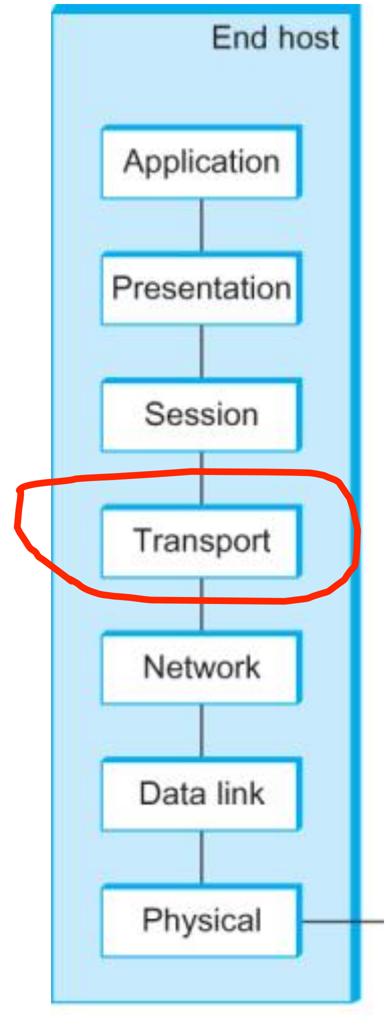
Transport layer and TCP

Overview

- Principles underlying transport layer
  - Multiplexing/demultiplexing
  - Detecting errors
  - Reliable delivery
  - Flow control
- Major transport layer protocols:
  - User Datagram Protocol (UDP)
    - Simple unreliable message delivery
  - Transmission Control Protocol (TCP)
    - Reliable bidirectional stream of bytes



Transmission Control Protocol (TCP)

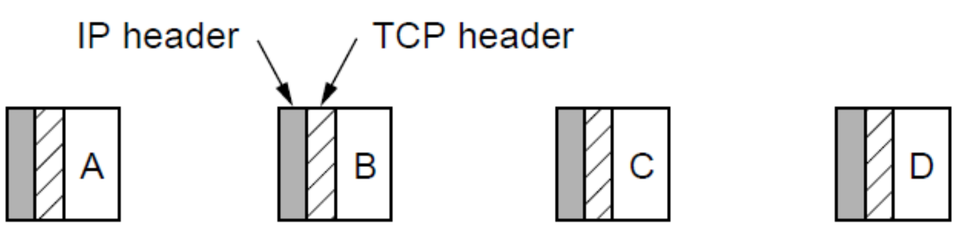
- Stream of bytes
  - Send and receive streams, not messages
- Reliable, in-order delivery
  - Checksums to detect corrupted data
  - Sequence numbers to detect losses and reorder
  - Acknowledgements and retransmission for reliability
- Connection-oriented
  - Explicit setup and teardown of connections
  - Full duplex, two streams one in each direction
- Flow control
  - Prevent overrunning receiver's buffer

Transmission Control Protocol (TCP)

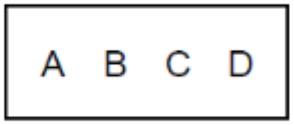
- Congestion control
  - Adapt for the greater good
- History:
  - RFC 793, TCP formally defined, September 1981
  - RFC 1122, clarification and bug fixes
  - RFC 1323, high performance extensions
  - RFC 2018, selective acknowledgements
  - RFC 2581, congestion control
  - RFC 2873, quality of service
  - RFC 2988, improved retransmission timers
  - RFC 3168, congestion notification
  - …
  - RFC 4614, guide to TCP RFCs

TCP service model

- Uses port number abstraction, same as UDP
- Demultiplexing key:
  - <source IP, source port, destination IP, destination port>
- Byte steam, no message boundaries
  - No way to know what size chunks given to SEND when other side does RECEIVE

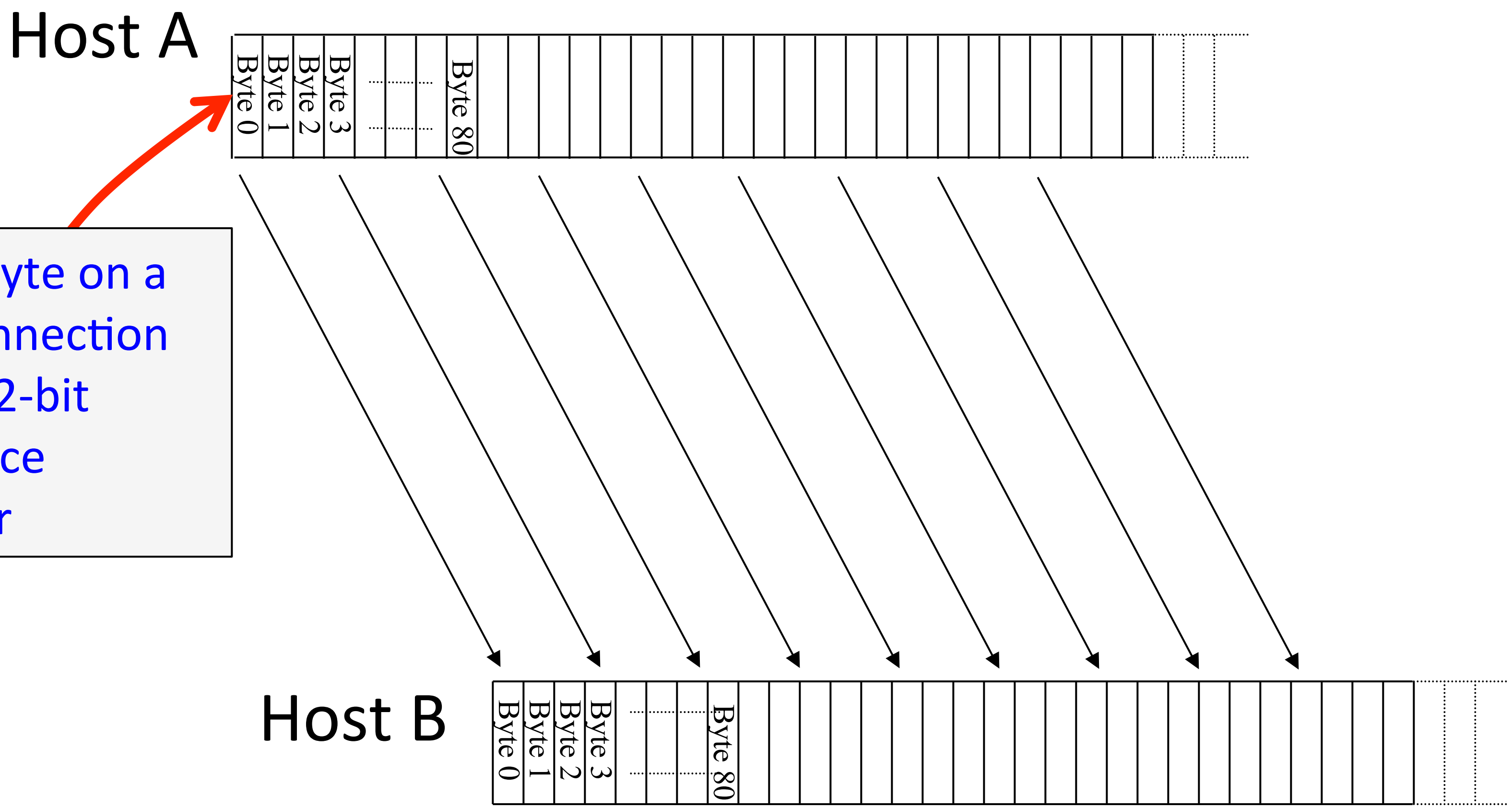


Four 512-byte segments sent as separate IP datagrams.



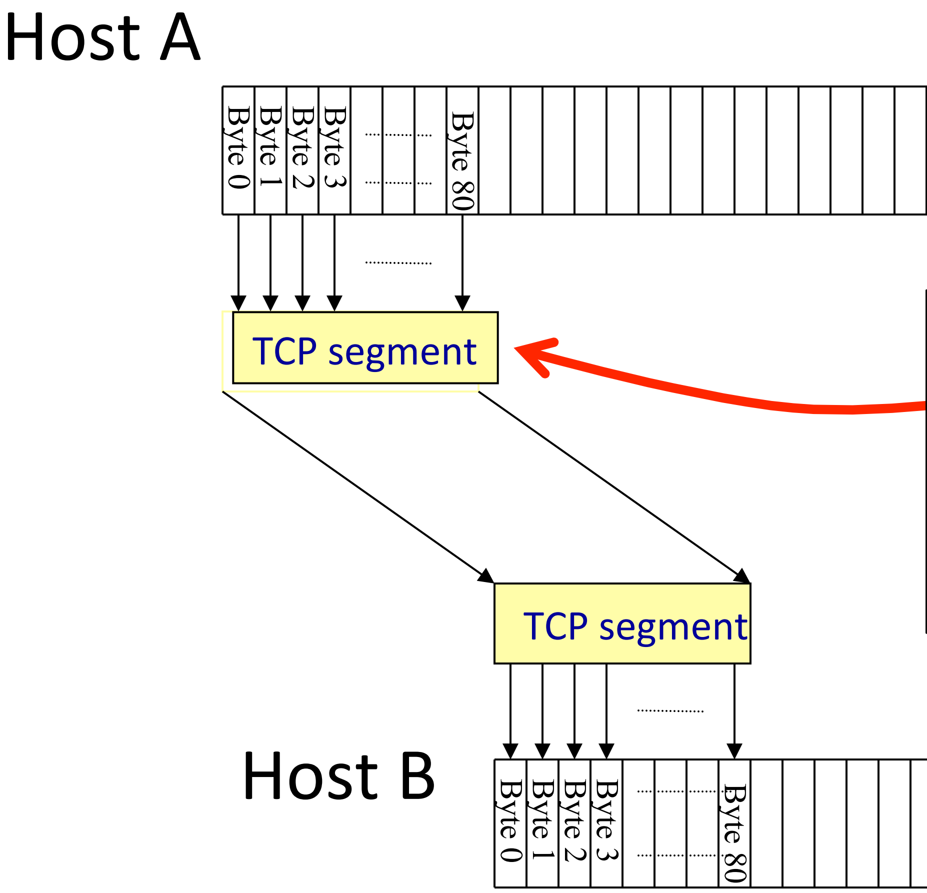
2048 bytes of data delivery to application in single READ call

TCP "stream of bytes" service



Every byte on a TCP connection has a 32-bit sequence number

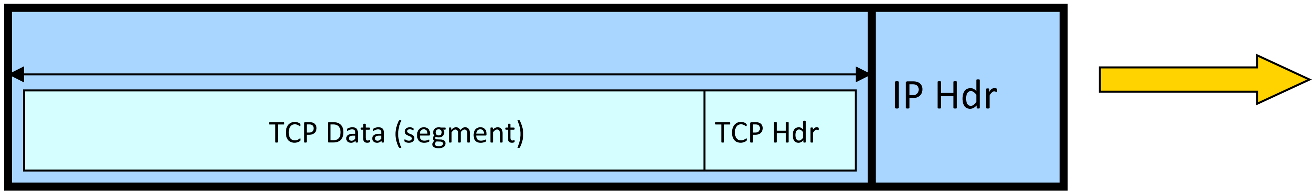
Emulating a byte stream



TCP segment sent when:

1) Segment full (hits the max segment size)
2) Hits a timeout value
3) Pushed by application

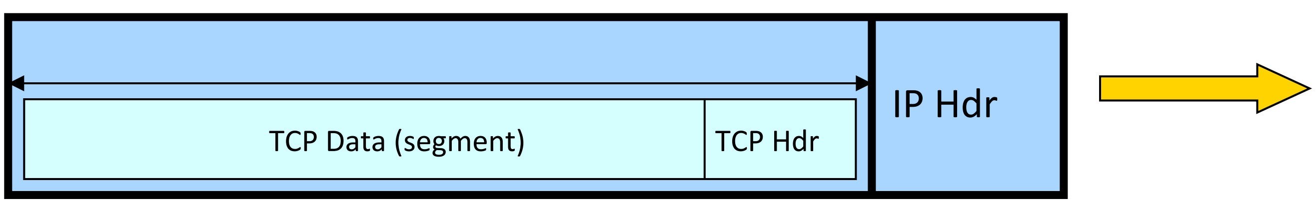
TCP Segment



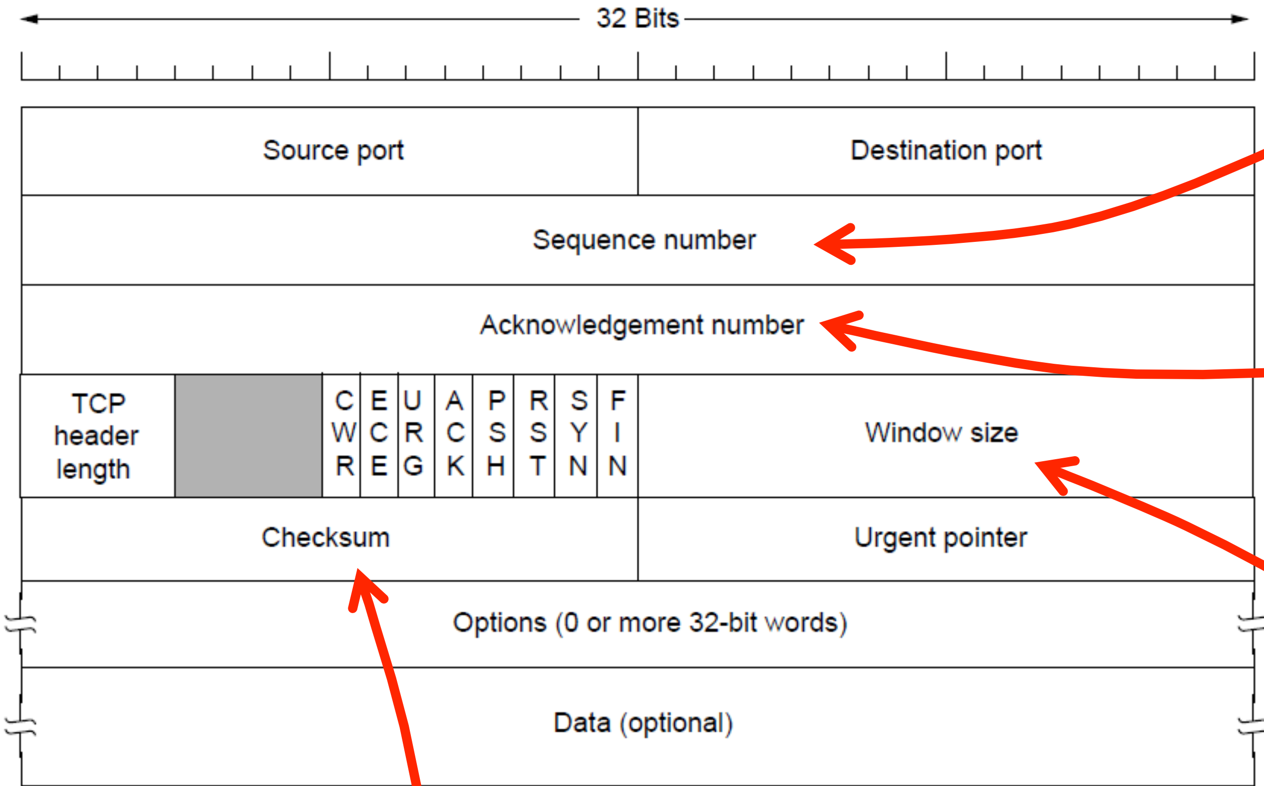
- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - Up to 1500 bytes on an Ethernet, 20 bytes
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header, 20 bytes long
- TCP segment
  - No more than Maximum Segment Size (MSS) bytes
  - Up to 1460 consecutive bytes from the stream

Determining MSS

- Maximum Segment Size (MSS)
  - Default size:
    - Nodes must support min IP MTU of 576 bytes
    - 536 bytes = 576 – 20 (IP header) – 20 (TCP header)
    - Usually doesn't fragment, unless IP/TCP options used
  - Nodes specify MSS during connection setup
    - Done via MSS option field of TCP segment header
    - Could be different in each direction



TCP header



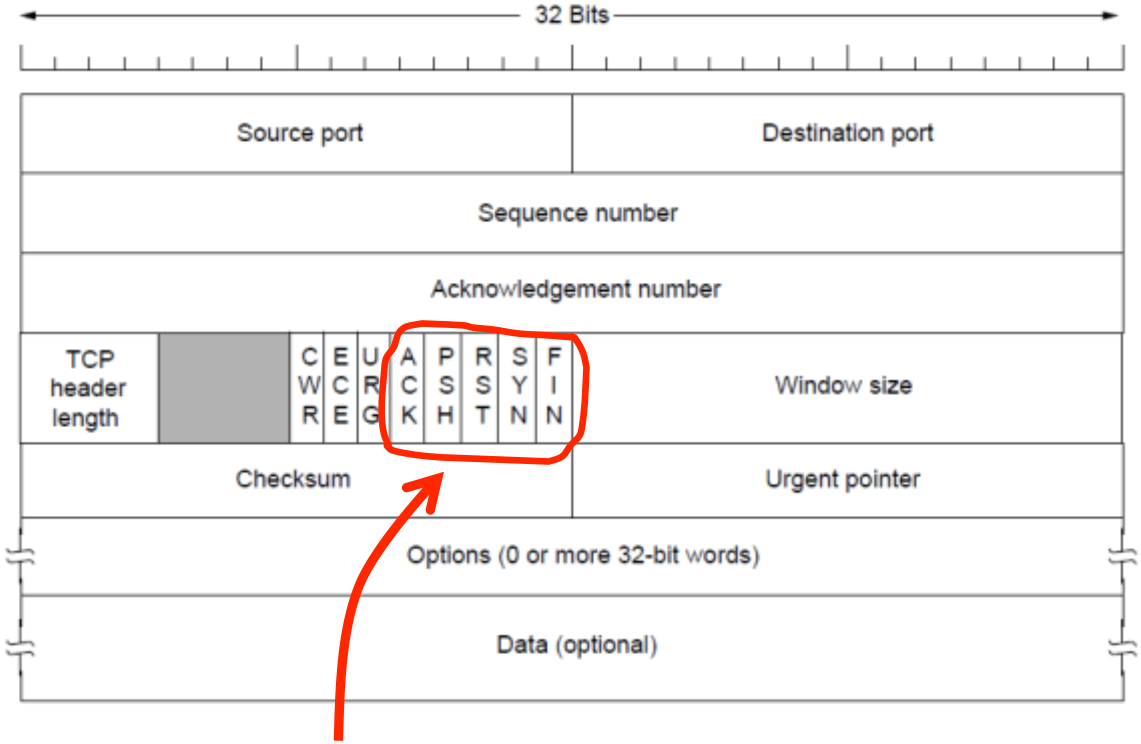
Sequence number of first byte in this segment

Next in-order byte expected

How many bytes may be sent next, 0 = no more data right now

Required in IPv4 and IPv6

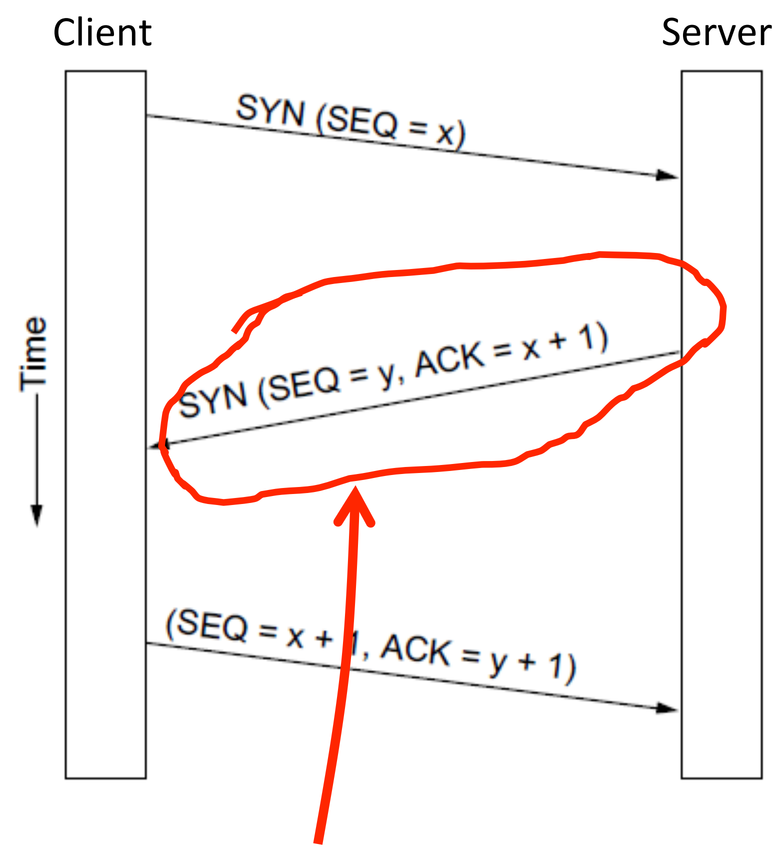
TCP flag bits



ACK	Acknowledgement number is valid (usually set)
PSH	Pushed data, receiver delivery to app without buffering
RST	Abruptly reset a confused connection
SYN	Used to establish connections
FIN	Used to release a connection

Connection: three-way handshake

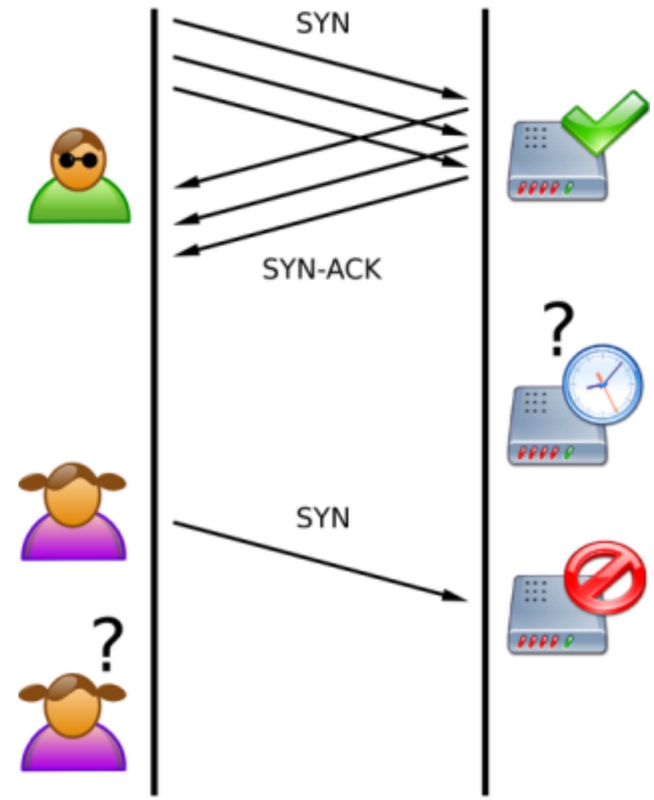
Client	Server
	LISTEN, ACCEPT Passively waits for incoming connection
CONNECT Sends TCP segment to (IP, port) with SYN bit on, ACK bit off	
	Receives segment. OS hands off to process that has done LISTEN on port. If process accepts, send TCP with SYN and ACK bit set.



Server has to remember it's sequence number in step 2

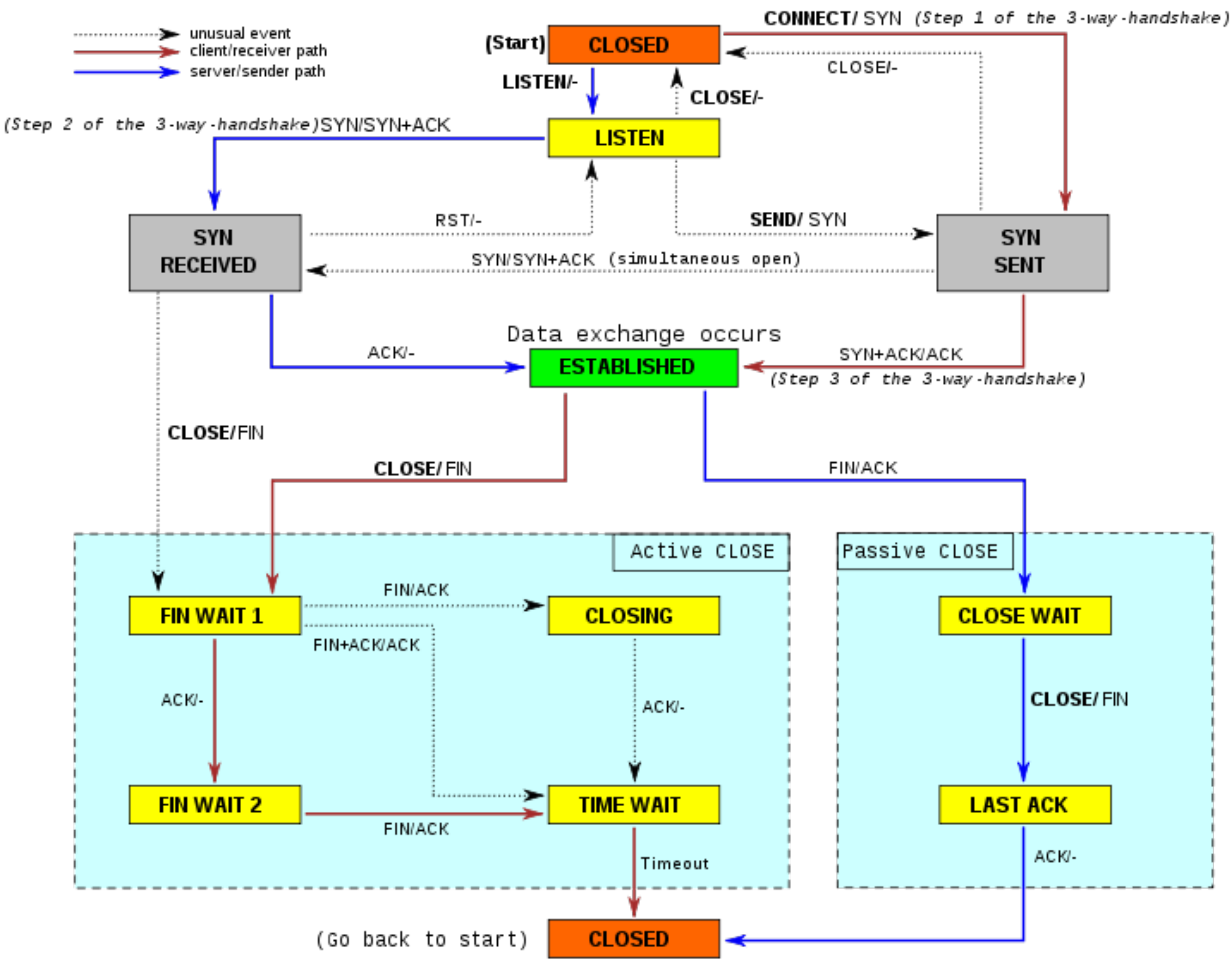
SYN flooding

- SYN flooding
  - Denial-of-service attack
    - Attacker sends large number of SYN requests
    - Never responds or spoofs source IP address
  - Server runs out of resources
    - Server has to track assigned sequence number
    - Fills with half-open connections



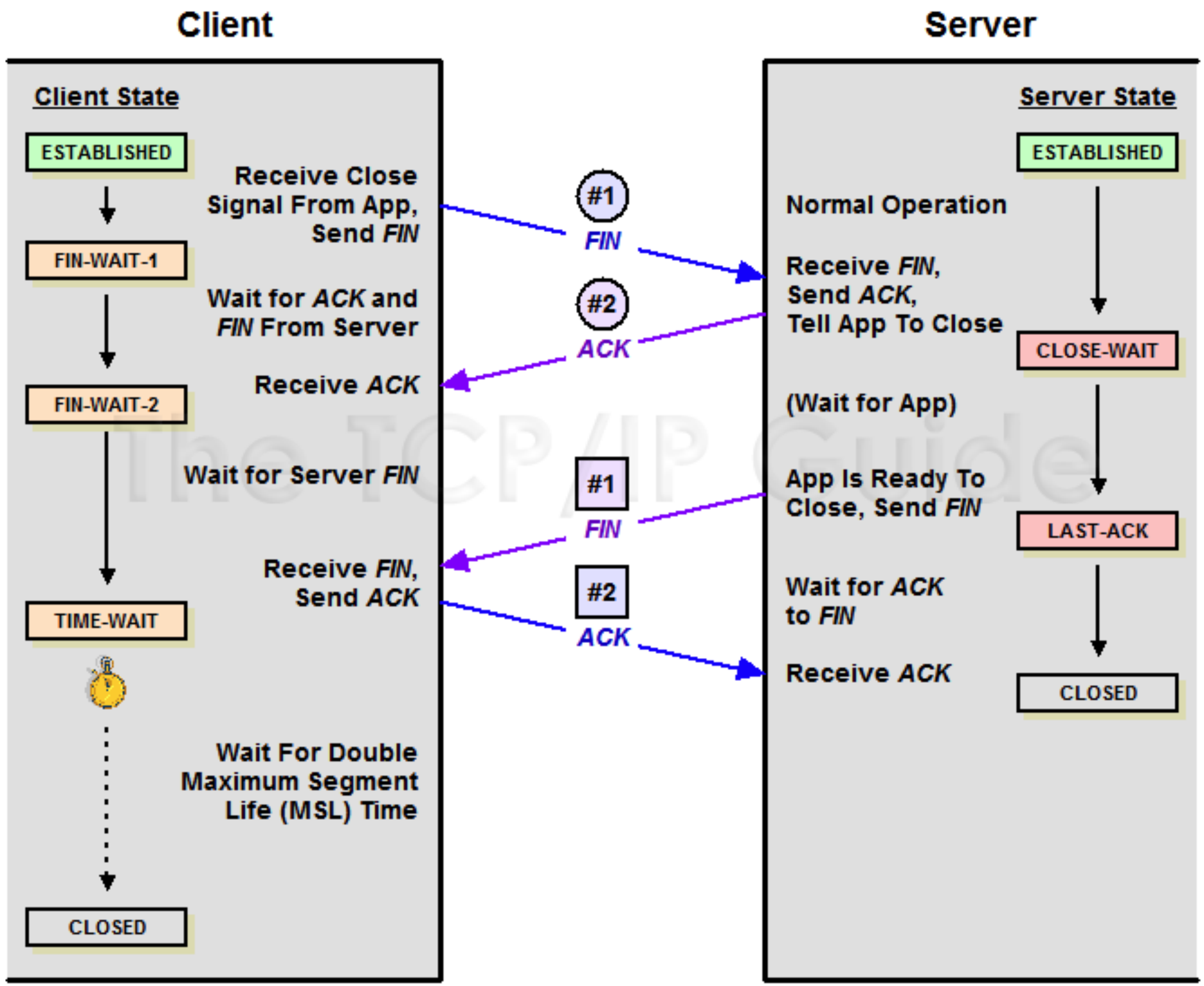
SYN cookies

- Server generates sequence number
  - Uses cryptographic process
  - Combine counter, MSS requested, and secret generated from client/server IP and ports
- Fires off response, forgetting number
- Can recover original sequence number if client responds



unusual event
client/receiver path
server/sender path
(Start)
CLOSED
CONNECT/ SYN (Step 1 of the 3-way-handshake)
CLOSE/-
LISTEN/-
CLOSE/-
LISTEN
(Step 2 of the 3-way-handshake)SYN/SYN+ACK
RST/-
SEND/ SYN
SYN RECEIVED
SYN/SYN+ACK (simultaneous open)
SYN SENT
Data exchange occurs
ACK/-
ESTABLISHED
SYN+ACK/ACK
(Step 3 of the 3-way-handshake)
CLOSE/FIN
CLOSE/ FIN
FIN/ACK
Active CLOSE
Passive CLOSE
FIN WAIT 1
FIN/ACK
CLOSING
CLOSE WAIT
FIN+ACK/ACK
ACK/-
ACK/-
CLOSE/ FIN
FIN WAIT 2
TIME WAIT
LAST ACK
FIN/ACK
Timeout
ACK/-
(Go back to start)
CLOSED

Connection release



http://www.tcpipguide.com/free/t_TCPConnectionTermination-2.htm

TCP extensions

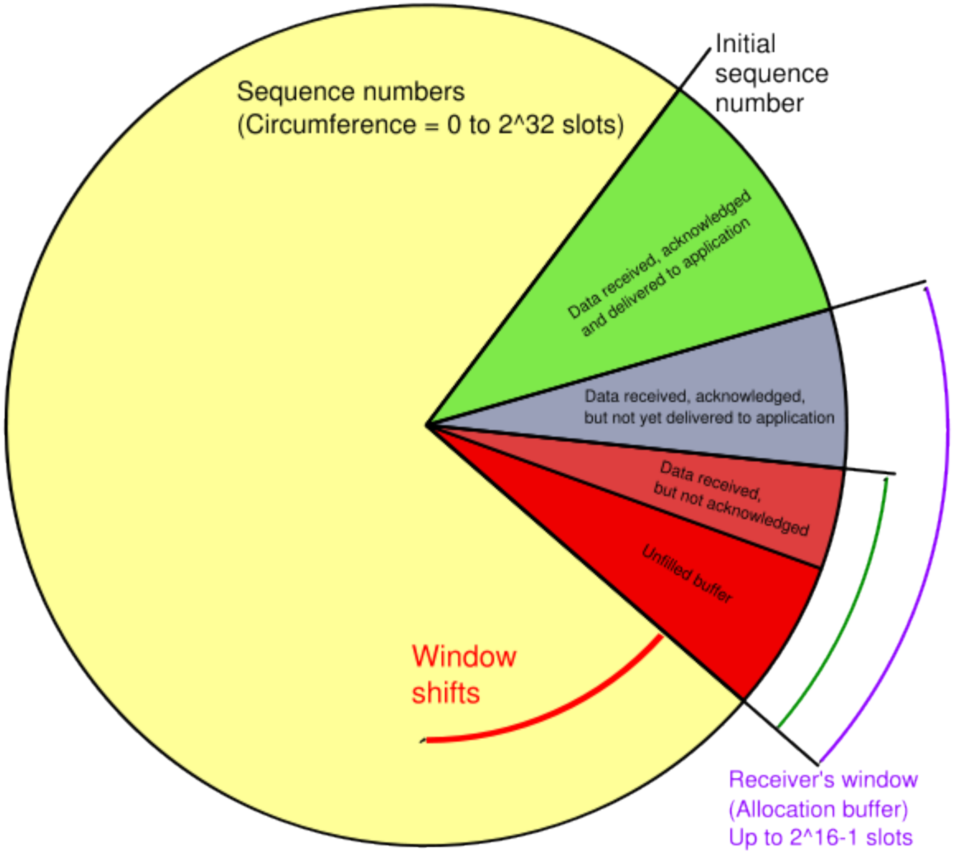
- Timestamp option
  - Timestamp added to segment by the sender
  - Echoed by the received
  - Sender can then compute RTT
  - Also can be combined with sequence number
    - Protects against wraparound
- Large window option
  - Use a scale factor
  - Left shift window size field by up to 14 bits
  - Windows of up to 2^{30} bytes

TCP extensions

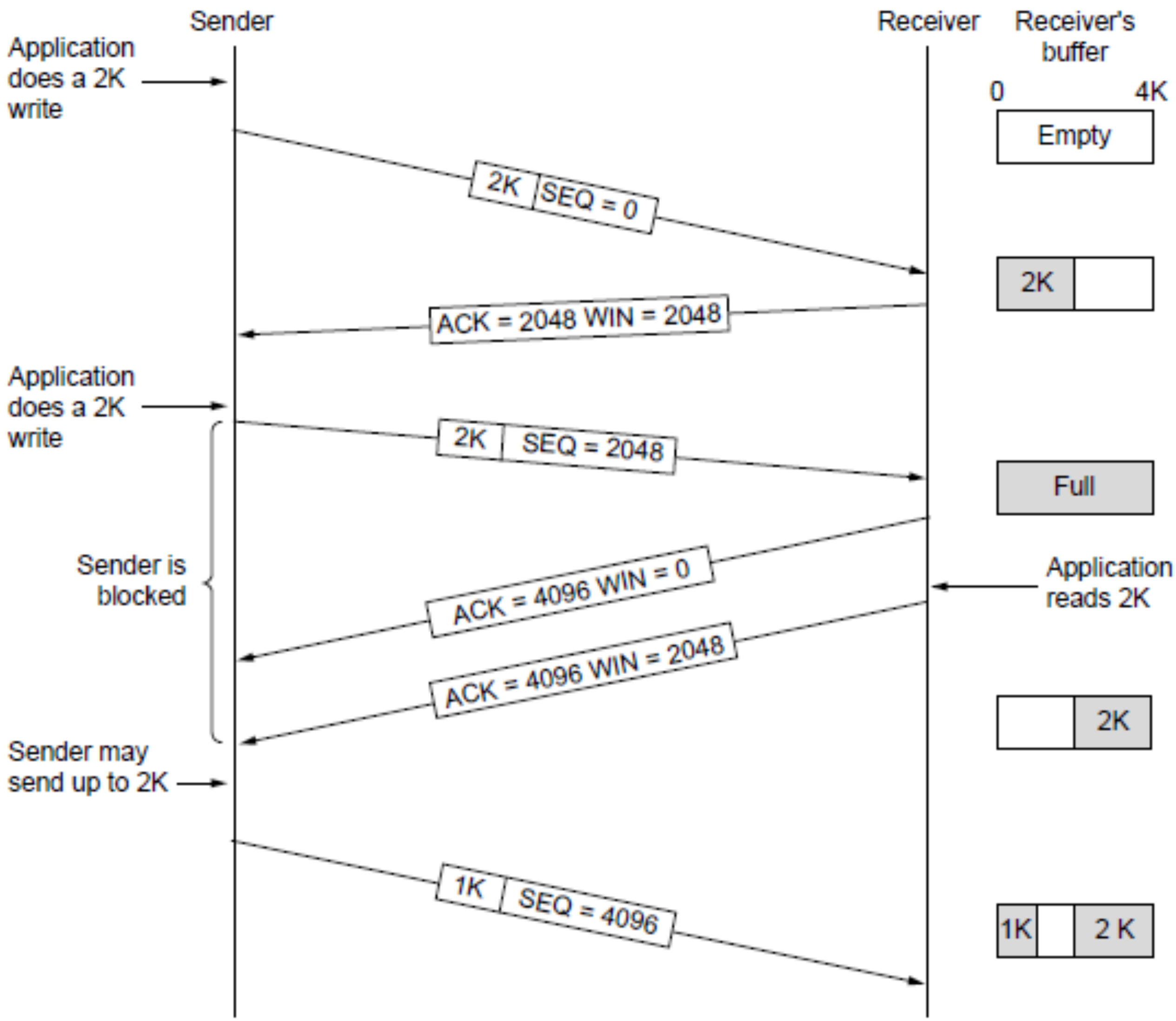
- Selective acknowledgements (SACK)
  - Optional header fields used to acknowledge additional blocks
  - Sender can then resubmit only missing blocks
- Maximum Segment Size (MSS)
  - Only valid extension during connection setup
  - Set a non-default value for maximum segment size

Flow & congestion control

- Flow control
  - Prevent senders from overrunning receiver
  - Window size in segments
- Congestion control
  - Prevent injecting too much data into network
  - Don't want to overload links



TCP sliding window



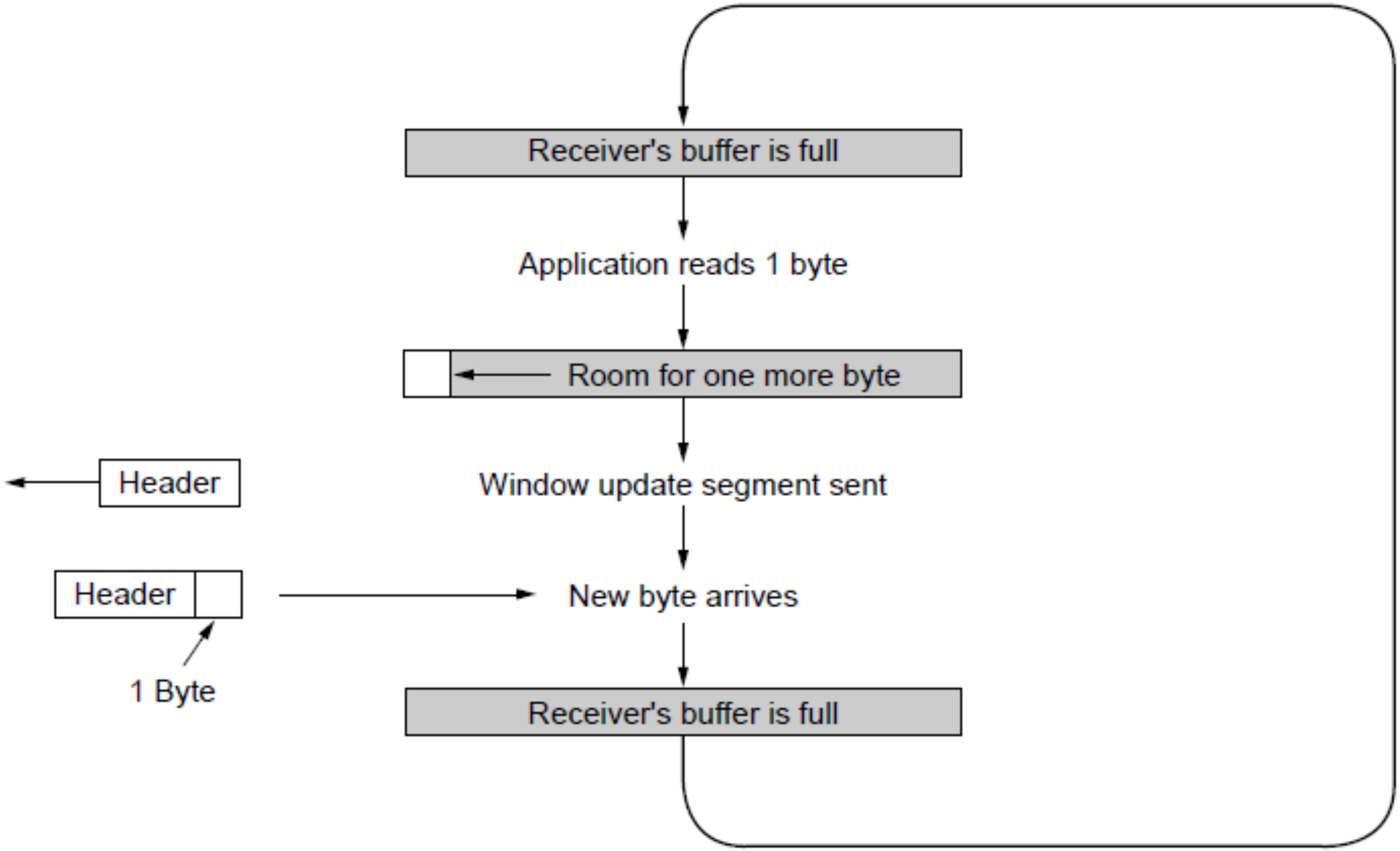
TCP sliding window

- Windows size = 0
  - Bytes up to an including ACK # - 1 have been received
  - Receiver has not consumed data so don't send more
  - When ready, receiver issues same ACK # and non-zero
  - Provides the flow-control in TCP
- Sender can still send:
  - Urgent requests (kill the process)
  - Periodic window probe frames, see if window has opened
    - Prevents deadlock should the receiver's windows update get lost
    - Persistence timer

Improving performance

- TCP does not require:
  - Senders send data immediately
  - Receivers deliver data immediately
- Delayed acknowledgements
  - Receiver has pending ACK
    - Wait before sending (< 500ms)
  - If data arrives, piggyback on ACK
  - Reduces load on network by receiver

Silly Window Syndrome



Nagle's Algorithm

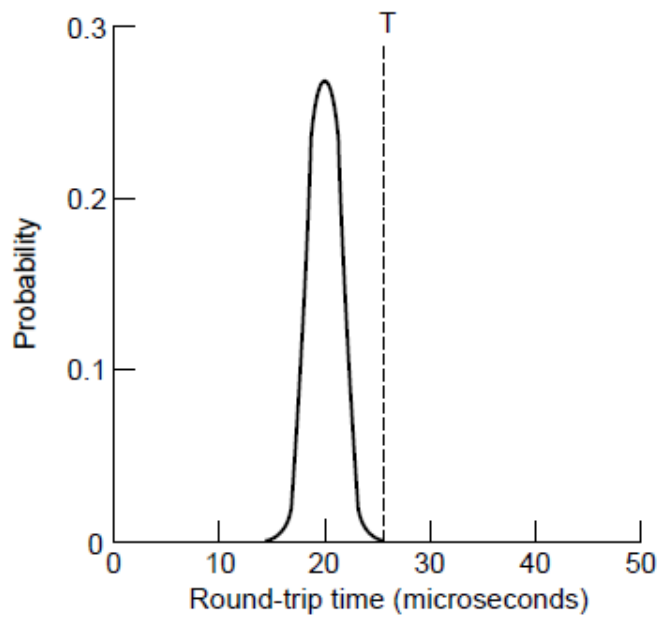
- Sender-side silly window avoidance
- Application produces data to send
  - If >= MSS, send segment
  - If no segments in flight, send the segment
  - Otherwise queue the data
- Limits to one small segment in network
  - But bad for interactive apps like gaming
  - Especially bad if combined with delayed ACKs
    - write byte, write byte, read byte
  - Can be disabled, TCP_NODELAY option

Clark's solution

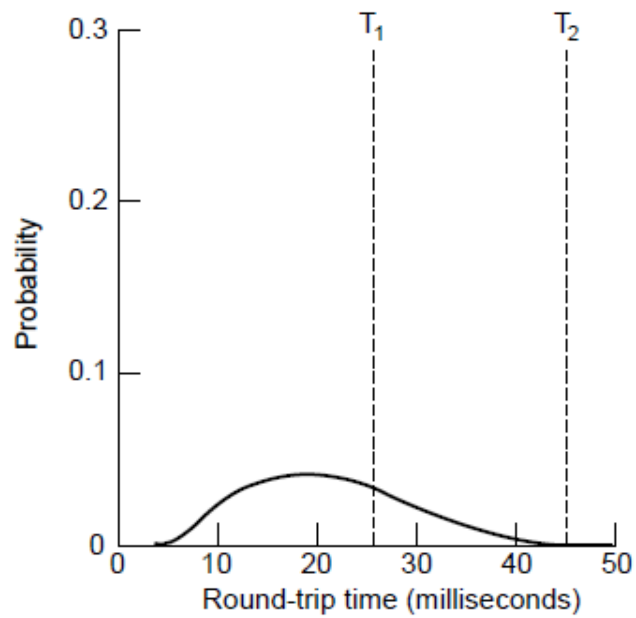
- Receiver-side silly window avoidance
- Do not send window size update unless:
  - It can handle full MSS size
  - Half of its buffer is empty

TCP timeouts

- TCP is reliable transport
  - Transmits data if ACK not recv'd in certain time
  - But what timeout value to use?
  - RTT times vary widely on the Internet



ACK arrival times, point-to-point



ACK arrival times, Internet

Simple adaptive timeout

- Smoothed Round-Trip Time (SRTT)
  - Start timer whenever you send segment
  - When ACK arrives, let R = RTT of segment
  - Exponentially weighted moving average:

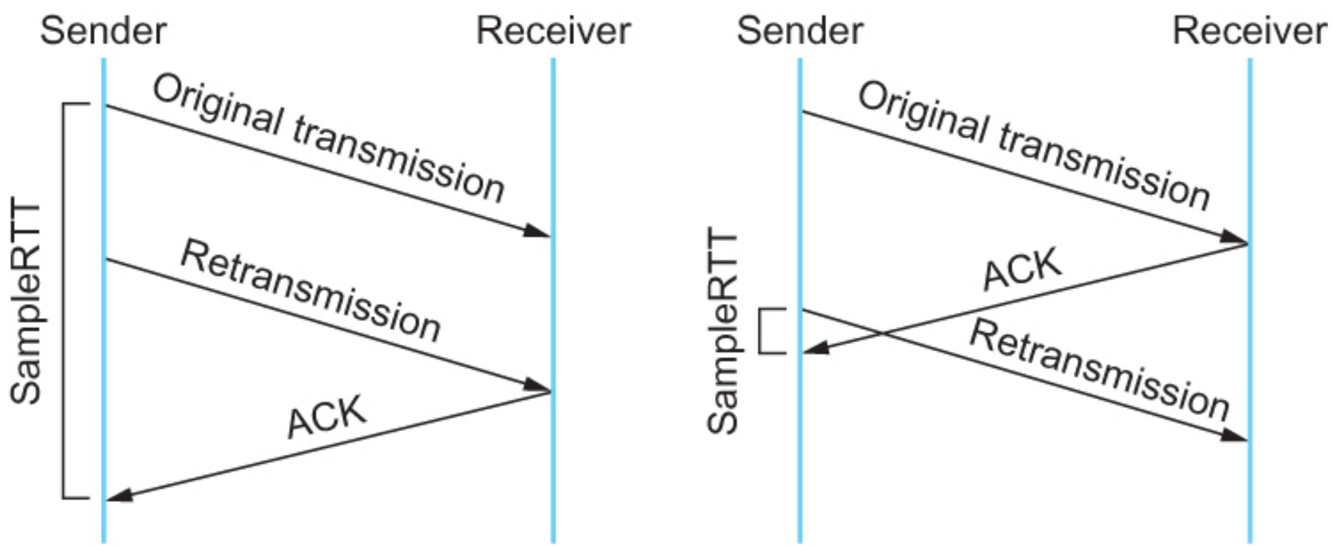
$$SRTT = \alpha(SRTT) + (1-\alpha)R$$

$$Timeout = 2(SRTT)$$

α typically 0.8 or 0.9

Simple adaptive timeout

- Problem 1: Timeout formula uses constant value (2)
  - Does not respond to variance in data
  - Delays become highly variable under high load
  - Timing out early just makes things worse
- Problem 2: Updating SRTT on retransmitted frames



Improved adaptive timeout

- Problem 1: Timeout formula uses constant value (2)
- Solution 1: Take variance into account
  - Jacobson/Karel's algorithm

$SRTT = \alpha(SRTT) + (1-\alpha)R$

$RTTVAR = \beta(RTTVAR) + (1-\beta)|SRTT-R|$

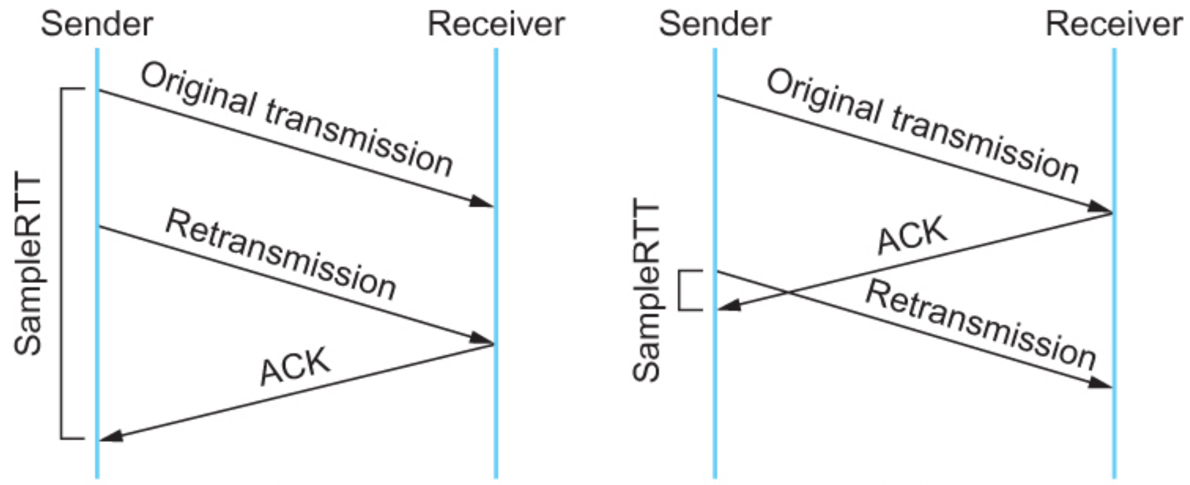
$Timeout = SRTT + 4(RTTVAR)$

α typically 0.8 or 0.9

β typically 0.75

Improved adaptive timeout

- Problem 2: Updating SRTT on retransmitted frames
- Solution 2: Don't do that
  - Karn/Partidge algorithm, 1987
  - Ignore RTT's of packet that were retransmitted
  - Also double timeout value when retransmitting (exponential backoff)



Staying Alive

- TCP keep-alive timer
  - If connection is idle > timeout, send a frame to see if other side still alive
  - Checking for dead peer
  - Prevent disconnection due to inactivity
    - NAT box might drop your state if you don't communicate once in awhile

Summary

- TCP protocol
  - Reliable byte-oriented delivery
  - TCP segments
  - Connection setup/shutdown
  - Flow control via window size feedback
  - Avoiding silly window syndrome
    - Nagel's algorithm, Clark's algorithm
  - Adaptive timeouts
    - Jacobson/Karel's algorithm, Karn/Partidge algorithm