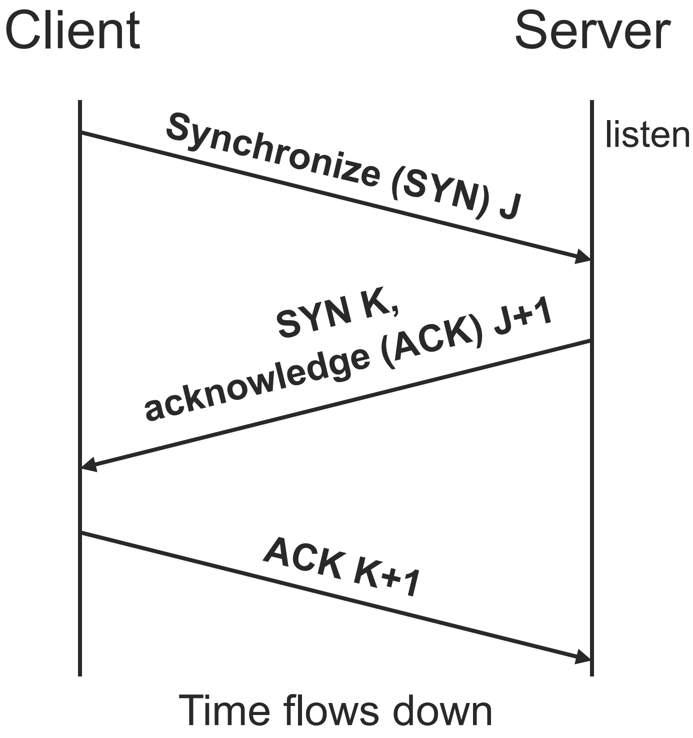
TCP Internals

TCP Usage Model

- Connection setup
  - 3-way handshake
- Data transport
  - Sender writes data
  - TCP
    - Breaks data into segments
    - Sends each segment over IP
    - Retransmits, reorders and removes duplicates as necessary
  - Receiver reads some data
- Teardown
  - 4 step exchange

TCP Connection Establishment

- 3-Way Handshake
  - Sequence Numbers
    - J,K
  - Message Types
    - Synchronize (SYN)
    - Acknowledge (ACK)
  - Passive Open
    - Server listens for connection from client
  - Active Open
    - Client initiates connection to server

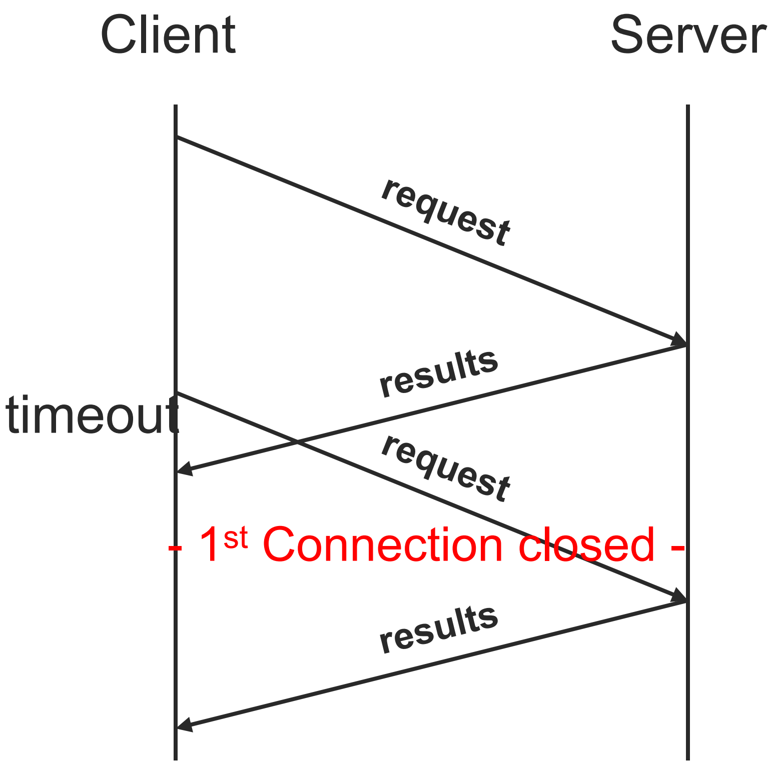


Purpose of the handshake

- Why use a handshake before sending / processing data?
- Suppose we don’t wait for the handshake
  - send data (e.g., HTTP request) along with SYN
  - deliver to application
  - send some results (e.g., index.html) along with SYN ACK
- What could go wrong?
  - Hint: remember packets can be delayed, dropped, duplicated, …

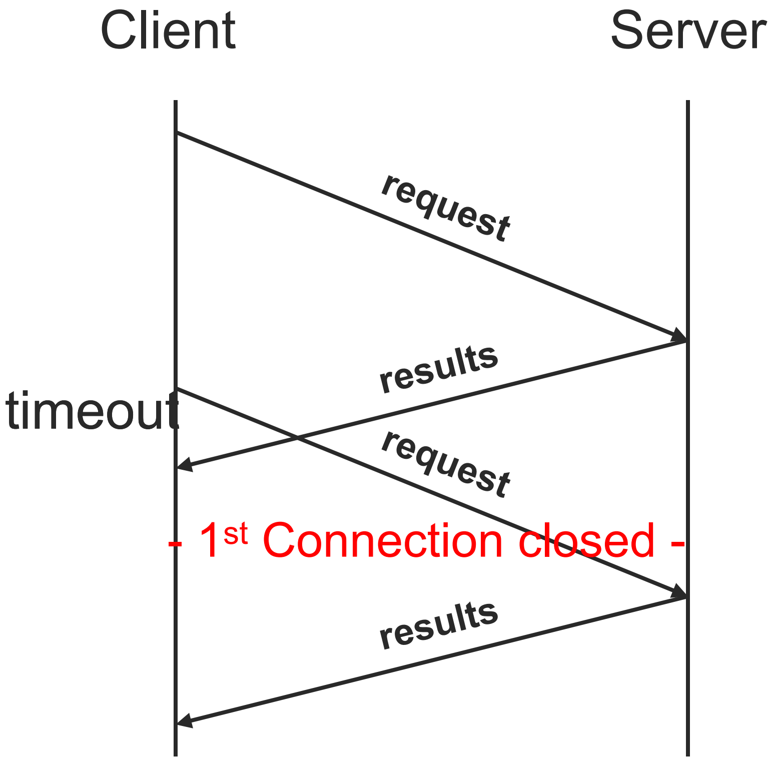
Purpose of the handshake

- Why use a handshake before sending / processing data?
- Duplicated packet causes data to be sent to application twice
- Why does handshake fix this?



Purpose of the handshake

- If server receives request a second time, it responds with SYN ACK a second time
- But sender will not subsequently respond with ACK ("what is this garbage I just received??")

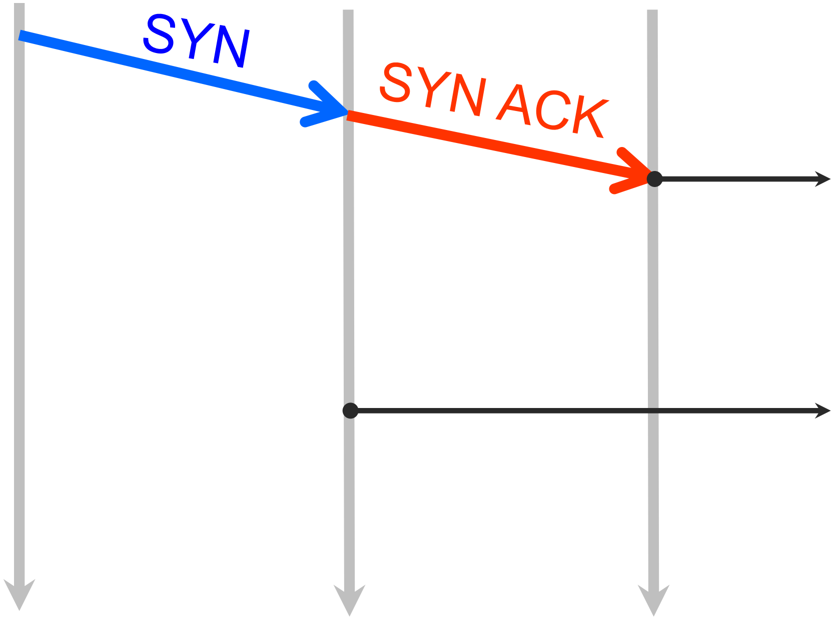


Another purpose of the handshake

- No handshake == security hole
  - Attacker sends request
  - …but spoofs source address, using address of a victim (C)
  - Server happily sends massive amounts of data to victim
  - Attacker repeats for 10,000 web servers
  - Massive denial of service attack, almost free and anonymous for the attacker!
- Used in the largest distributed denial of service (DDoS) attacks in 2008, 2009, and 2010
  - Use services that lack handshake (e.g., DNS over UDP)
  - Amplification factor 1:76 in 2008!

Another purpose of the handshake

- Handshake lets server verify source address is real



Doesn’t match a connection initiated by C: ignore (or reply with *reset*)

No ACK received after timeout: drop connection without sending data

Q: does this prevent reflection attack?

A: No, but at least it prevents amplification

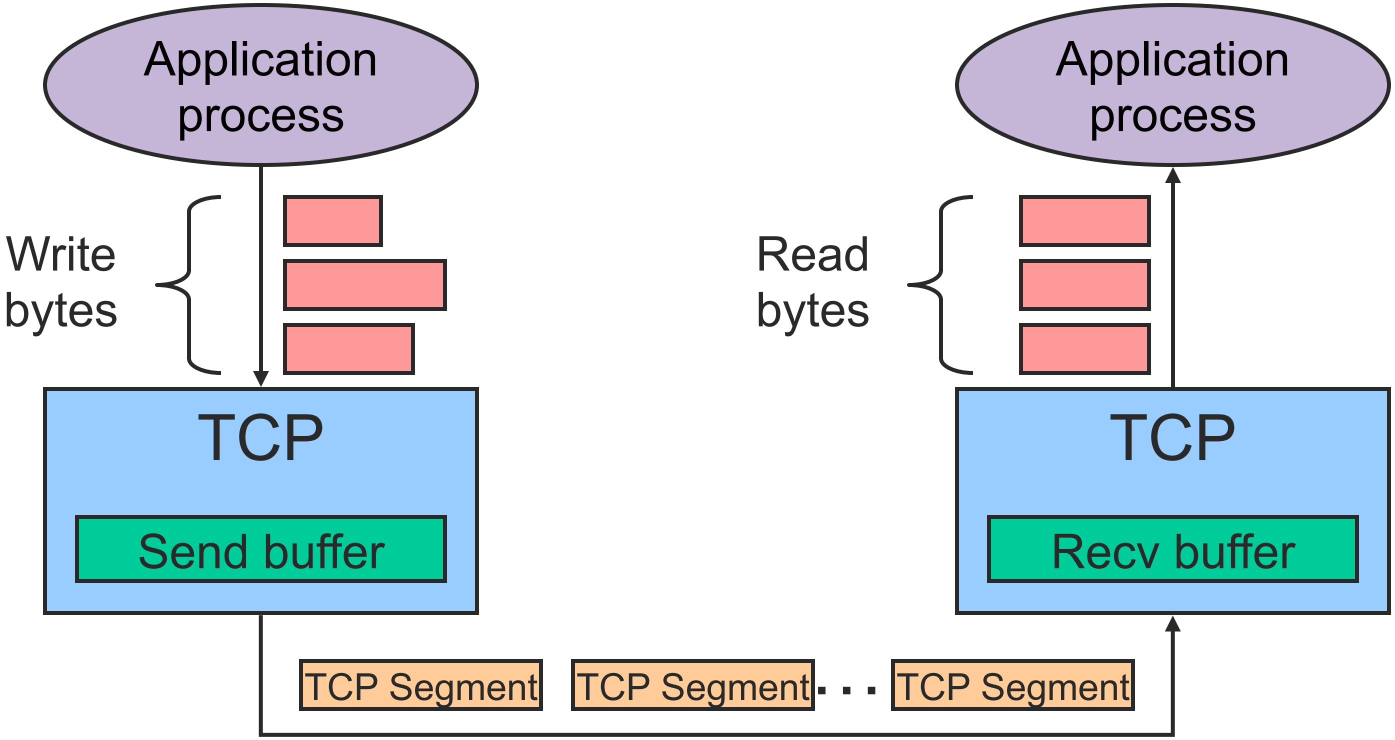
Handshaking

- Internet was not designed for accountability
  - Hard to tell where a packet came from
  - ISPs filter suspicious packets: sometimes easy, sometimes hard, and sometimes not done
    - And the Internet is not secure until everyone filters
- More generally, Internet was not designed for security
  - Vulnerabilities in most of the core protocols
  - Even with handshake, early designs are vulnerable
    - Had predictable Initial Sequence Number (why’s that bad?)
    - Because security was not initial goal of the handshake

TCP Data Transport

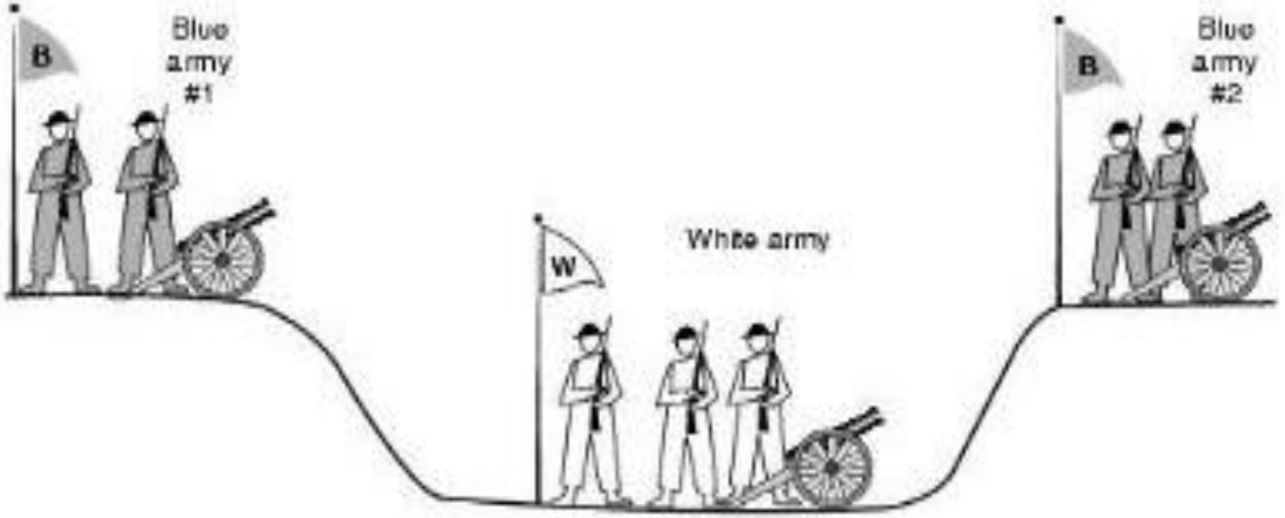
- Data broken into segments
  - Limited by maximum segment size (MSS)
  - Defaults to 352 bytes
  - Negotiable during connection setup
  - Typically set to
    - MTU of directly connected network – size of TCP and IP headers
- Three events cause a segment to be sent
  - ≥ MSS bytes of data ready to be sent
  - Explicit PUSH operation by application
  - Periodic timeout

TCP Byte Stream



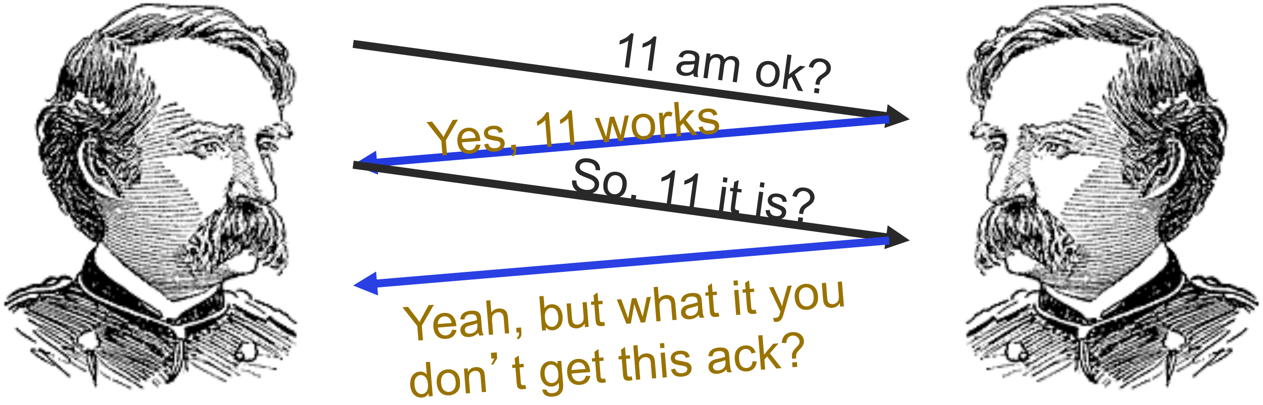
TCP Connection Termination

- Two generals problem
  - Enemy camped in valley
  - Two generals’ hills separated by enemy
  - Communication by unreliable messengers
  - Generals need to agree whether to attack or retreat



Two generals problem

- Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
  - No, even if all messages get through



11 am ok?

Yes, 11 works

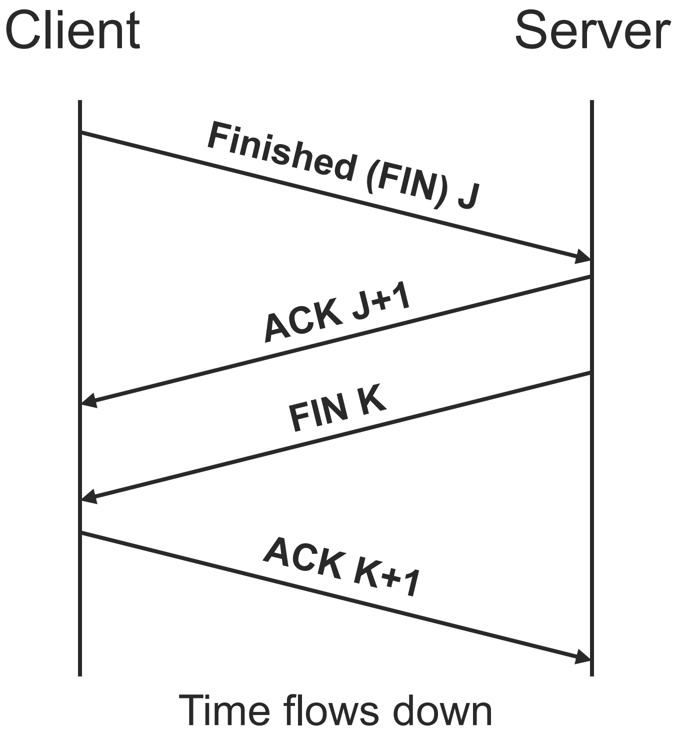
So, 11 it is?

Yeah, but what it you don’ t get this ack?

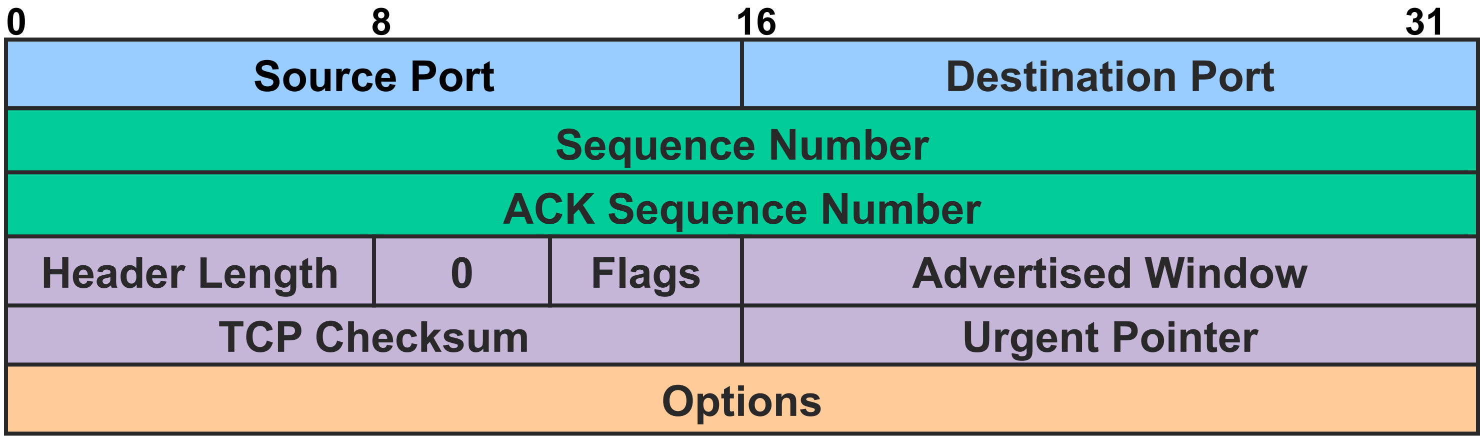
- No way to be sure last message gets through!

TCP Connection Termination

- Message Types
  - Finished (FIN)
  - Acknowledge (ACK)
- Active Close
  - Sends no more data
- Passive close
  - Accepts no more data



TCP Segment Header Format



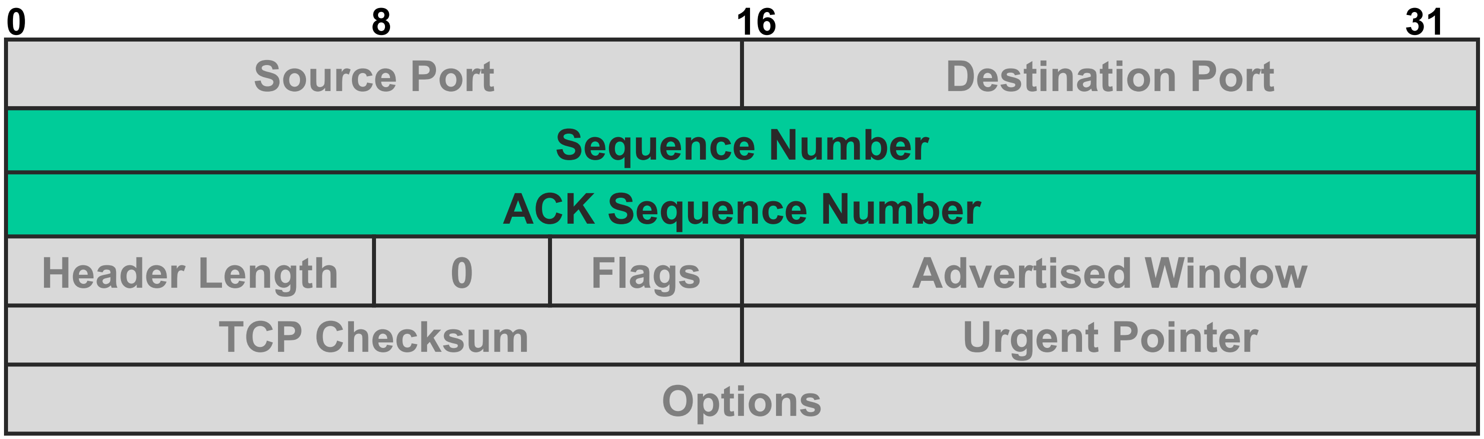
0	8		16	31
Source Port			Destination Port	
Sequence Number				
ACK Sequence Number				
Header Length	0	Flags	Advertised Window	
TCP Checksum			Urgent Pointer	
Options				

TCP Segment Header Format

<table>
<tr><th>0</th><th>8</th><th></th><th>16</th><th>31</th></tr>
<tr><td colspan="3"><b>Source Port</b></td><td colspan="2"><b>Destination Port</b></td></tr>
<tr><td colspan="5">Sequence Number</td></tr>
<tr><td colspan="5">ACK Sequence Number</td></tr>
<tr><td>Header Length</td><td>0</td><td>Flags</td><td colspan="2">Advertised Window</td></tr>
<tr><td colspan="3">TCP Checksum</td><td colspan="2">Urgent Pointer</td></tr>
<tr><td colspan="5">Options</td></tr>
</table>

- 16-bit source and destination ports

TCP Segment Header Format

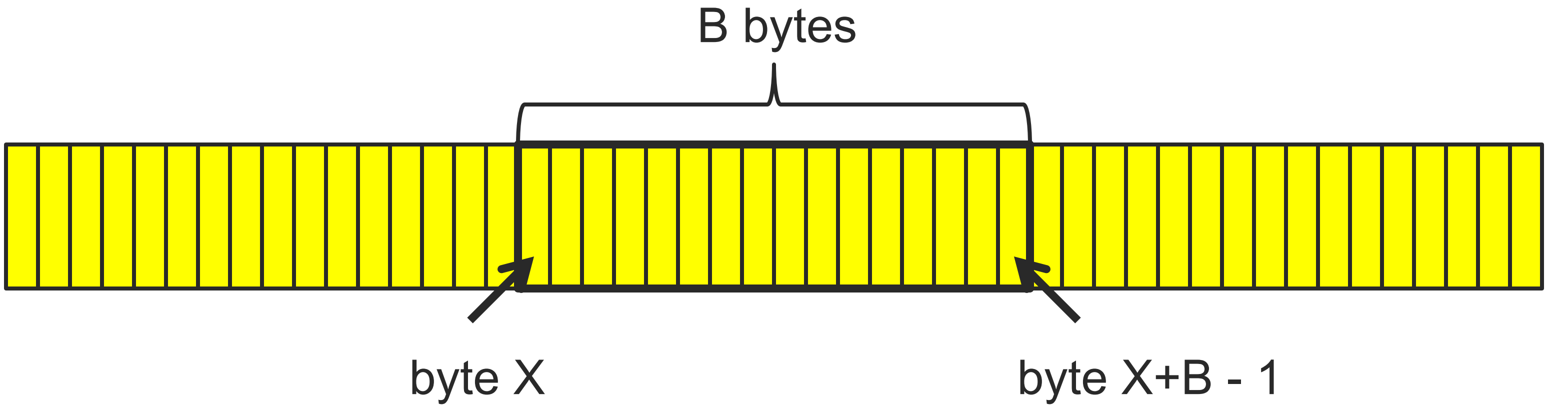


<table>
<tr><th>0</th><th>8</th><th></th><th>16 31</th></tr>
<tr><td colspan="3">Source Port</td><td>Destination Port</td></tr>
<tr><td colspan="4">**Sequence Number**</td></tr>
<tr><td colspan="4">**ACK Sequence Number**</td></tr>
<tr><td>Header Length</td><td>0</td><td>Flags</td><td>Advertised Window</td></tr>
<tr><td colspan="3">TCP Checksum</td><td>Urgent Pointer</td></tr>
<tr><td colspan="4">Options</td></tr>
</table>

- 32-bit send and ACK sequence numbers

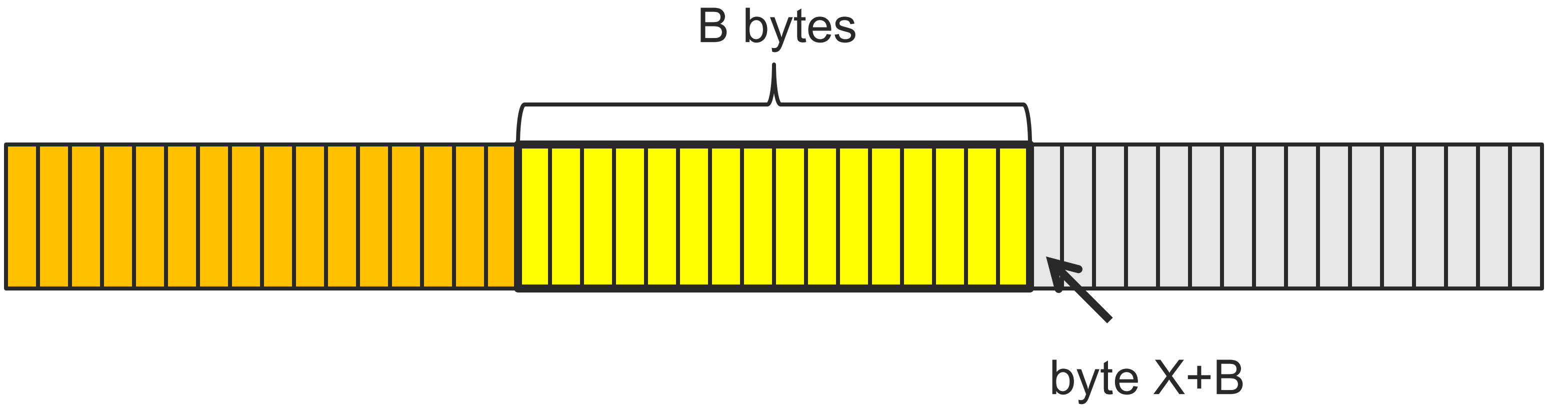
ACKing and Sequence Numbers

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



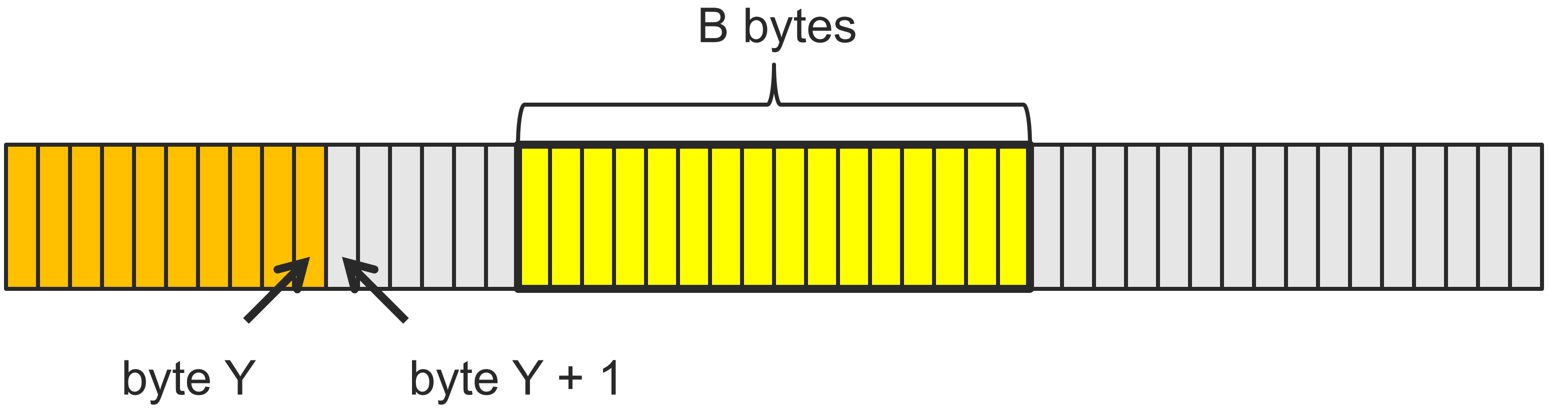
ACKing and Sequence Numbers

- 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)

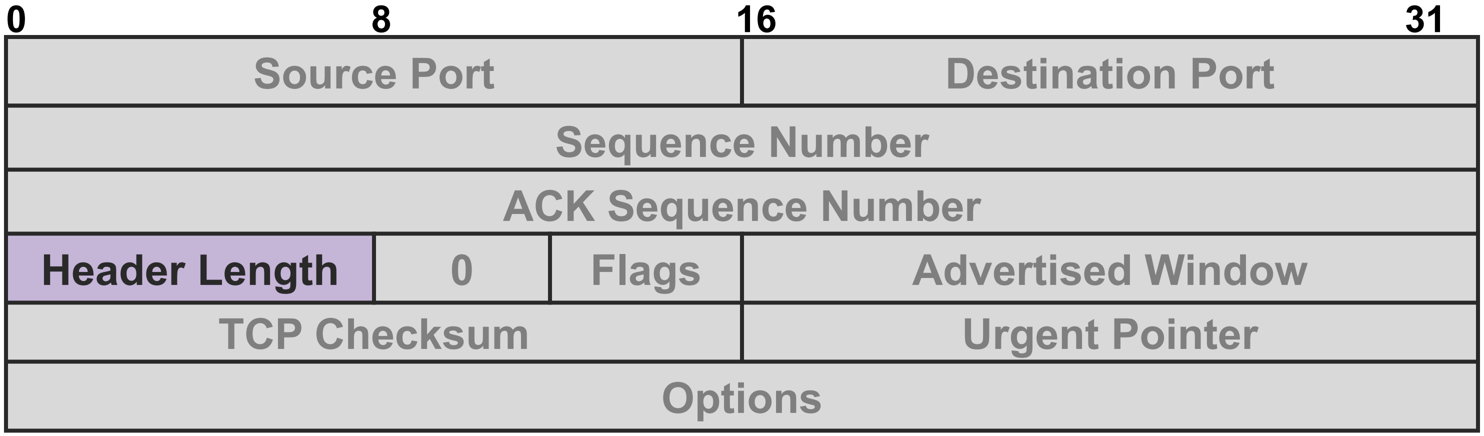


ACKing and Sequence Numbers

- Upon receipt of packet, receiver sends an ACK
  - If highest byte already received is some smaller value Y
    - ACK acknowledges Y+1
    - Even if this has been ACKed before



TCP Segment Header Format



0 8 16 31

Source Port | Destination Port

Sequence Number

ACK Sequence Number

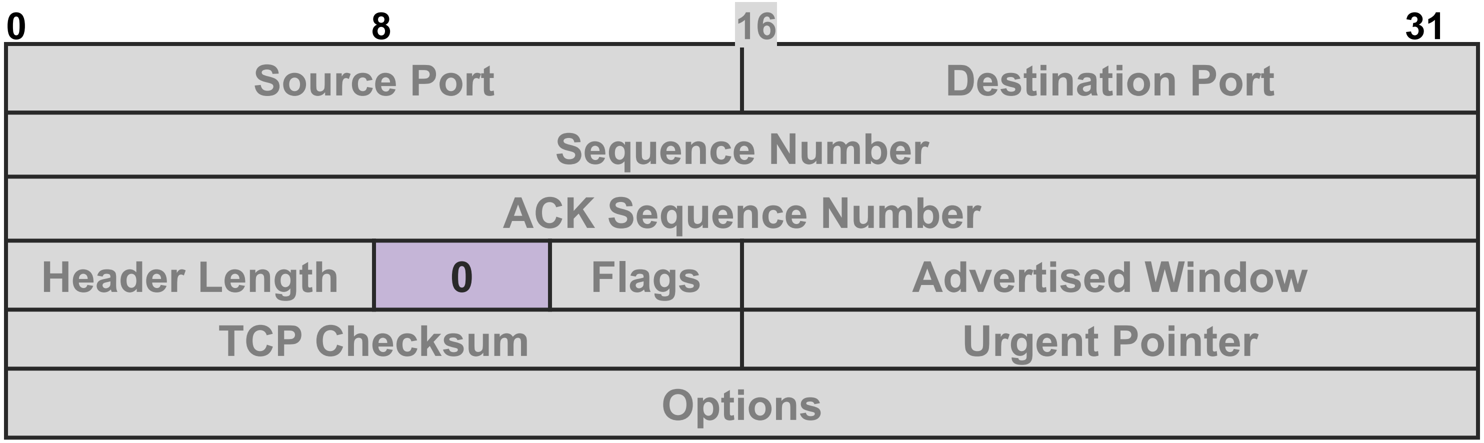
Header Length | 0 | Flags | Advertised Window

TCP Checksum | Urgent Pointer

Options

- 4-bit header length in 4-byte words
  - Minimum 5 bytes
  - Offset to first data byte

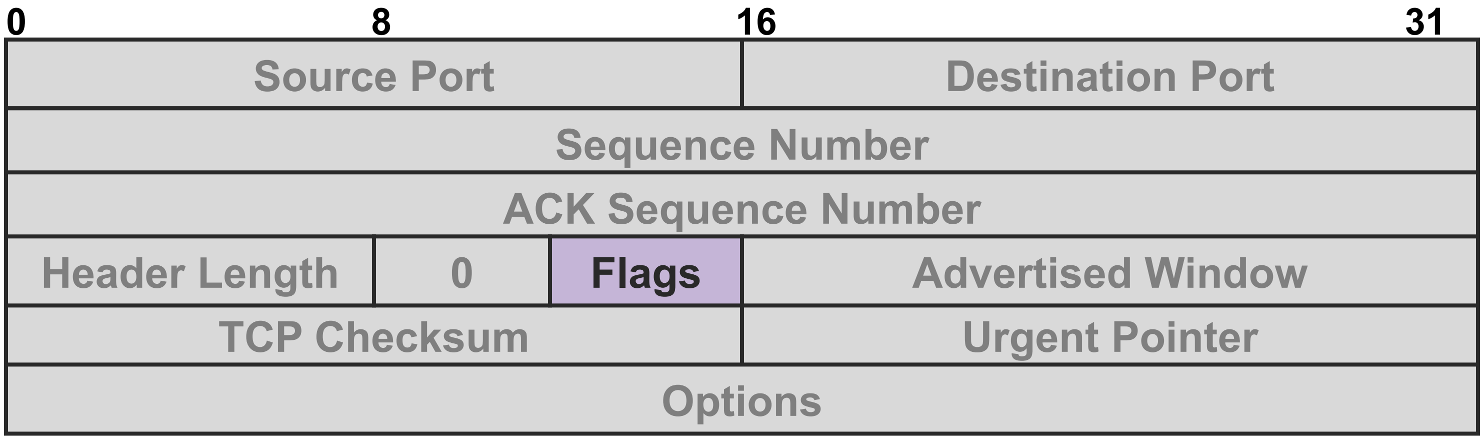
TCP Segment Header Format



<table>
<tr><td>0</td><td>8</td><td></td><td>16</td><td>31</td></tr>
<tr><td colspan="3">Source Port</td><td colspan="2">Destination Port</td></tr>
<tr><td colspan="5">Sequence Number</td></tr>
<tr><td colspan="5">ACK Sequence Number</td></tr>
<tr><td>Header Length</td><td>0</td><td>Flags</td><td colspan="2">Advertised Window</td></tr>
<tr><td colspan="3">TCP Checksum</td><td colspan="2">Urgent Pointer</td></tr>
<tr><td colspan="5">Options</td></tr>
</table>

- Reserved
  - Must be 0

TCP Segment Header Format



0 8 16 31

Source Port | Destination Port

Sequence Number

ACK Sequence Number

Header Length | 0 | Flags | Advertised Window

TCP Checksum | Urgent Pointer

Options

- 6 1-bit flags

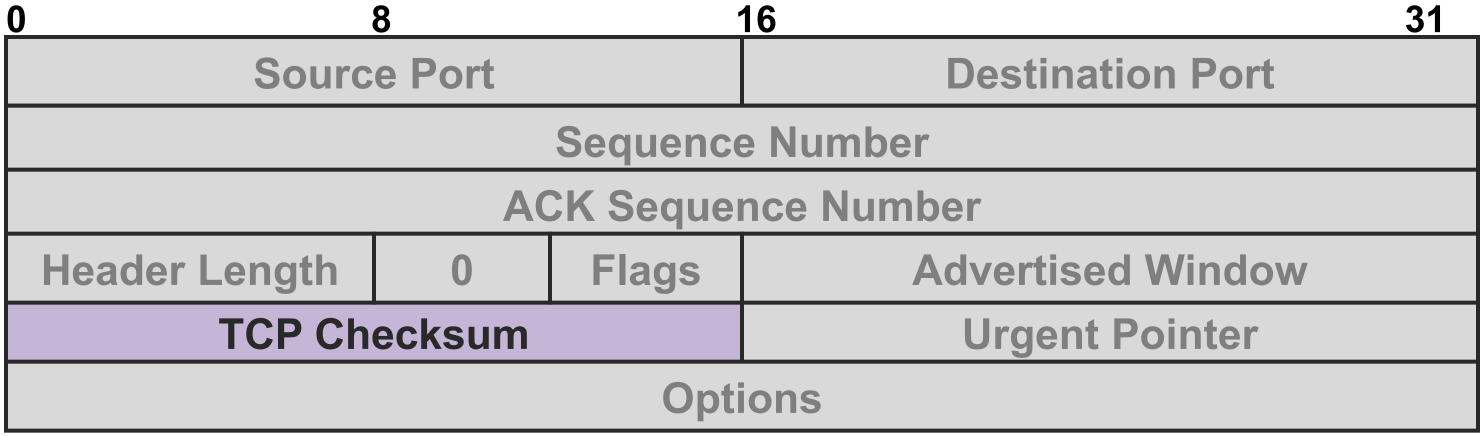
URG:	Contains urgent data	RST:	Reset connection
ACK:	Valid ACK seq. number	SYN:	Synchronize for setup
PSH:	Do not delay data delivery	FIN:	Final segment for teardown

TCP Segment Header Format

<table>
<tr><th>0</th><th>8</th><th></th><th>16</th><th>31</th></tr>
<tr><td colspan="3">Source Port</td><td colspan="2">Destination Port</td></tr>
<tr><td colspan="5">Sequence Number</td></tr>
<tr><td colspan="5">ACK Sequence Number</td></tr>
<tr><td>Header Length</td><td>0</td><td>Flags</td><td colspan="2">**Advertised Window**</td></tr>
<tr><td colspan="3">TCP Checksum</td><td colspan="2">Urgent Pointer</td></tr>
<tr><td colspan="5">Options</td></tr>
</table>

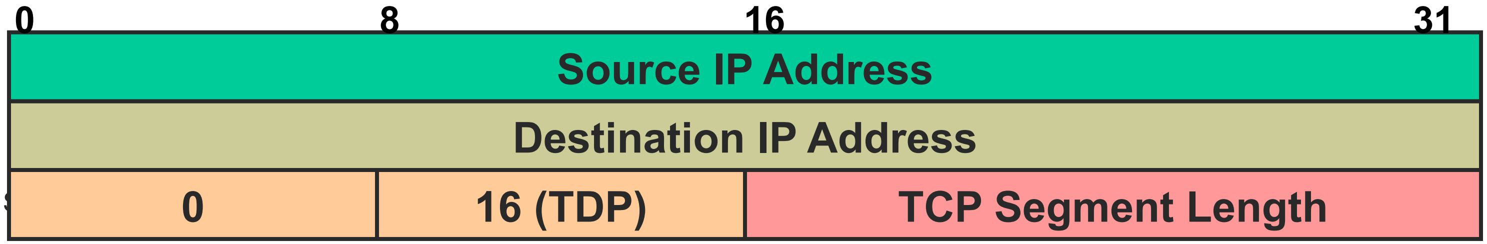
- 16-bit advertised window
  - Space remaining in receive window

TCP Segment Header Format



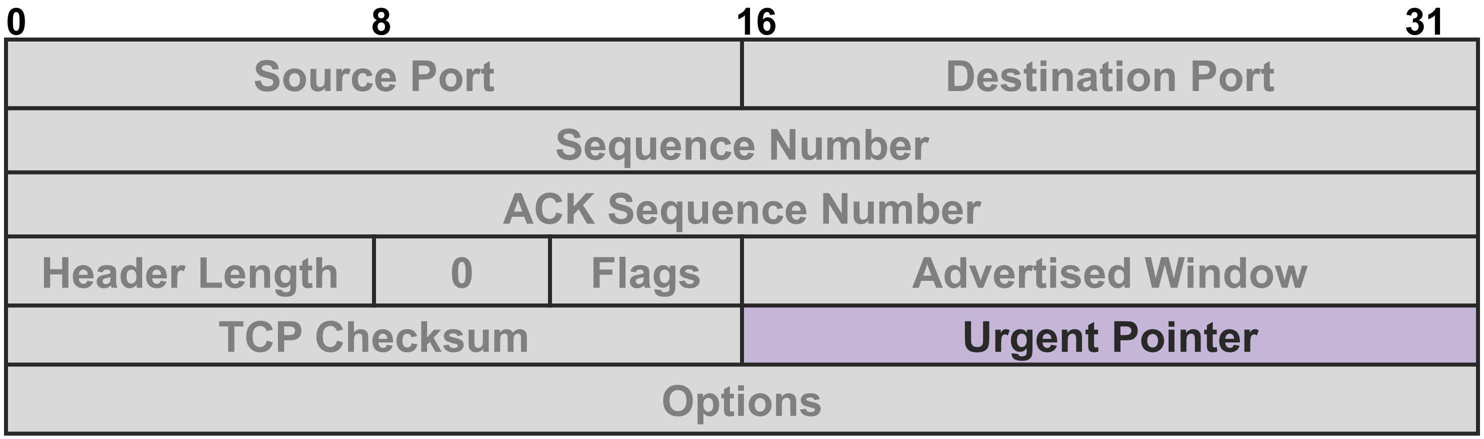
0	8	16	31
Source Port		Destination Port	
Sequence Number			
ACK Sequence Number			
Header Length	0	Flags	Advertised Window
TCP Checksum		Urgent Pointer	
Options			

- 16-bit checksum
  - Uses IP checksum algorithm
  - Computed on header, data and pseudo header



0	8	16	31
Source IP Address			
Destination IP Address			
0	16 (TDP)	TCP Segment Length	

TCP Segment Header Format



0 8 16 31

Source Port | Destination Port

Sequence Number

ACK Sequence Number

Header Length | 0 | Flags | Advertised Window

TCP Checksum | **Urgent Pointer**

Options

- 16-bit urgent data pointer
  - If URG = 1
  - Index of last byte of urgent data in segment

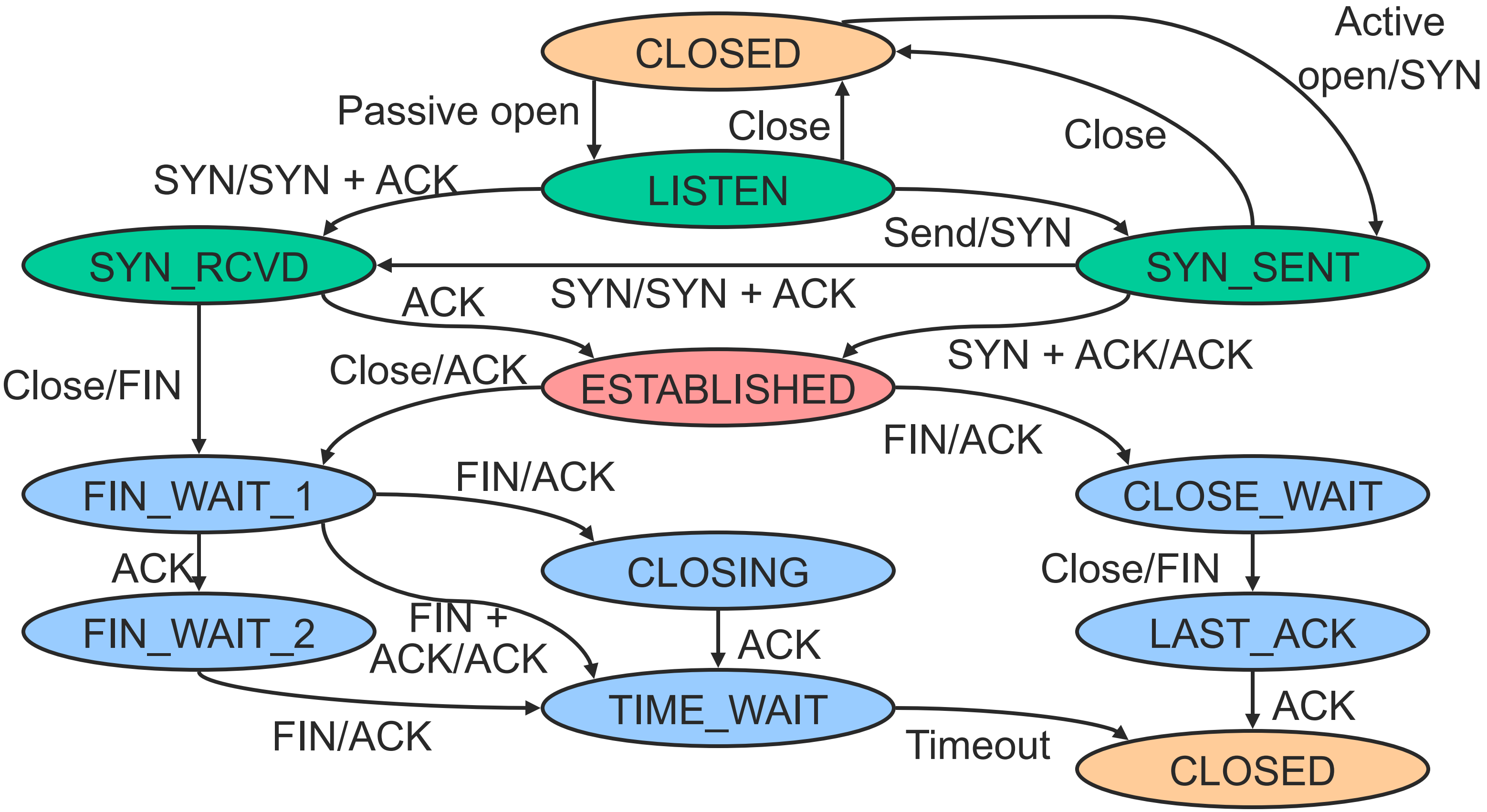
TCP Options

- Negotiate maximum segment size (MSS)
  - Each host suggests a value
  - Minimum of two values is chosen
  - Prevents IP fragmentation over first and last hops
- Packet timestamp
  - Allows RTT calculation for retransmitted packets
  - Extends sequence number space for identification of stray packets
- Negotiate advertised window granularity
  - Allows larger windows
  - Good for routes with large bandwidth-delay products

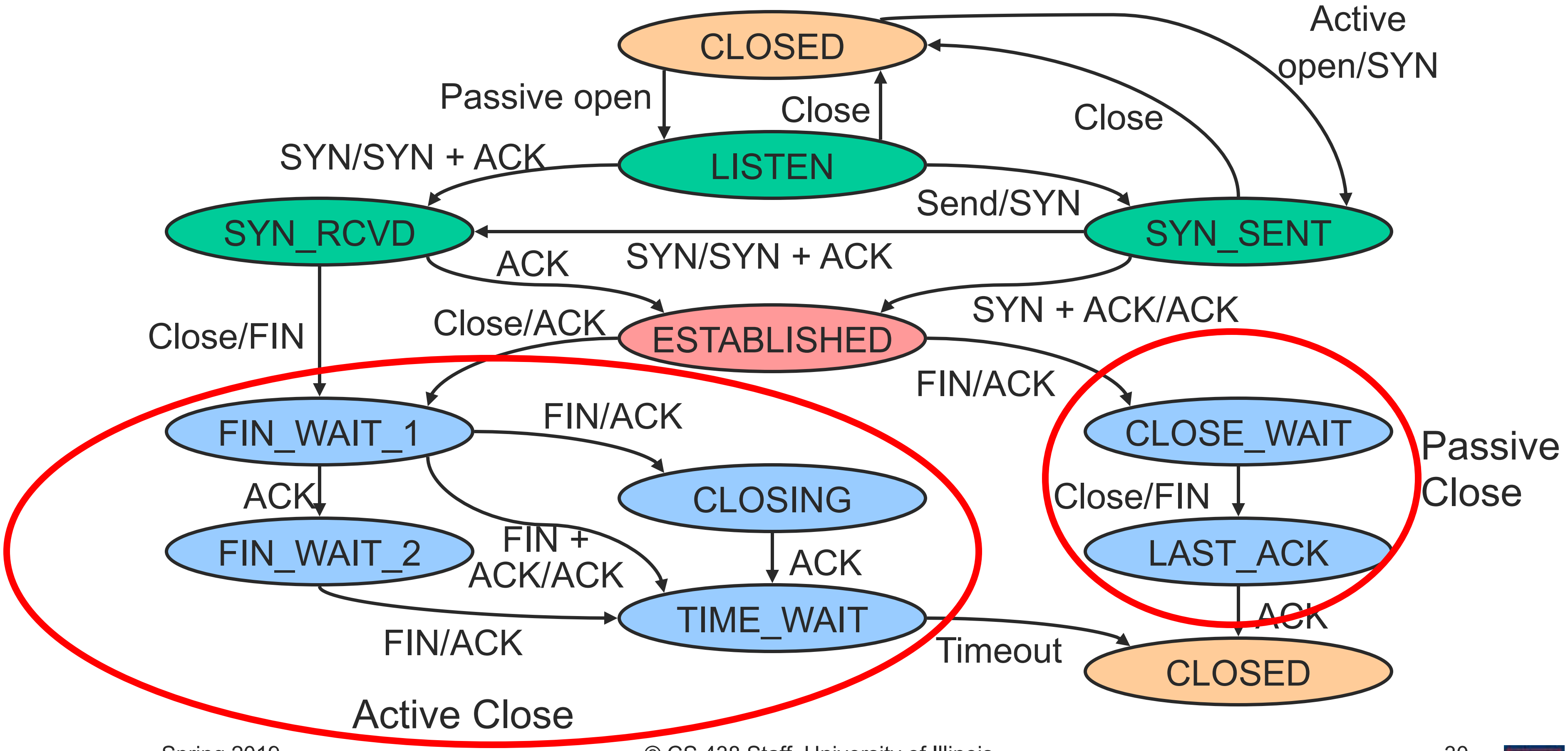
TCP State Descriptions

CLOSED	Disconnected
LISTEN	Waiting for incoming connection
SYN_RCVD	Connection request received
SYN_SENT	Connection request sent
ESTABLISHED	Connection ready for data transport
CLOSE_WAIT	Connection closed by peer
LAST_ACK	Connection closed by peer, closed locally, await ACK
FIN_WAIT_1	Connection closed locally
FIN_WAIT_2	Connection closed locally and ACK’d
CLOSING	Connection closed by both sides simultaneously
TIME_WAIT	Wait for network to discard related packets

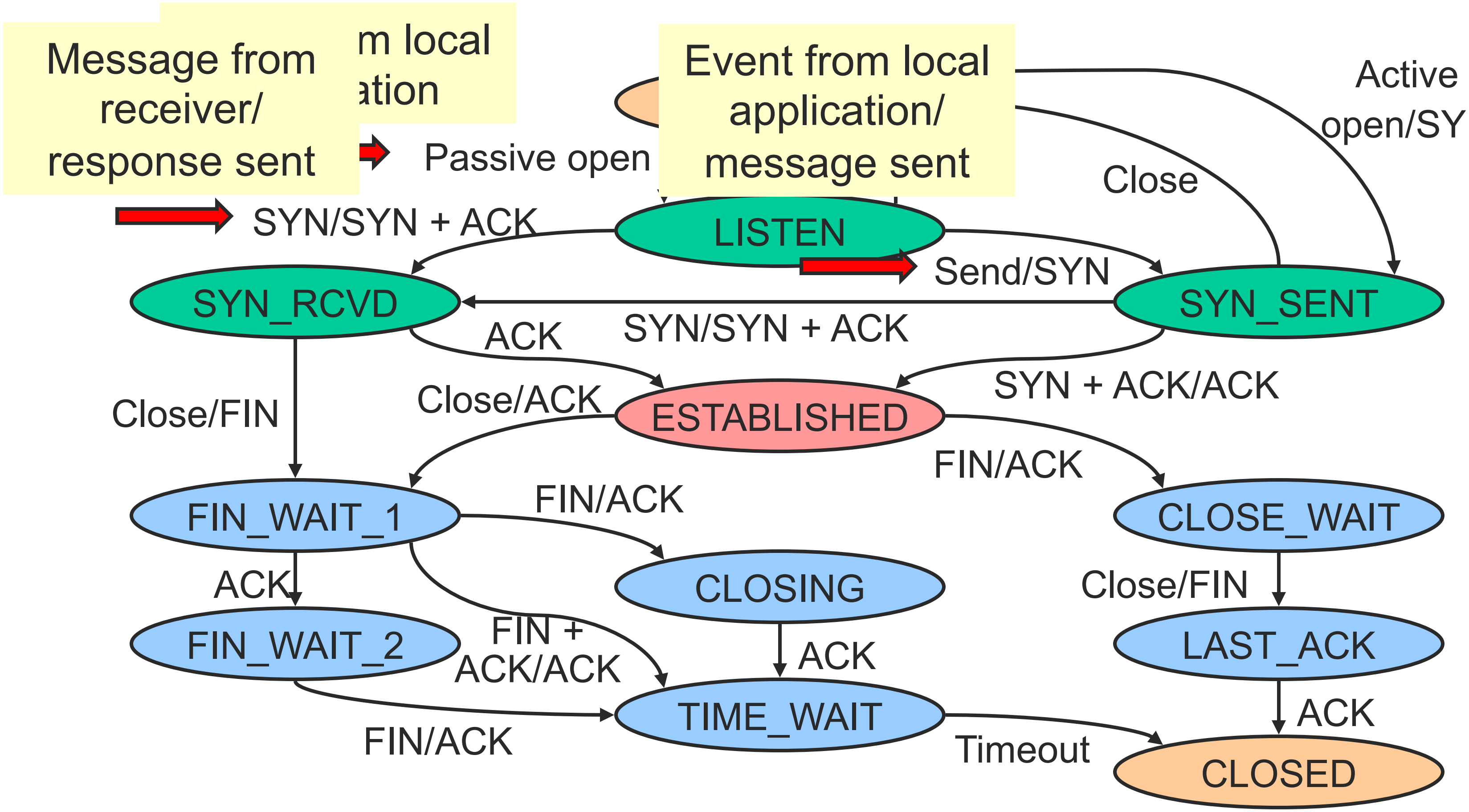
TCP State Transition Diagram



TCP State Transition Diagram



TCP State Transition Diagram



Message from receiver/ response sent

m local ation

Event from local application/ message sent

Passive open

SYN/SYN + ACK

Active open/SYN

Close

Send/SYN

LISTEN

SYN_RCVD

SYN_SENT

ACK

SYN/SYN + ACK

SYN + ACK/ACK

Close/FIN

Close/ACK

ESTABLISHED

FIN/ACK

FIN_WAIT_1

FIN/ACK

CLOSE_WAIT

ACK

CLOSING

Close/FIN

FIN_WAIT_2

FIN + ACK/ACK

ACK

LAST_ACK

TIME_WAIT

ACK

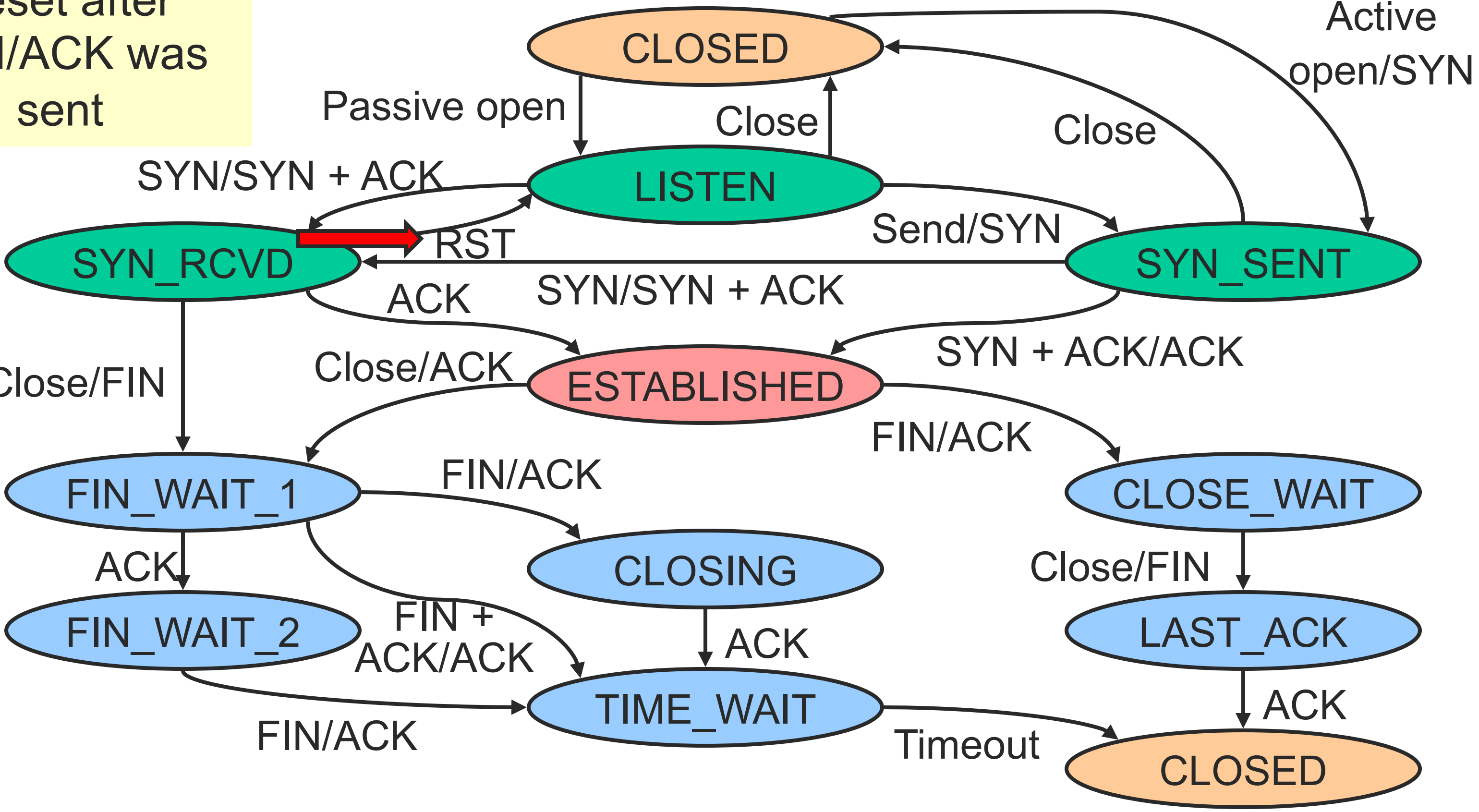
FIN/ACK

Timeout

CLOSED

TCP State Transition Diagram

Reset after SYN/ACK was sent

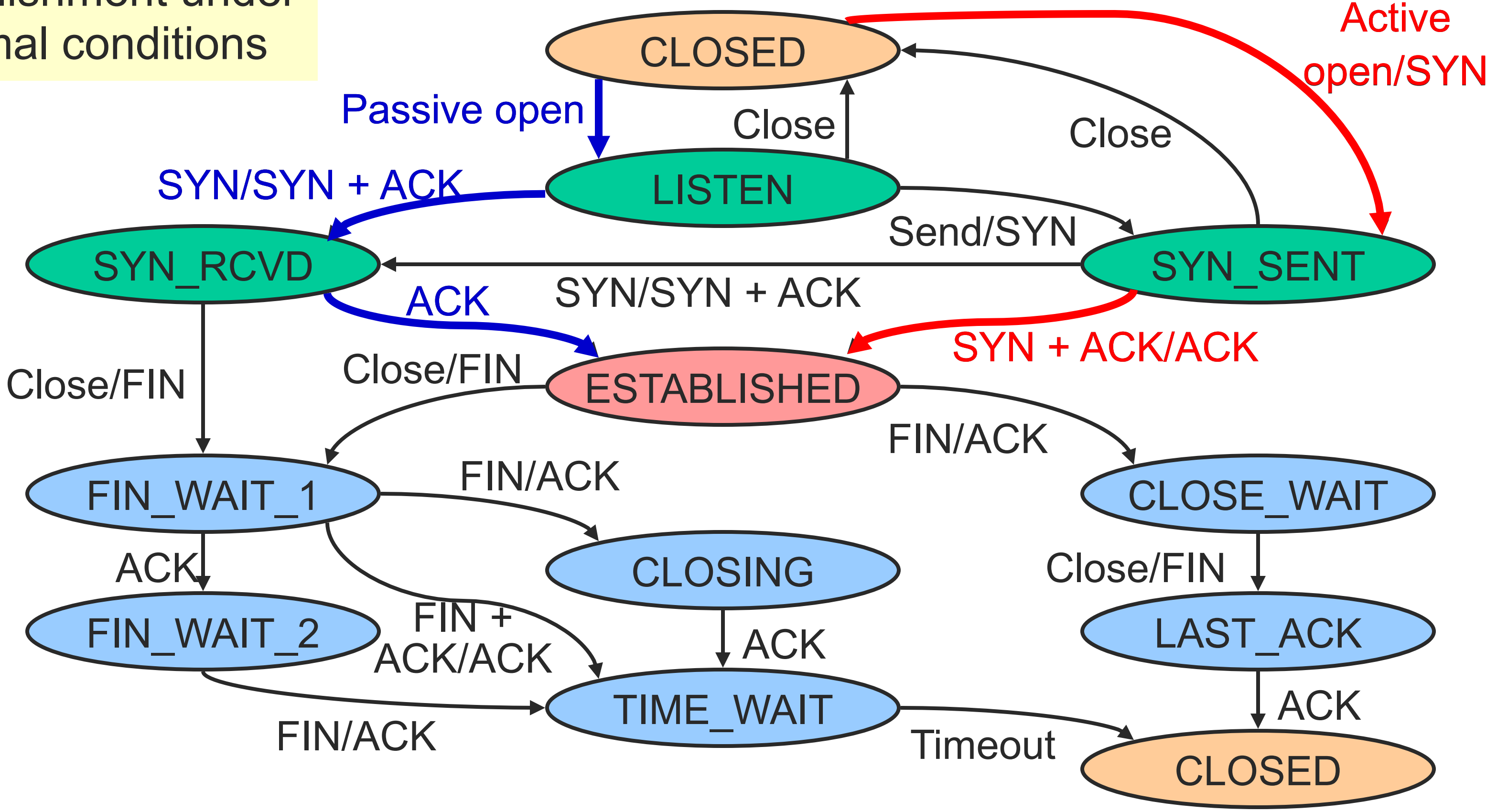


TCP State Transition Diagram

- Questions
  - State transitions
    - Describe the path taken by a server under normal conditions
    - Describe the path taken by a client under normal conditions
    - Describe the path taken assuming the client closes the connection first

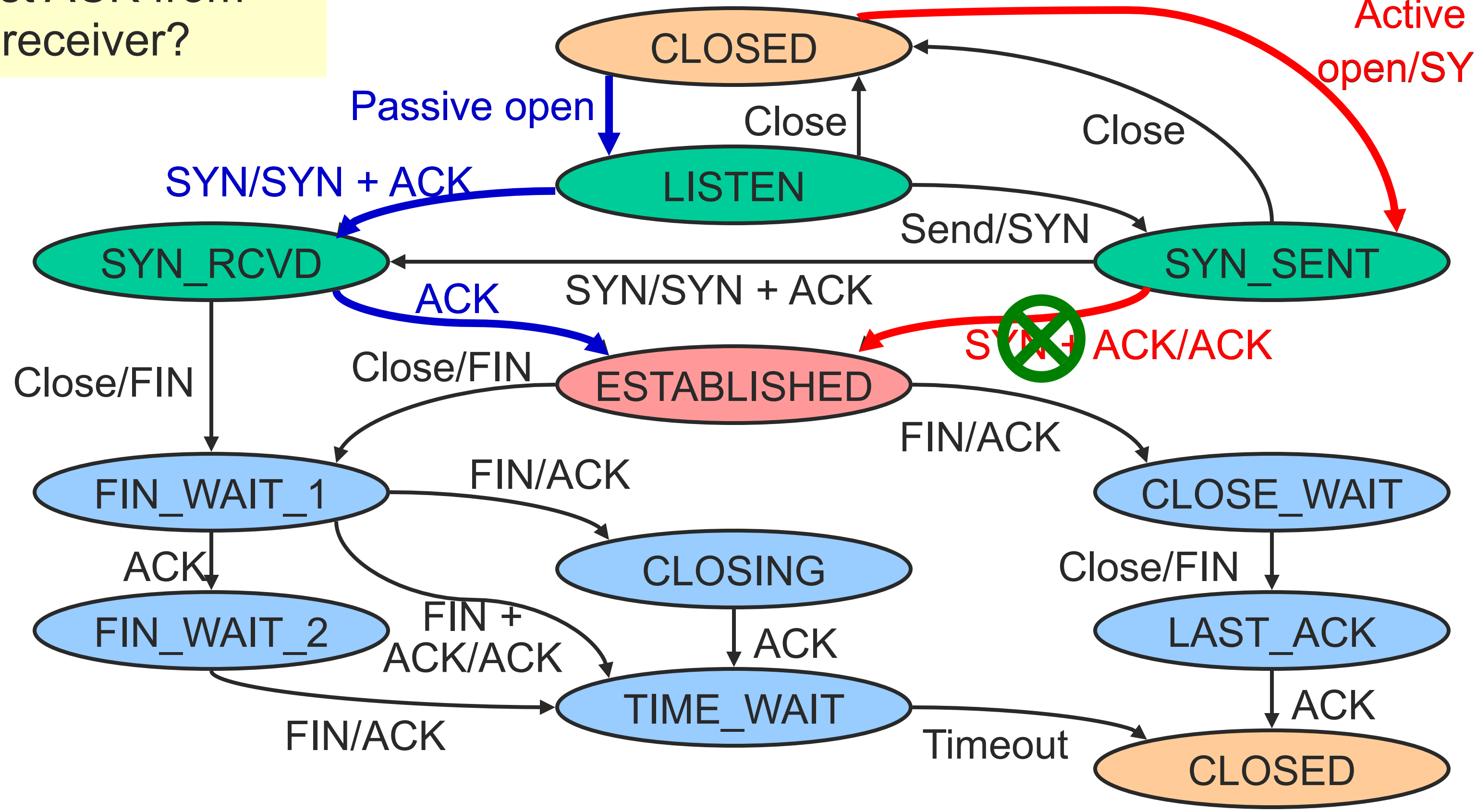
TCP State Transition Diagram

Establishment under normal conditions



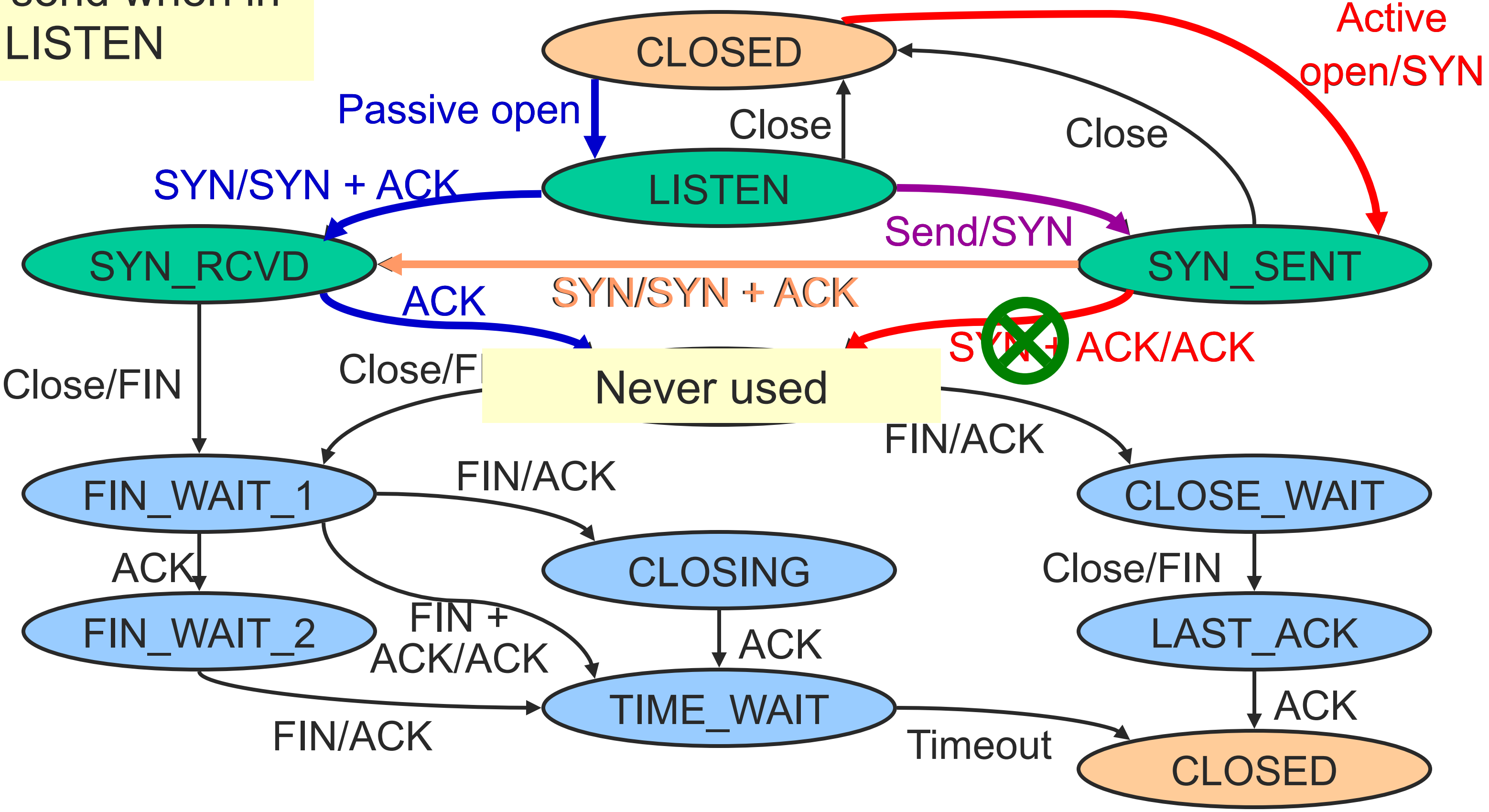
TCP State Transition Diagram

Lost ACK from receiver?



TCP State Transition Diagram

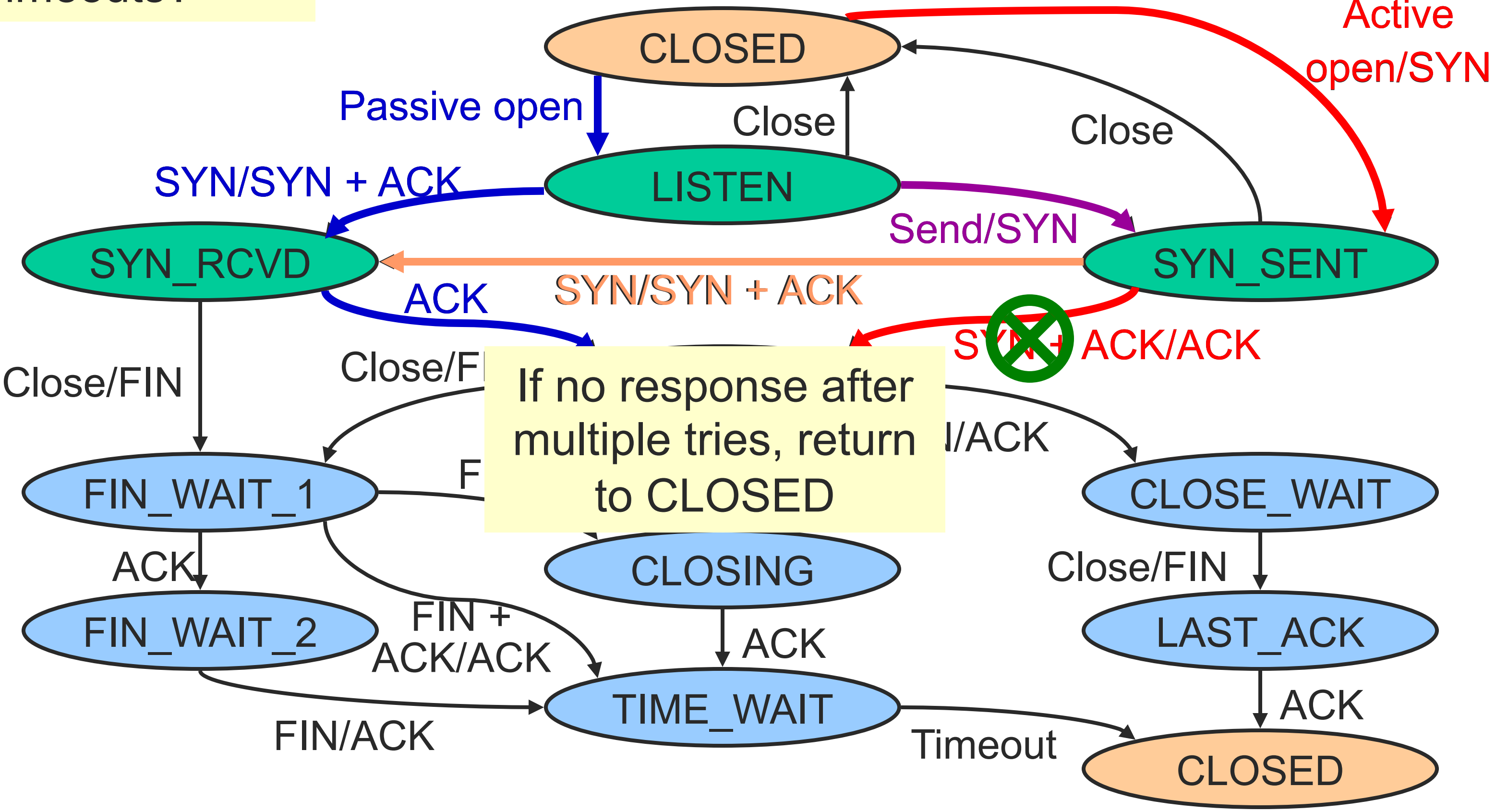
Local send when in LISTEN



Never used

TCP State Transition Diagram

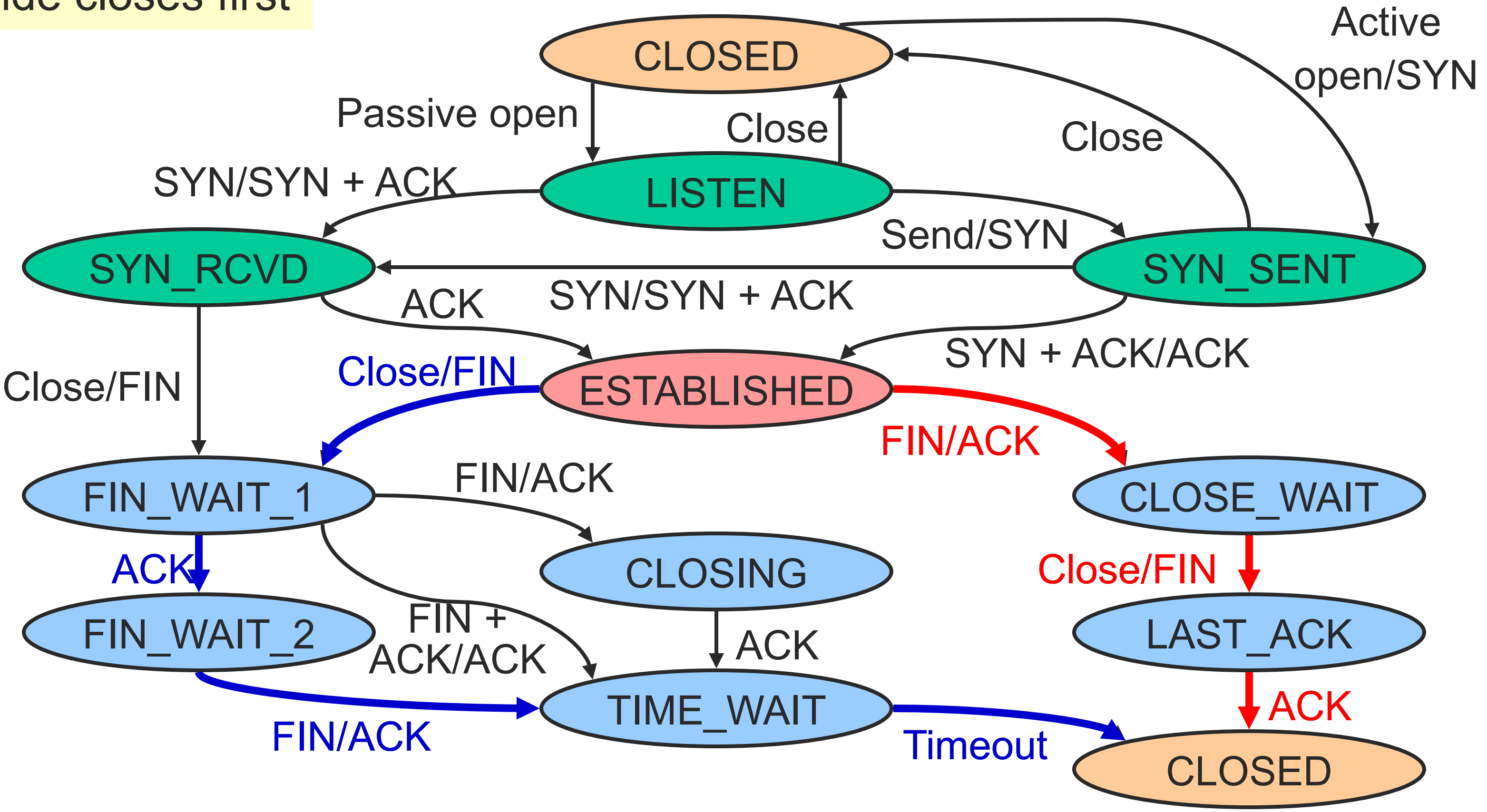
Timeouts?



If no response after multiple tries, return to CLOSED

TCP State Transition Diagram

One side closes first

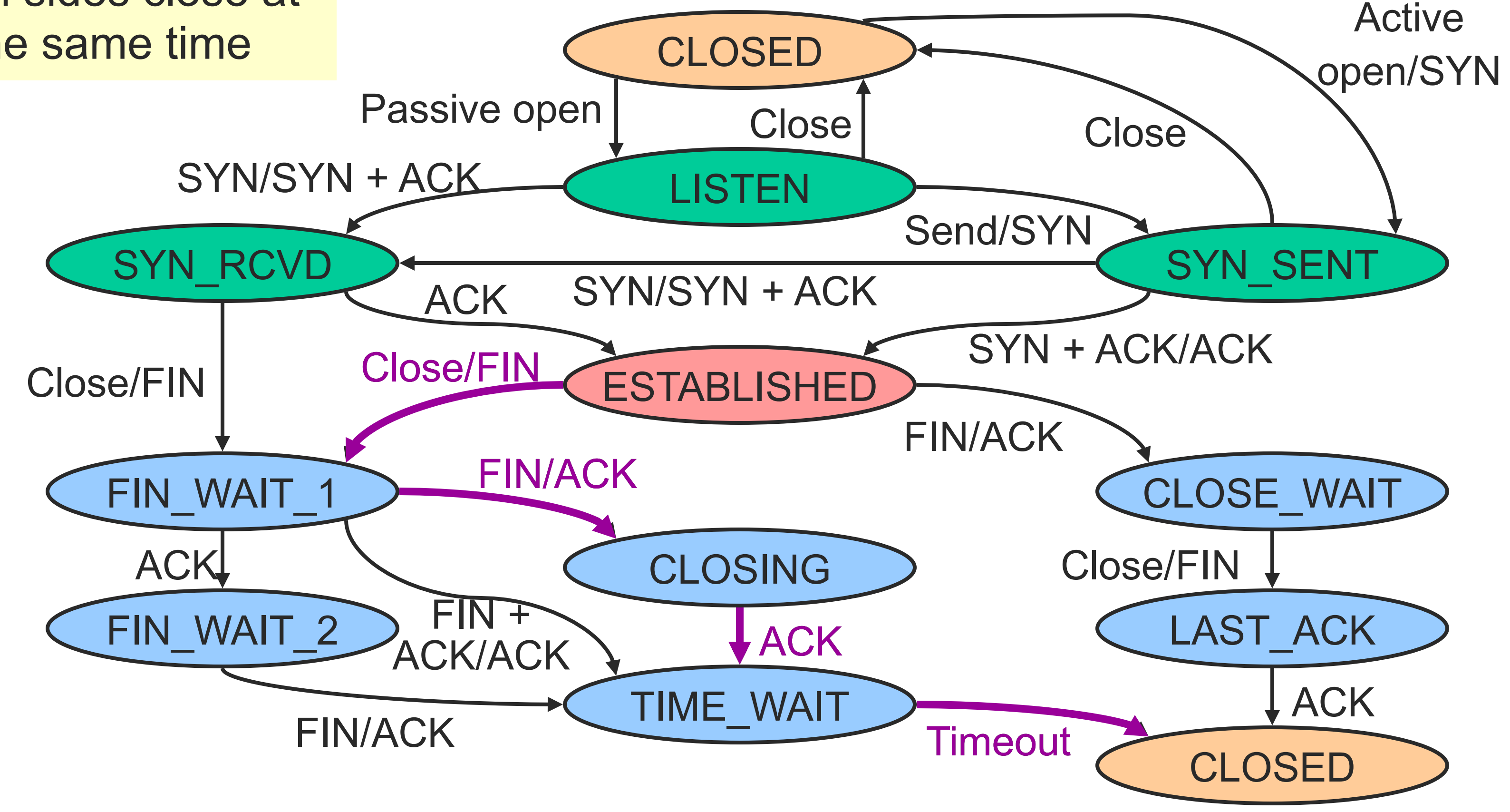


TCP TIME_WAIT State

- What purpose does the TIME_WAIT stae serve?
- Problem
  - What happens if a segment from an old connection arrives at a new connection?
- Maximum Segment Lifetime
  - Max time an old segment can live in the Internet
- TIME_WAIT State
  - Connection remains in this state from two times the maximum segment lifetime

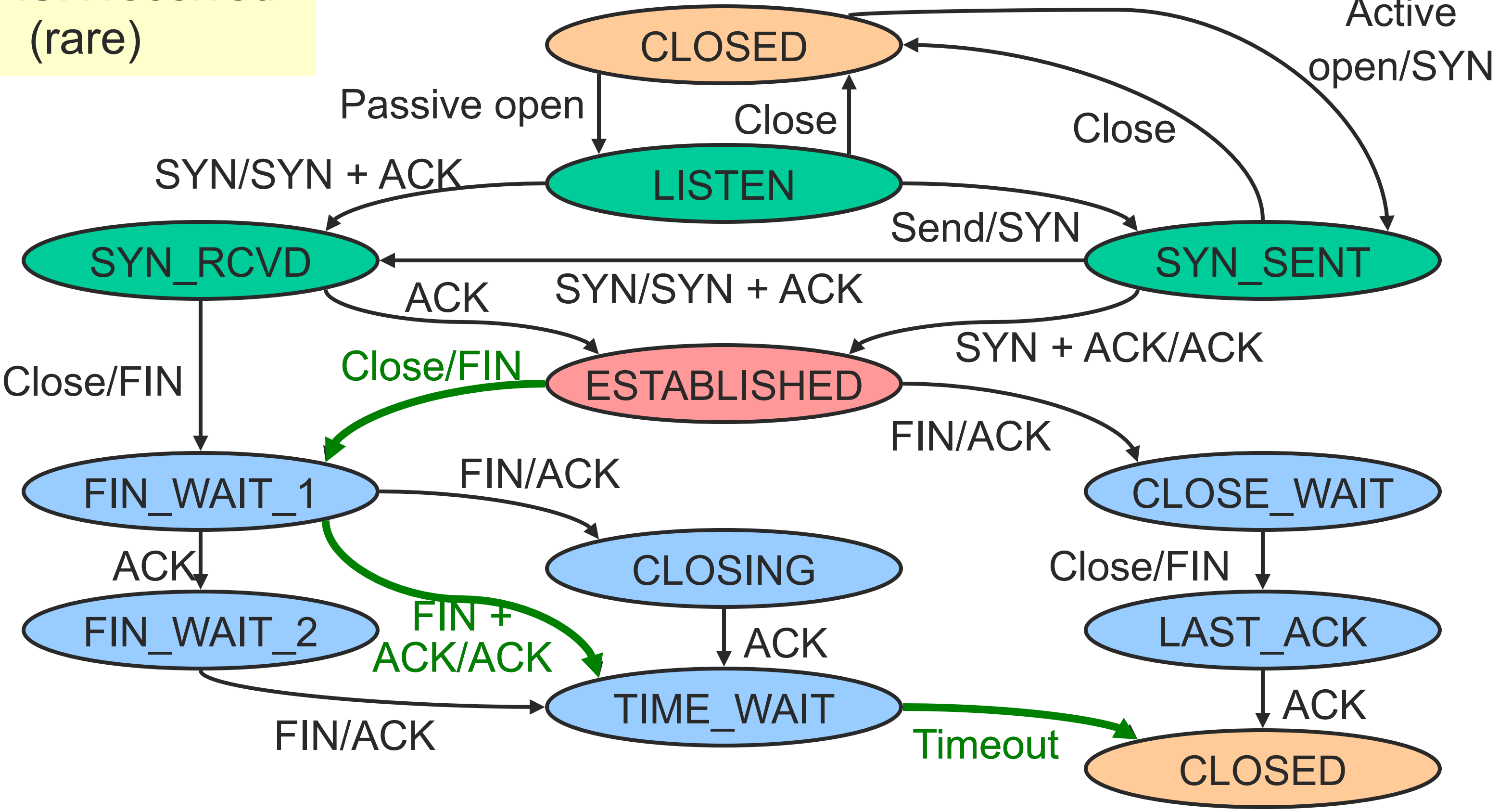
TCP State Transition Diagram

Both sides close at the same time



TCP State Transition Diagram

FIN_ACK received
(rare)



TCP Sliding Window Protocol

- Sequence numbers
  - Indices into byte stream
- ACK sequence number
  - Actually next byte expected as opposed to last byte received

TCP Sliding Window Protocol

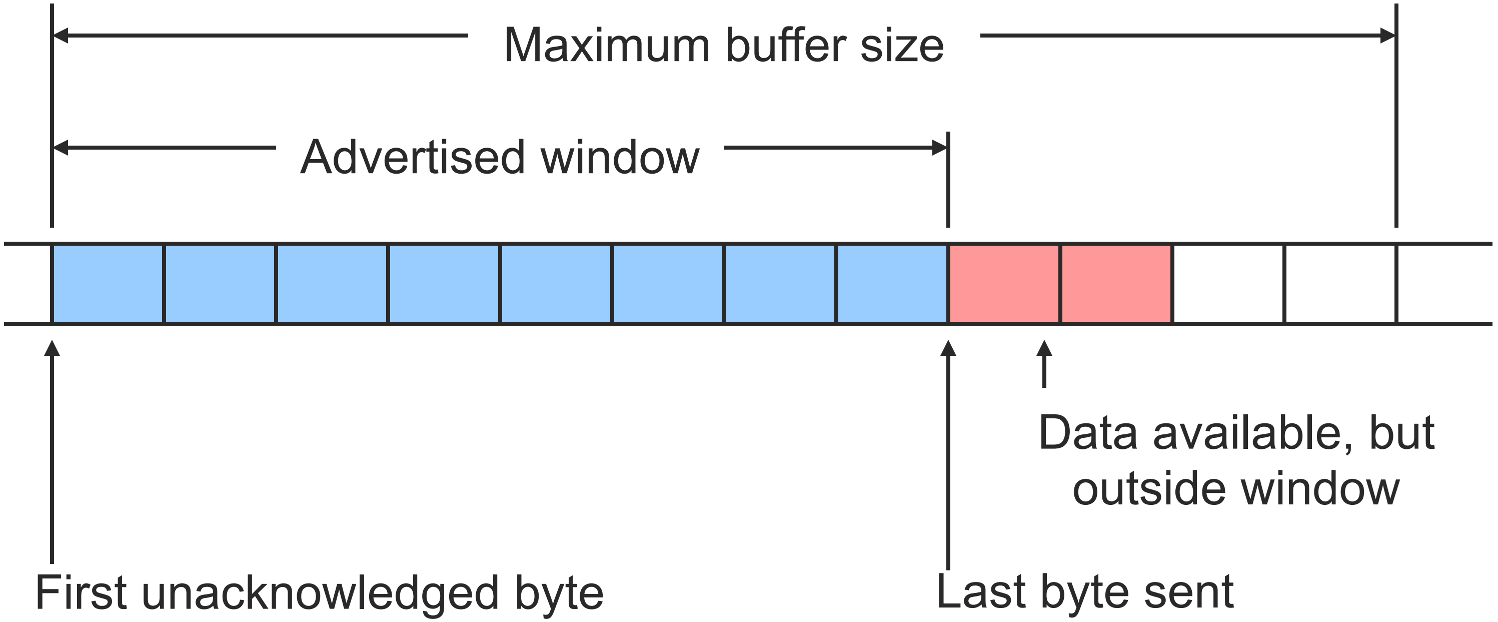
- Initial Sequence Number
  - Why not just use 0?
- Practical issue
  - IP addresses and port #s uniquely identify a connection
  - Eventually, though, these port #s do get used again
  - … small chance an old packet is still in flight
  - … and might be associated with new connection
- TCP requires (RFC793) 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

TCP Sliding Window Protocol

- Advertised window
  - Enables dynamic receive window size
- Receive buffers
  - Data ready for delivery to application until requested
  - Out-of-order data to maximum buffer capacity
- Sender buffers
  - Unacknowledged data
  - Unsent data out to maximum buffer capacity

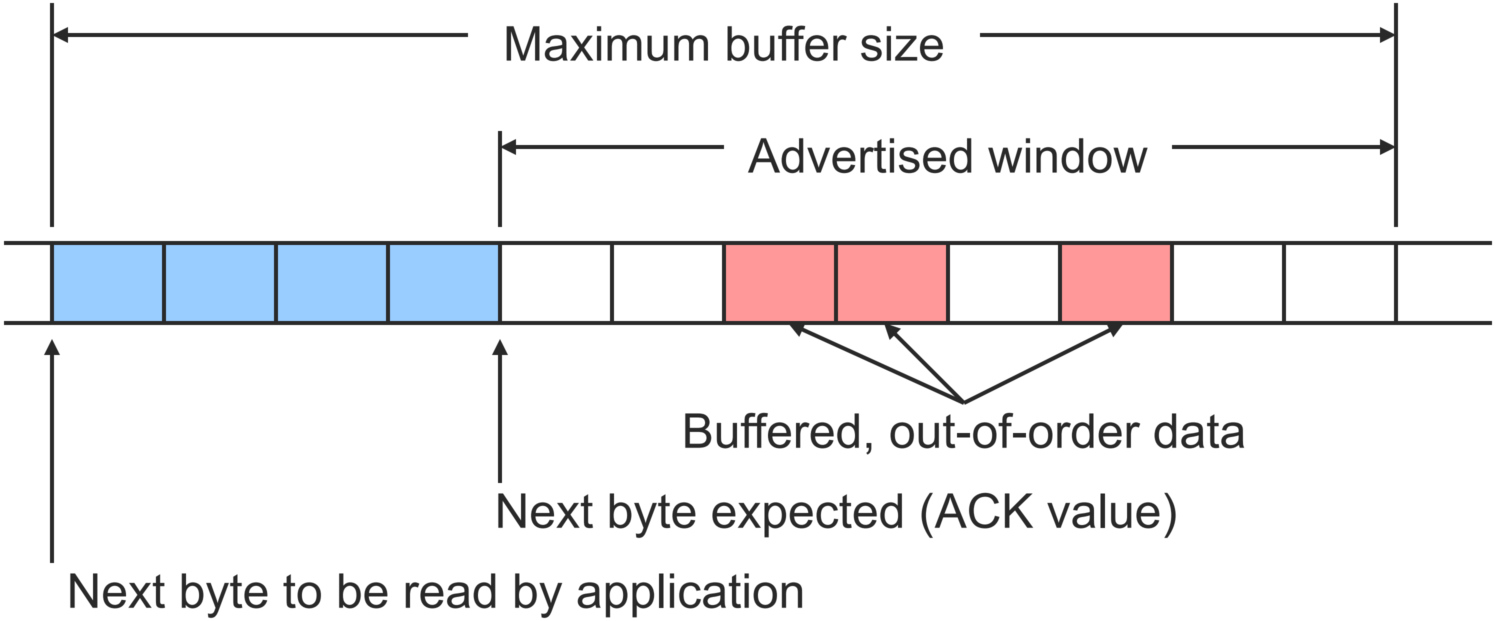
TCP Sliding Window Protocol – Sender Side

- `LastByteAcked <= LastByteSent`
- `LastByteSent <= LastByteWritten`
- Buffer bytes between `LastByteAcked` and `LastByteWritten`



TCP Sliding Window Protocol – Receiver Side

- `LastByteRead <  NextByteExpected`
- `NextByteExpected <= LastByteRcvd + 1`
- Buffer bytes between **`NextByteRead`** and **`LastByteRcvd`**


Maximum buffer size
Advertised window
Buffered, out-of-order data
Next byte expected (ACK value)
Next byte to be read by application

Flow Control vs. Congestion Control

- Flow control
  - Preventing senders from overrunning the capacity of the receivers
- Congestion control
  - Preventing too much data from being injected into the network, causing switches or links to become overloaded
- Which one does TCP provide?
- TCP provides both
  - Flow control based on advertised window
  - Congestion control discussed later in class

Advertised Window Limits Rate

- W = window size
  - Sender can send no faster than W/RTT bytes/sec
  - Receiver implicitly limits sender to rate that receiver can sustain
  - If sender is going too fast, window advertisements get smaller & smaller

TCP Flow Control: Receiver

- Receive buffer size
  - `= MaxRcvBuffer`
  - `LastByteRcvd - LastByteRead < = MaxRcvBuf`
- Advertised window
  - `= MaxRcvBuf - (NextByteExp - NextByteRead)`
  - Shrinks as data arrives and
  - Grows as the application consumes data

TCP Flow Control: Sender

- Send buffer size
  - `= MaxSendBuffer`
  - `LastByteSent - LastByteAcked < = AdvertWindow`
- Effective buffer
  - `= AdvertWindow - (LastByteSent - LastByteAck)`
  - `EffectiveWindow > 0 to send data`

- Relationship between sender and receiver
  - `LastByteWritten - LastByteAcked < = MaxSendBuffer`
  - `block sender if (LastByteWritten - LastByteAcked) + y > MaxSenderBuffer`

TCP Flow Control

- Problem: Slow receiver application
  - Advertised window goes to 0
  - Sender cannot send more data
  - Non-data packets used to update window
  - Receiver may not spontaneously generate update or update may be lost
- Solution
  - Sender periodically sends 1-byte segment, ignoring advertised window of 0
  - Eventually window opens
  - Sender learns of opening from next ACK of 1-byte segment

TCP Flow Control

- Problem: Application delivers tiny pieces of data to TCP
  - Example: telnet in character mode
  - Each piece sent as a segment, returned as ACK
  - Very inefficient
- Solution
  - Delay transmission to accumulate more data
  - Nagle’s algorithm
    - Send first piece of data
    - Accumulate data until first piece ACK’d
    - Send accumulated data and restart accumulation
    - Not ideal for some traffic (e.g., mouse motion)

TCP Flow Control

- Problem: Slow application reads data in tiny pieces
  - Receiver advertises tiny window
  - Sender fills tiny window
  - Known as silly window syndrome
- Solution
  - Advertise window opening only when MSS or ½ of buffer is available
  - Sender delays sending until window is MSS or ½ of receiver’s buffer (estimated)

TCP Bit Allocation Limitations

- Sequence numbers vs. packet lifetime
  - Assumed that IP packets live less than 60 seconds
  - Can we send 2^{32} bytes in 60 seconds?
  - Less than an STS-12 line
- Advertised window vs. delay-bandwidth
  - Only 16 bits for advertised window
  - Cross-country RTT = 100 ms
  - Adequate for only 5.24 Mbps!

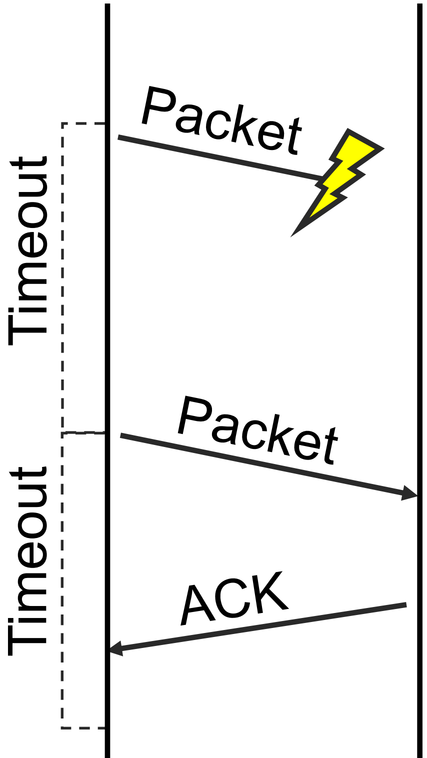
TCP Sequence Numbers – 32-bit

Bandwidth	Speed	Time until wrap around
T1	1.5 Mbps	6.4 hours
Ethernet	10 Mbps	57 minutes
T3	45 Mbps	13 minutes
FDDI	100 Mbps	6 minutes
STS-3	155 Mbps	4 minutes
STS-12	622 Mbps	55 seconds
STS-24	1.2 Gbps	28 seconds

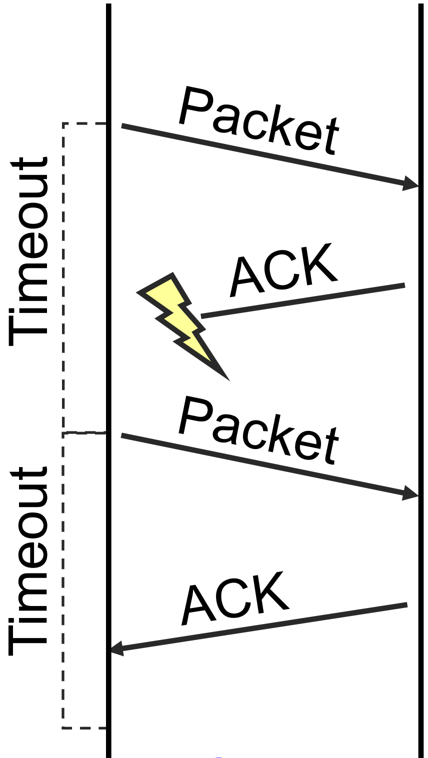
TCP Advertised Window – 16-bit

Bandwidth	Speed	Delay x Bandwidth Product
T1	1.5 Mbps	18 KB
Ethernet	10 Mbps	122 KB
T3	45 Mbps	549 KB
FDDI	100 Mbps	1.2 MB
STS-3	155 Mbps	1.8 MB
STS-12	622 Mbps	7.4 MB
STS-24	1.2 Gbps	14.8 MB

Reasons for Retransmission

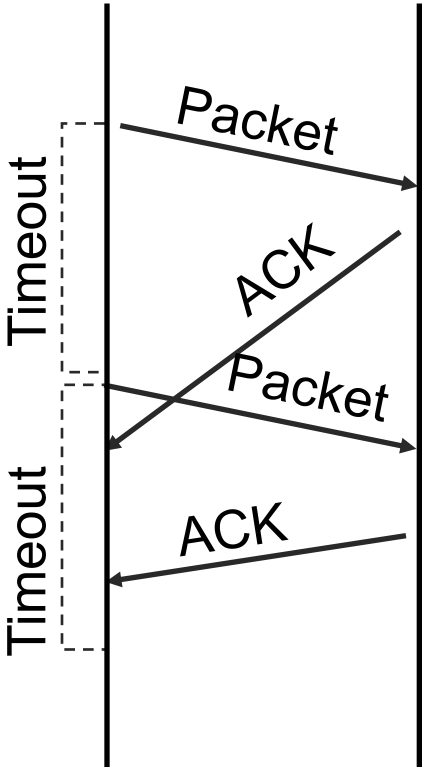


Packet lost



ACK lost

DUPLICATE PACKET



Early timeout

DUPLICATE PACKETS

How Long Should Sender Wait?

- Sender sets a timeout to wait for an ACK
    - Too short
        - wasted retransmissions
    - Too long
        - excessive delays when packet lost

TCP Round Trip Time and Timeout

- How should TCP set its timeout value?
  - Longer than RTT
    - But RTT varies
  - Too short
    - Premature timeout
    - Unnecessary retransmissions
  - Too long
    - Slow reaction to segment loss

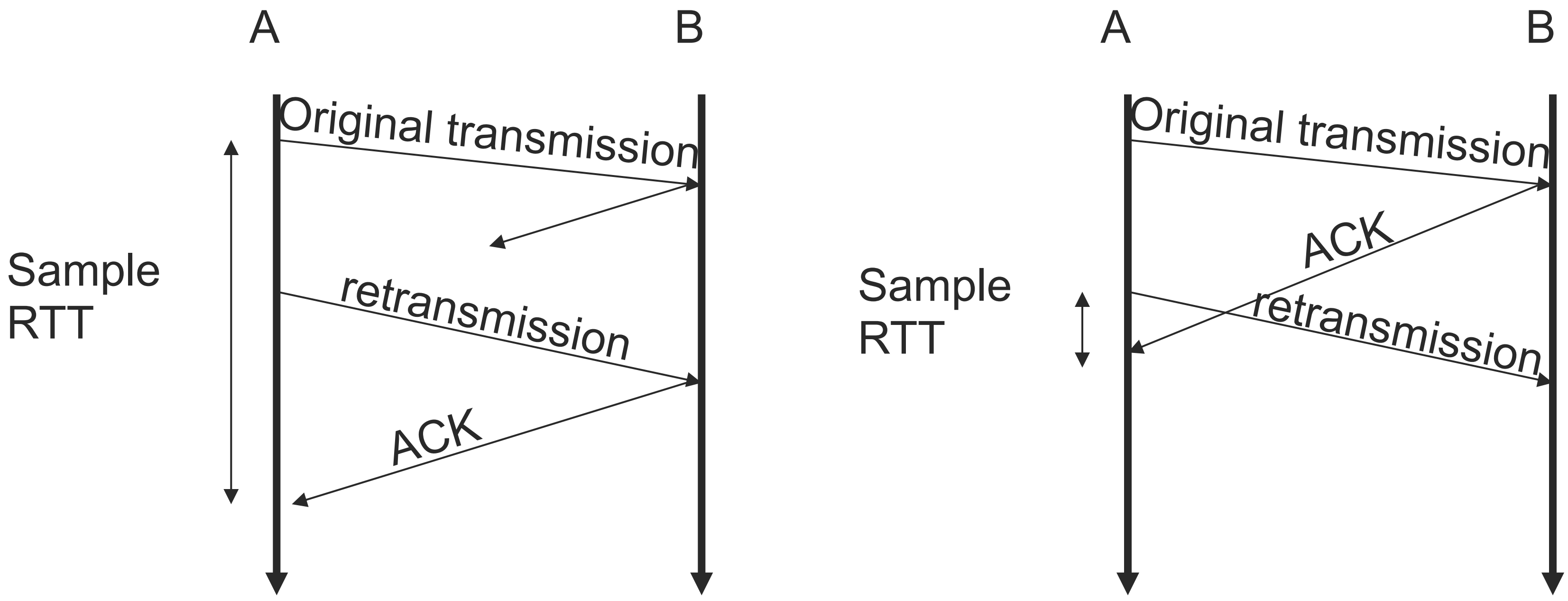
- Estimating RTT
  - SampleRTT
    - Measured time from segment transmission until ACK receipt
    - Will vary
    - Want smoother estimated RTT
  - Average several recent measurements
    - Not just current SampleRTT

TCP Adaptive Retransmission Algorithm - Original

- Theory
  - Estimate RTT
  - Multiply by 2 to allow for variations
- Practice
  - Use exponential moving average (α = 0.1 to 0.2)
  - Estimate = (α) * measurement + (1- α) * estimate
  - Influence of past sample decreases exponentially fast

TCP Adaptive Retransmission Algorithm - Original

- Problem: What does an ACK really ACK?
  - Was ACK in response to first, second, etc transmission?



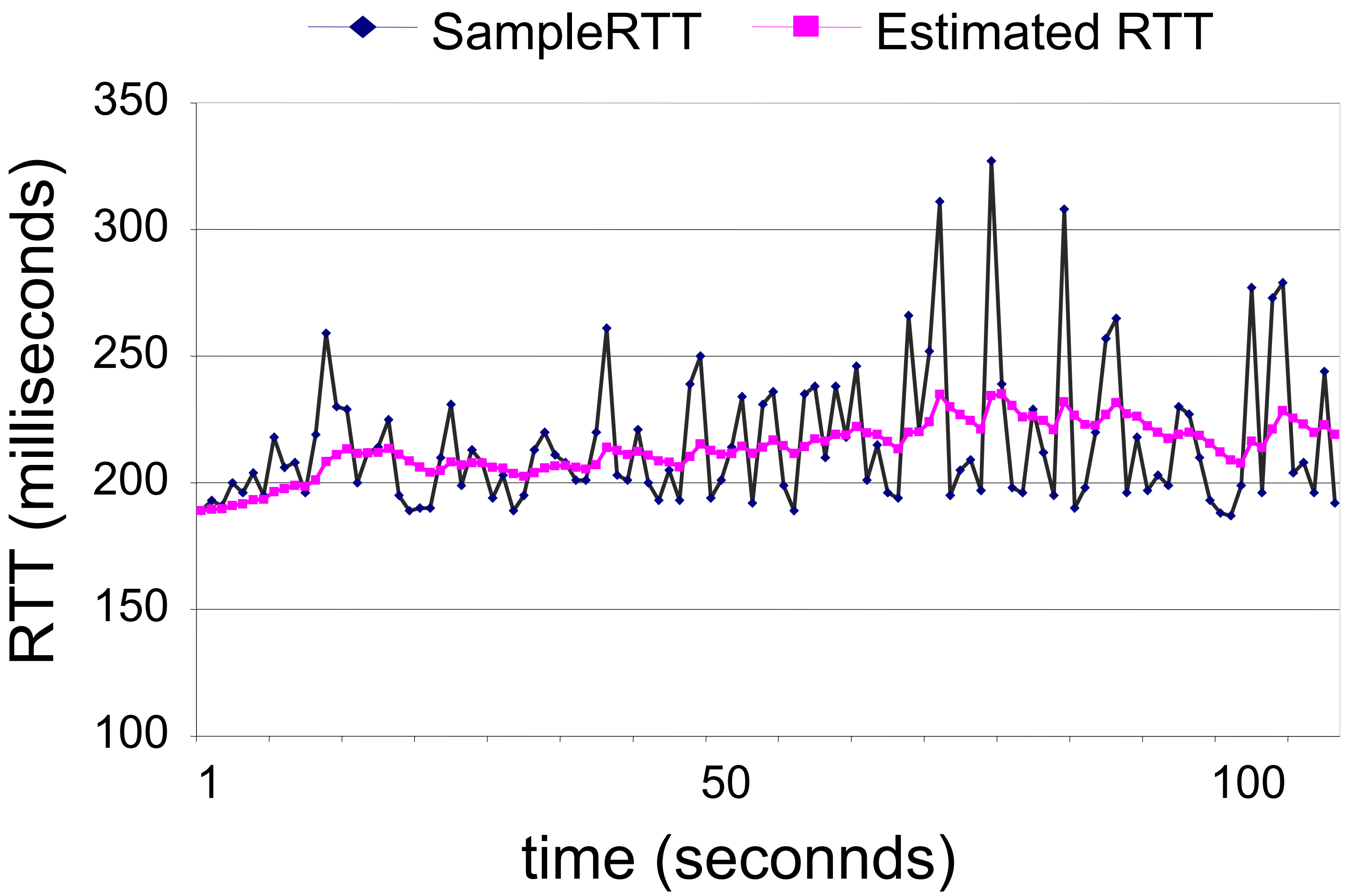
TCP Adaptive Retransmission Algorithm – Karn-Partridge

- Algorithm
  - Exclude retransmitted packets from RTT estimate
  - For each retransmission
    - Double RTT estimate
    - Exponential backoff from congestion

TCP Adaptive Retransmission Algorithm – Karn-Partridge

- Problem
  - Still did not handle variations well
  - Did not solve network congestion problems as well as desired
    - At high loads round trip variance is high

Example RTT Estimation



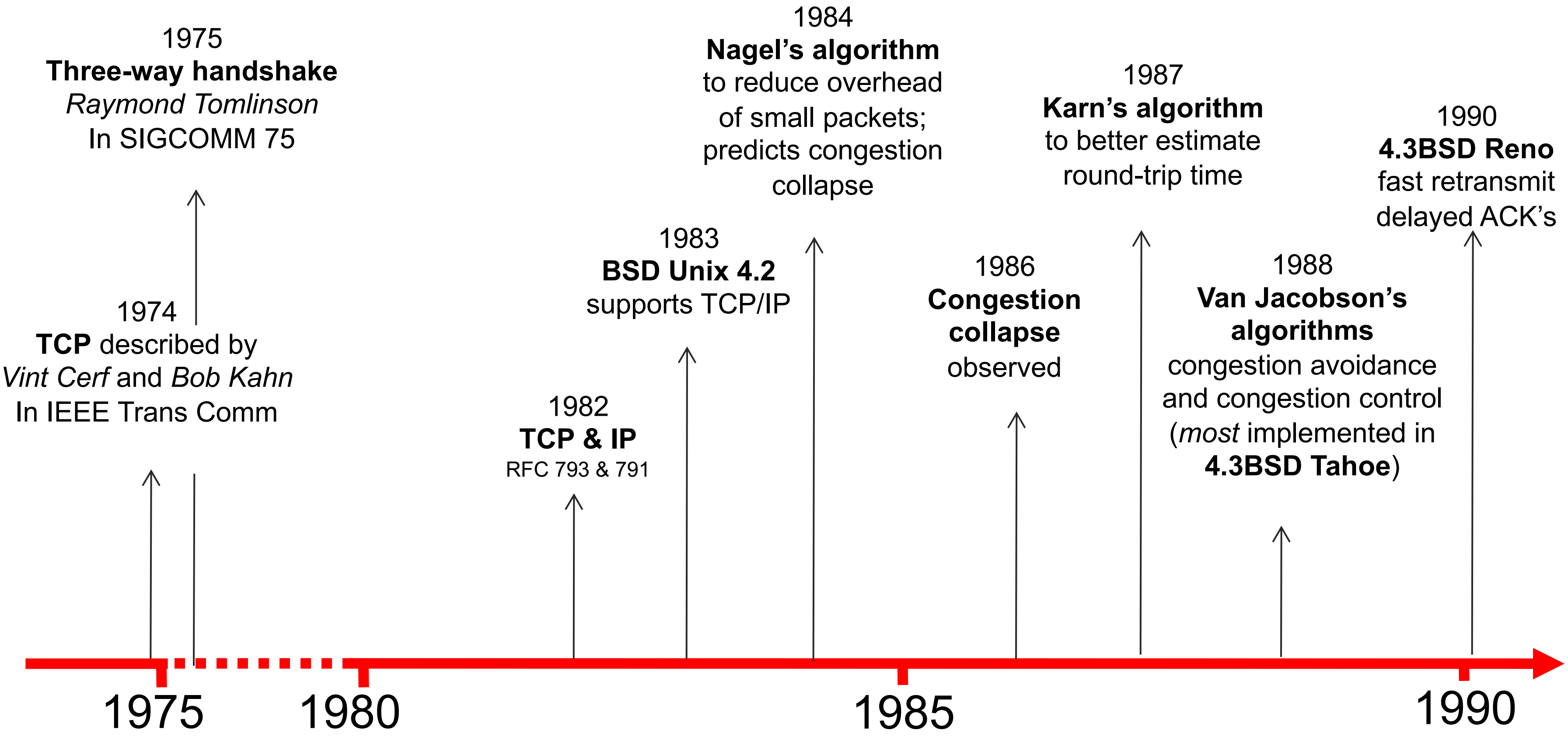
TCP Adaptive Retransmission Algorithm – Jacobson

- Algorithm
    - Estimate variance of RTT
        - Calculate mean interpacket RTT deviation to approximate variance
        - Use second exponential moving average
        - Dev = (β) * |RTT_Est – Sample| + (1–β) * Dev
        - β = 0.25, A = 0.125 for RTT_est
    - Use variance estimate as component of RTT estimate
        - Next_RTT = RTT_Est + 4 * Dev
    - Protects against high jitter

TCP Adaptive Retransmission Algorithm – Jacobson

- Notes
  - Algorithm is only as good as the granularity of the clock
  - Accurate timeout mechanism is important for congestion control

Evolution of TCP



1974
TCP described by
Vint Cerf and *Bob Kahn*
In IEEE Trans Comm

1975
Three-way handshake
Raymond Tomlinson
In SIGCOMM 75

1982
TCP & IP
RFC 793 & 791

1983
BSD Unix 4.2
supports TCP/IP

1984
Nagel's algorithm
to reduce overhead
of small packets;
predicts congestion
collapse

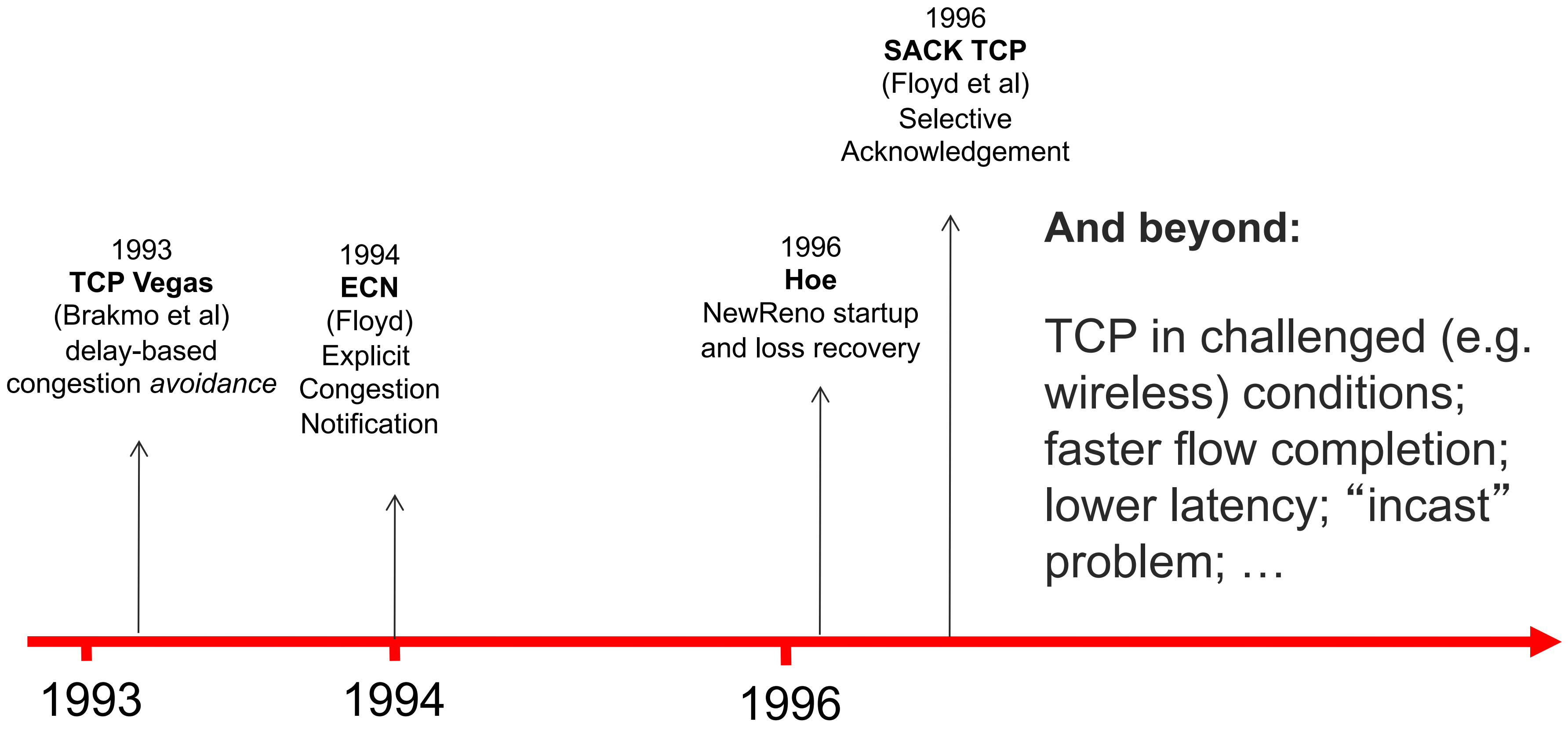
1986
Congestion
collapse
observed

1987
Karn's algorithm
to better estimate
round-trip time

1988
Van Jacobson's
algorithms
congestion avoidance
and congestion control
(*most* implemented in
4.3BSD Tahoe)

1990
4.3BSD Reno
fast retransmit
delayed ACK's

TCP Through the 1990s



1993
TCP Vegas
(Brakmo et al)
delay-based
congestion *avoidance*

1994
ECN
(Floyd)
Explicit
Congestion
Notification

1996
Hoe
NewReno startup
and loss recovery

1996
SACK TCP
(Floyd et al)
Selective
Acknowledgement

And beyond:

TCP in challenged (e.g. wireless) conditions; faster flow completion; lower latency; “incast” problem; …