Διαδικτυακά Πρωτόκολλα Μεταφοράς (Transport Protocols)

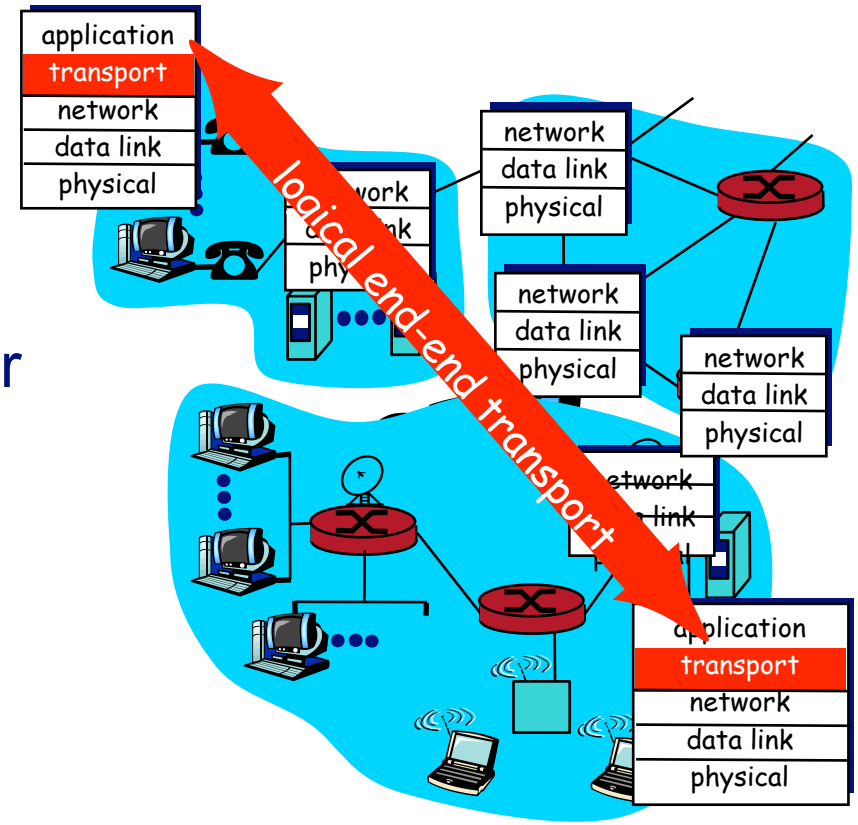
Το πρωτόκολλο TCP

Role of Transport Layer

- Application layer
  - Communication for specific applications
  - E.g., HyperText Transfer Protocol (HTTP), File Transfer Protocol (FTP), Network News Transfer Protocol (NNTP)
- Transport layer
  - Communication between processes (e.g., socket)
  - Relies on network layer and serves the application layer
  - E.g., TCP and UDP
- Network layer
  - Logical communication between nodes
  - Hides details of the link technology
  - E.g., IP

Transport Protocols

- Provide *logical communication* between application processes running on different hosts
- Run on end hosts
  - Sender: breaks application messages into segments, and passes to network layer
  - Receiver: reassembles segments into messages, passes to application layer
- Multiple transport protocol available to applications
  - Internet: TCP and UDP

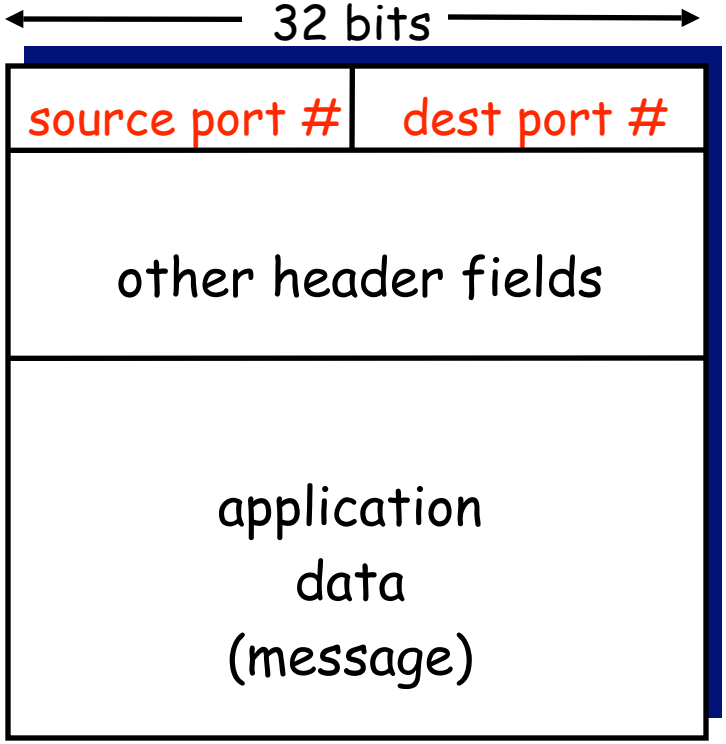


Internet Transport Protocols

- Datagram messaging service (UDP)
  - No-frills extension of “best-effort” IP
- Reliable, in-order delivery (TCP)
  - Connection set-up
  - Discarding of corrupted packets
  - Retransmission of lost packets
  - Flow control
  - Congestion control
- Other services not available
  - Delay guarantees
  - Bandwidth guarantees

Multiplexing and Demultiplexing

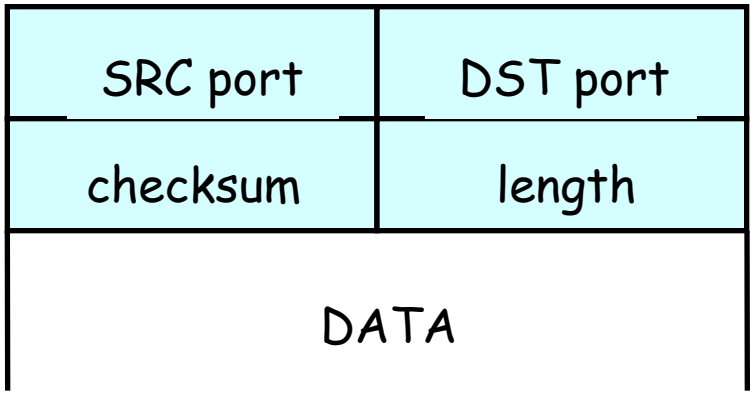
- Host receives IP datagrams
  - Each datagram has source and destination IP address,
  - Each datagram carries one transport-layer segment
  - Each segment has source and destination port number
- Host uses IP addresses and port numbers to direct the segment to appropriate socket



TCP/UDP segment format

Unreliable Message Delivery Service

- Lightweight communication between processes
  - Avoid overhead and delays of ordered, reliable delivery
  - Send messages to and receive them from a socket
- User Datagram Protocol (UDP)
  - IP plus port numbers to support (de)multiplexing
  - Optional error checking on the packet contents



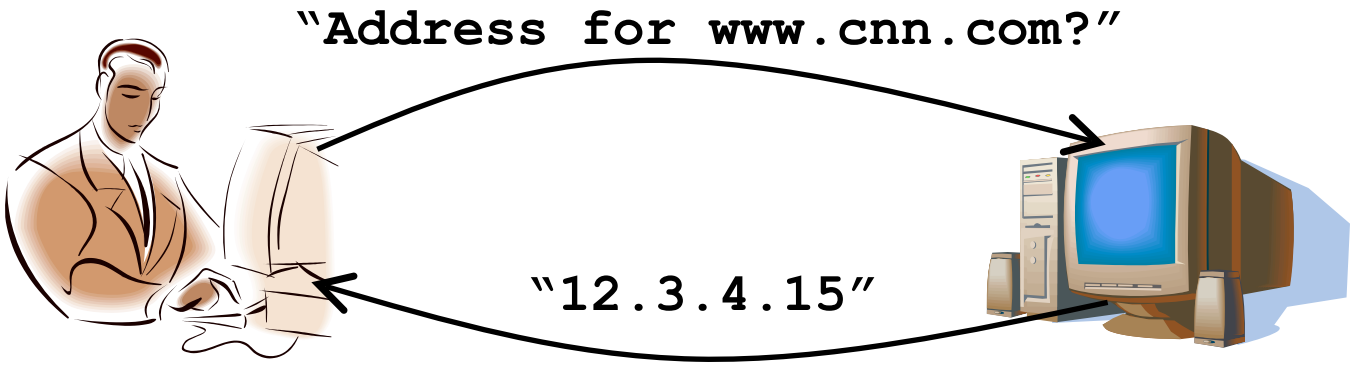
Why Would Anyone Use UDP?

- Finer control over what data is sent and when
  - As soon as an application process writes into the socket
  - … UDP will package the data and send the packet
- No delay for connection establishment
  - UDP just blasts away without any formal preliminaries
  - … which avoids introducing any unnecessary delays
- No connection state
  - No allocation of buffers, parameters, sequence #s, etc.
  - … making it easier to handle many active clients at once
- Small packet header overhead
  - UDP header is only eight-bytes long

Popular Applications That Use UDP

- Multimedia streaming
  - Retransmitting lost/corrupted packets is not worthwhile
  - By the time the packet is retransmitted, it's too late
  - E.g., telephone calls, video conferencing, gaming

- Simple query protocols like Domain Name System
  - Overhead of connection establishment is overkill
  - Easier to have application retransmit if needed



Transmission Control Protocol (TCP)

- Connection oriented
  - Explicit set-up and tear-down of TCP session
  - State maintained at both hosts
- Stream-of-bytes service
  - Sends and receives a stream of bytes, not messages
  - Division of data into datagrams, headers etc are invisible to the application above
- Reliable, in-order delivery
  - Checksums to detect corrupted data
  - Acknowledgments & retransmissions for reliable delivery
  - Sequence numbers to detect losses and reorder data
  - Adapt to network congestion for the greater good
- Full duplex transfer
  - Allows data transfer between two applications in both directions at the same time

Transmission Control Protocol (TCP)

- Flow control
  - Prevent overflow of the receiver's buffer space
- Congestion control
  - Regulates the rate at which the network can transfer the data

Challenges of Reliable Data Transfer

- Over a perfectly reliable channel
  - All of the data arrives in order, just as it was sent
  - Simple: sender sends data, and receiver receives data
- Over a channel with bit errors
  - All of the data arrives in order, but some bits corrupted
  - Receiver detects errors and says “please repeat that”
  - Sender retransmits the data that were corrupted
- Over a lossy channel with bit errors
  - Some data are missing, and some bits are corrupted
  - Receiver detects errors but cannot always detect loss
  - Sender must wait for acknowledgment (“ACK” or “OK”)
  - … and retransmit data after some time if no ACK arrives

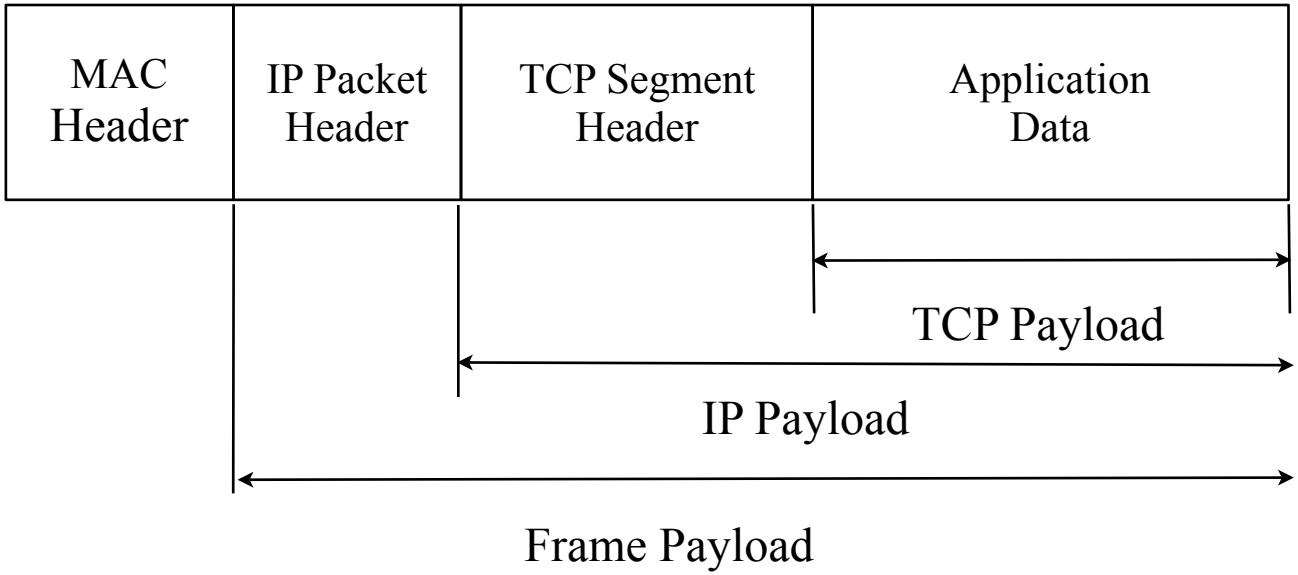
TCP Support for Reliable Delivery

- Checksum
  - Used to detect corrupted data at the receiver
  - …leading the receiver to drop the packet
- Sequence numbers
  - Used to detect missing data
  - ... and for putting the data back in order
- Retransmission
  - Sender retransmits lost or corrupted data
  - Timeout based on estimates of round-trip time
  - Fast retransmit algorithm for rapid retransmission

TCP Segments

TCP Segments

- TCP breaks the data stream into *segments*, which the network layer encapsulates into IP datagrams.
- TCP segment header supports: connection management, reliable delivery, flow control

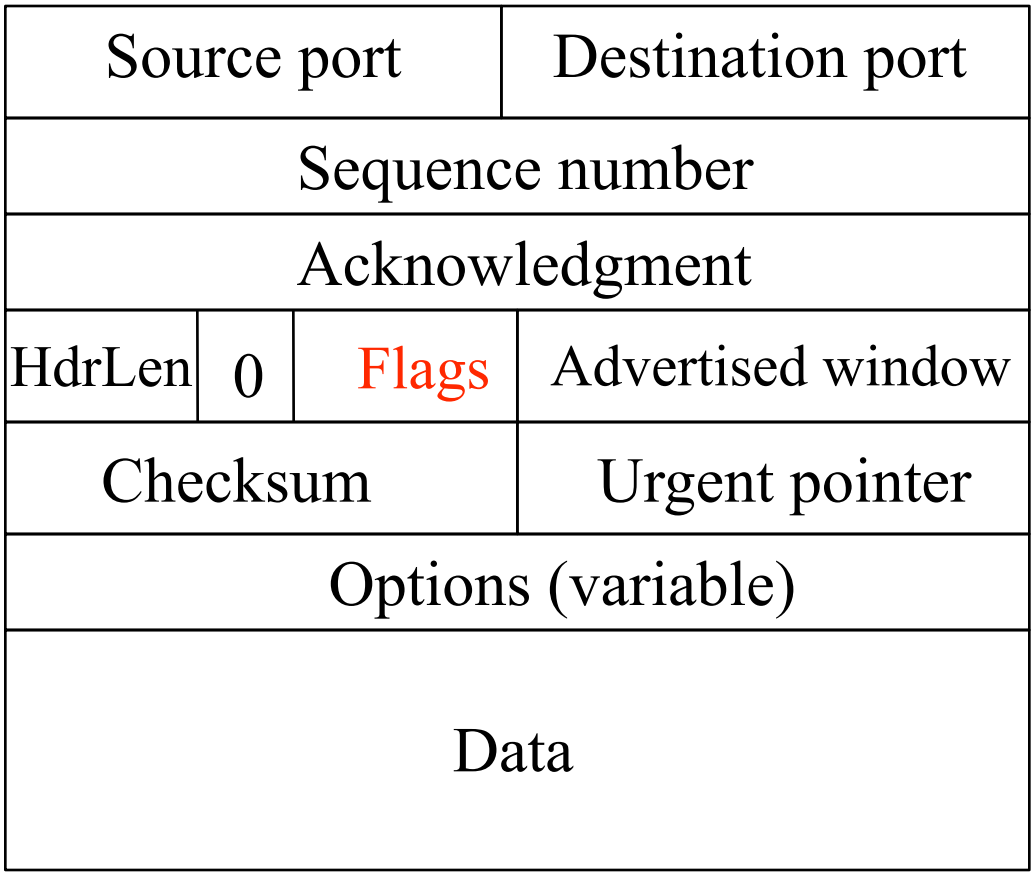


TCP Header Fields

- Source and Destination port numbers: for multiplexing/ de-multiplexing at the hosts
- Sequence number: the offset of a segment in a byte stream (the sequence number of the first segment byte in the byte stream)
- Acknowledgment number: confirms that the sender of the segment has received all bytes up to this number from the other host (valid when the ACK bit is set)
- Code bits field (flags): deal with connection management and urgency of the content of the segment (SYN, FIN, RST, ACK)
- Window field: used for flow control

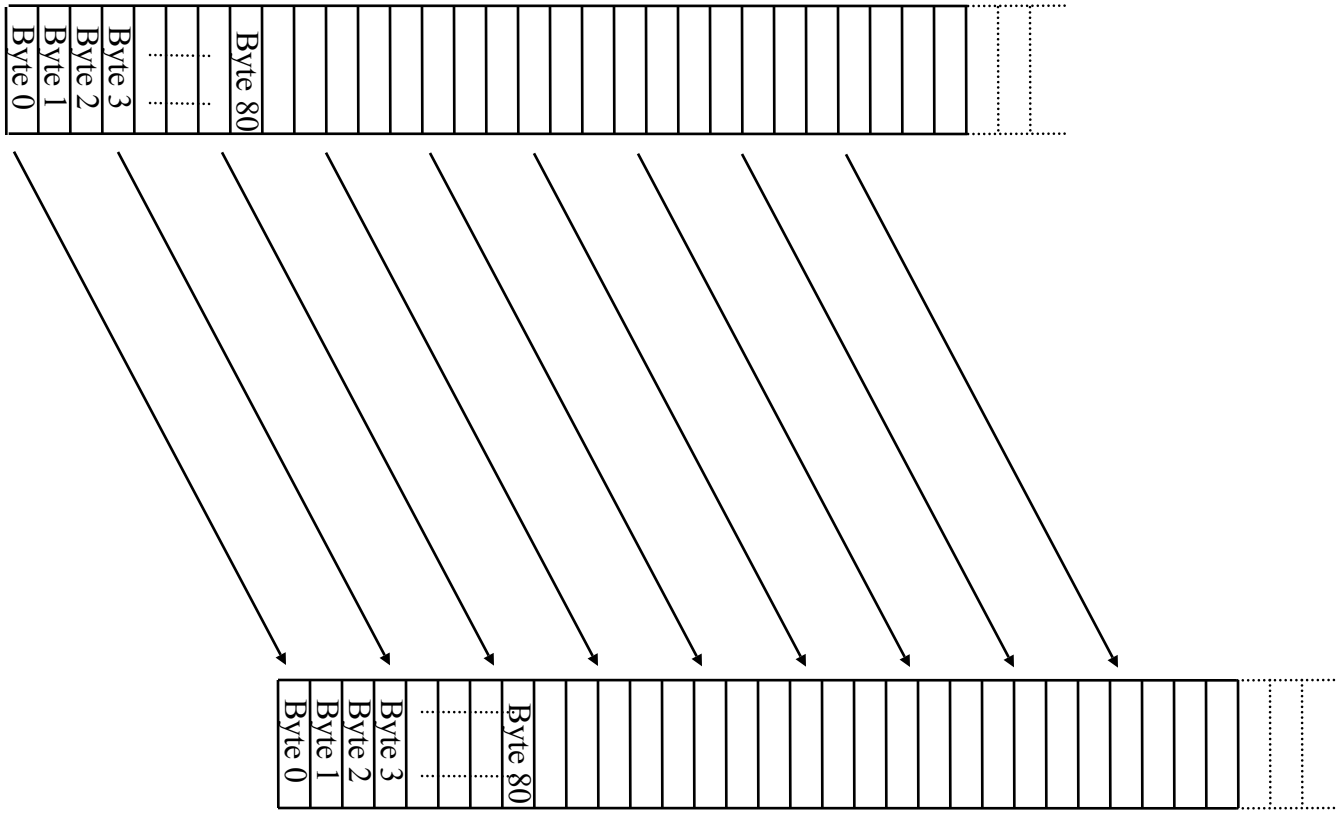
TCP Header

Flags: SYN
FIN
RST
PSH
URG
ACK



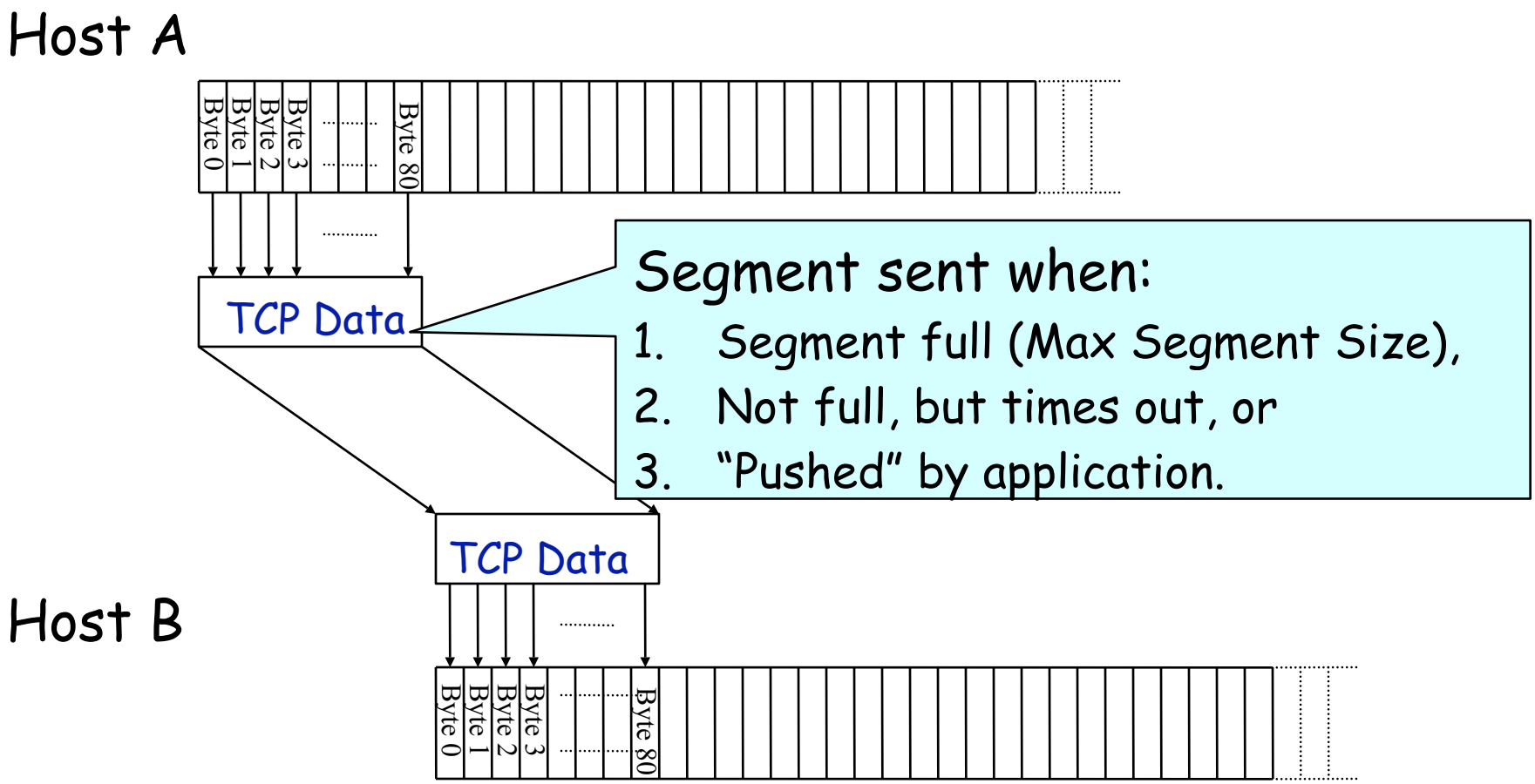
TCP "Stream of Bytes" Service

Host A



Host B

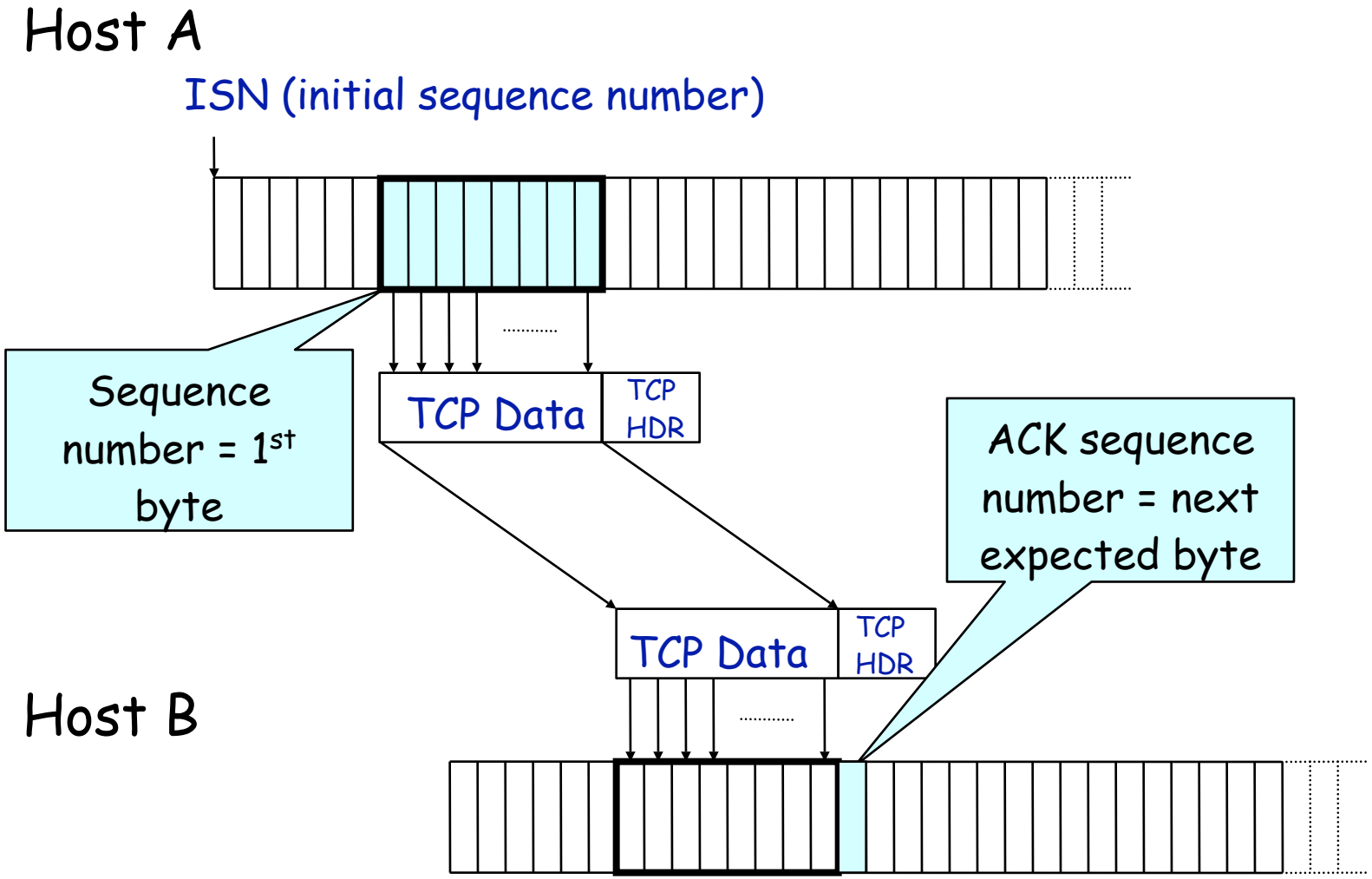
…Emulated Using TCP "Segments"



TCP Segment

- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes on an Ethernet
- 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

Sequence Numbers



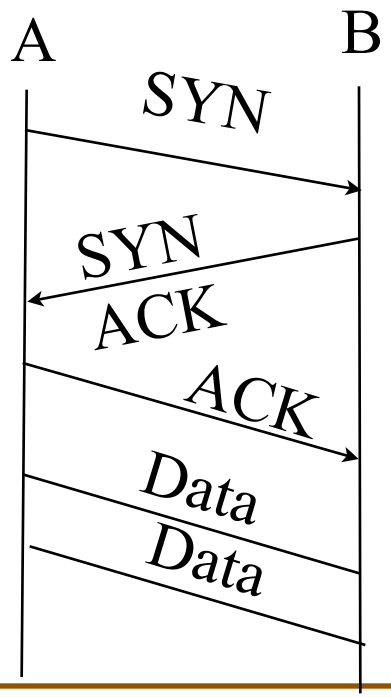
Initial Sequence Number (ISN)

- Sequence number for the very first byte
  - E.g., Why not a de facto ISN of 0?
- Practical issue
  - IP addresses and port #s uniquely identify a connection
  - Eventually, though, these port #s do get used again
  - … and there is a chance an old packet is still in flight
  - … and might be associated with the new connection
- So, TCP requires changing the ISN over time
  - Set from a 32-bit clock that ticks every 4 microseconds
  - … which only wraps around once every 4.55 hours!
- But, this means the hosts need to exchange ISNs

TCP Three-Way Handshake

Establishing a TCP Connection

- 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



Each host tells its ISN to the other host.

Step 1: A's Initial SYN Packet

Flags: SYN
FIN
RST
PSH
URG
ACK

A's port			B's port
A's Initial Sequence Number			
Acknowledgment			
20	0	Flags	Advertised window
Checksum			Urgent pointer
Options (variable)			

A tells B it wants to open a connection…

Step 2: B's SYN-ACK Packet

Flags: SYN FIN RST PSH URG ACK

B's port			A's port
B's Initial Sequence Number			
A's ISN plus 1			
20	0	Flags	Advertised window
Checksum			Urgent pointer
Options (variable)			

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

A's port			B's port
Sequence number			
B's ISN plus 1			
20	0	Flags	Advertised window
Checksum			Urgent pointer
Options (variable)			

A tells B it is okay to start sending

… upon receiving this packet, B can start sending data

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-ACK if needed
- 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
  - Some TCPs use a default of 3 or 6 seconds

SYN Loss and Web Downloads

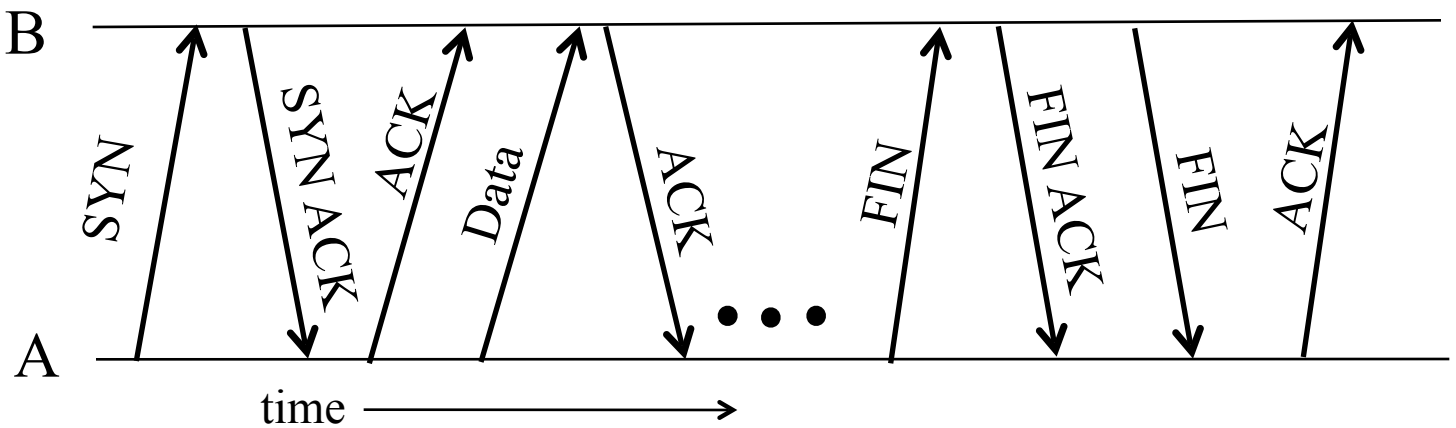
- User clicks on a hypertext link
  - Browser creates a socket and does a “connect”
  - The “connect” triggers the OS to transmit a SYN
- If the SYN is lost…
  - The 3-6 seconds of delay may be very long
  - The user may get impatient
  - … and click the hyperlink again, or click “reload”
- User triggers an “abort” of the “connect”
  - Browser creates a new socket and does a “connect”
  - Essentially, forces a faster send of a new SYN packet!
  - Sometimes very effective, and the page comes fast

Performance implications

- TCP performance is suboptimal for transfers of small Web objects:
  - nearly all TCP implementations send the first data after the handshake is completed
  - this adds an additional *round-trip time (RTT)* to the application-layer data transfer

Tearing Down the Connection

Tearing Down the Connection



- Closing 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 of 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

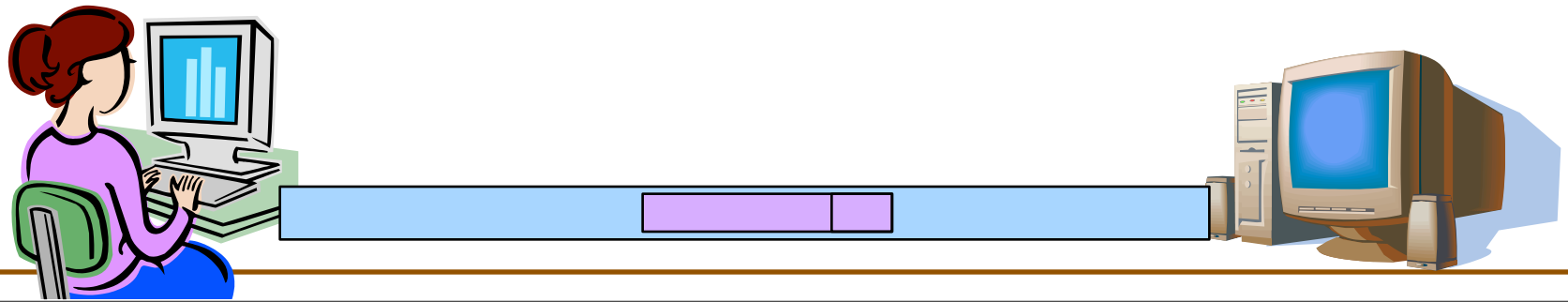
Flow Control

Principles of TCP Flow Control

- Flow control limits the transmission rate to a rate that the receiver can absorb the data
- TCP makes the sender wait for an ACK after transmitting a certain amount of data
- To decide how much data to send before waiting, TCP uses the sliding window protocol
- With every ACK segment, the receiver advertises a window in bytes, using the WINDOW header field
  - this window size indicates how many more bytes the receiver is able to accept into its buffers

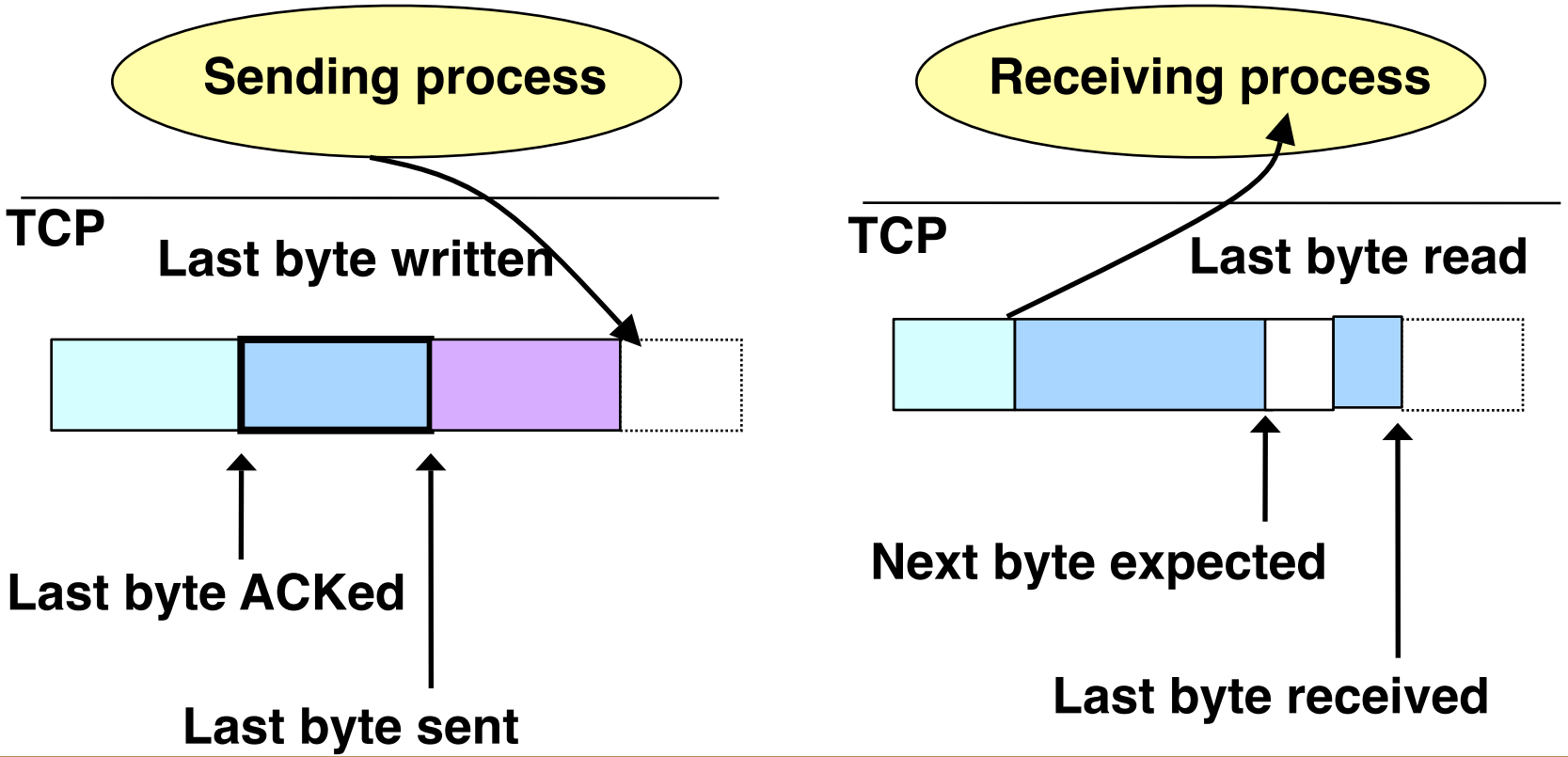
Motivation for Sliding Window

- Stop-and-wait is inefficient
  - Only one TCP segment is "in flight" at a time
  - Especially bad when delay-bandwidth product is high
- Numerical example
  - 1.5 Mbps link with a 45 msec round-trip time (RTT)
    - Delay-bandwidth product is 67.5 Kbits (or 8 KBytes)
  - But, sender can send at most one packet per RTT
    - Assuming a segment size of 1 KB (8 Kbits)
    - … leads to 8 Kbits/segment / 45 msec/segment ➔ 177 Kbps
    - That's just one-eighth of the 1.5 Mbps link capacity



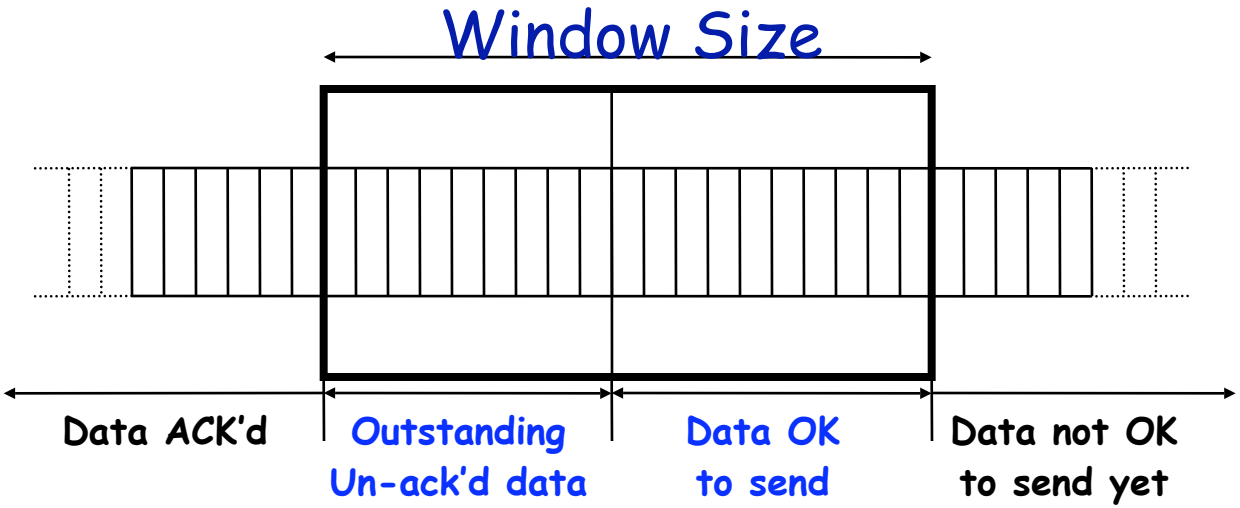
Sliding Window

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



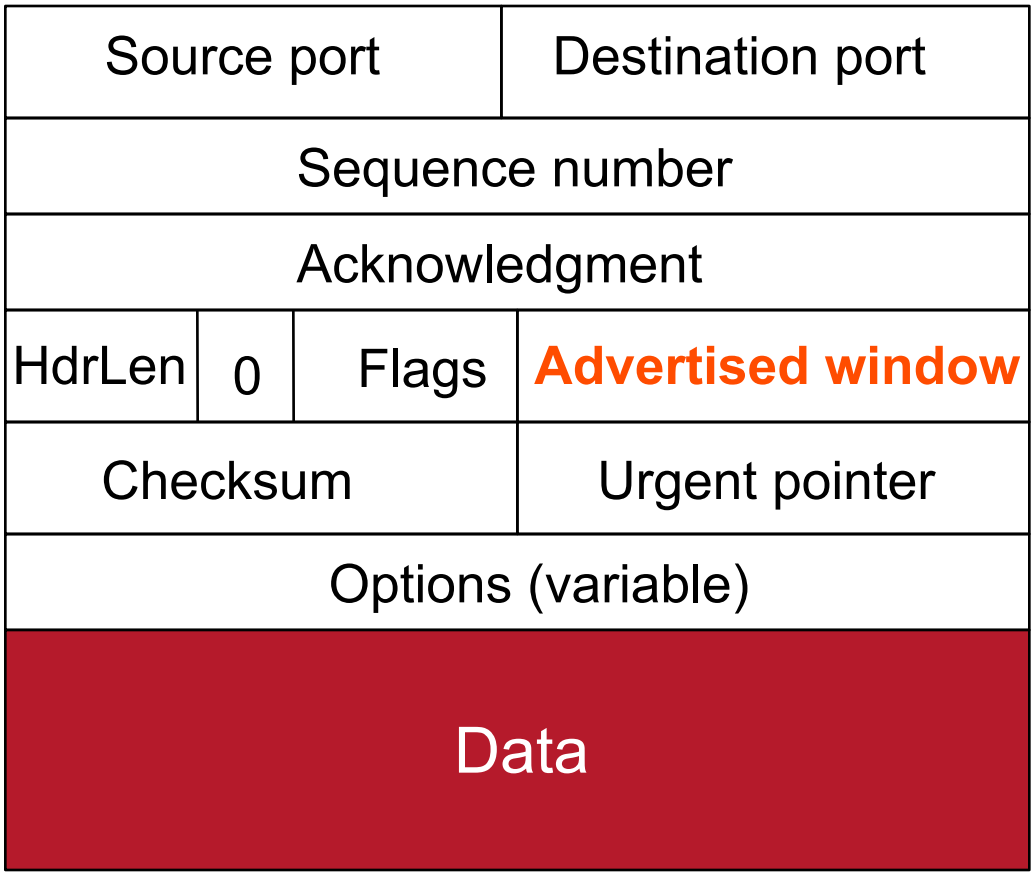
Receiver Buffering

- Window size
  - Amount that can be sent without acknowledgment
  - Receiver needs to be able to store this amount of data
- Receiver advertises the window to the sender
  - Tells the sender the amount of free space left
  - … and the sender agrees not to exceed this amount



TCP Header for Receiver Buffering

Flags: SYN
FIN
RST
PSH
URG
ACK



TCP congestion control

- Applied in order to avoid packet drops and retransmissions that would further increase the congestion.
- TCP congestion control mechanism:
  - Congestion window: specifies the *number of unacknowledged segments* one host may send to the other
- Basic idea:
  - sender monitors the loss of packets; when this occurs, the sender reduces quickly (multiplicatively) its transmission rate.
  - if there are no lost packets, sender increases slowly its transmission rate - one segment per congestion window of data acknowledged
  - additive increase, multiplicative decrease algorithm

Slow start mechanism

- Start with a small transfer rate (allow only two unacknowledged segments)
- Initially, increase the congestion window by one segment for each acknowledged segment:
  - this leads to an exponential increase of transfer rate
- If a packet is lost, reduce drastically the transfer rate by *reducing the congestion window by half*
- Later: increase the congestion window by one segment only after the sender receives acks for the number of segments equal to the congestion window
- Several packet losses in quick succession may trigger the repeat of the slow start from scratch

Performance implications

- The throughput of one large TCP transfer is usually higher than the throughput of many short ones, with the exception of very short connections that fit into the initial congestion window of two segments (~3KB)
- Short TCP transfers are affected mostly by the TCP handshake overhead, which cannot be amortized over the life of a connection

TCP Retransmissions

Retransmissions

- A mechanism to deal with unreliability (loss of packets)
- TCP sender retransmits segments that have not been acknowledged by the receiver within a certain time after the initial transmission (timeout)
- The timeout value is adapted according to previously observed RTT and RTT variance between two communicating hosts
- Initial timeout value set to a predefined value

Automatic Repeat reQuest (ARQ)

Automatic Repeat Request
Receiver sends acknowledgment (ACK) when it receives packet
Sender waits for ACK and timeouts if it does not arrive within some time period
Simplest ARQ protocol
Stop and wait
Send a packet, stop and wait until ACK arrives