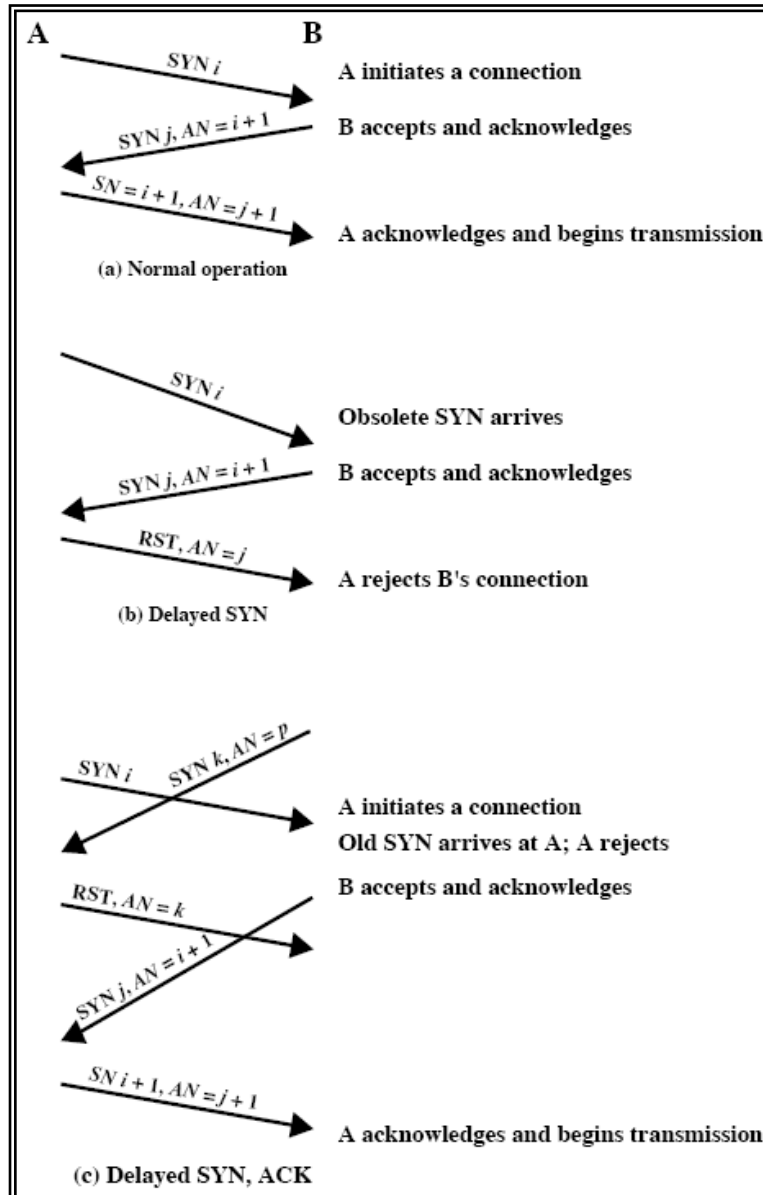


SYN loss and web downloads

- 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



Three-Way Handshake: Examples

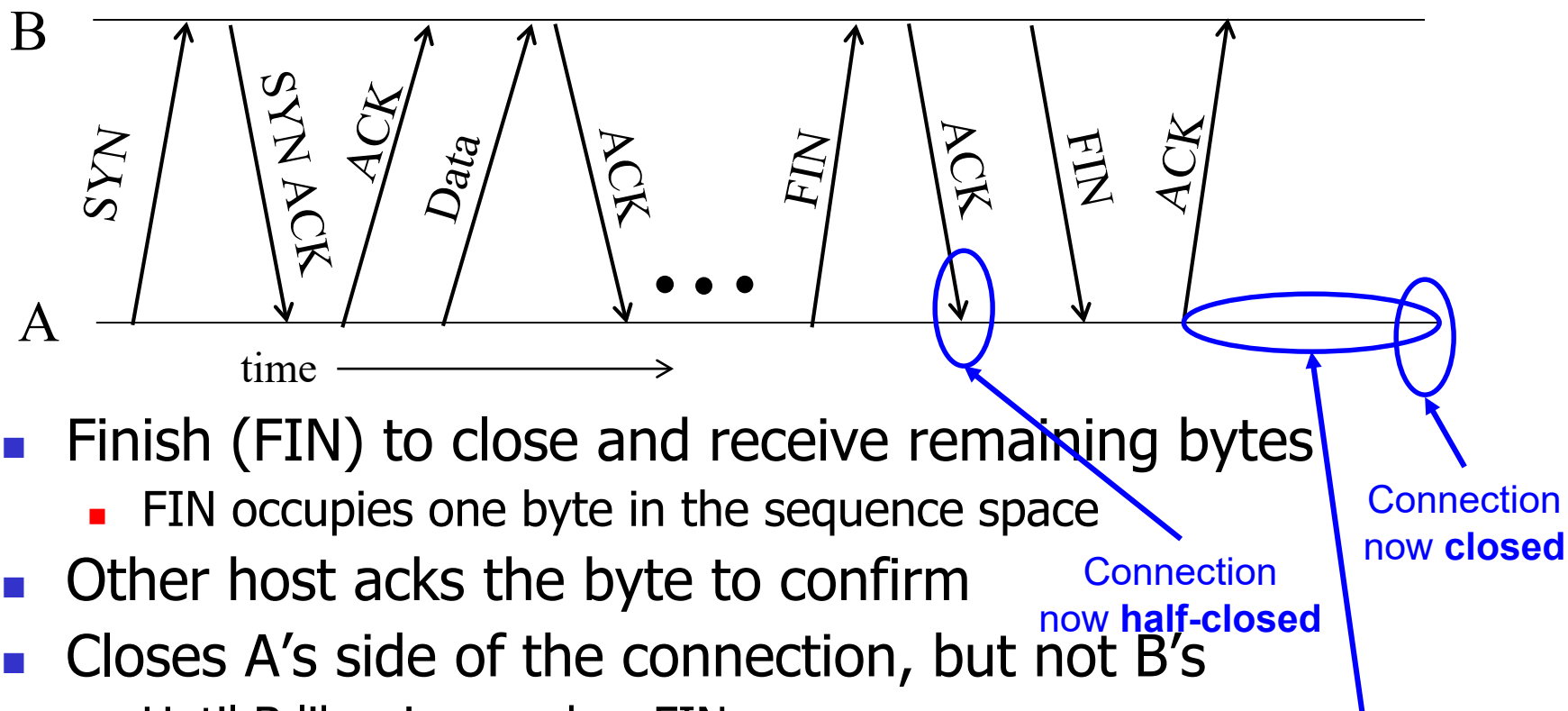




TCP connection teardown



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

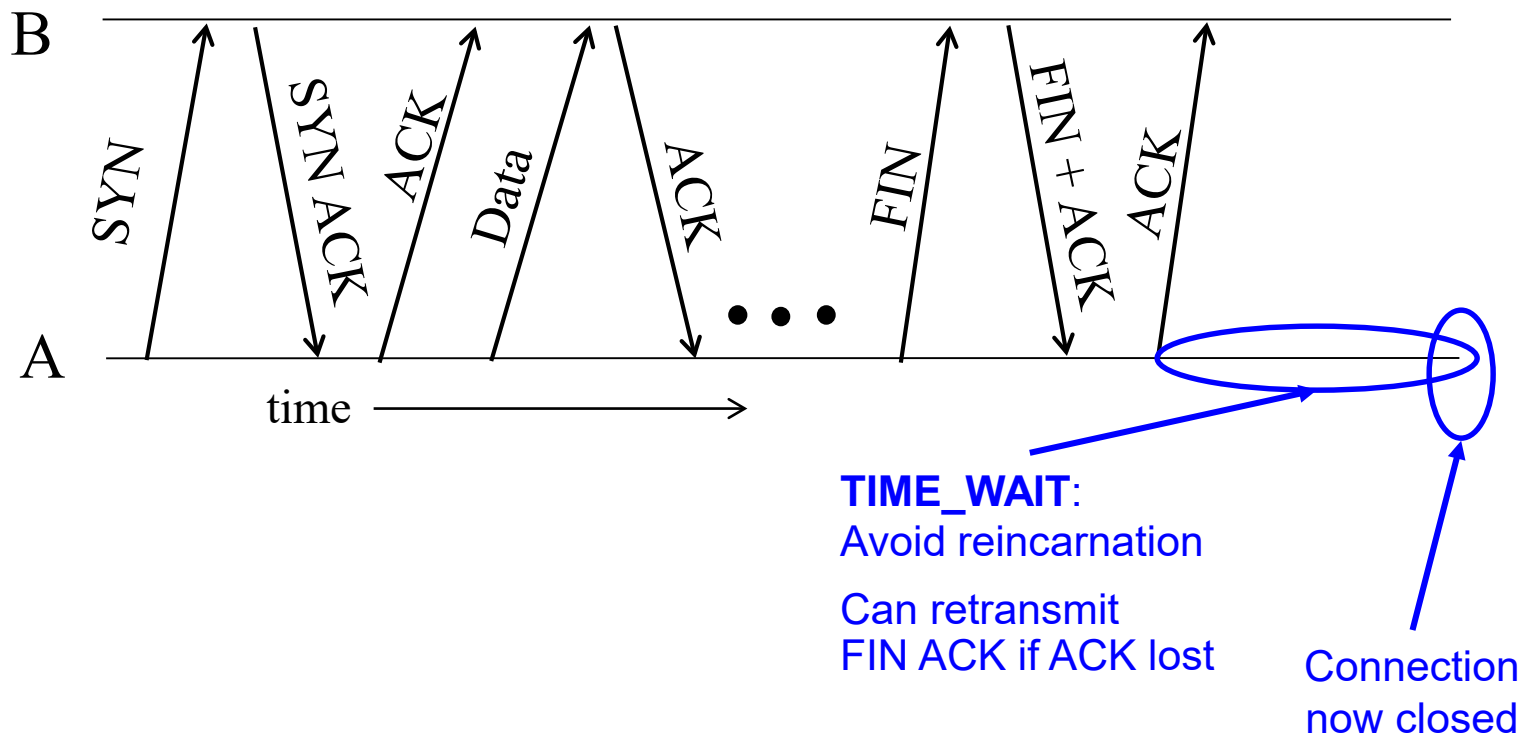
TIME_WAIT:

Avoid reincarnation

B will retransmit FIN
if ACK is lost



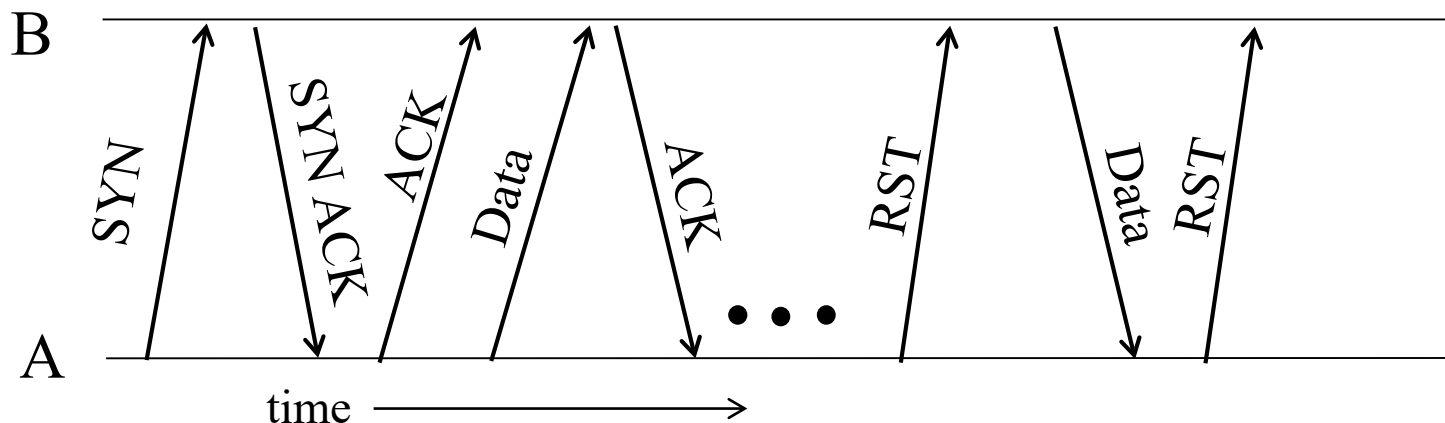
Normal termination, both together



- 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



TCP header

Flags:

SYN

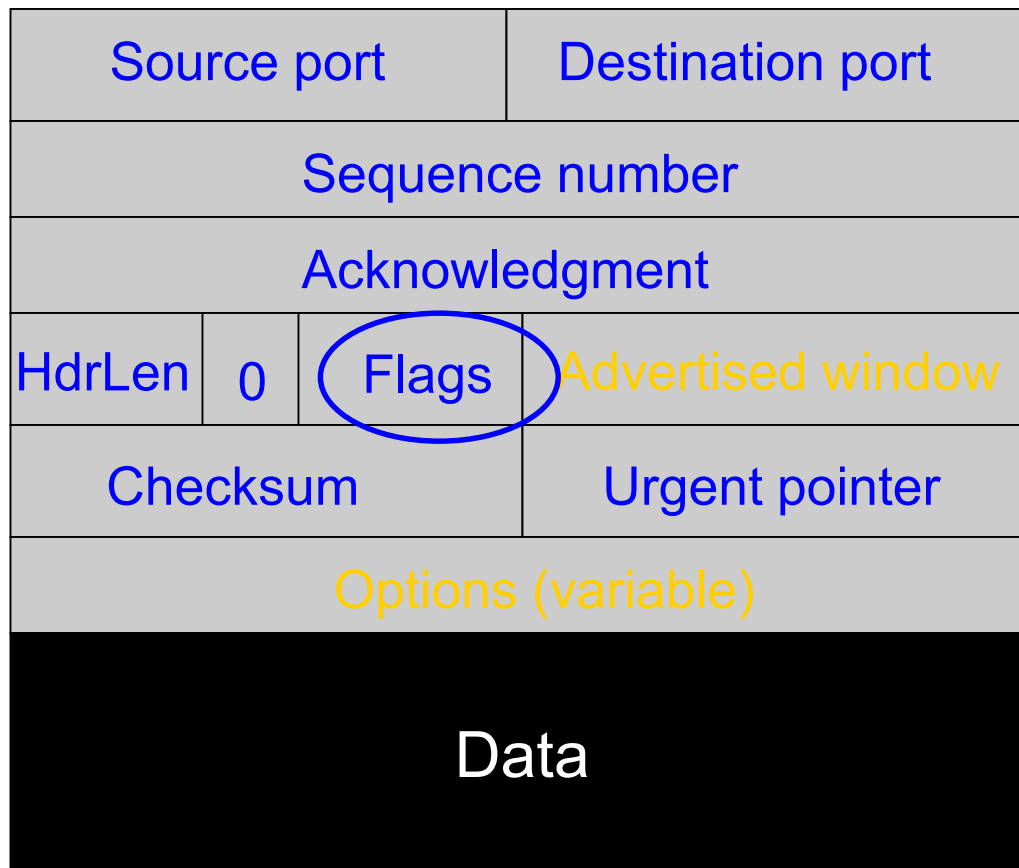
ACK

FIN

RST

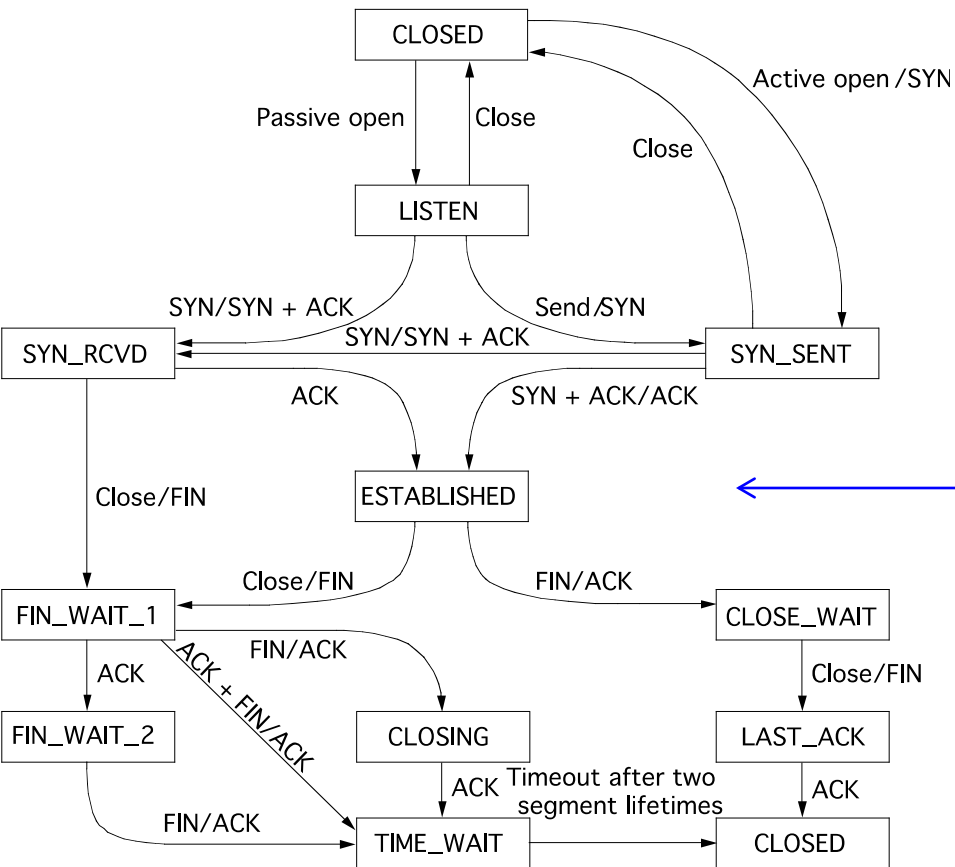
PSH

URG





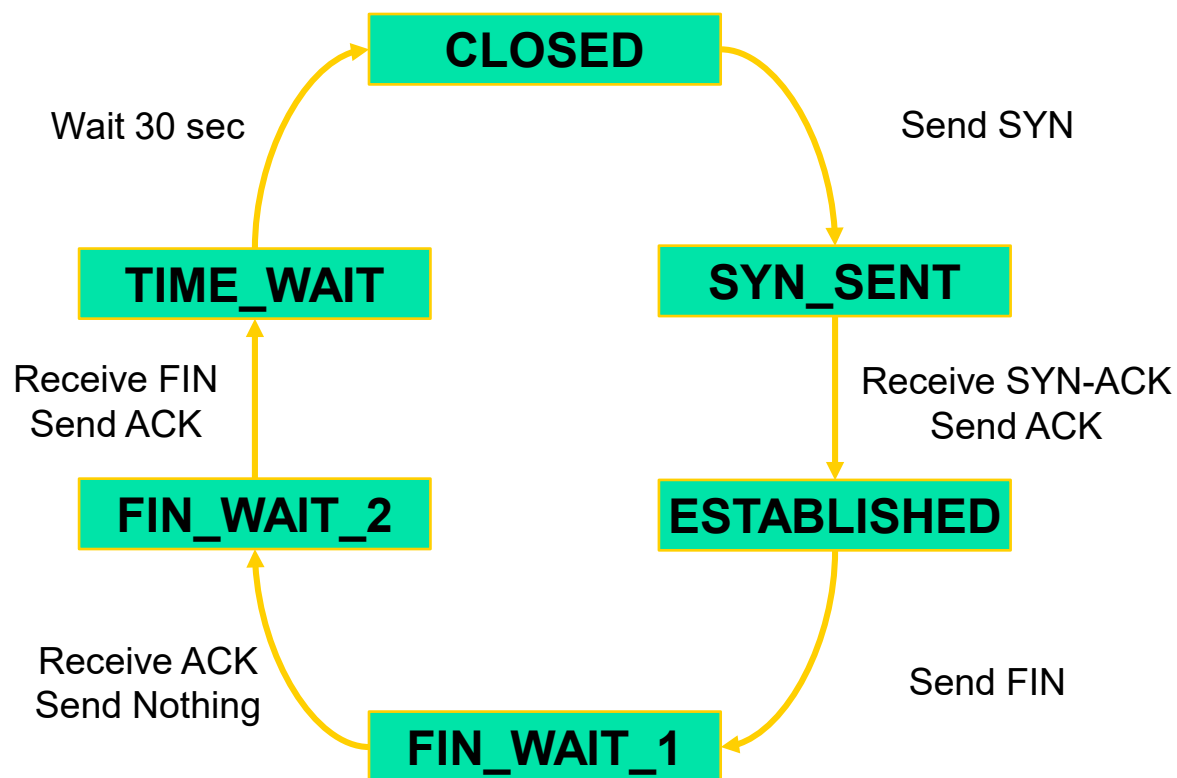
TCP state transitions



Data, ACK exchanges are in here

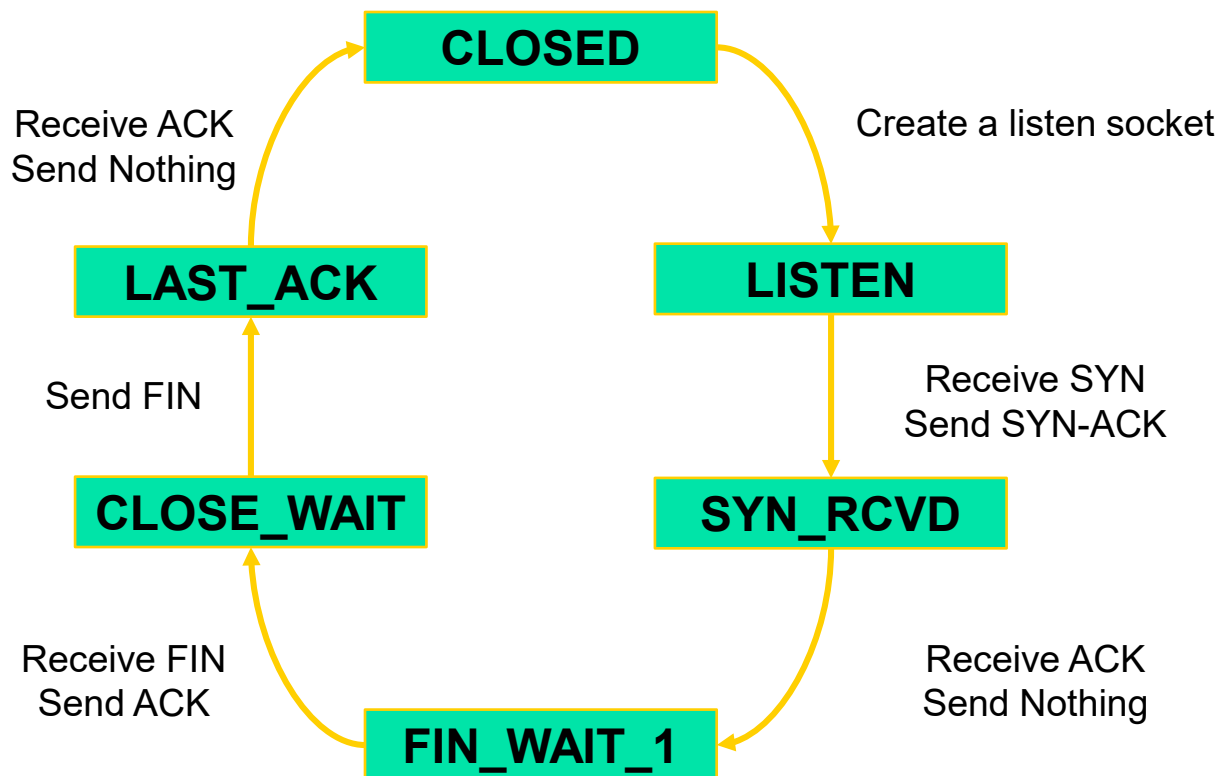


TCP client lifecycle





TCP server lifecycle





Summary

- UDP
- TCP
- TCP header fields
- TCP Connection Establishment
 - 2-way and 3-way handshake
- TCP connection teardown