
University of Colorado Boulder

Fundamentals of Data Communications

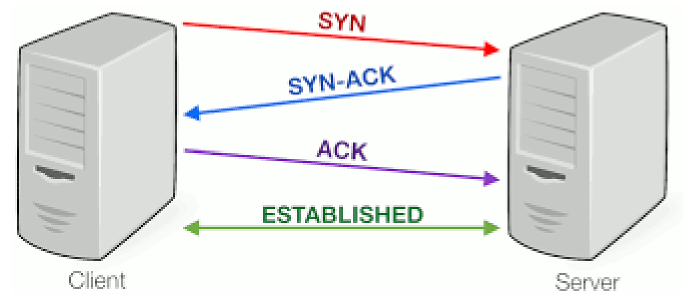
Transmission Control Protocol (TCP)

Levi Perigo, Ph.D.
University of Colorado Boulder
Department of Computer Science
Network Engineering

Review

Key topics

- **Establishing a connection**
  - Three-way Handshake
  - Sockets



- **Transmitting data**
  - Sequencing and acknowledging
  - Flow control
    - ***Sliding window***
  - Congestion control

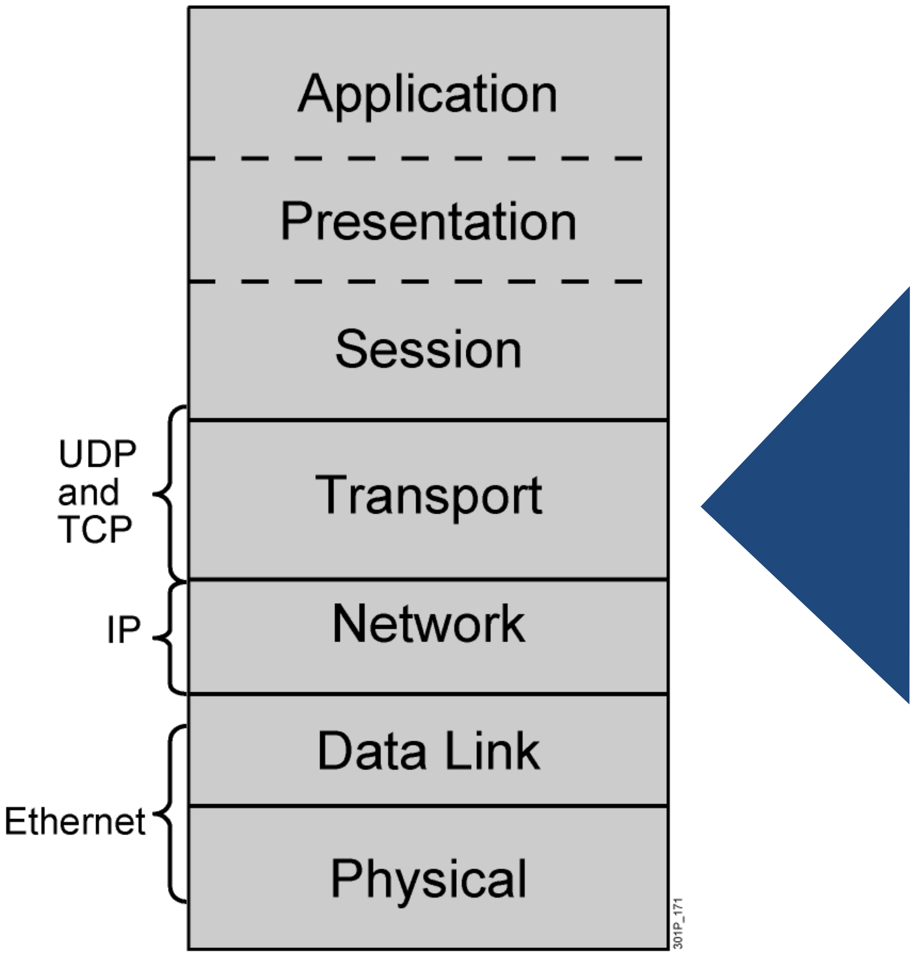
Network Connection Established – What next?

- **Once computers have found each other, how do they communicate?**
  - They break down the data into segments/packets
    - ***Transmit to sockets***
    - ***Must have a 3-way handshake connection***

    - ***But don't know the reliability of the connection, so they don't want to try to send everything at once***
      - Starts sending in groups (**flow control**) (and groups get progressively bigger)
        - So now it sends (double) packets 2 & 3

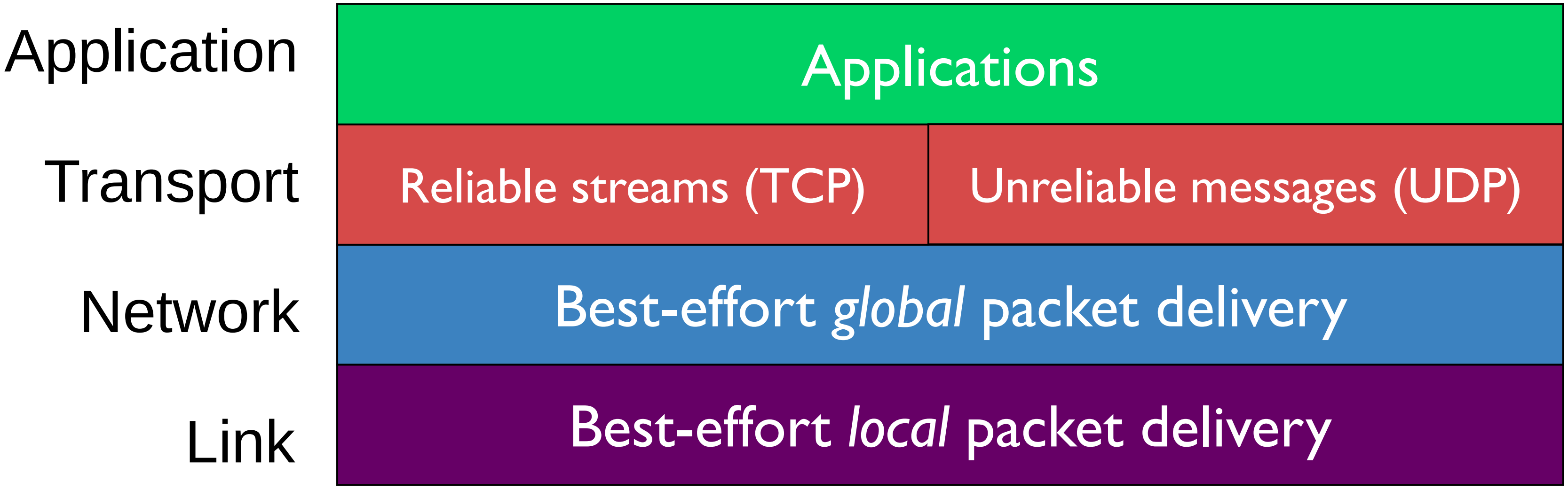
      - What about a failure?
        - Cut window in half (from where you are) and start over at 1 window size (**congestion control**)

Transport Layer – TCP & UDP



- Session multiplexing
- Segmentation
- Flow control (when required)
- Connection-oriented (when required)
- Reliability (when required)

Transport Layer Services



- Transport layer is where we "deliver"
  – Provide applications with good abstractions
  – Without support or feedback from the network

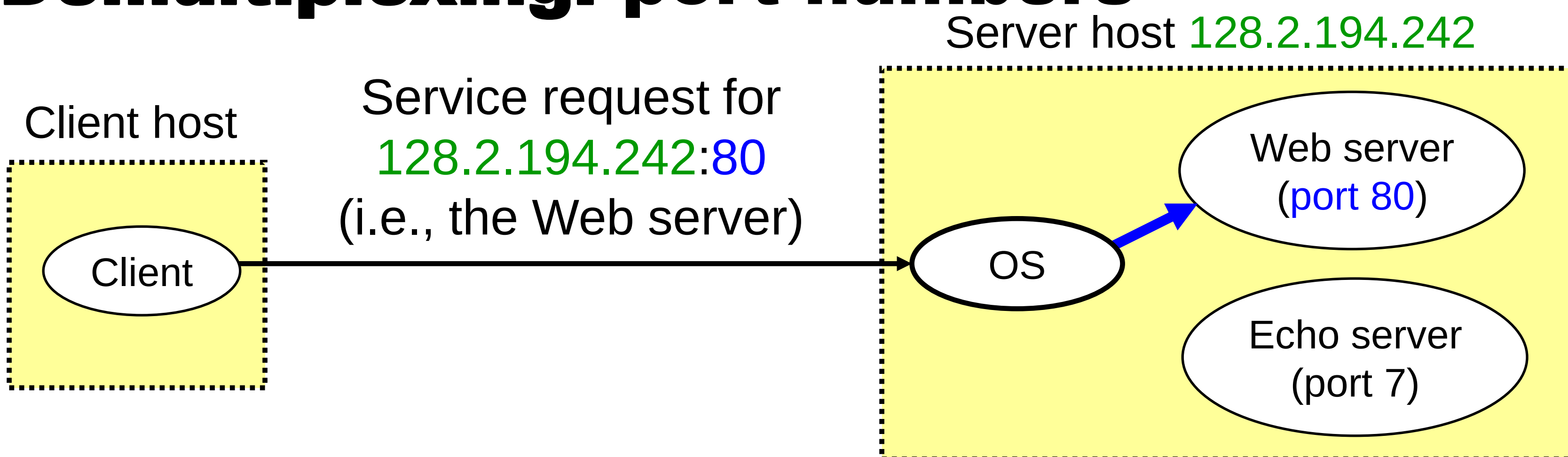
Transport Protocols

- Logical communication between processes
  - Sender divides a message into segments
  - Receiver reassembles segments into message
    - ***Post Office analogy***

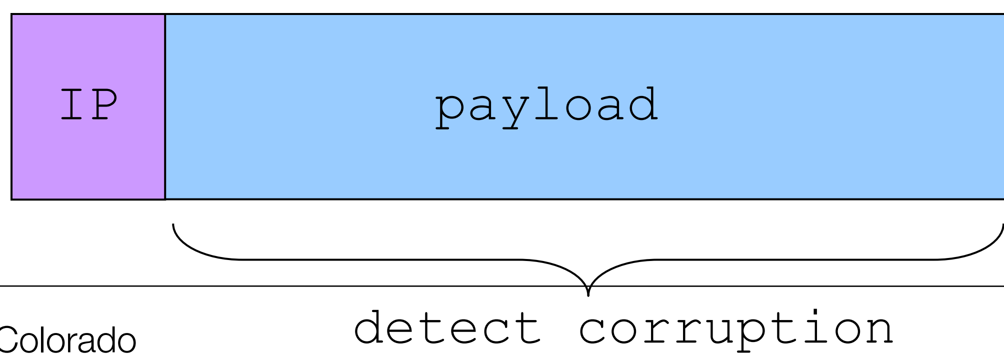
- Transport services
  - (De)multiplexing packets
  - Detecting corrupted data
  - Optionally: reliable delivery, flow control, …

Two Basic Transport Features

- **Demultiplexing: port numbers**



- **Error detection: checksums**

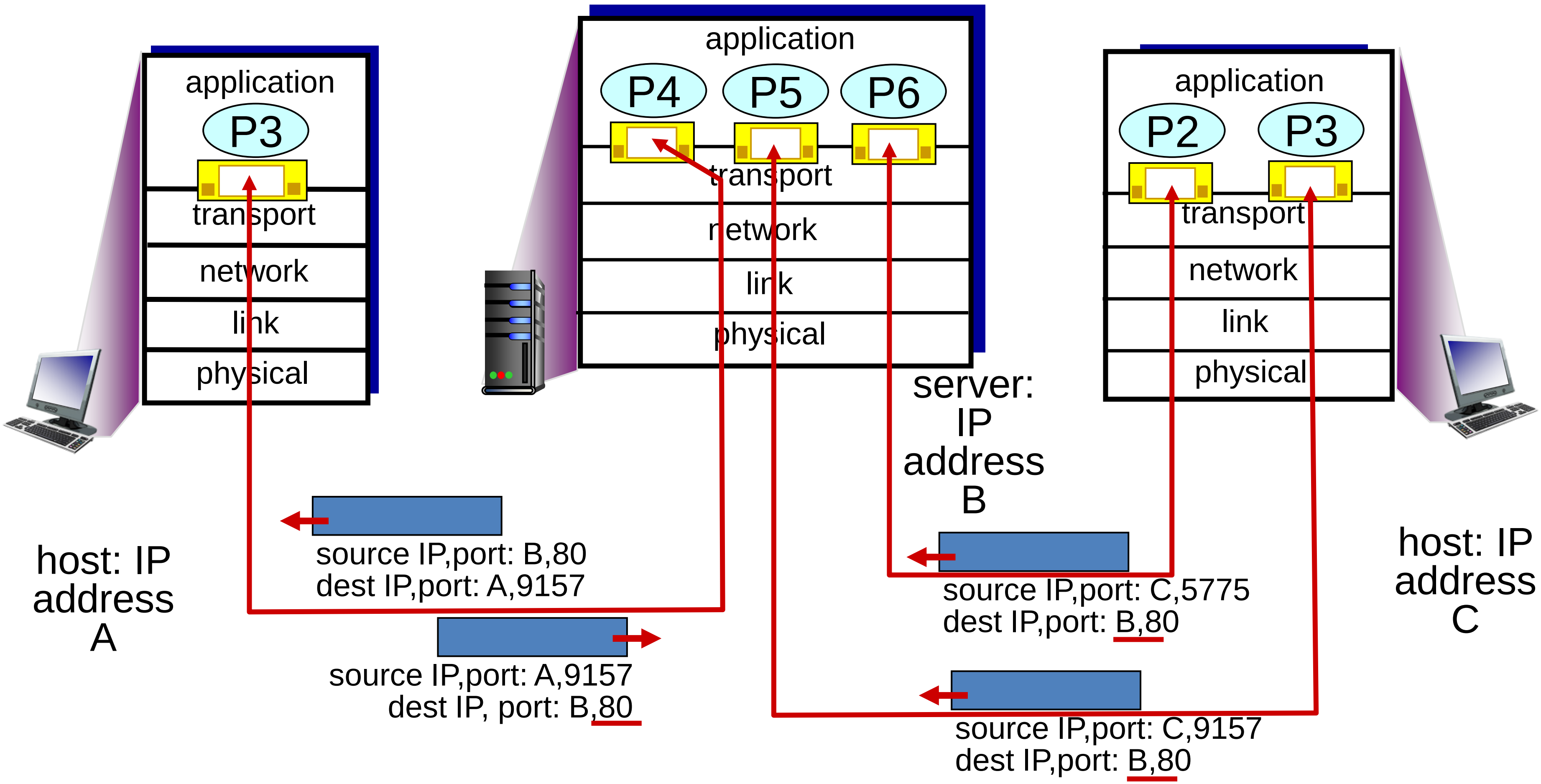


Connection-oriented Demux (sockets)

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- Demux: Receiver uses all four values to direct segment to appropriate socket

- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - Non-persistent HTTP will have different socket for each request

Connection-oriented Demux: Example



three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

Transport Layer – TCP & UDP

- **Hide the network requirements from the application layer**

- **Connection-oriented**
  - Reliable transport
  - TCP
    - ***connection-oriented***
    - ***provides error checking***
    - ***delivers data reliably***
    - ***operates in full-duplex mode***

- **Connectionless**
  - Best-effort transport
  - UDP
    - ***provides applications with access to the network layer without the overhead of the reliability mechanisms of TCP.***
    - ***Connectionless / best-effort***

UDP Characteristics

Transport layer of OSI and TCP/IP models

Provides applications with access to the network layer without the overhead of reliability mechanisms

Is a **connectionless** protocol
- ***Faster***

Provides limited error checking

Provides **best-effort delivery**
- ***Real-time communications***
  - **Voice & Video**

Has no data-recovery features

TCP Characteristics

Transport layer of the OSI and TCP/IP stack

Provides applications with access to the network layer without the overhead of reliability mechanisms

Connection-oriented protocol
- reliable

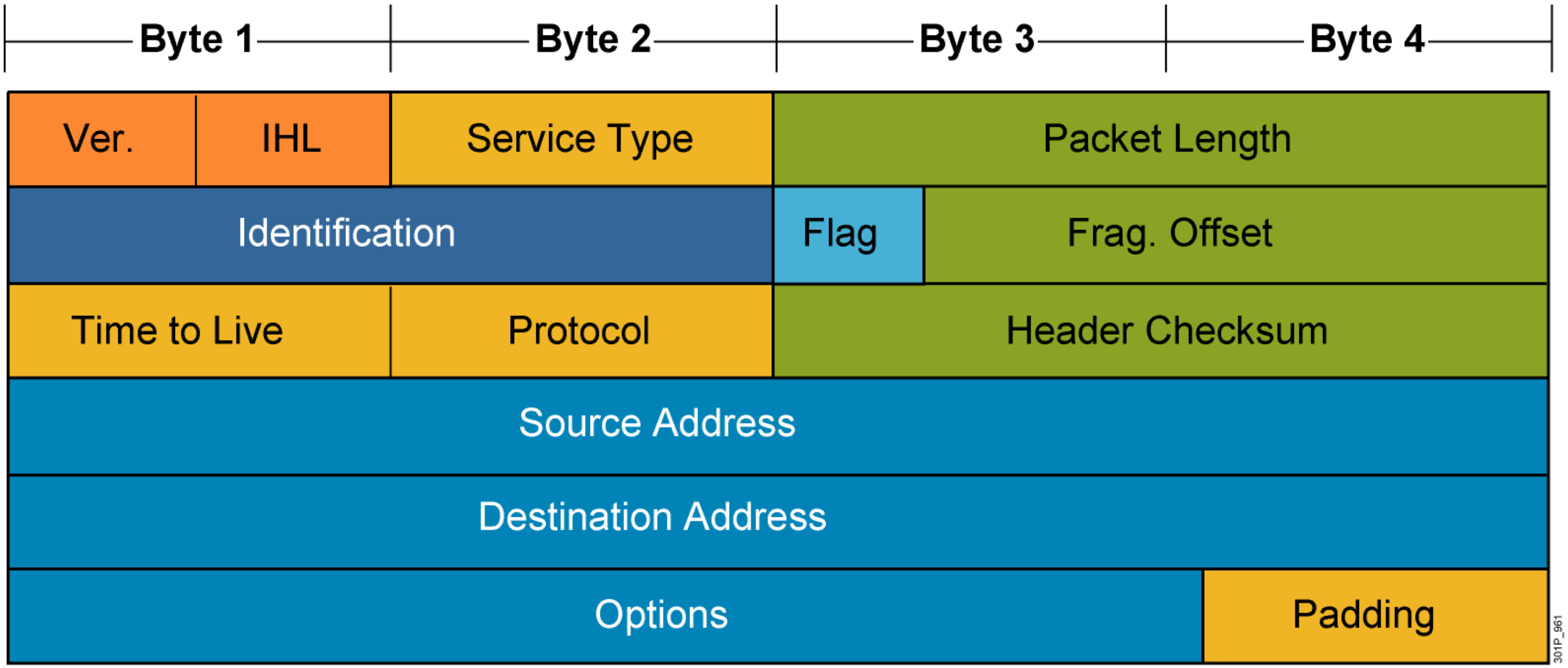
Error checking

Sequencing of data packets

Acknowledgement of receipt

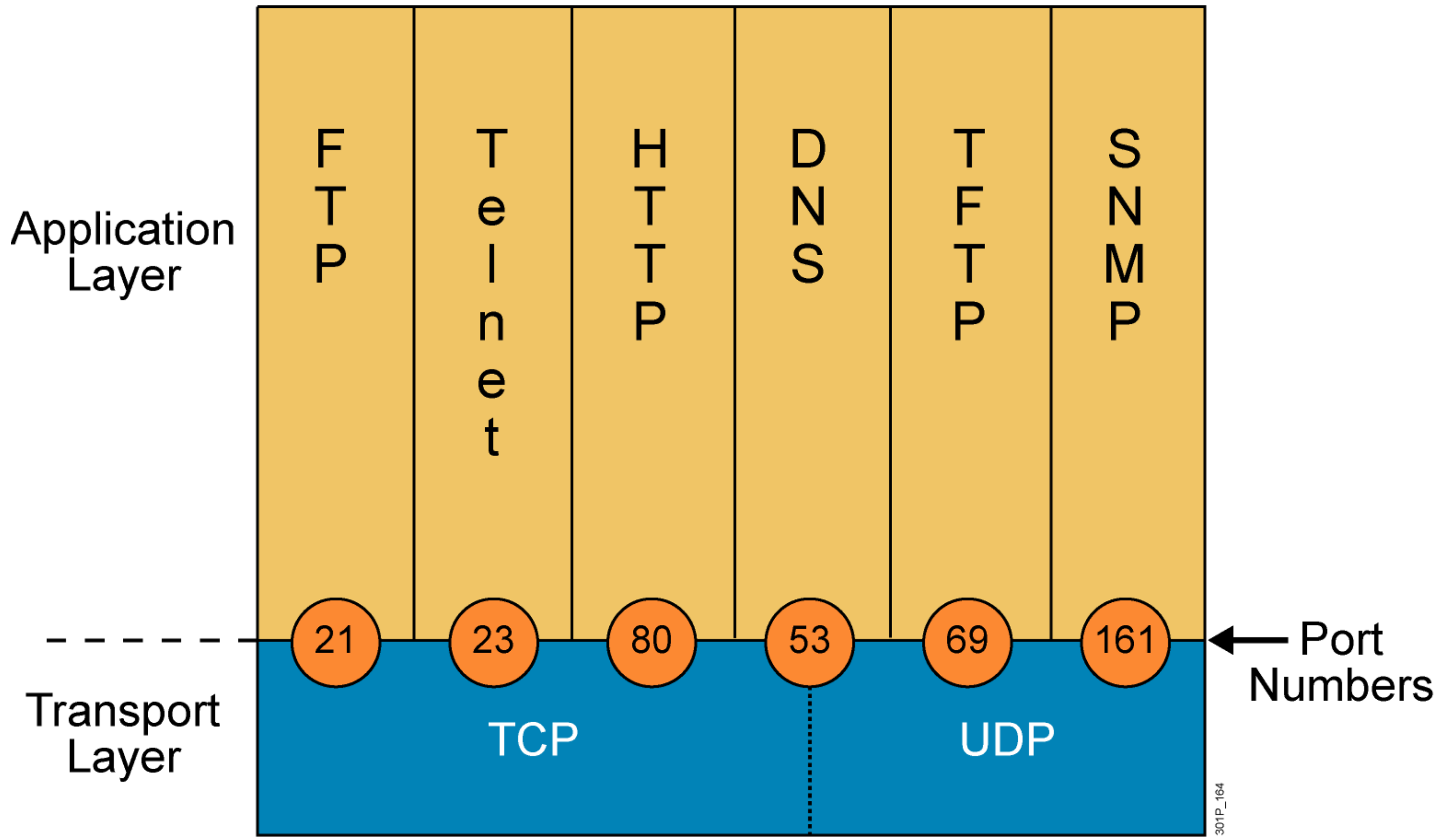
Data-recovery features

Mapping Layer 3 to Layer 4



- Layer 3 - IP Header
  - Protocol Field (TCP or UDP)

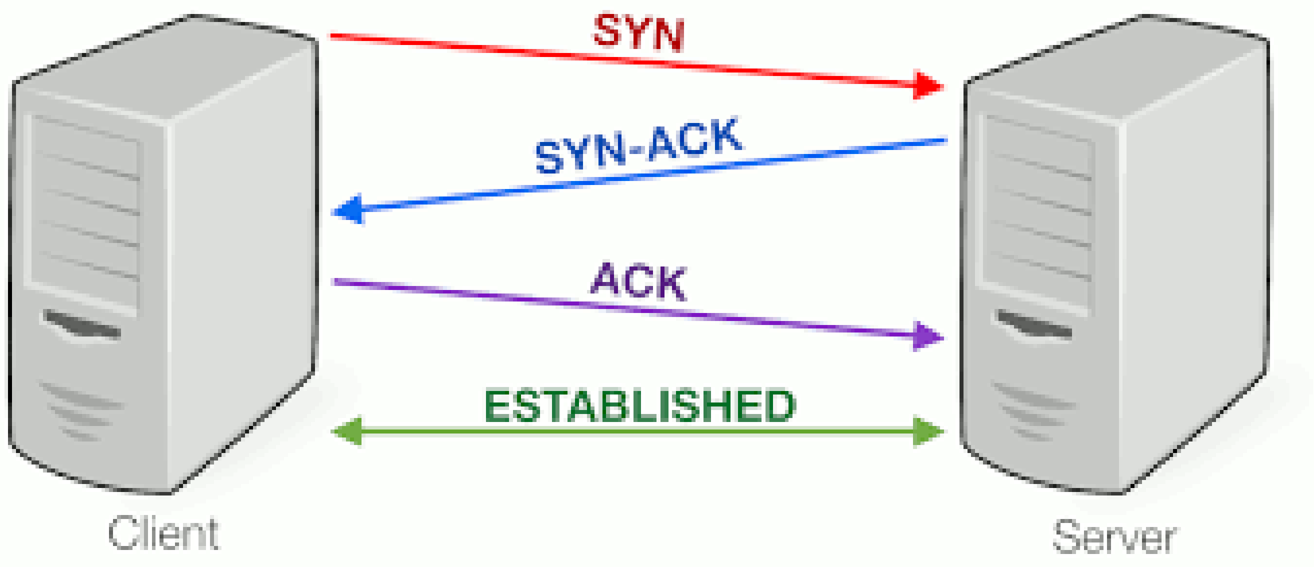
Mapping Layer 4 to Applications



TCP

- Sequencing of segments / acknowledgment
  - ***When a single segment is sent, receipt is acknowledged, and the next segment is then sent***
  - ***Send <-> Ack***
- Window size
  - ***Allows a specified number of unacknowledged segments to be sent (in flight)***
  - ***Sliding window***
    - Window that can change size dynamically to accommodate the flow of segments
- Flow control
  - ***Prevents sending to a host and overflowing the buffer***
    - Results in slowing down the network

Establishing a TCP Connection



Each host tells its ISN to the other host.

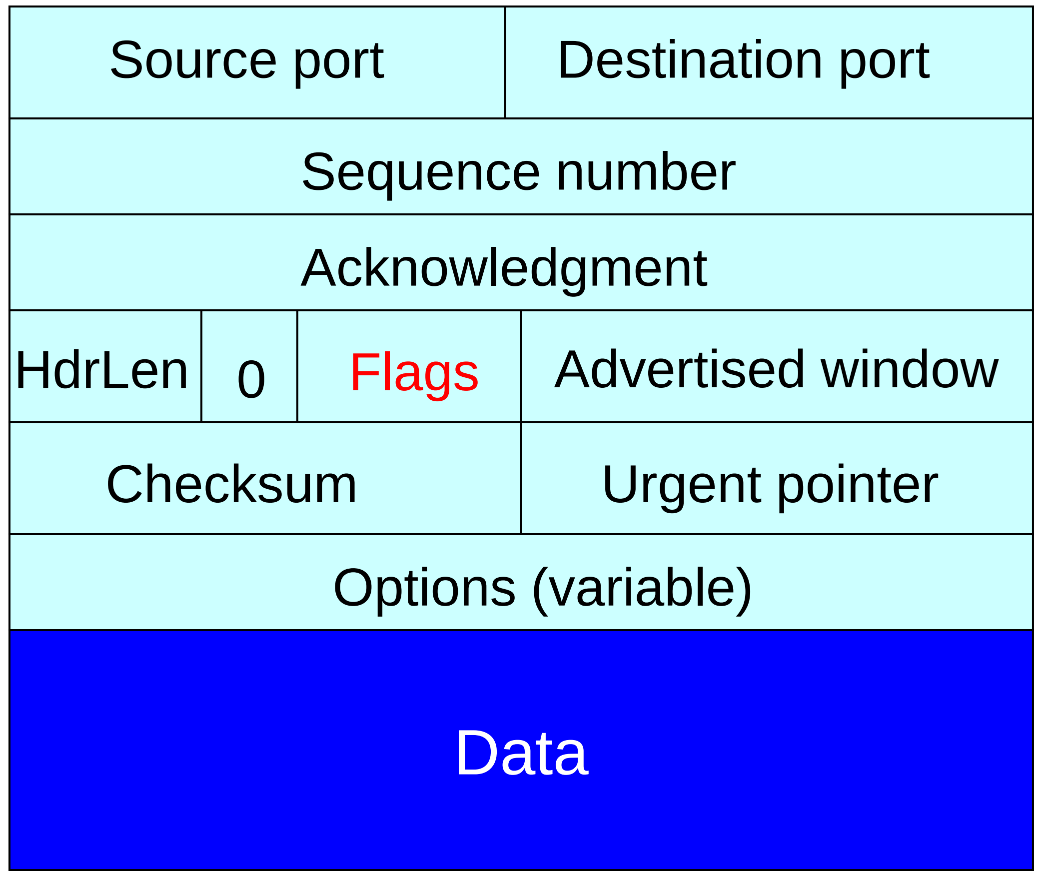
- Three-way handshake to establish connection
  - Host A sends a **SYN** (open) to the host B
  - Host B returns a SYN acknowledgment (**SYN ACK**)
  - Host A sends an **ACK** to acknowledge the SYN ACK

Establishing a TCP Connection

- **SYN (client)**
  – Client randomly chooses an initial sequence number (ISN)
    - ***Places the client ISN in the sequence number field***
- **SYNACK (server)**
  – ACK field = client ISN + 1
    - ***(confirming receipt)***
  – Sever randomly chooses own initial sequence number (ISN)
    - ***Places the server ISN in the sequence number field***
- **ACK (client)**
  – Acknowledges the server's connection
    - ***Puts the server ISN + 1 in the acknowledgement field***

TCP Header

Flags: SYN
FIN
RST
PSH
URG
ACK



Step 1: A’s Initial SYN Packet

<table>
<tr><td colspan="3">A’s port</td><td>B’s port</td></tr>
<tr><td colspan="4">A’s Initial Sequence Number</td></tr>
<tr><td colspan="4">Acknowledgment</td></tr>
<tr><td>20</td><td>0</td><td>Flags</td><td>Advertised window</td></tr>
<tr><td colspan="3">Checksum</td><td>Urgent pointer</td></tr>
<tr><td colspan="4">Options (variable)</td></tr>
</table>

Flags:
- SYN
- FIN
- RST
- PSH
- URG
- ACK

A tells B it wants to open a connection…

Step 2: B’s SYN-ACK Packet

Flags: SYN
FIN
RST
PSH
URG
ACK

<table>
<tr><td colspan="3">B’s port</td><td>A’s port</td></tr>
<tr><td colspan="4">B’s Initial Sequence Number</td></tr>
<tr><td colspan="4">A’s ISN plus 1</td></tr>
<tr><td>20</td><td>0</td><td>Flags</td><td>Advertised window</td></tr>
<tr><td colspan="3">Checksum</td><td>Urgent pointer</td></tr>
<tr><td colspan="4">Options (variable)</td></tr>
</table>

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

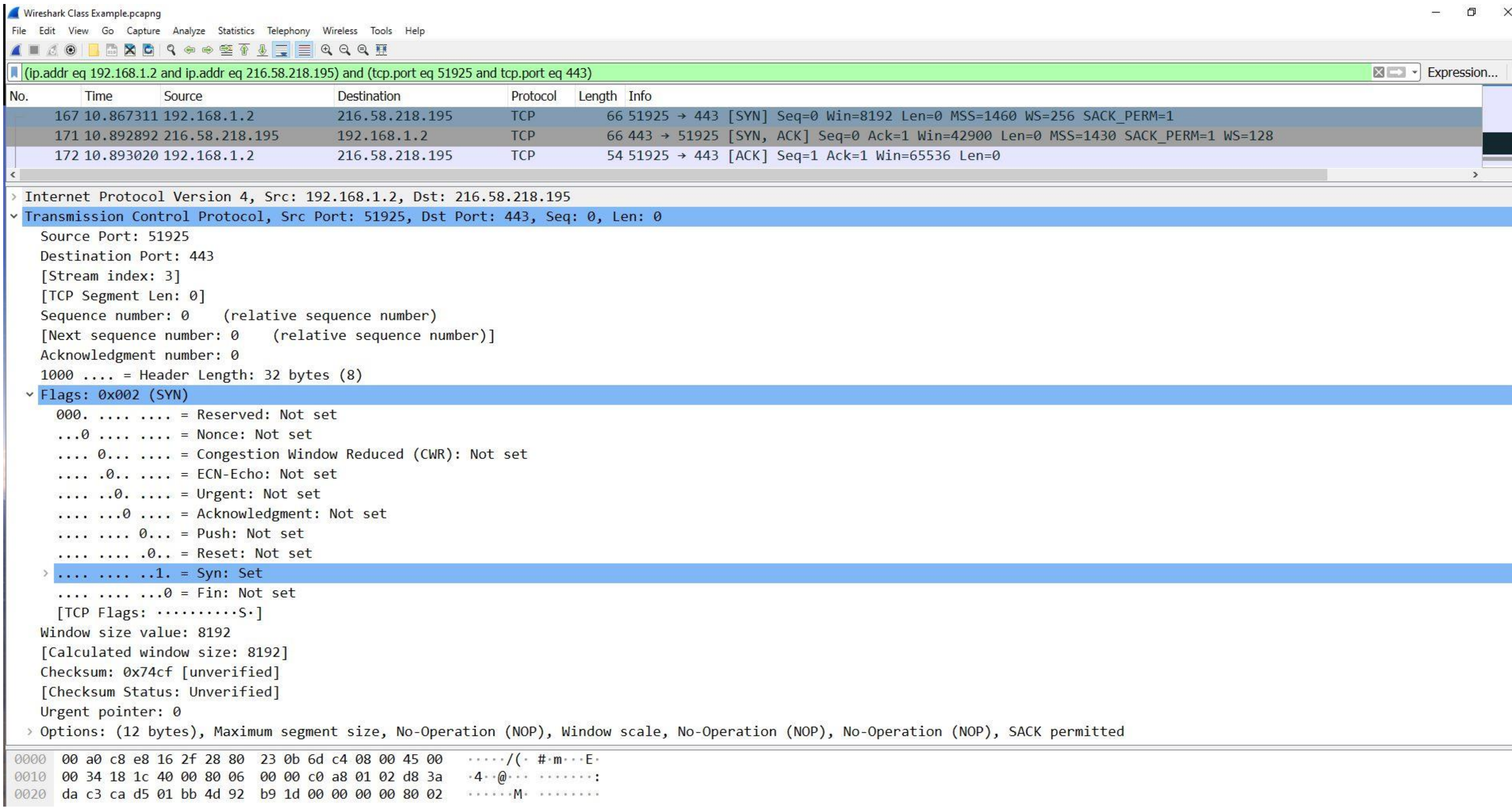
Flags:
- SYN
- FIN
- RST
- PSH
- URG
- ACK

<table>
<tr><td colspan="3">A’s port</td><td>B’s port</td></tr>
<tr><td colspan="4">Sequence number</td></tr>
<tr><td colspan="4">B’s ISN plus 1</td></tr>
<tr><td>20</td><td>0</td><td>Flags</td><td>Advertised window</td></tr>
<tr><td colspan="3">Checksum</td><td>Urgent pointer</td></tr>
<tr><td colspan="4">Options (variable)</td></tr>
</table>

A tells B it is okay to start sending

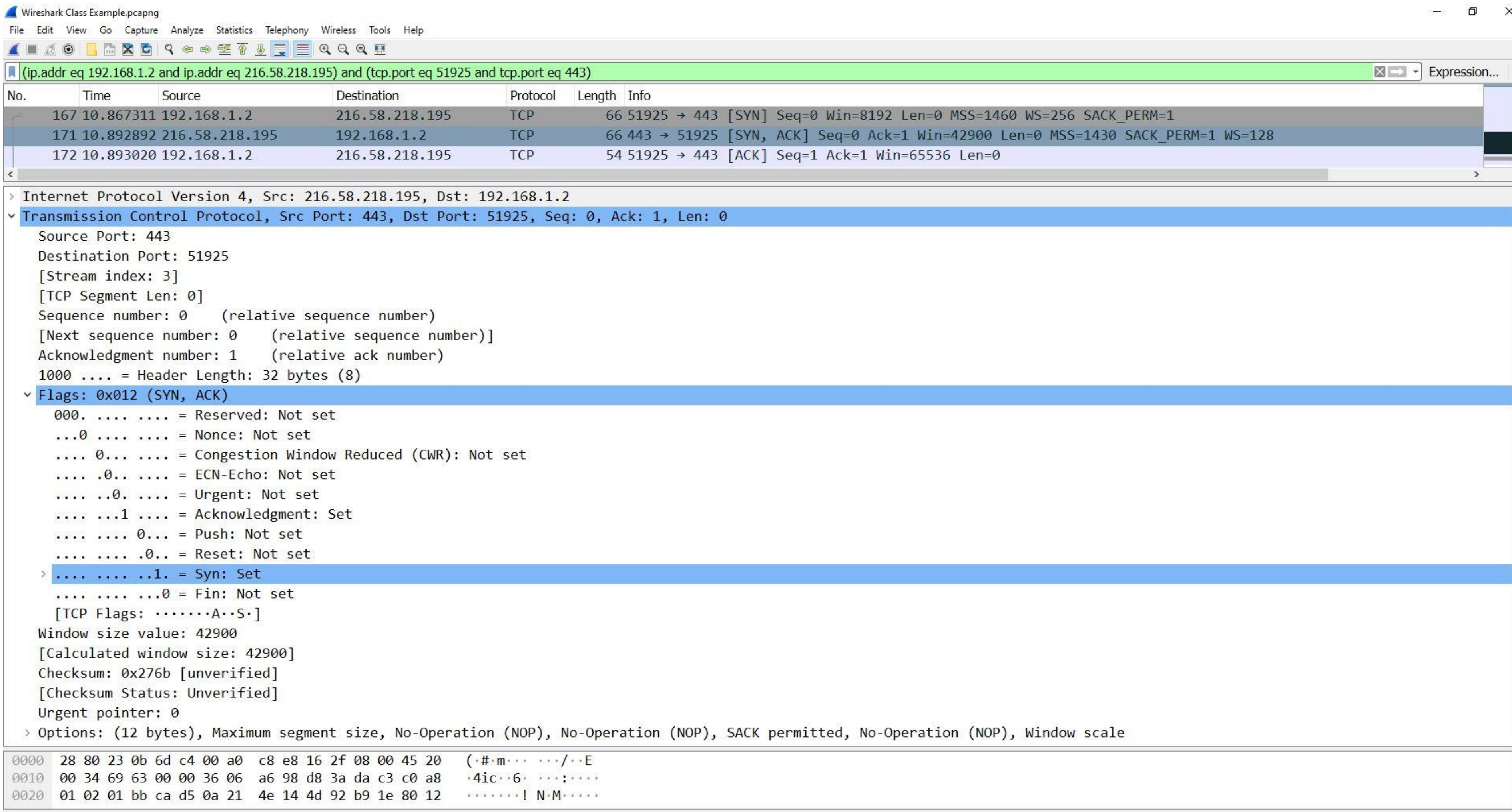
… upon receiving this packet, B can start sending data

TCP SYN



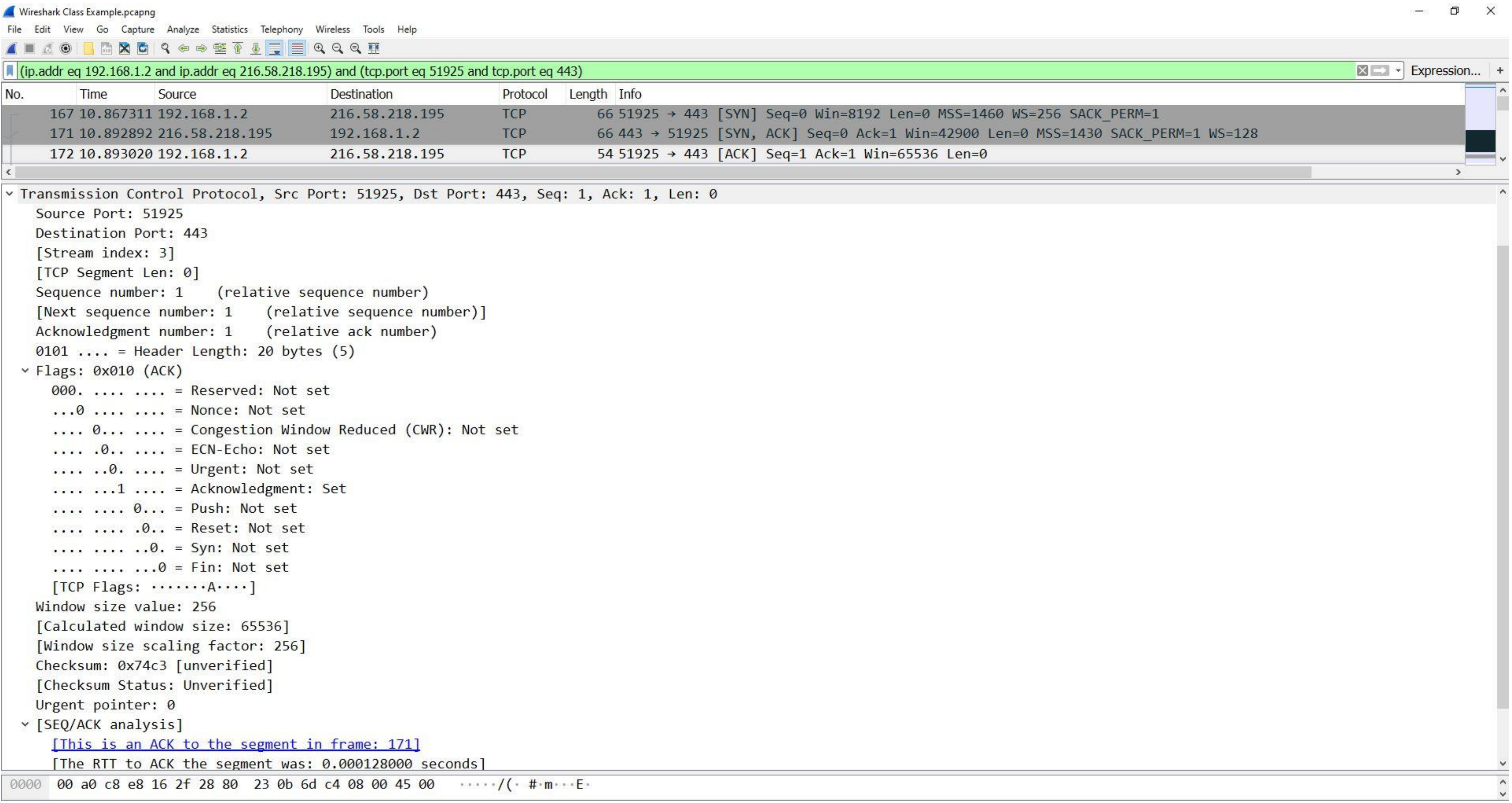
Wireshark Class Example.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
(ip.addr eq 192.168.1.2 and ip.addr eq 216.58.218.195) and (tcp.port eq 51925 and tcp.port eq 443)
Expression...
No. Time Source Destination Protocol Length Info
167 10.867311 192.168.1.2 216.58.218.195 TCP 66 51925 → 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
171 10.892892 216.58.218.195 192.168.1.2 TCP 66 443 → 51925 [SYN, ACK] Seq=0 Ack=1 Win=42900 Len=0 MSS=1430 SACK_PERM=1 WS=128
172 10.893020 192.168.1.2 216.58.218.195 TCP 54 51925 → 443 [ACK] Seq=1 Ack=1 Win=65536 Len=0
Internet Protocol Version 4, Src: 192.168.1.2, Dst: 216.58.218.195
Transmission Control Protocol, Src Port: 51925, Dst Port: 443, Seq: 0, Len: 0
Source Port: 51925
Destination Port: 443
[Stream index: 3]
[TCP Segment Len: 0]
Sequence number: 0 (relative sequence number)
[Next sequence number: 0 (relative sequence number)]
Acknowledgment number: 0
1000 .... = Header Length: 32 bytes (8)
Flags: 0x002 (SYN)
000. .... = Reserved: Not set
...0 .... = Nonce: Not set
.... 0... .... = Congestion Window Reduced (CWR): Not set
.... .0.. .... = ECN-Echo: Not set
.... ..0. .... = Urgent: Not set
....0 .... = Acknowledgment: Not set
.... 0... = Push: Not set
.... .0.. = Reset: Not set
....1. = Syn: Set
....0 = Fin: Not set
[TCP Flags: ··········S·]
Window size value: 8192
[Calculated window size: 8192]
Checksum: 0x74cf [unverified]
[Checksum Status: Unverified]
Urgent pointer: 0
Options: (12 bytes), Maximum segment size, No-Operation (NOP), Window scale, No-Operation (NOP), No-Operation (NOP), SACK permitted
0000 00 a0 c8 e8 16 2f 28 80 23 0b 6d c4 08 00 45 00 ·····/(· #·m···E·
0010 00 34 18 1c 40 00 80 06 00 00 c0 a8 01 02 d8 3a ·4··@··· ·······:
0020 da c3 ca d5 01 bb 4d 92 b9 1d 00 00 00 00 80 02 ······M· ········

TCP SYN ACK



Wireshark Class Example.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
(ip.addr eq 192.168.1.2 and ip.addr eq 216.58.218.195) and (tcp.port eq 51925 and tcp.port eq 443)
Expression...
No. Time Source Destination Protocol Length Info
167 10.867311 192.168.1.2 216.58.218.195 TCP 66 51925 → 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
171 10.892892 216.58.218.195 192.168.1.2 TCP 66 443 → 51925 [SYN, ACK] Seq=0 Ack=1 Win=42900 Len=0 MSS=1430 SACK_PERM=1 WS=128
172 10.893020 192.168.1.2 216.58.218.195 TCP 54 51925 → 443 [ACK] Seq=1 Ack=1 Win=65536 Len=0
Internet Protocol Version 4, Src: 216.58.218.195, Dst: 192.168.1.2
Transmission Control Protocol, Src Port: 443, Dst Port: 51925, Seq: 0, Ack: 1, Len: 0
Source Port: 443
Destination Port: 51925
[Stream index: 3]
[TCP Segment Len: 0]
Sequence number: 0 (relative sequence number)
[Next sequence number: 0 (relative sequence number)]
Acknowledgment number: 1 (relative ack number)
1000 .... = Header Length: 32 bytes (8)
Flags: 0x012 (SYN, ACK)
000. .... = Reserved: Not set
...0 .... = Nonce: Not set
.... 0... .... = Congestion Window Reduced (CWR): Not set
.... .0.. .... = ECN-Echo: Not set
.... ..0. .... = Urgent: Not set
....1 .... = Acknowledgment: Set
.... 0... = Push: Not set
.... .0.. = Reset: Not set
....1. = Syn: Set
....0 = Fin: Not set
[TCP Flags: ·······A··S·]
Window size value: 42900
[Calculated window size: 42900]
Checksum: 0x276b [unverified]
[Checksum Status: Unverified]
Urgent pointer: 0
Options: (12 bytes), Maximum segment size, No-Operation (NOP), No-Operation (NOP), SACK permitted, No-Operation (NOP), Window scale
0000 28 80 23 0b 6d c4 00 a0 c8 e8 16 2f 08 00 45 20 (·#·m··· ···/··E
0010 00 34 69 63 00 00 36 06 a6 98 d8 3a da c3 c0 a8 ·4ic··6· ···:····
0020 01 02 01 bb ca d5 0a 21 4e 14 4d 92 b9 1e 80 12 ·······! N·M·····

TCP ACK



Wireshark Class Example.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
(ip.addr eq 192.168.1.2 and ip.addr eq 216.58.218.195) and (tcp.port eq 51925 and tcp.port eq 443)
Expression...
No. Time Source Destination Protocol Length Info
167 10.867311 192.168.1.2 216.58.218.195 TCP 66 51925 → 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
171 10.892892 216.58.218.195 192.168.1.2 TCP 66 443 → 51925 [SYN, ACK] Seq=0 Ack=1 Win=42900 Len=0 MSS=1430 SACK_PERM=1 WS=128
172 10.893020 192.168.1.2 216.58.218.195 TCP 54 51925 → 443 [ACK] Seq=1 Ack=1 Win=65536 Len=0
Transmission Control Protocol, Src Port: 51925, Dst Port: 443, Seq: 1, Ack: 1, Len: 0
Source Port: 51925
Destination Port: 443
[Stream index: 3]
[TCP Segment Len: 0]
Sequence number: 1 (relative sequence number)
[Next sequence number: 1 (relative sequence number)]
Acknowledgment number: 1 (relative ack number)
0101 .... = Header Length: 20 bytes (5)
Flags: 0x010 (ACK)
000. .... = Reserved: Not set
...0 .... = Nonce: Not set
.... 0... .... = Congestion Window Reduced (CWR): Not set
.... .0.. .... = ECN-Echo: Not set
.... ..0. .... = Urgent: Not set
....1 .... = Acknowledgment: Set
.... 0... = Push: Not set
.... .0.. = Reset: Not set
....0. = Syn: Not set
....0 = Fin: Not set
[TCP Flags: ·······A····]
Window size value: 256
[Calculated window size: 65536]
[Window size scaling factor: 256]
Checksum: 0x74c3 [unverified]
[Checksum Status: Unverified]
Urgent pointer: 0
[SEQ/ACK analysis]
[This is an ACK to the segment in frame: 171]
[The RTT to ACK the segment was: 0.000128000 seconds]
0000 00 a0 c8 e8 16 2f 28 80 23 0b 6d c4 08 00 45 00 ·····/(· #·m···E·

What if the SYN Packet Gets Lost?

- Suppose the SYN packet gets lost
  - Packet is lost inside the network, or
  - Server rejects the packet (e.g., listen queue is full)

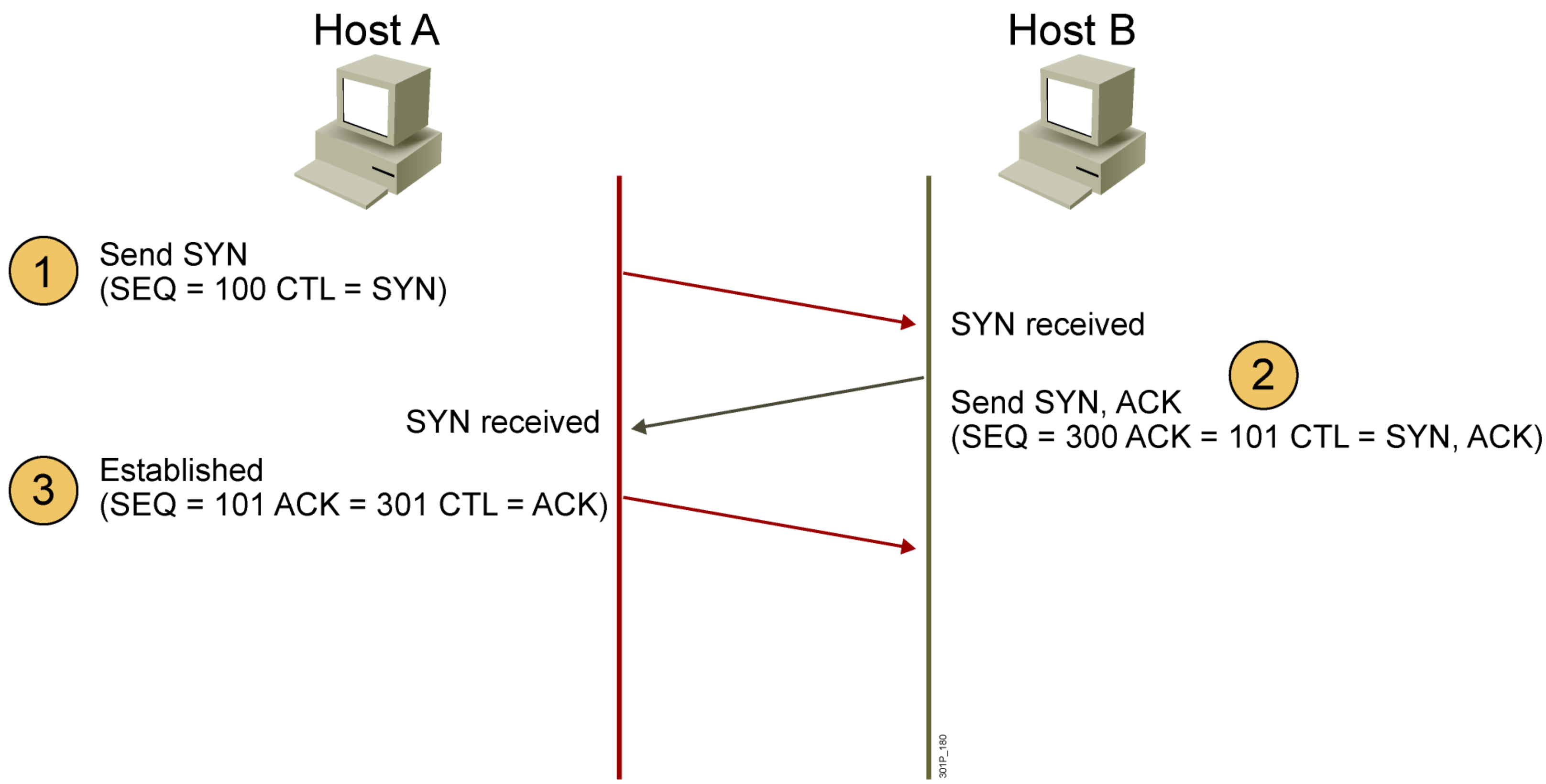
- Eventually, no SYN-ACK arrives
  - Sender sets a timer and wait for the SYN-ACK
    - ***and retransmits the SYN if needed***

- How should the TCP sender set the timer?
  - Sender has no idea how far away the receiver is
  - Some TCPs use a default of 3 or 6 seconds - depends

SYN Loss and Web Downloads

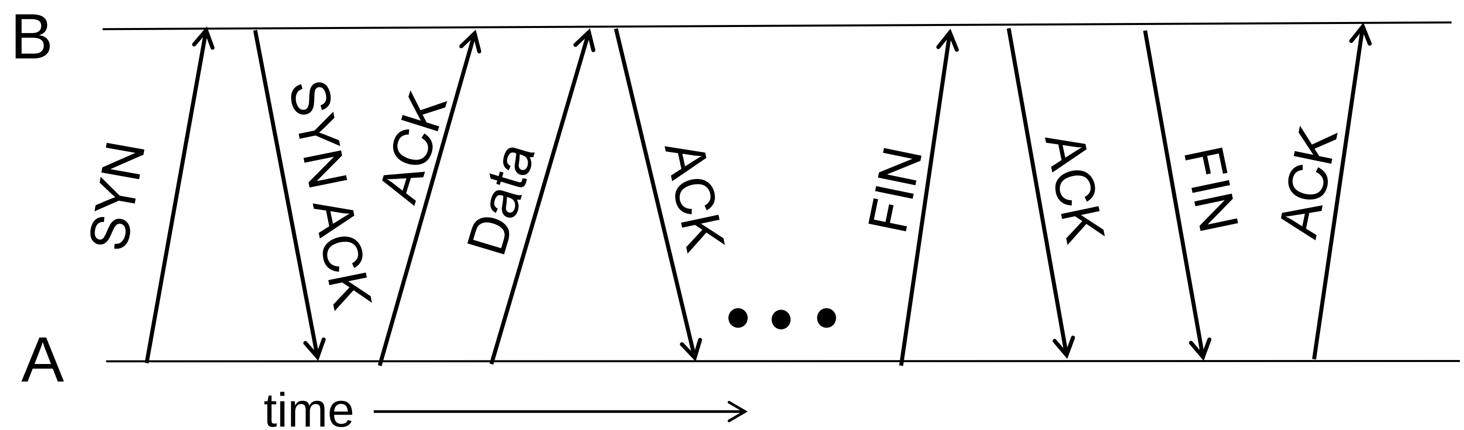
- User clicks on a hypertext link
  - Browser creates a socket and does a "connect"
  - The "connect" triggers the OS to transmit a SYN
- If the SYN is lost…
  - The 3-6 seconds of delay is very long
  - The impatient user may click "reload"
- User triggers an "abort" of the "connect"
  - Browser "connects" on a new socket
  - Essentially, forces a fast send of a new SYN!

Three-Way Handshake



CTL = Which control bits in the TCP header are set to 1

Tearing Down the Connection (FIN)



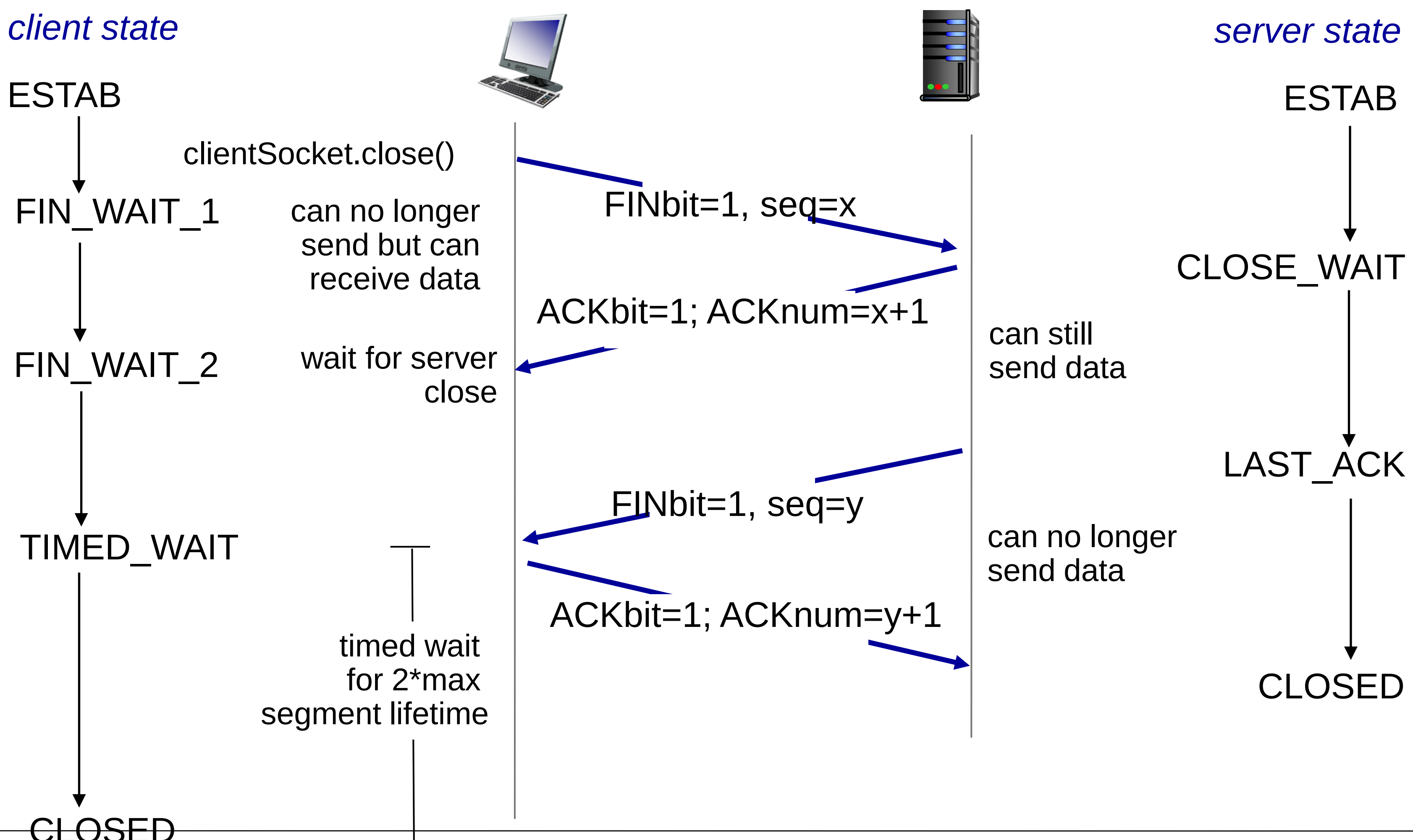
- Closing (each end of) the connection
  - Finish (FIN) to close and receive remaining bytes
  - And other host sends a FIN ACK to acknowledge
  - Reset (RST) to close and not receive remaining bytes

Sending/Receiving the FIN Packet

- Sending a FIN: close()
  - Process is done sending data via the socket
  - Process invokes "close()" to close the socket
  - Once TCP has sent all the outstanding bytes…
    - ***then TCP sends a FIN***

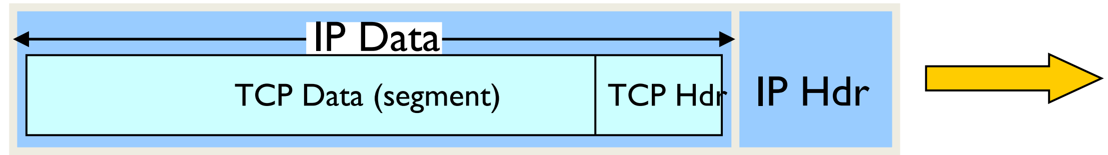
- Receiving a FIN: EOF
  - Process is reading data from the socket
  - Eventually, the attempt to read returns an EOF

TCP: closing a connection



Transmitting Data

TCP Segment



- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
    - ***E.g., up to 1500 bytes on an Ethernet & PPP***
    - ***“largest link layer frame”***

- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header is *typically* 20 bytes long

- TCP segment
  - No more than Maximum Segment Size (MSS) bytes
    - ***E.g., up to 1460 consecutive bytes from the stream***
      - TCP segment in IP datagram + TCP header (40 bytes) into single link layer frame = typically 1460

- Find MTU and make sure MSS fits into it

TCP Header

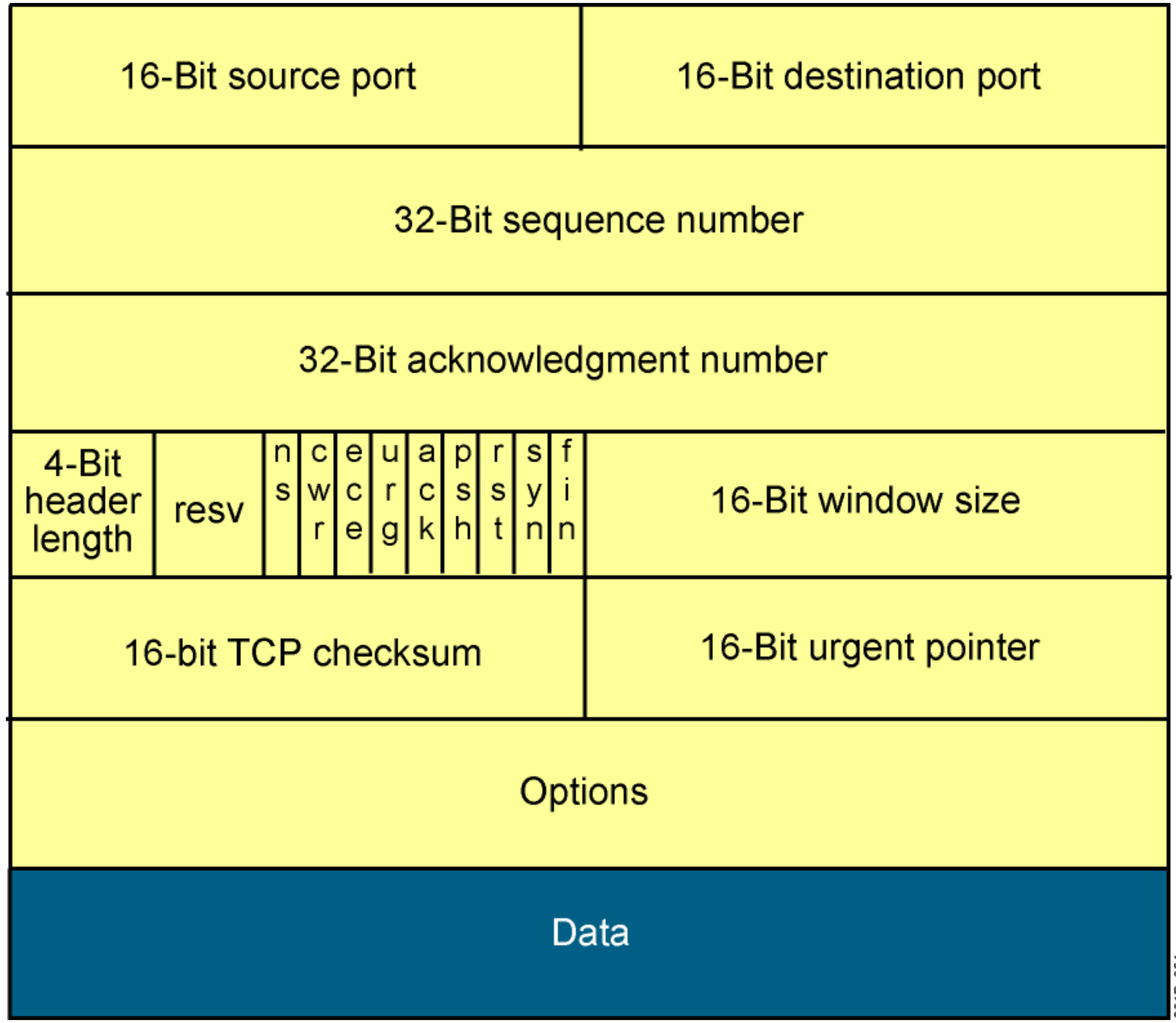
URG: urgent data (generally not used)

ACK: ACK # valid

PSH: push data now (generally not used)

RST, SYN, FIN: connection estab (setup, teardown commands)

Internet checksum (as in UDP)



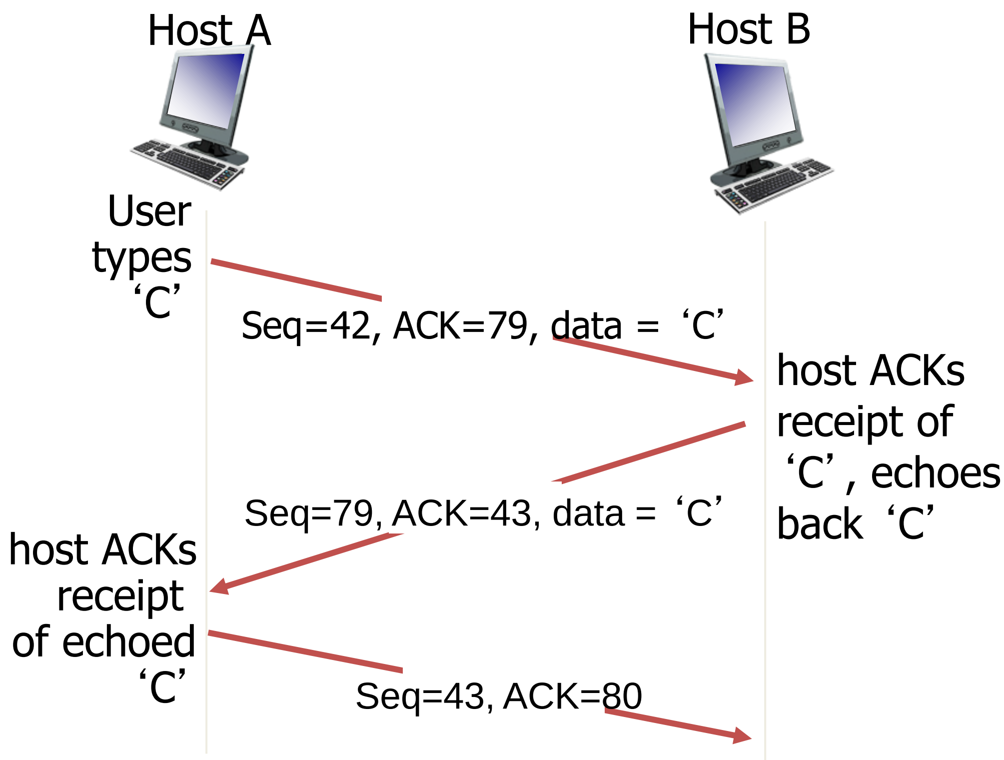
Initial Sequence Number (ISN)

- Sequence number for the very first byte
  - Why random?
    - ***E.g., Why not a de facto ISN of 0?***

- Practical issue: reuse of port numbers
  - Port numbers must (eventually) get used again
    - ***an old packet may still be in flight***
    - ***associated with the new connection***

- TCP must change the ISN over time
  - Set from a 32-bit clock that ticks every 4 microsec
    - ***which wraps around once every 4.55 hours***

TCP Seq. Numbers, ACKs



simple telnet scenario

TCP Seq. Numbers, ACKs

Sequence numbers:

–byte stream “number” of first byte in segment’s data

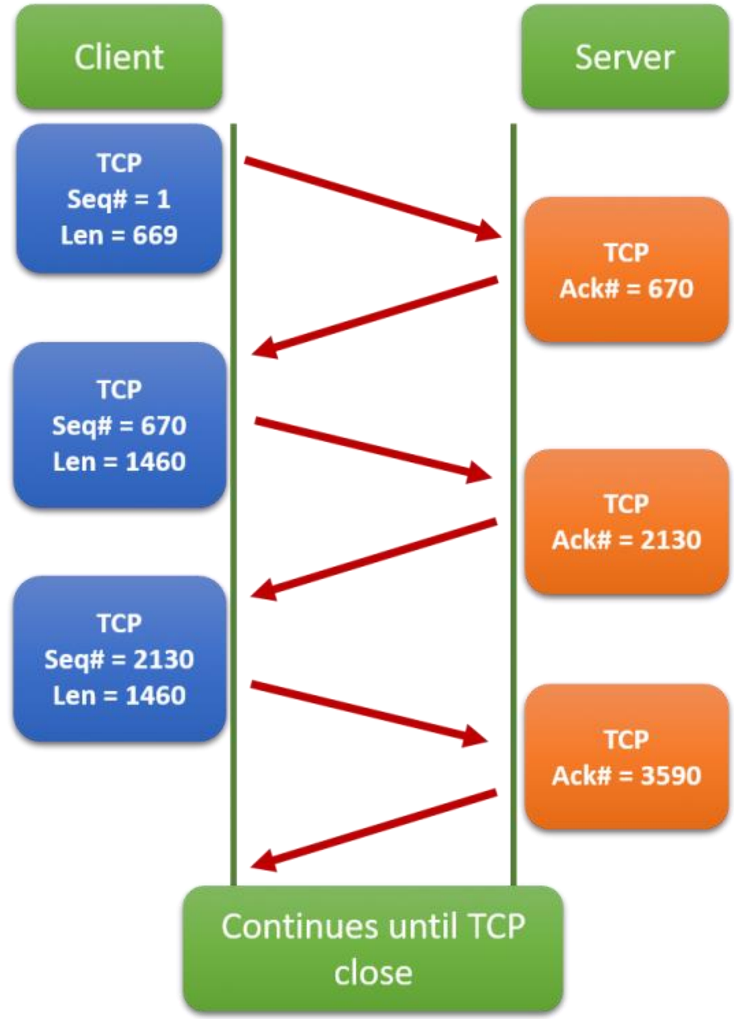
Acknowledgements:

–seq # of next byte expected from other side

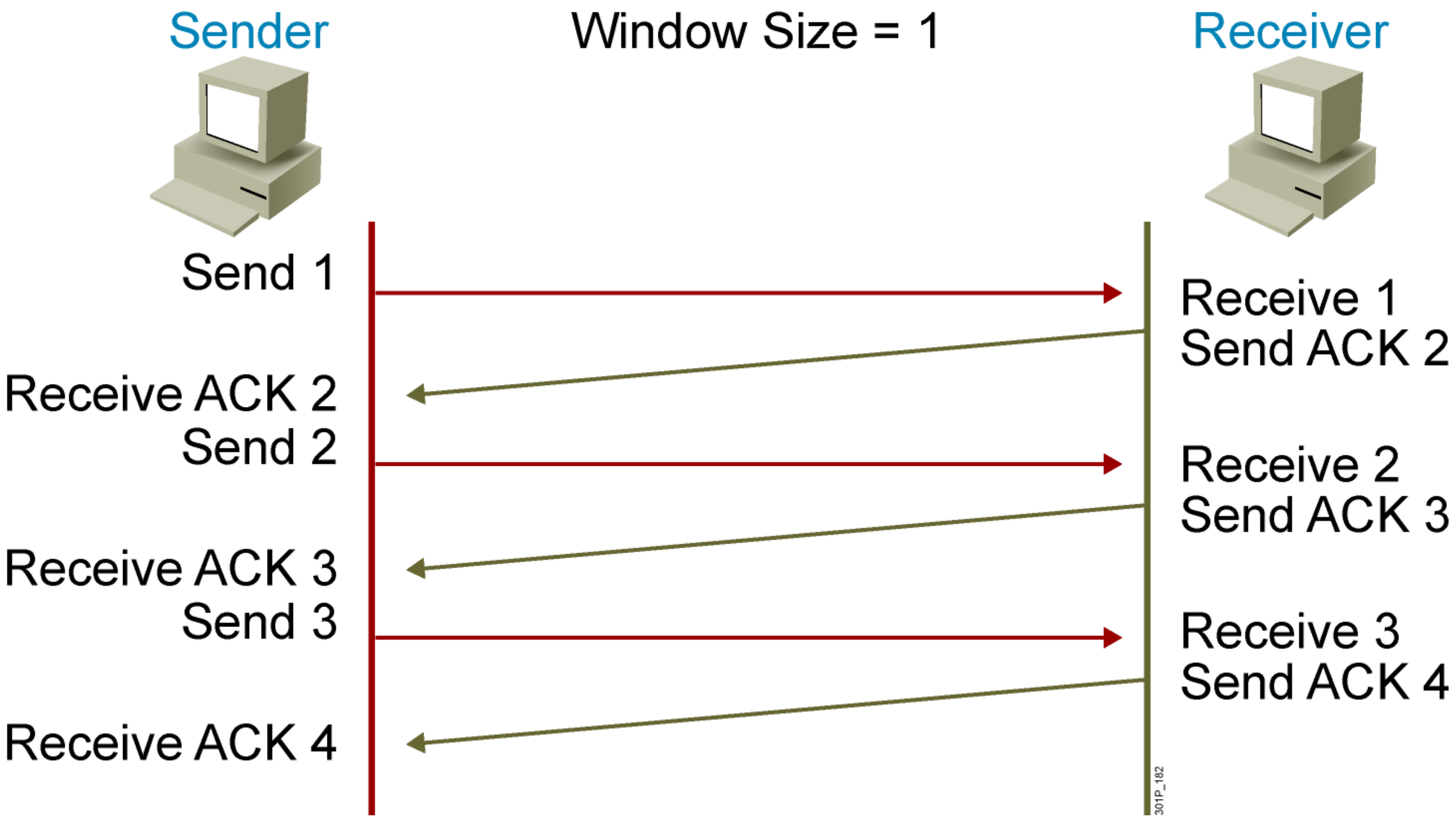
–cumulative ACK

Q: How does receiver handle out-of-order segments?

–A: TCP spec doesn’t say, - up to implementer

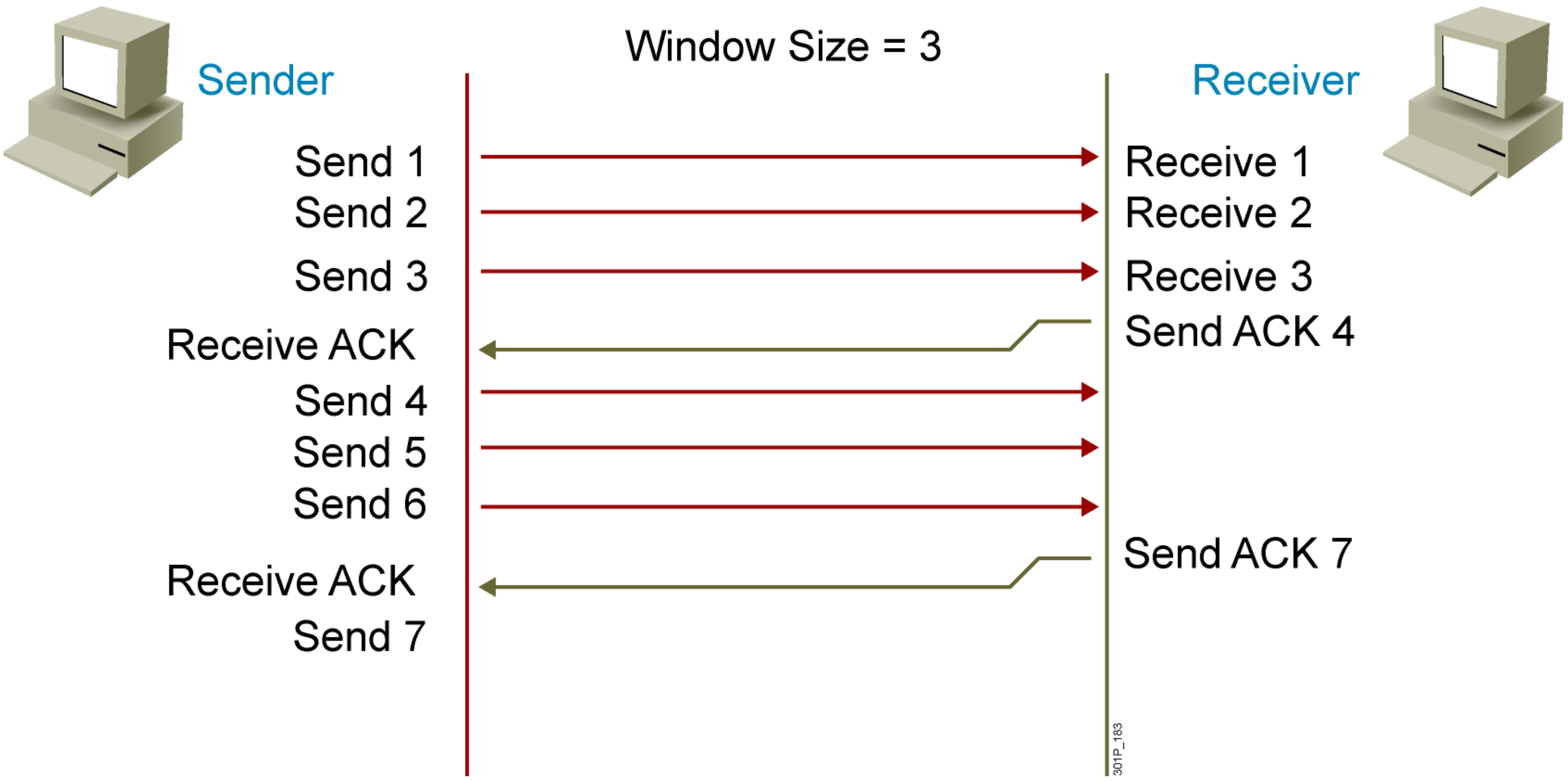


TCP Acknowledgment

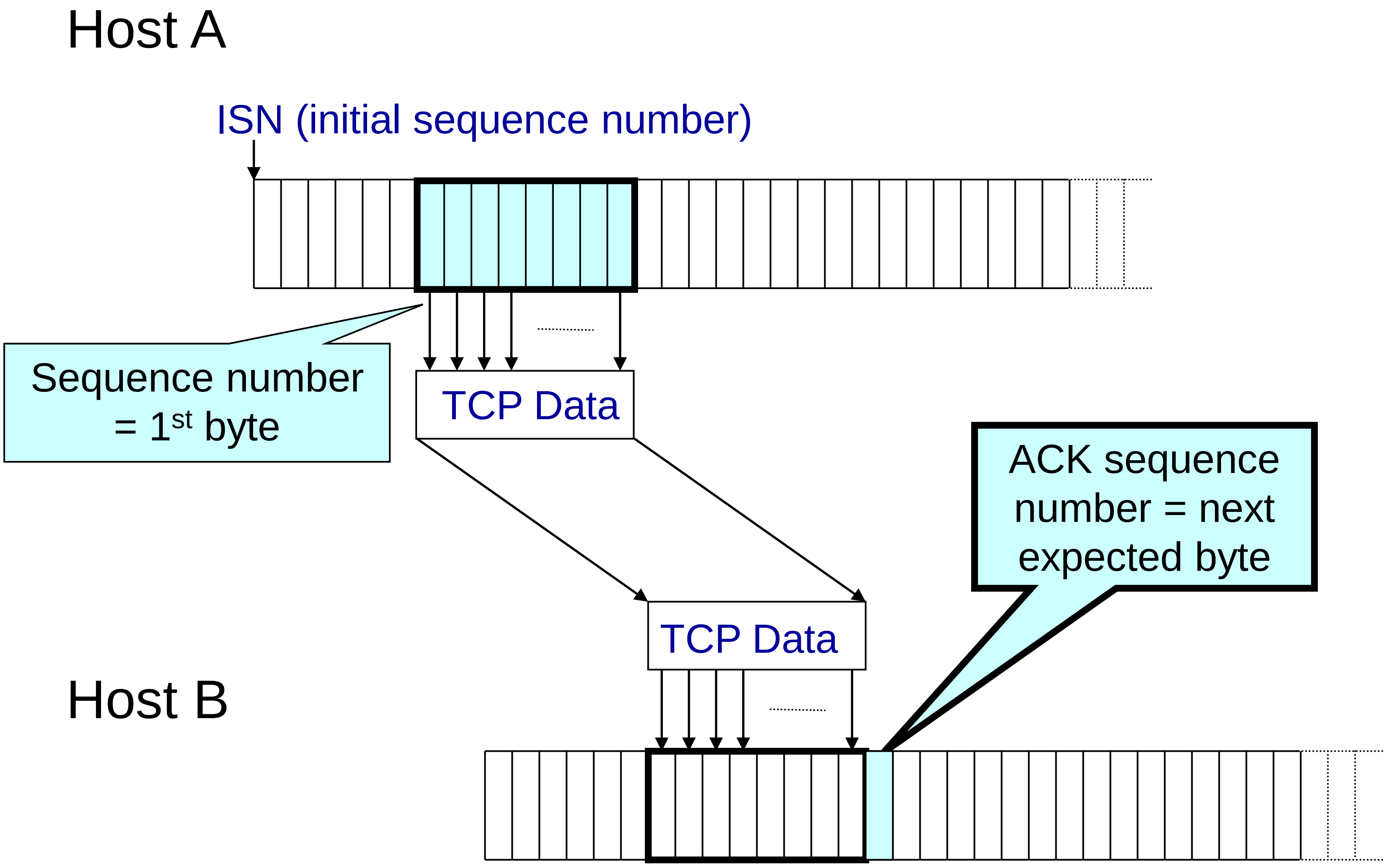


University of Colorado
Boulder

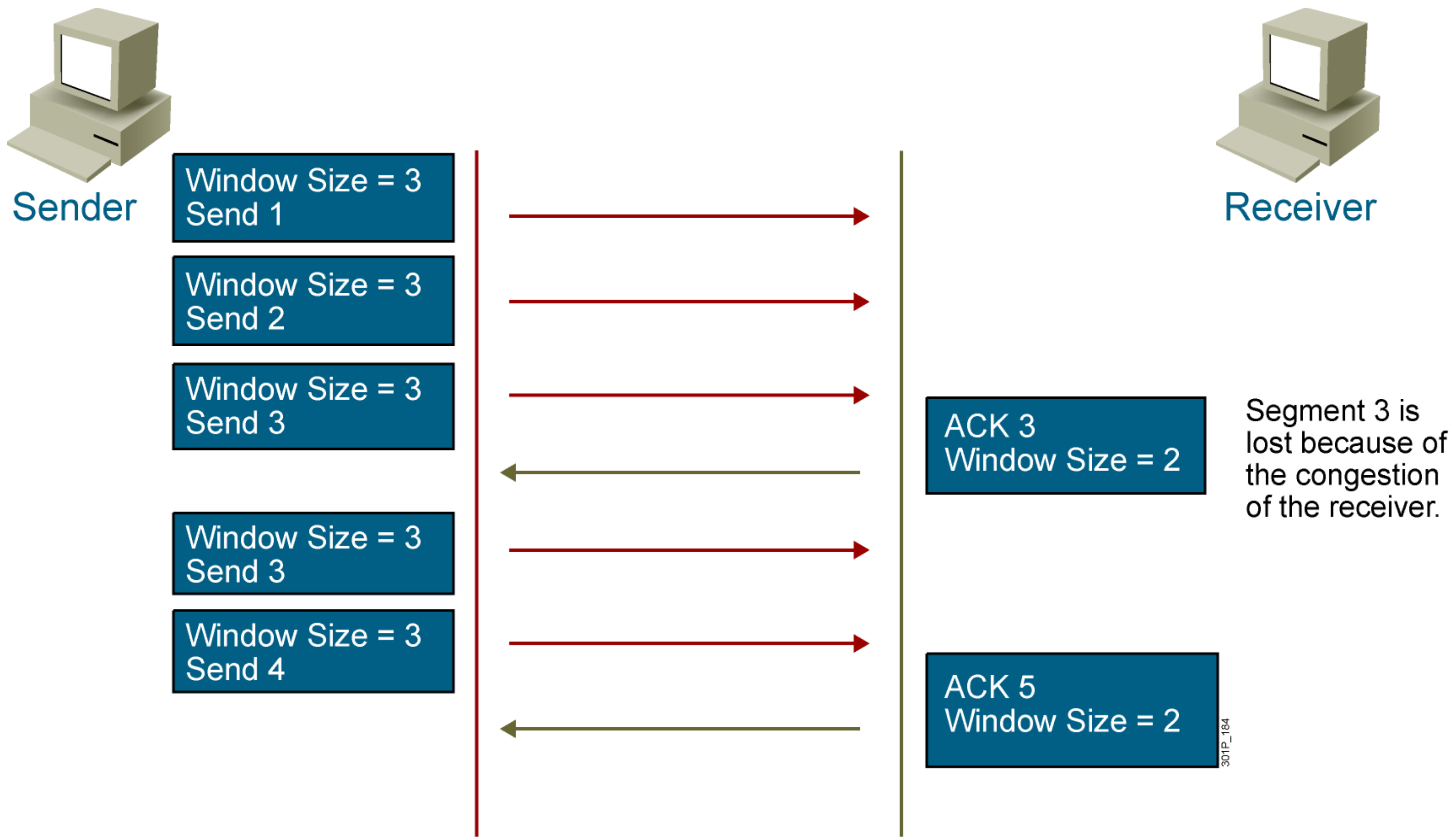
Fixed Windowing



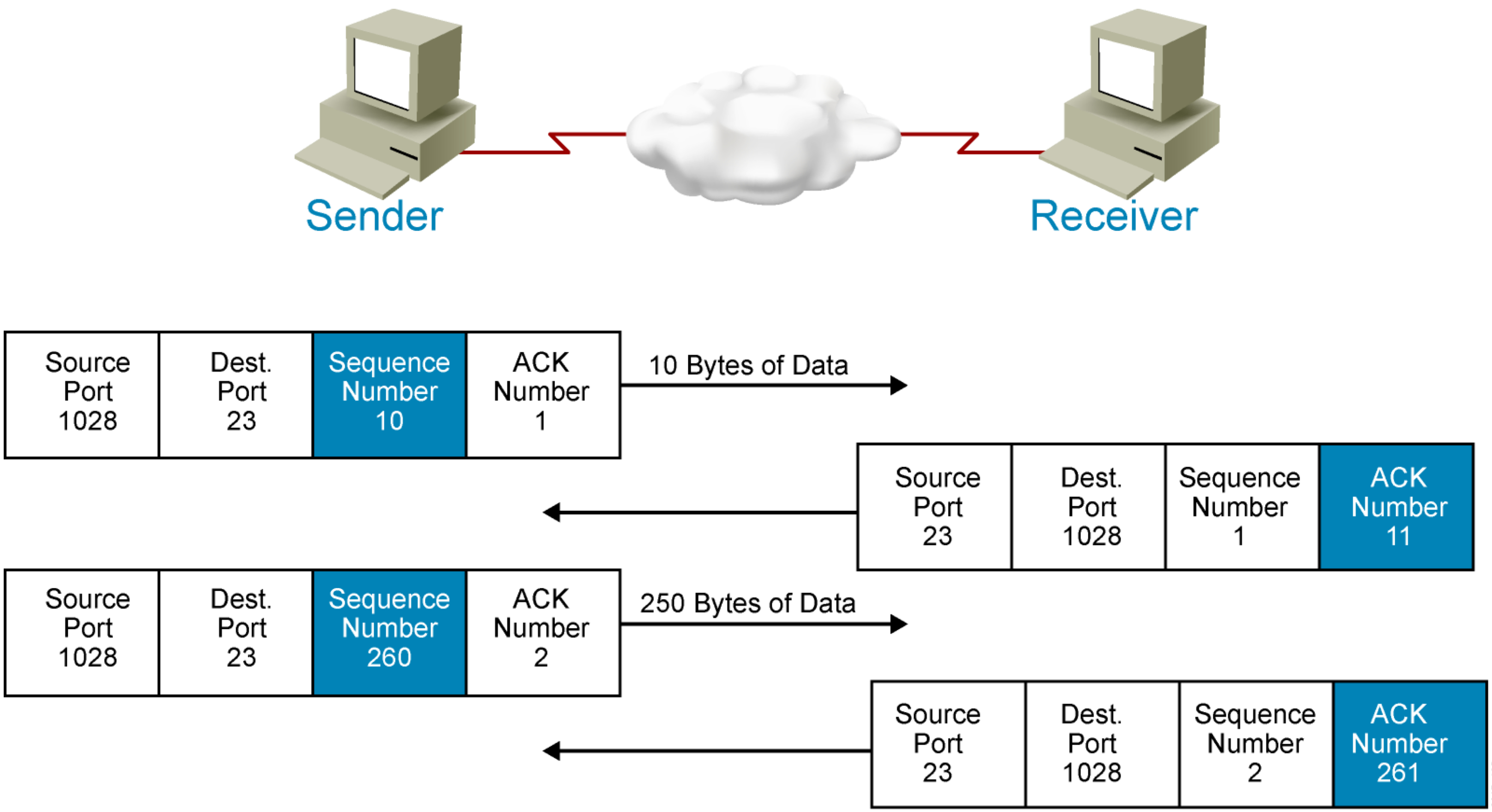
TCP Acknowledgments



TCP Sliding Windowing



TCP Sequence and Acknowledgment Numbers



Challenges of Reliable Data Transfer

- **Over a perfectly reliable channel**
  – Easy: sender sends and receiver receives
- **Over a channel with bit errors**
  – Receiver detects errors and requests retransmission
- **Over a lossy channel with bit errors**
  – Some data are missing, and others corrupted
  – Receiver cannot always detect loss
- **Over a channel that may reorder packets**
  – Receiver cannot distinguish loss from out-of-order

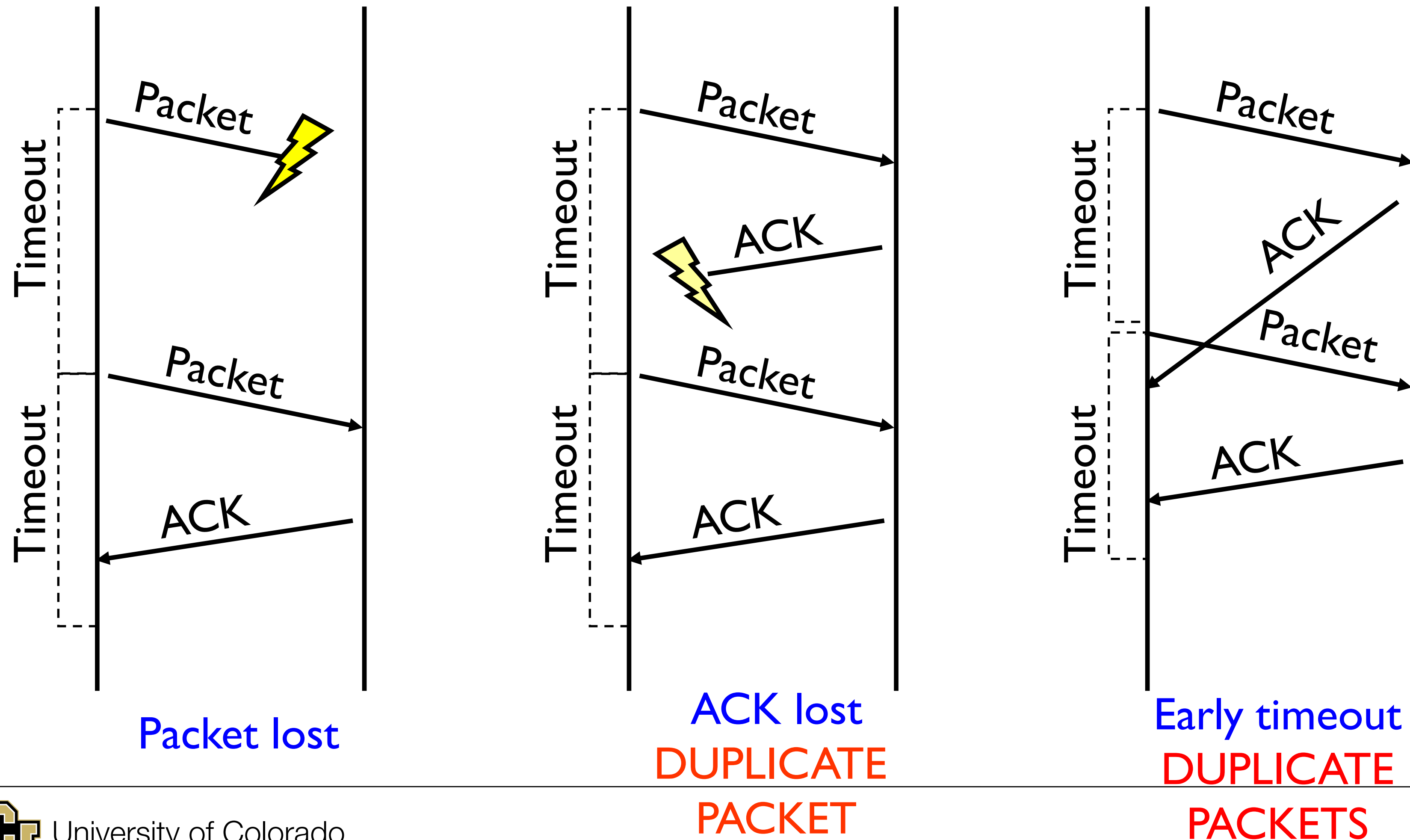
TCP Support for Reliable Delivery

- **Detect bit errors:** checksum
  - Used to detect corrupted data at the receiver
    - ***leading the receiver to drop the packet***

- **Detect missing data:** sequence number
  - Used to detect a gap in the stream of bytes
    - ***and for putting the data back in order***

- **Recover from lost data:** retransmission
  - Sender retransmits lost or corrupted data
  - Two main ways to detect lost packets

Reasons for Retransmission



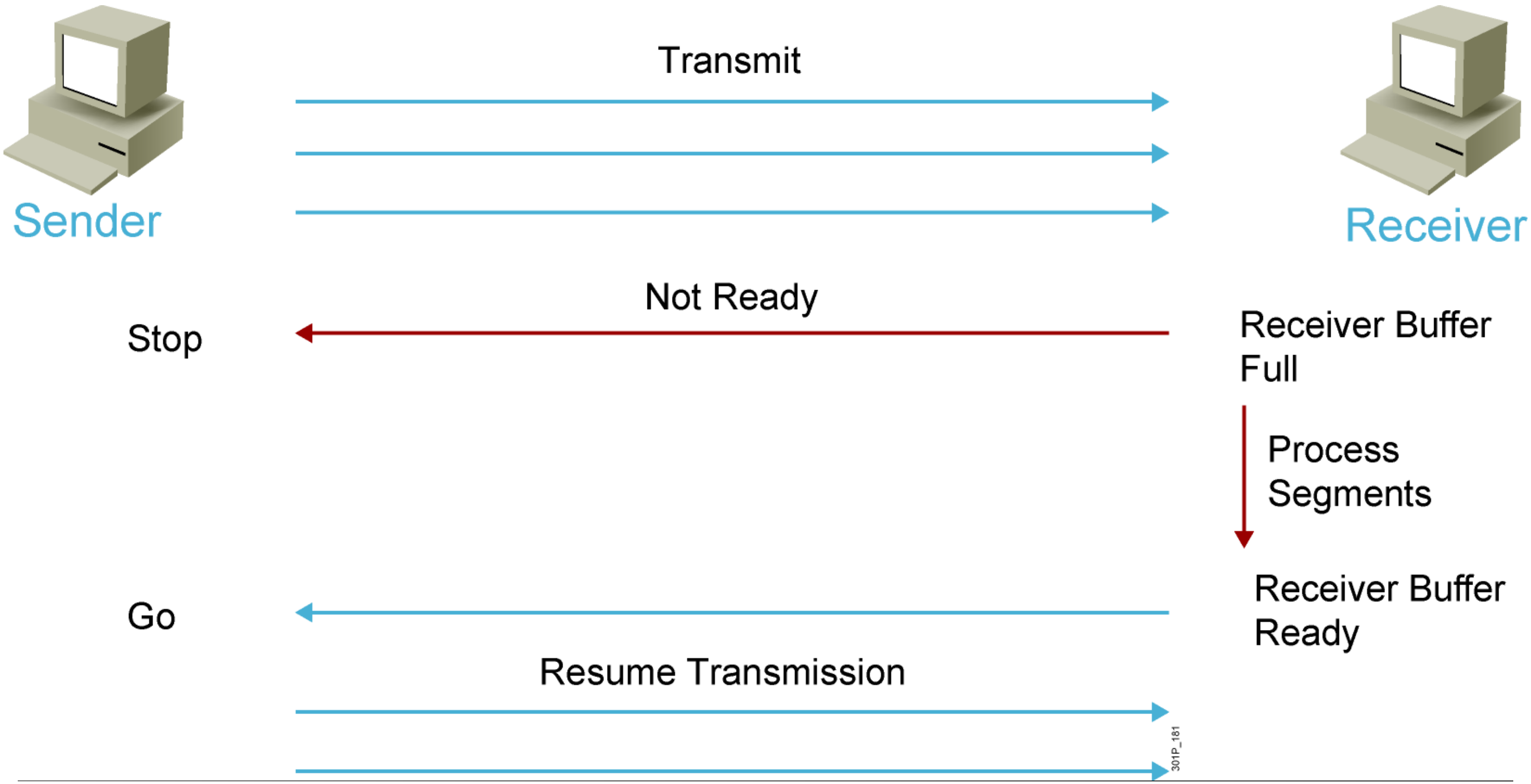
How Long Should Sender Wait?

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

- TCP sets timeout as a function of the RTT
  - Expect ACK to arrive after a “round-trip time”
    - ***plus a fudge factor to account for queuing***

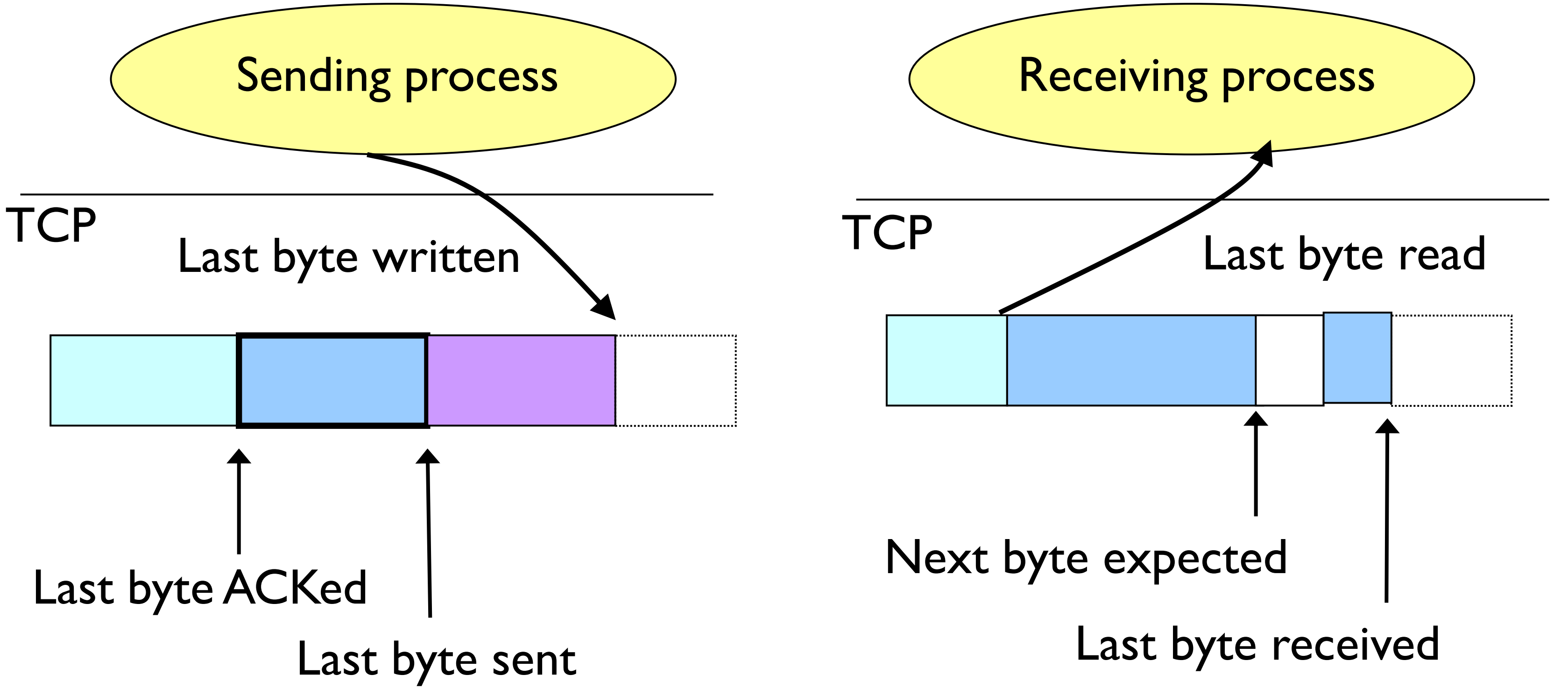
- But, how does the sender know the RTT?
  - Running average of delay to receive an ACK

Flow Control

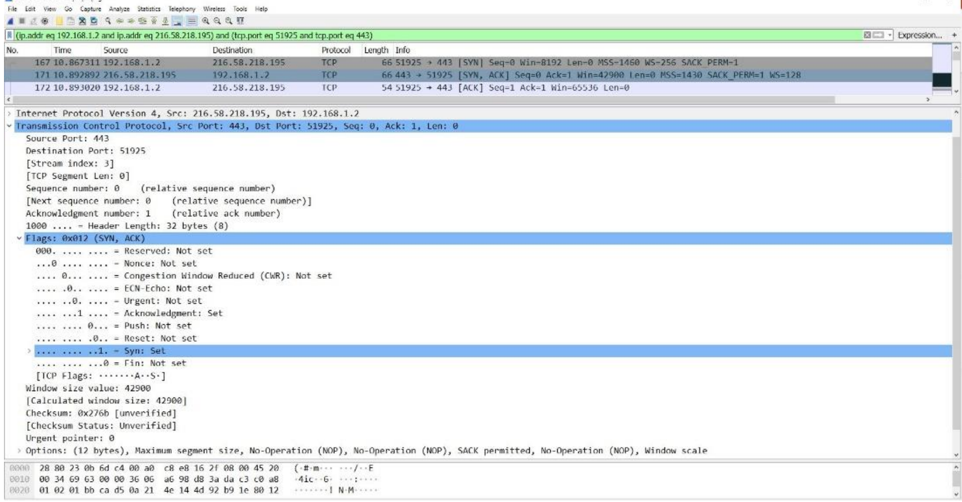


Sliding Window

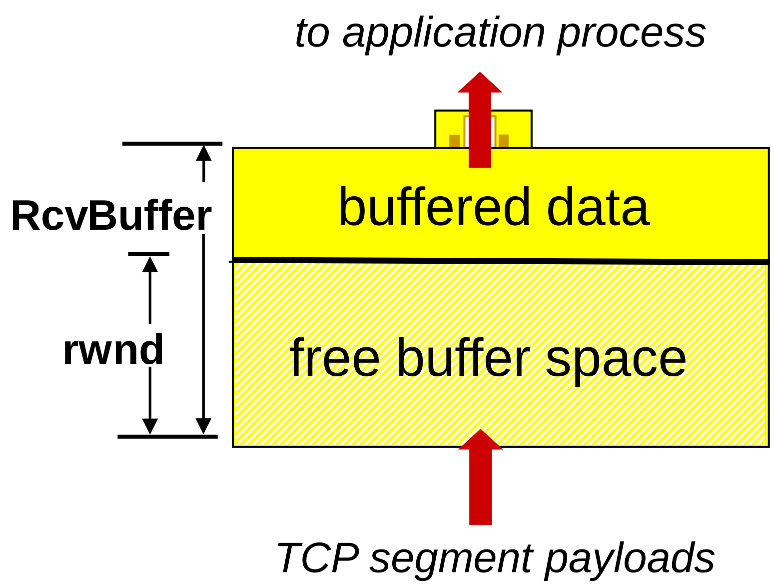
- Allow a larger amount of data "in flight"
  - Allow sender to get ahead of the receiver
    - ***though not too far ahead***



TCP Flow Control



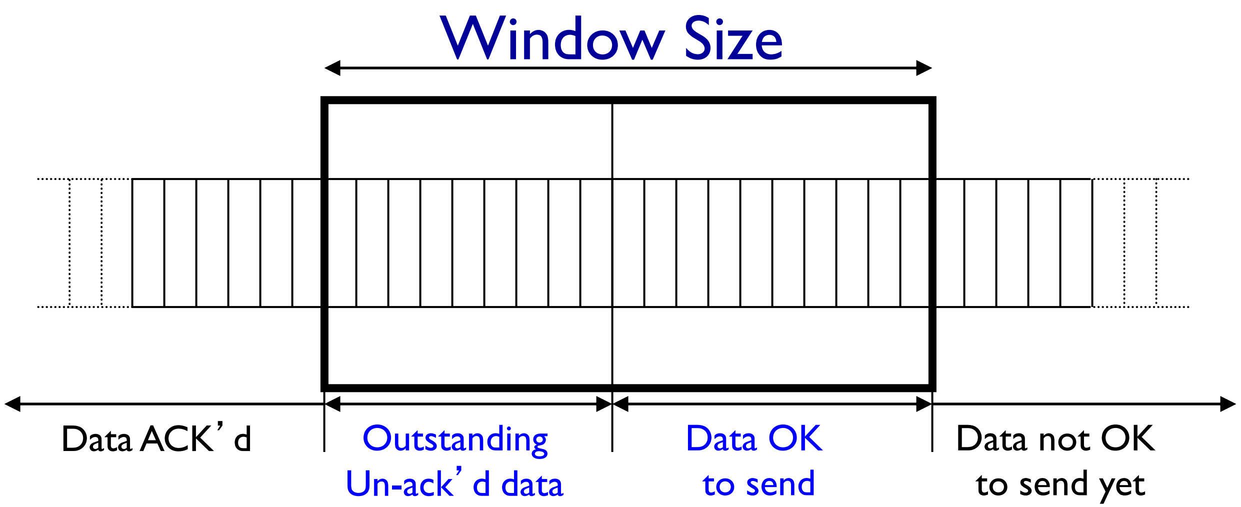
- Receiver "advertises" free buffer space by including **rwnd** (receiver window) value in TCP header of receiver-to-sender segments
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- Sender limits amount of unACKed ("in-flight") data to receiver's **rwnd** value
- Guarantees receive buffer will not overflow



receiver-side buffering

Flow Control – Receiver Buffer

- Receive window size
  - Amount that can be sent without ACK
  - Receiver must be able to store this amount of data
- Receiver tells the sender the window
  - Tells the sender the amount of free space left



Receiver Window vs. Congestion Window

- **Flow control**
  - Keep a *fast sender* from overwhelming *a slow receiver*
- **Congestion control**
  - Keep a *set of senders* from overloading the *network*
- **Different concepts, but similar mechanisms**
  - TCP flow control: receiver window
  - TCP congestion control: congestion window
  - Sender TCP window =

    min { congestion window, receiver window }

TCP Congestion Control

Principles of Congestion Control

Congestion:

- "Too many sources sending too much data too fast for the *network* to handle"

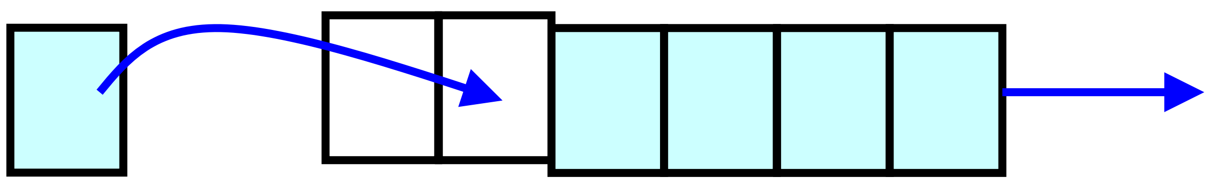
- Different from flow control!

- Manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)

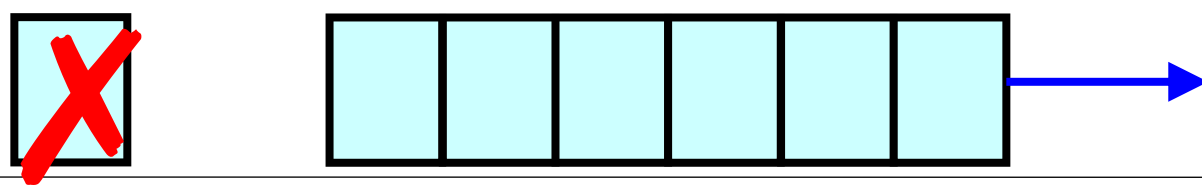
- Top-10 problem!

Where Congestion Happens: Links

- Simple resource allocation: FIFO queue & drop-tail
- Access to the bandwidth: first-in first-out queue
  - Packets transmitted in the order they arrive

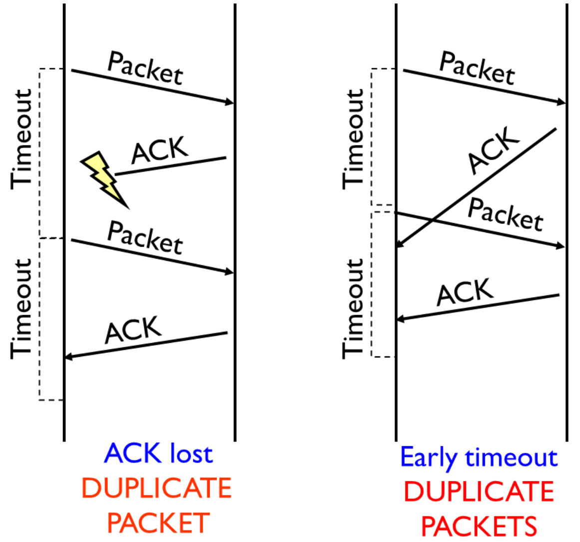


- Access to the buffer space: drop-tail queuing
  - If the queue is full, drop the incoming packet



How Congestion Looks to the End Host

- Delay: Packet experiences high delay
- Loss: Packet gets dropped along path
- How does TCP sender learn this?
  – Delay: Round-trip time estimate
  – Loss: Timeout and/or duplicate acknowledgments



TCP Congestion Window

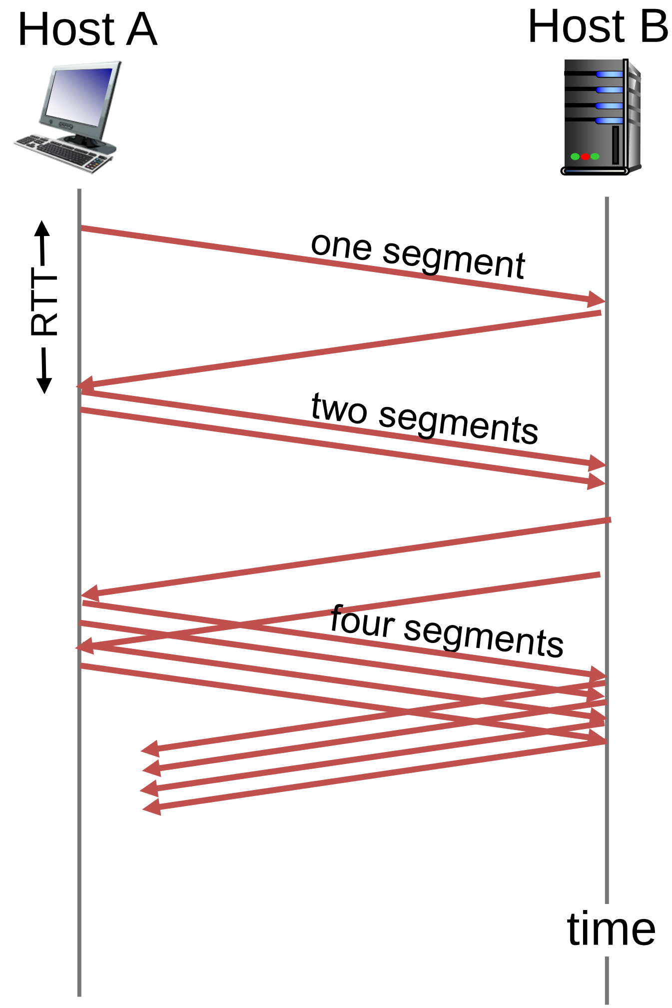
- Each TCP sender maintains a congestion window
  - Max number of bytes to have in transit (not yet ACK'd)

- Adapting the congestion window
  - Decrease upon losing a packet: backing off
  - Increase upon success: optimistically exploring
  - Always struggling to find right transfer rate

- Tradeoff
  - Pro: avoids needing explicit network feedback
  - Con: continually under- and over-shoots "right" rate

TCP Slow Start

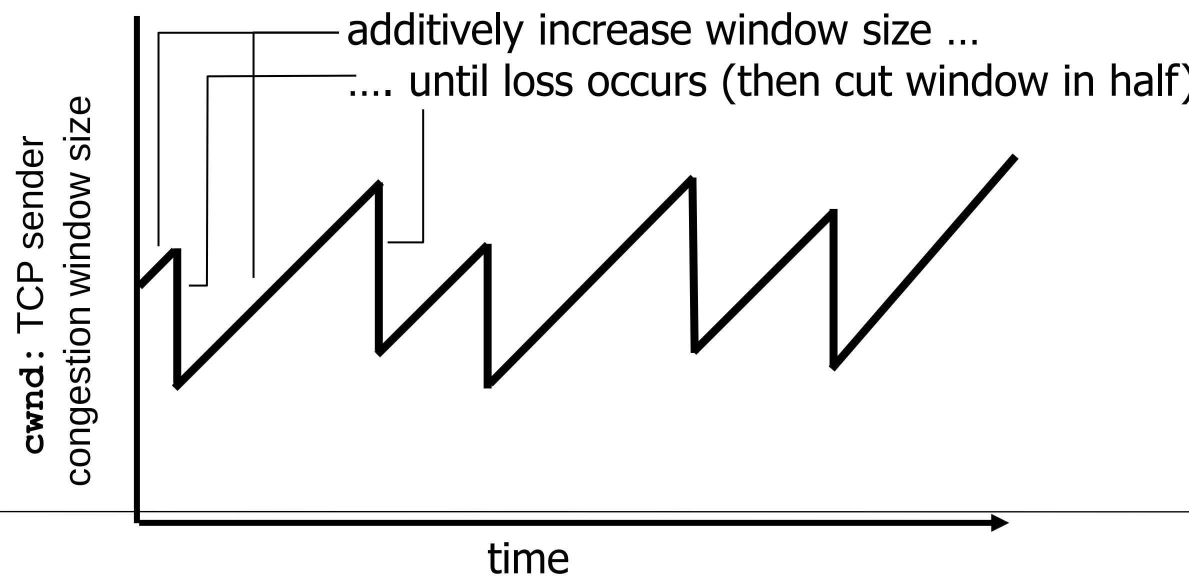
- When connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received
- <u>*Summary:*</u> initial rate is slow but ramps up exponentially fast



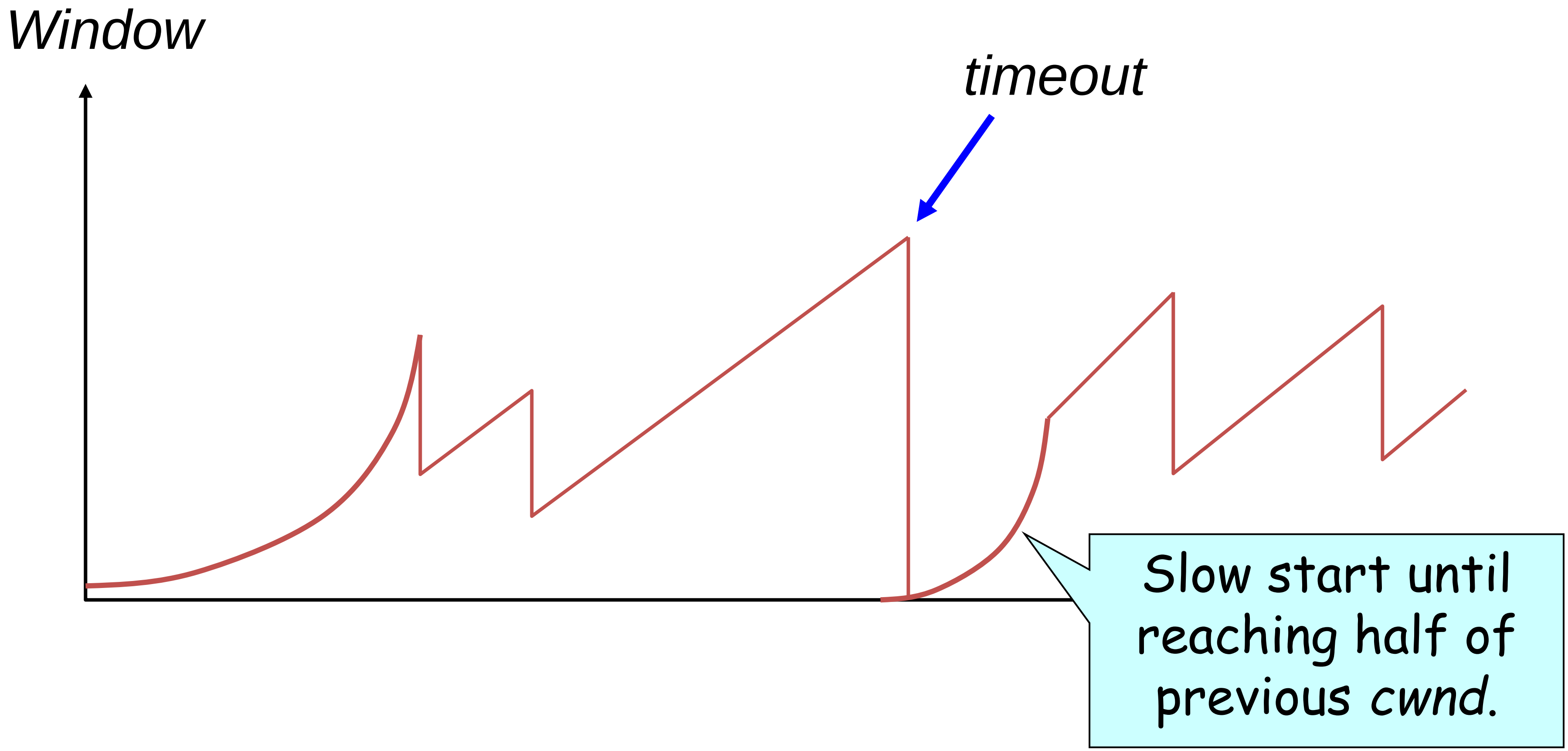
Additive Increase Multiplicative Decrease (AIMD)

- *Approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *Additive increase:* increase **cwnd** (congestion window) by 1 MSS every RTT until loss detected
  - *Multiplicative decrease*: cut **cwnd** in half after loss

AIMD saw tooth behavior: probing for bandwidth



Repeating Slow Start After Timeout



- Slow-start restart: Go back to CWND of 1, but take advantage of knowing the previous value of CWND.

Receiver Window vs. Congestion Window

- **Flow control**
  - Keep a *fast sender* from overwhelming *a slow receiver*
- **Congestion control**
  - Keep a *set of senders* from overloading the *network*
- **Different concepts, but similar mechanisms**
  - TCP flow control: receiver window
  - TCP congestion control: congestion window
  - Sender TCP window =

    min { congestion window, receiver window }

Sources of poor TCP performance

- The below conditions *may* result in:
  - (A) Higher packet latency
  - (B) Greater loss
  - (C) Lower throughput

1. Larger buffers in routers = A
2. Smaller buffers in routers = B
3. Smaller buffers on end-hosts = C
4. Slow application receivers = C

Questions?