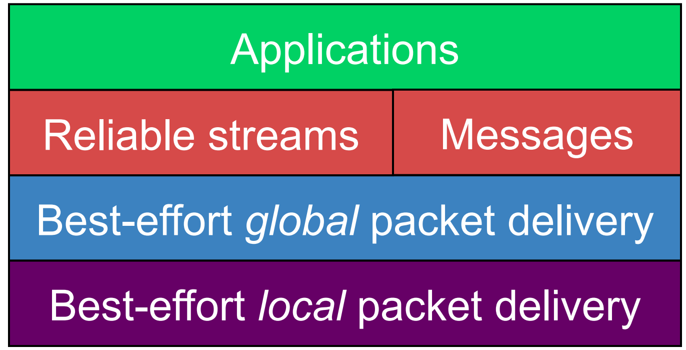
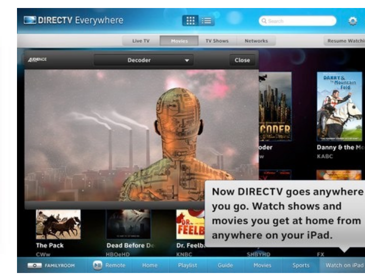


Transport Layer

Note: The slides are adapted from the materials from Prof. Richard Han at CU Boulder and Profs. Jennifer Rexford and Mike Freedman at Princeton University, and the networking book (Computer Networking: A Top Down Approach) from Kurose and Ross.

Popularity of Internet Applications



Reliability

Efficiency

Transport Layer Services

Application	Applications	
Transport	Reliable streams (TCP)	Unreliable messages (UDP)
Network	Best-effort *global* packet delivery	
Link	Best-effort *local* packet delivery	

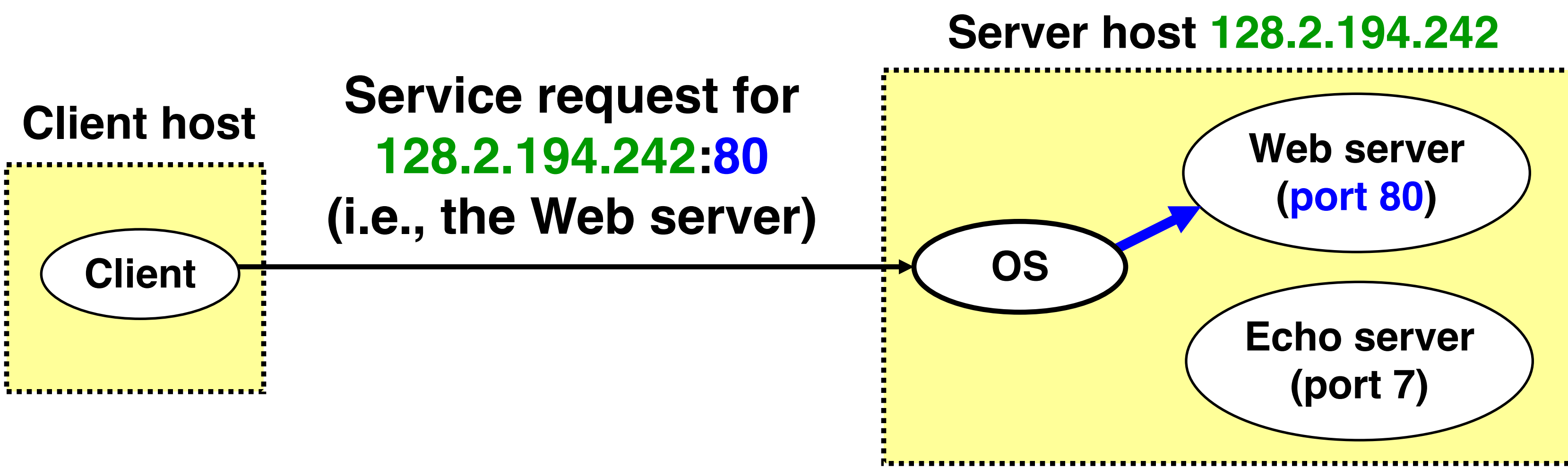
- Transport layer is where we “pay the piper”
  - Provide applications with good abstractions
  - Without support or feedback from the network

Transport Protocols

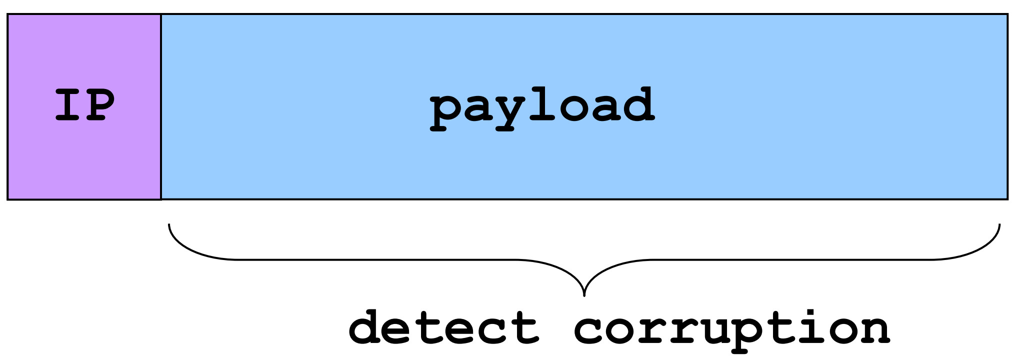
- Logical communication between processes
  - Sender divides a message into segments
  - Receiver reassembles segments into message
- Transport services
  - (De)multiplexing packets
  - Detecting corrupted data
  - Optionally: reliable delivery, flow control, …

Two Basic Transport Features

- **Demultiplexing:** port numbers



- **Error detection:** checksums

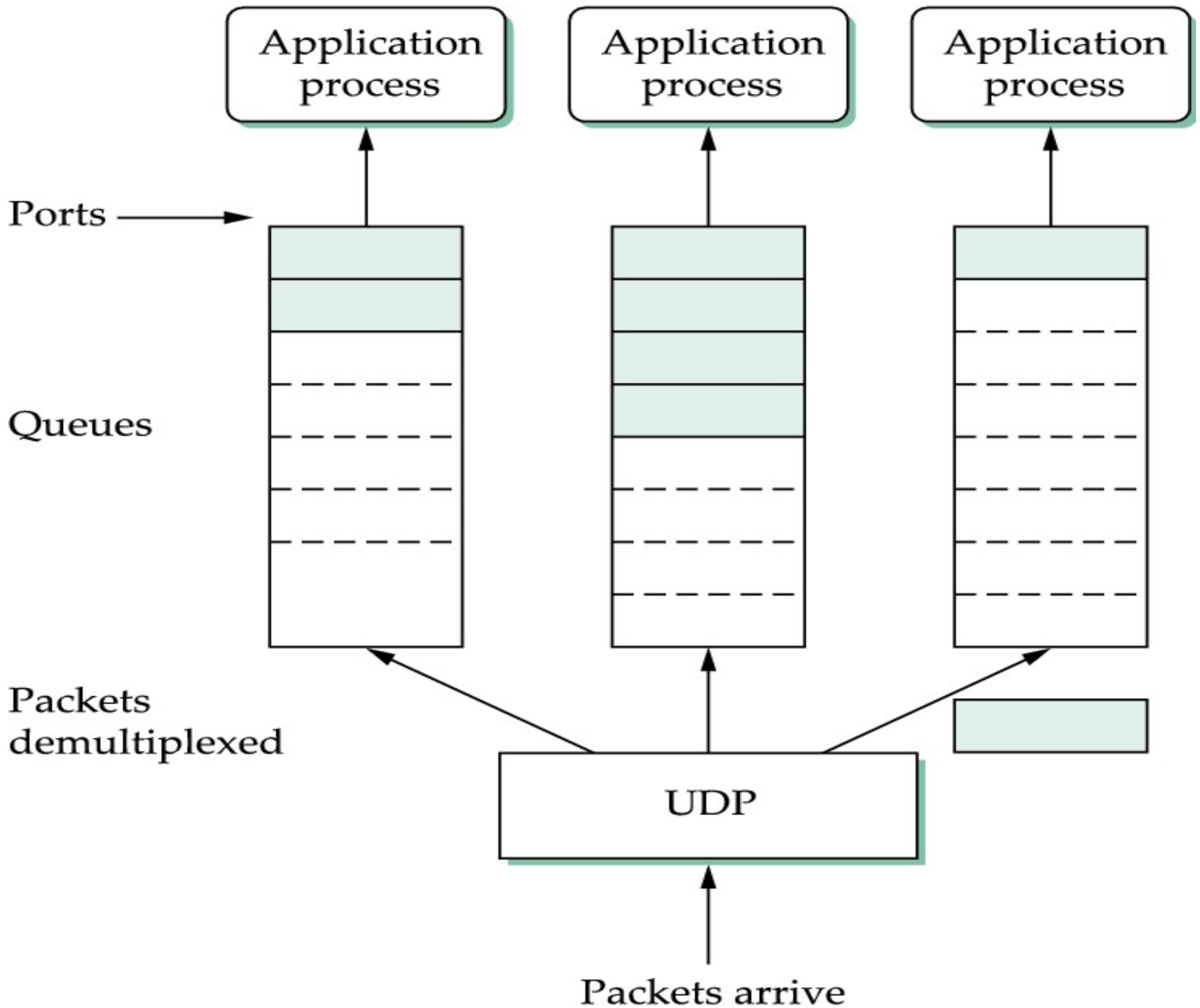


User Datagram Protocol (UDP)

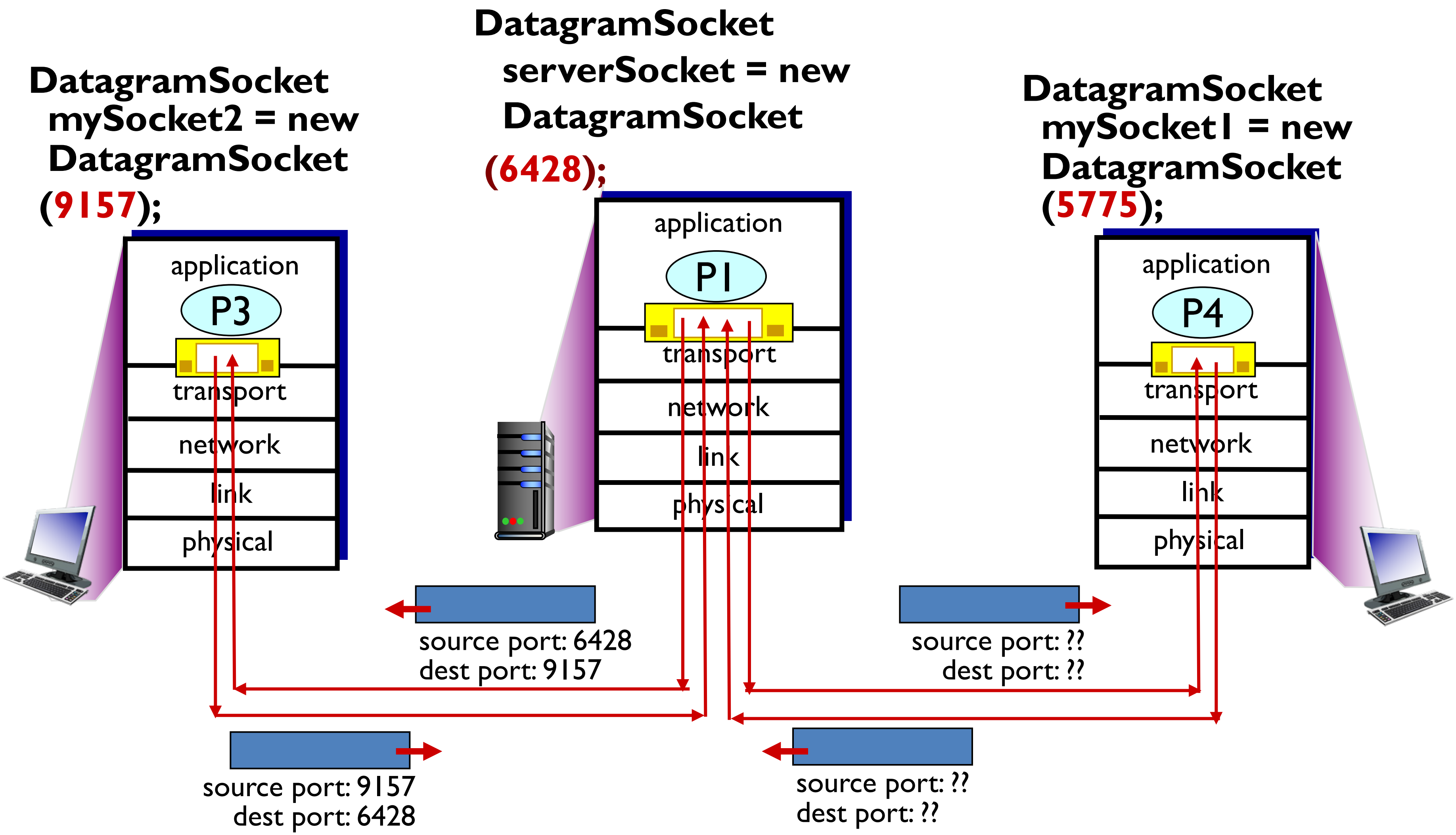
- Datagram messaging service
  - Demultiplexing: port numbers
  - Detecting corruption: checksum
- Lightweight communication between processes
  - Send and receive messages
  - Avoid overhead of ordered, reliable delivery

SRC port	DST port
checksum	length
DATA	

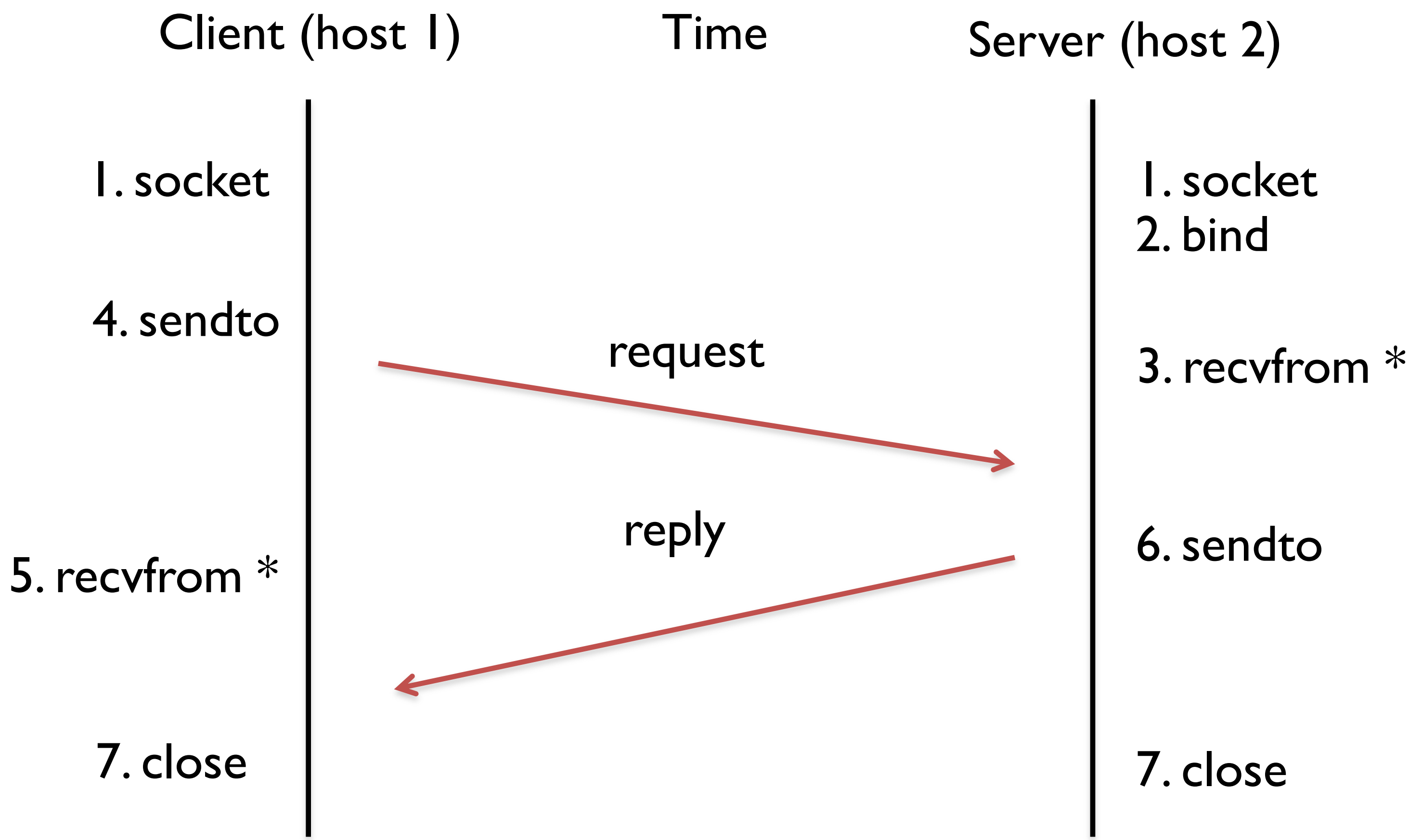
UDP Message Queue



Connectionless Demux: Example



UDP Datagram Socket



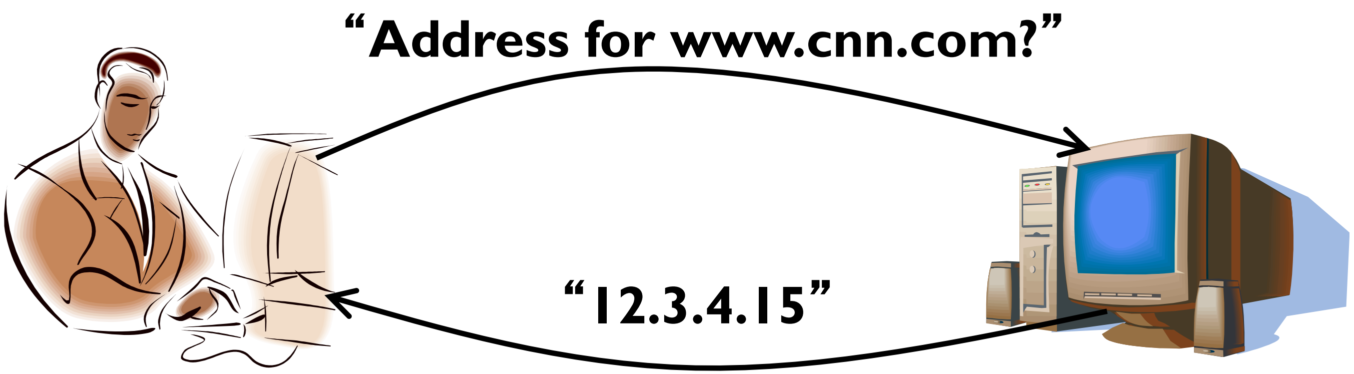
* : call block

Advantages of UDP

- Fine-grain control
  - UDP sends as soon as the application writes
- No connection set-up delay
  - UDP sends without establishing a connection
- No connection **state**
  - No buffers, parameters, sequence #s, etc.
- Small header overhead
  - UDP header is only eight-bytes long

Popular Applications That Use UDP

- Multimedia streaming
  - Retransmitting packets is not always worthwhile
  - E.g., phone calls, video conferencing, gaming, IPTV
- Simple query-response protocols
  - Overhead of connection establishment is overkill
  - E.g., Domain Name System (DNS), DHCP, etc.



Transmission Control Protocol (TCP)

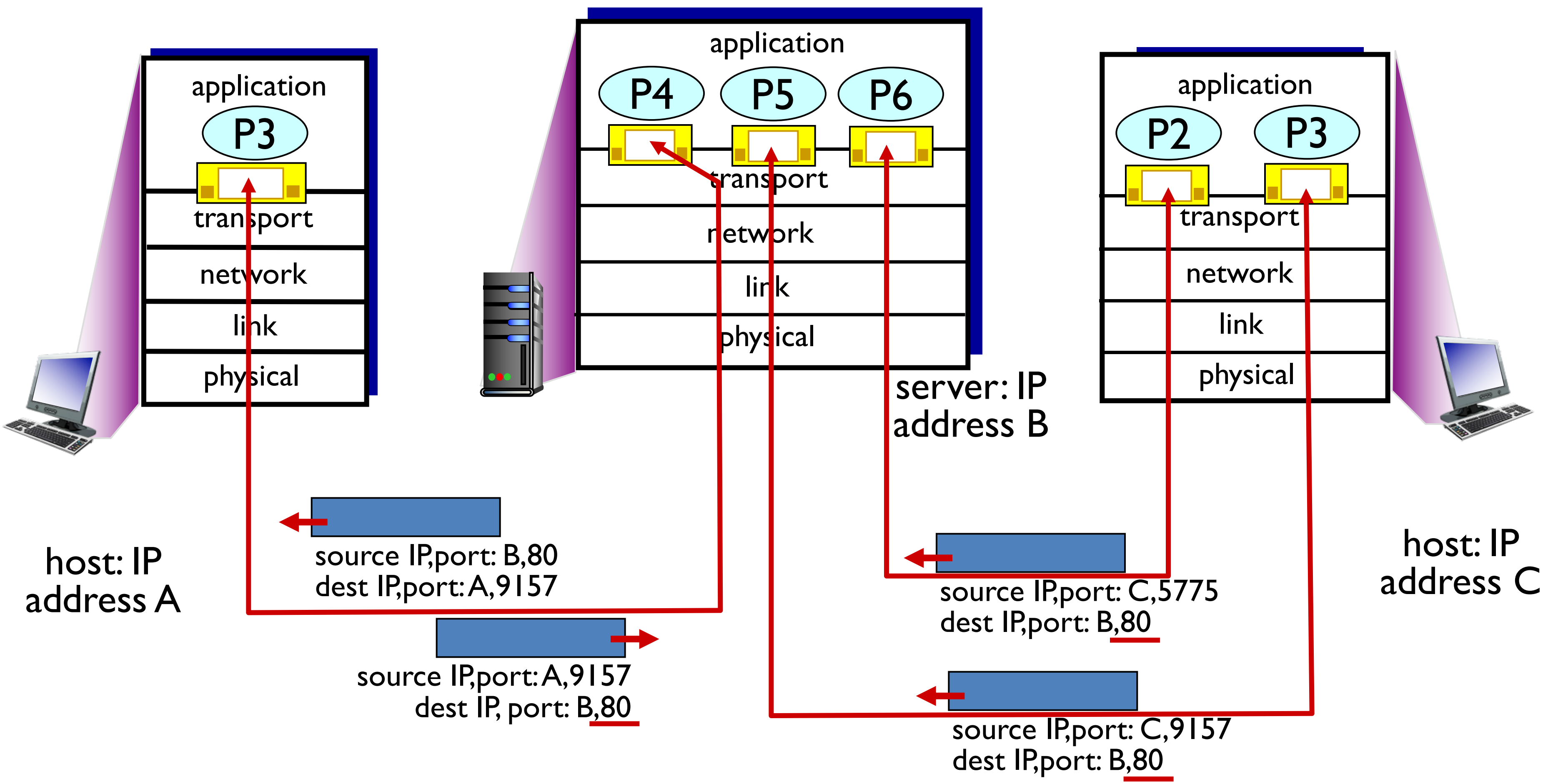
- Stream-of-bytes service
  - Sends and receives a stream of bytes
- Reliable, in-order delivery
  - Corruption: checksums
  - Detect loss/reordering: sequence numbers
  - Reliable delivery: acknowledgments and retransmissions
- Connection oriented
  - Explicit set-up and tear-down of TCP connection
  - Connection-oriented multiplexing
- Flow control
  - Prevent overflow of the receiver's buffer space
- Congestion control
  - Adapt to network congestion for the greater good

Connection-oriented Demux

- 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

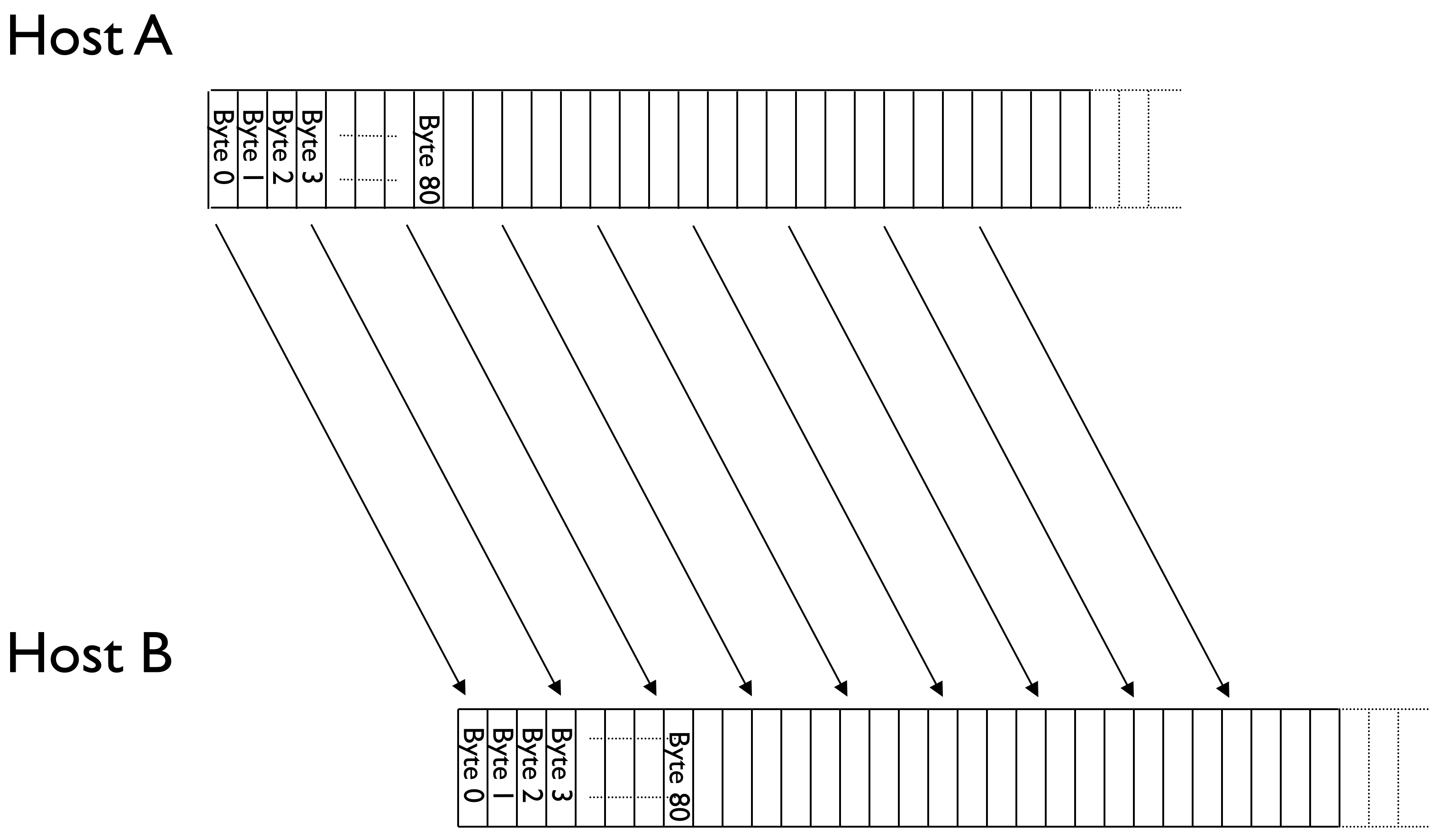
- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - Non-persistent HTTP will have different socket for each request

Connection-oriented Demux: Example

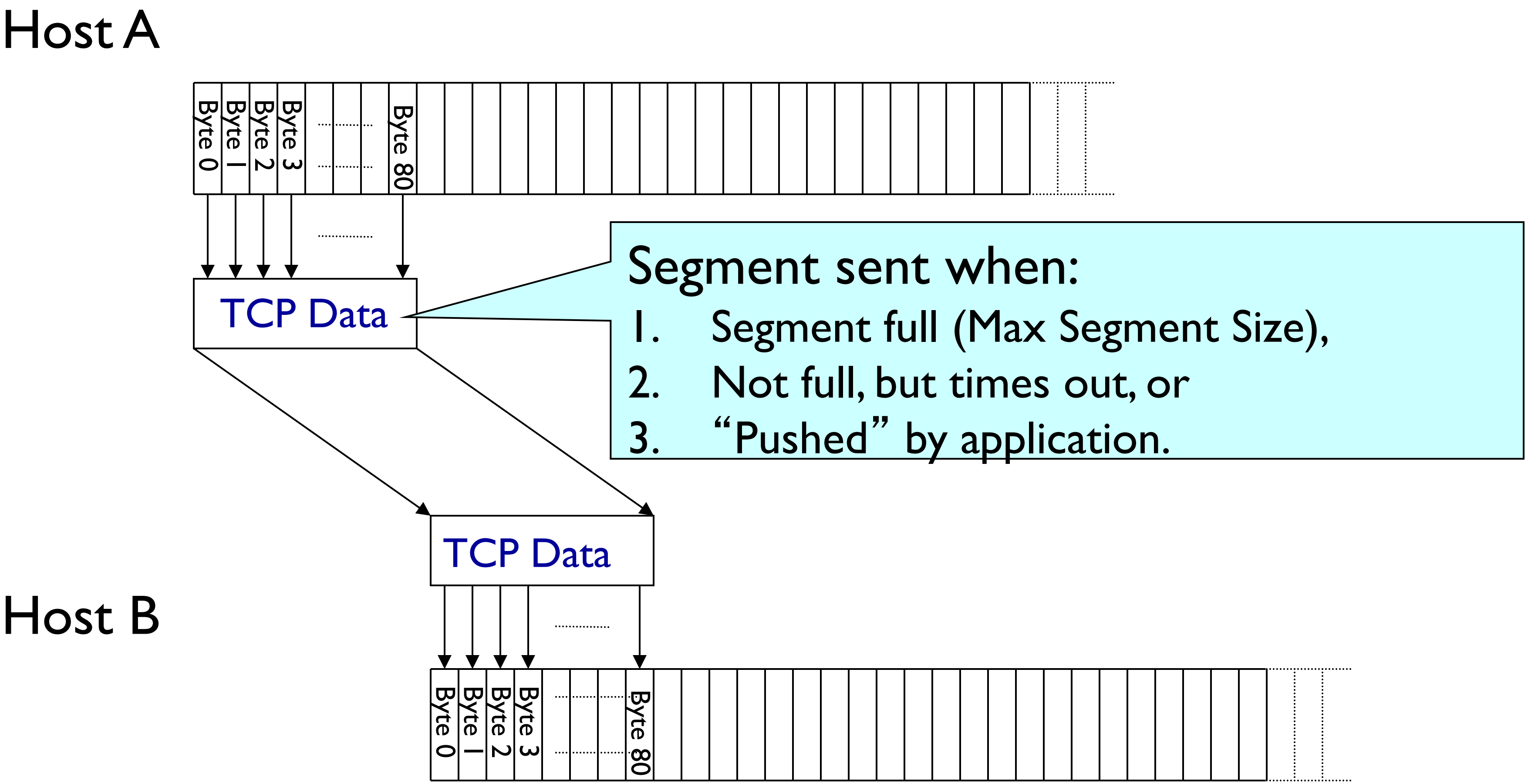


three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

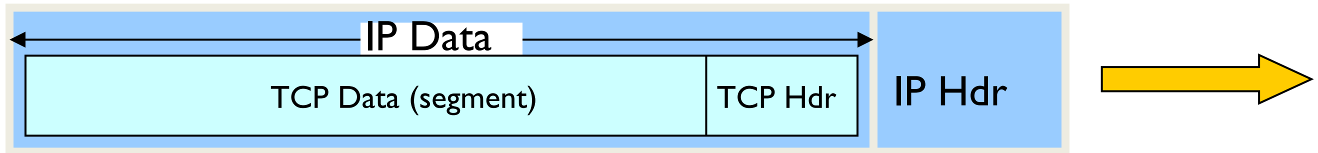
TCP "Stream of Bytes" Service



…Emulated Using TCP "Segments"

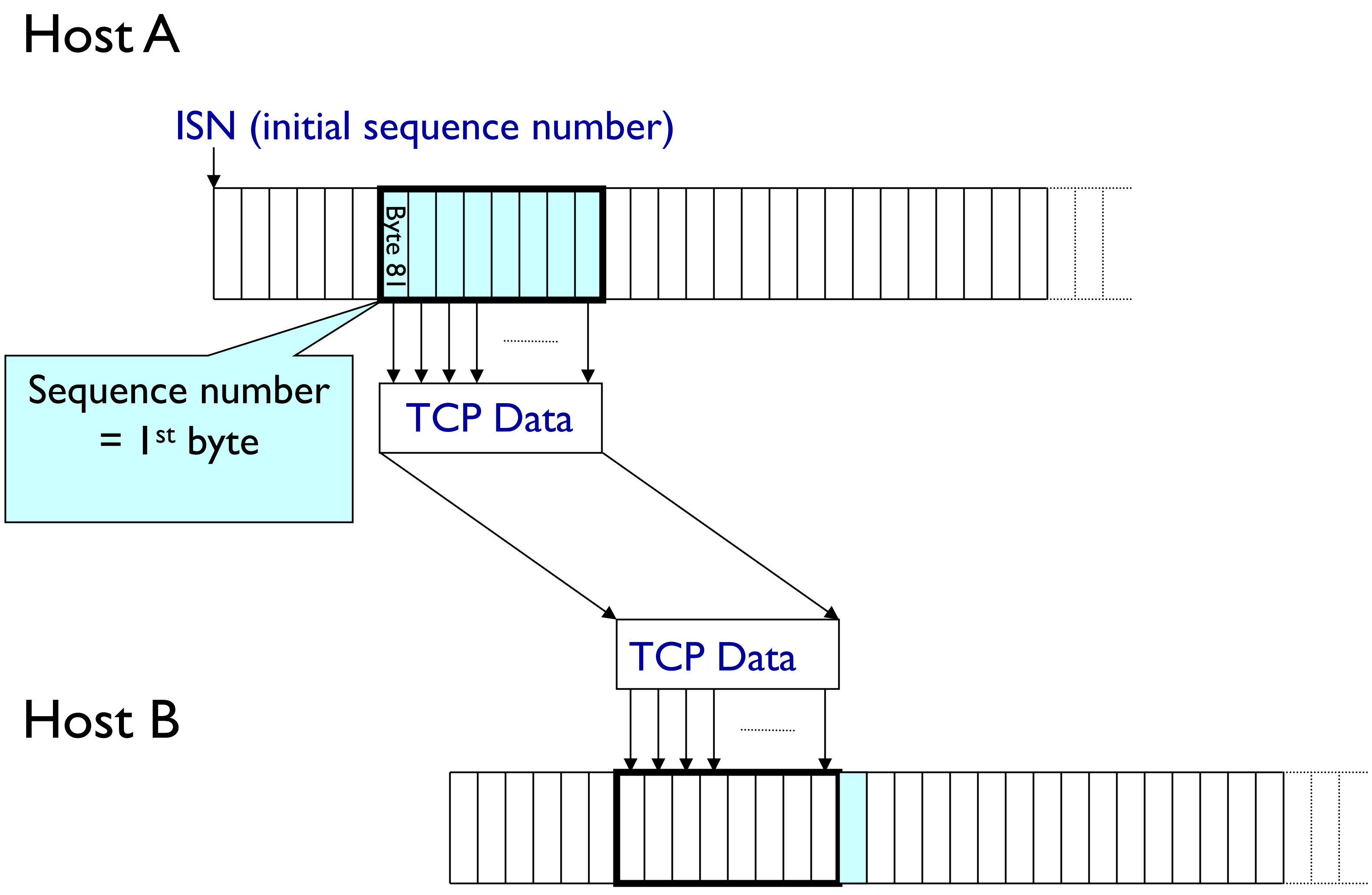


TCP Segment



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



Initial Sequence Number (ISN)

- Sequence number for the very first byte
  - E.g., Why not a de facto ISN of 0?
- Practical issue: reuse of port numbers
  - Port numbers must (eventually) get used again
  - … and an old packet may still be in flight
  - … and associated with the new connection
- So, 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!

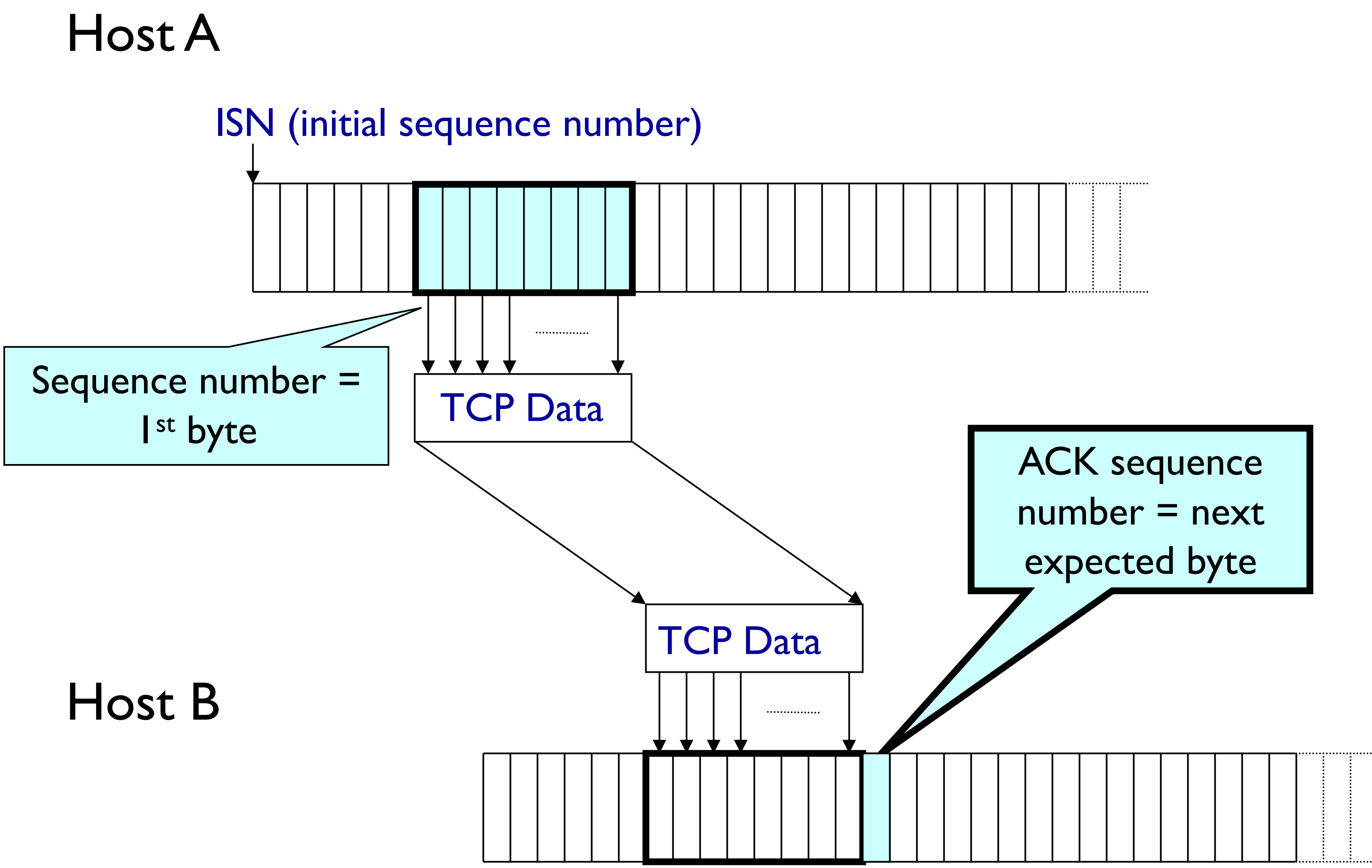
Challenges of Reliable Data Transfer

- 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

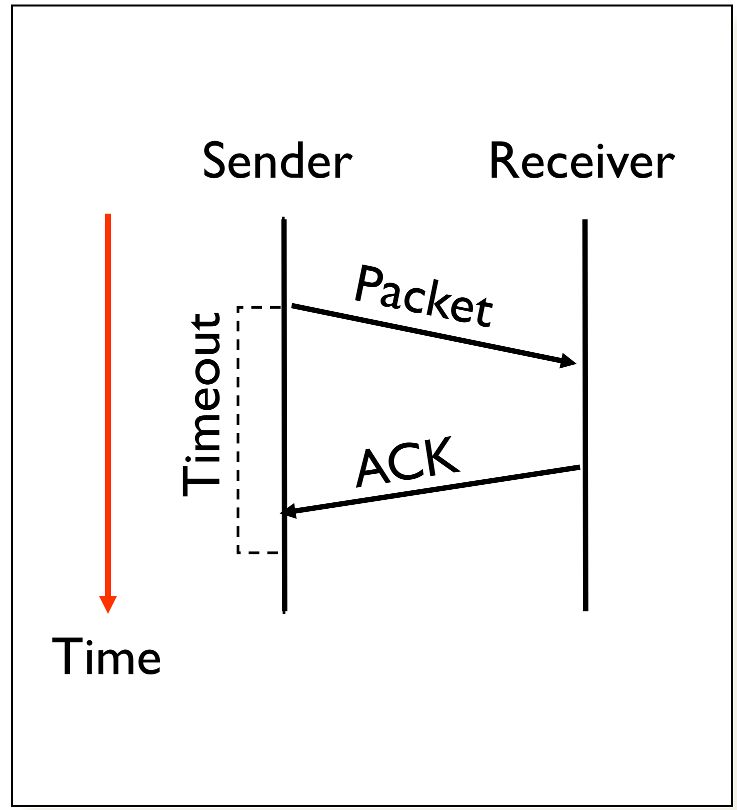
- **Detect bit errors:** checksum
  - Used to detect corrupted data at the receiver
  - …leading the receiver to drop the packet
- **Detect missing data:** sequence number
  - Used to detect a gap in the stream of bytes
  - ... and for putting the data back in order
- **Recover from lost data:** retransmission
  - Sender retransmits lost or corrupted data
  - Two main ways to detect lost packets

TCP Acknowledgments



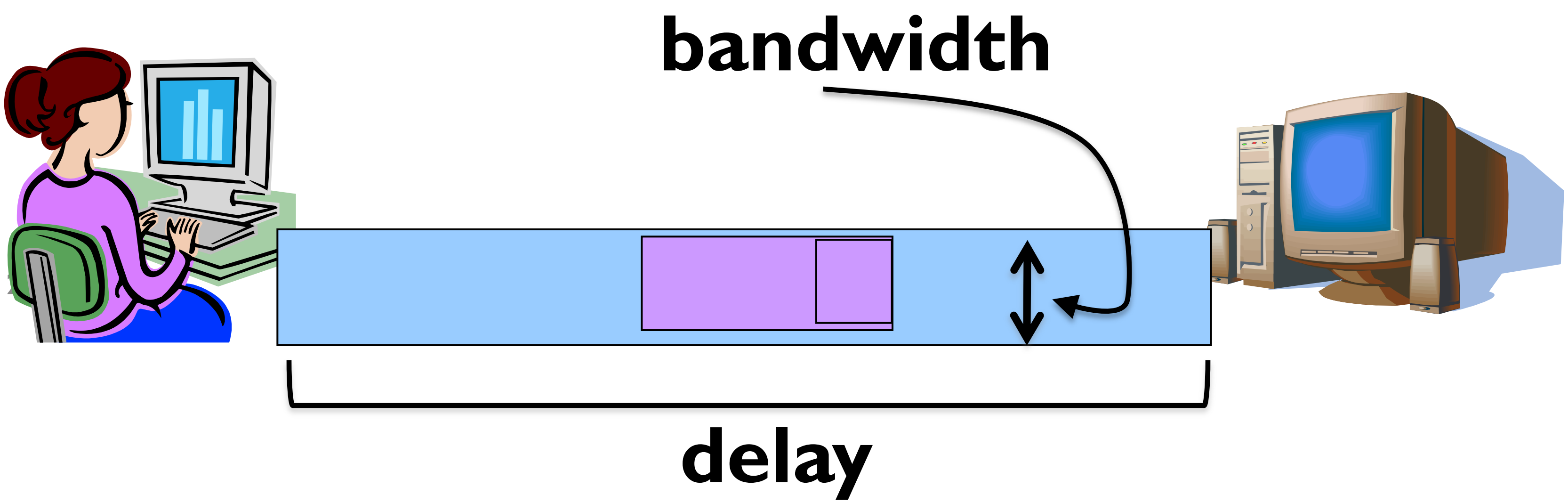
Automatic Repeat reQuest (ARQ)

- ACK and timeouts
  - Receiver sends ACK when
    it receives packet
  - Sender waits for ACK and times out
- Simplest ARQ protocol
  - Stop and wait
  - Send a packet, stop and wait until ACK arrives



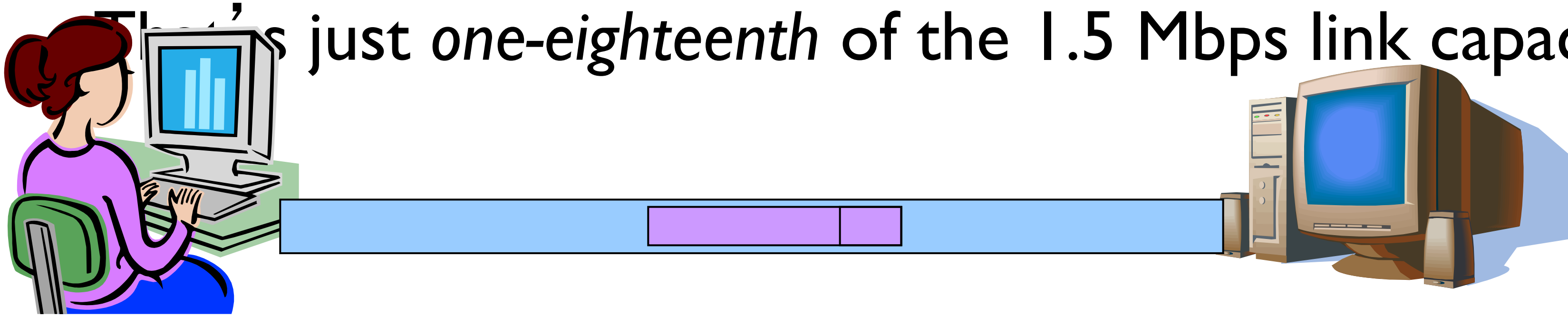
Motivation for Sliding Window

- Stop-and-wait is inefficient
  - Only one TCP segment is "in flight" at a time
  - Especially bad for high "delay-bandwidth product"



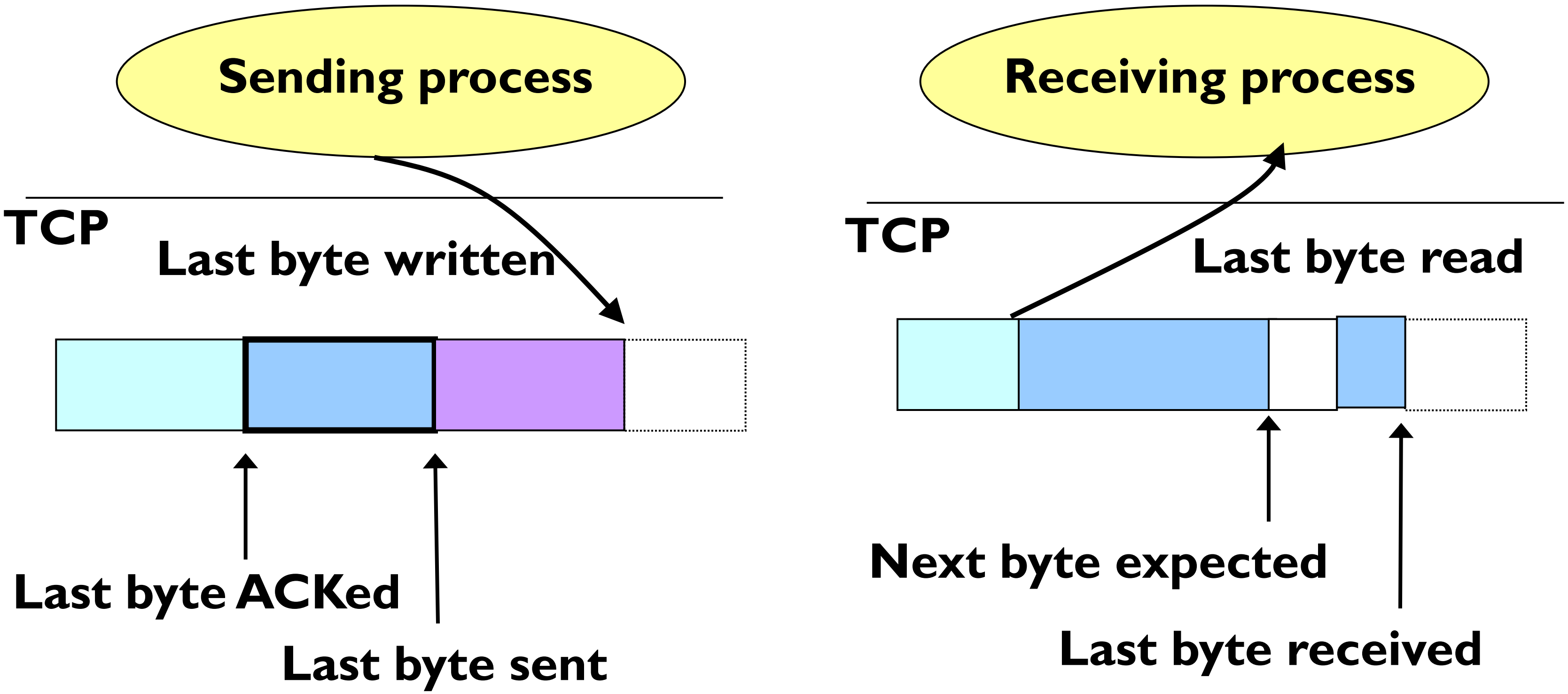
Numerical Example

- 1.5 Mbps link with 100msec round-trip time (RTT)
  - – Delay-bandwidth product is 150 Kbits (or 18 KBytes)
- Sender can send at most one packet per RTT
  - – Assuming a segment size of 1 KB (8 Kbits)
  - – 8 Kbits/segment at 100 msec/segment → 80 Kbps
  - – That's just *one-eighteenth* of the 1.5 Mbps link capacity



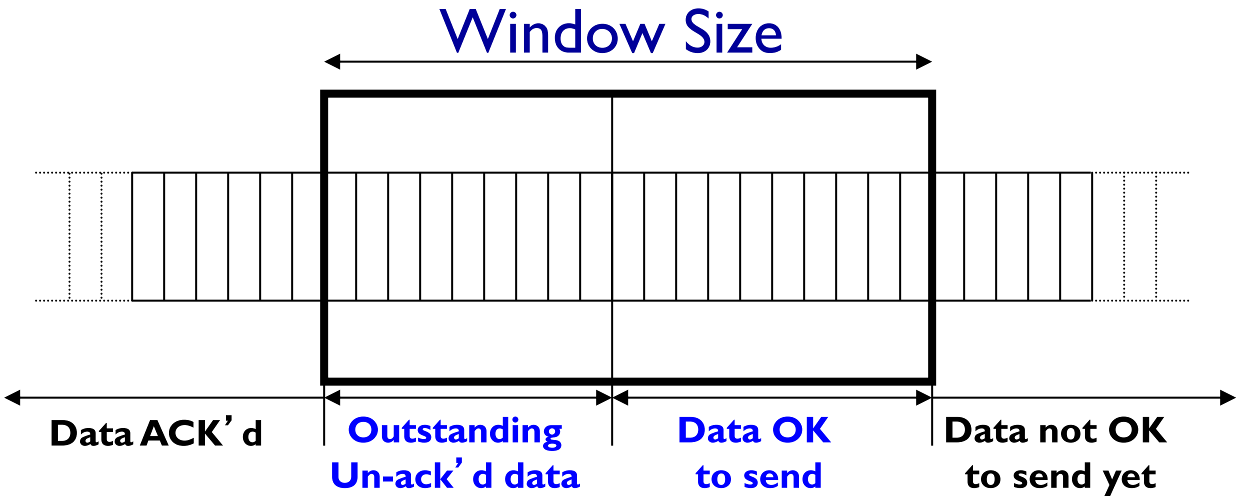
Sliding Window

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

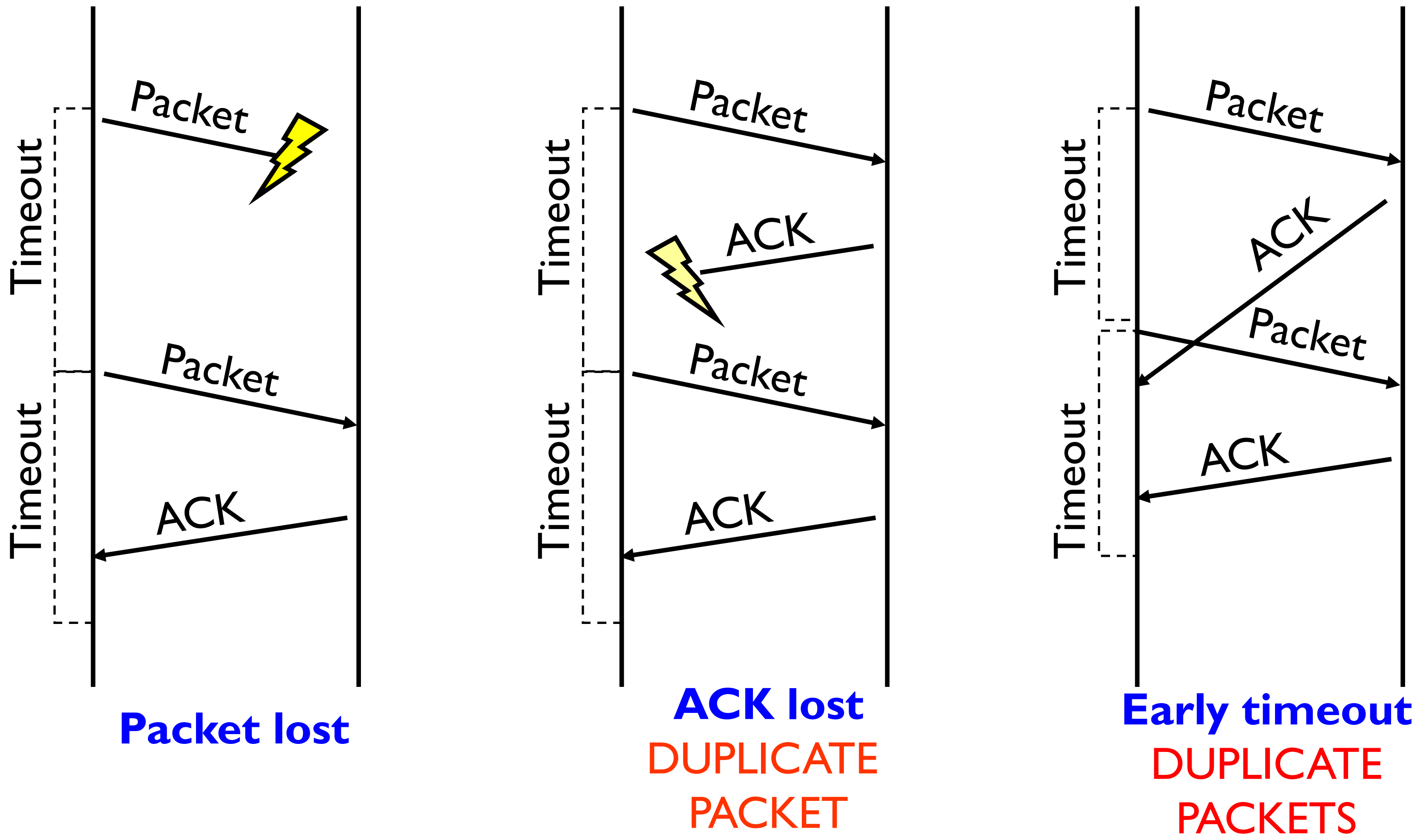


Flow Control – Receiver Buffer

- Receive window size
  - Amount that can be sent without acknowledgment
  - 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



Reasons for Retransmission



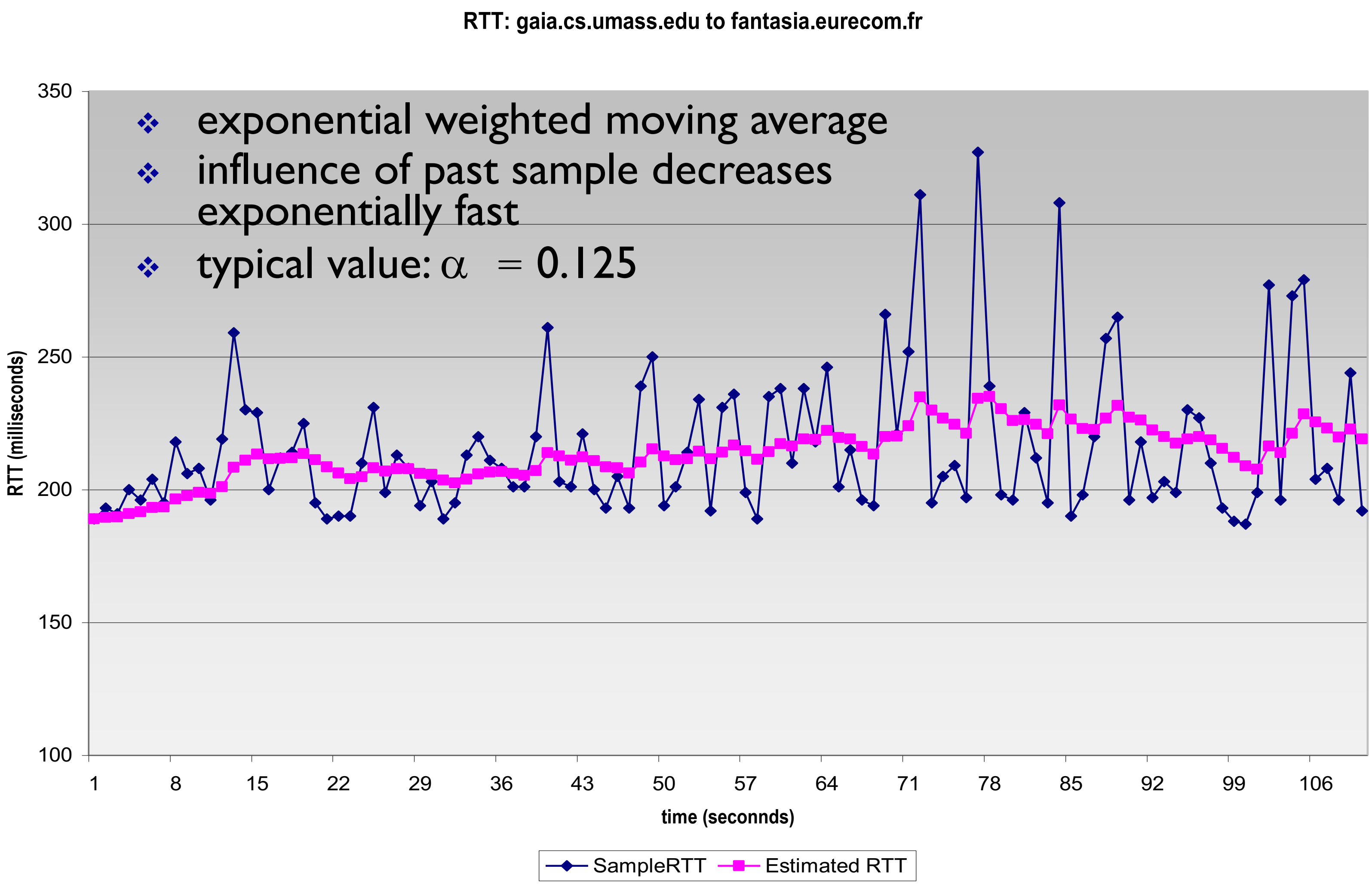
How Long Should Sender Wait?

- 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 an "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

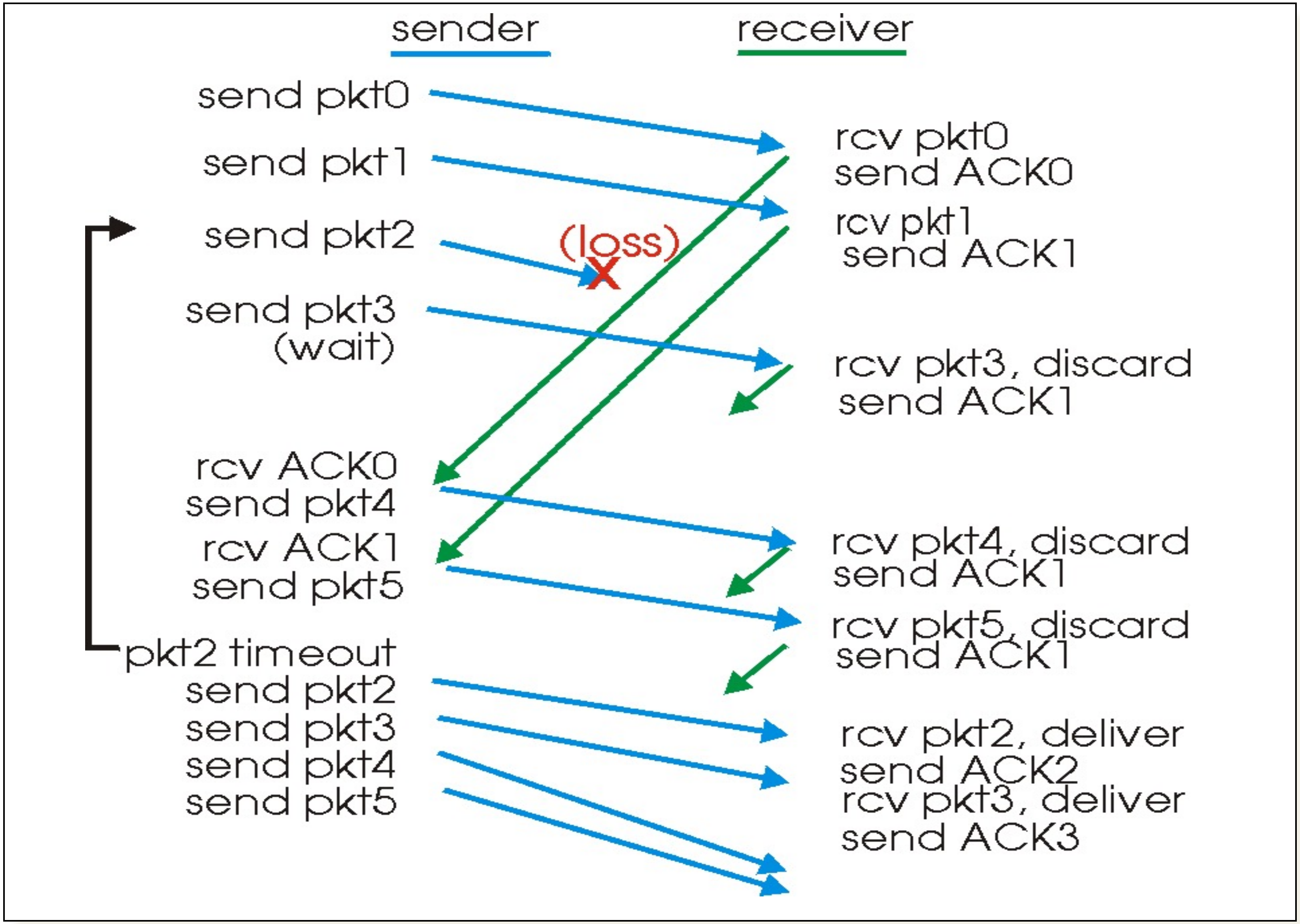
Example RTT Estimation

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

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$



Still, Timeouts are Inefficient



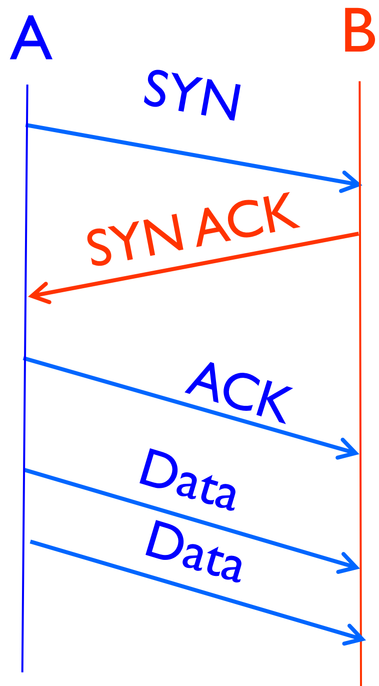
Fast Retransmission

- When packet n is lost…
  - … packets n+1, n+2, and so on may get through
- Exploit the ACKs of these packets
  - ACK says receiver is still awaiting nth packet
  - Duplicate ACKs suggest later packets arrived
  - Sender uses "duplicate ACKs" as a hint
- Fast retransmission
  - Retransmit after "triple duplicate ACK"

Effectiveness of Fast Retransmit

- When does Fast Retransmit work best?
  - High likelihood of many packets in flight
  - Long data transfers, large window size, …
- Implications for Web traffic
  - Most Web transfers are short (e.g., 10 packets)
    - So, often there aren't many packets in flight
  - Making fast retransmit is less likely to "kick in"
    - Forcing users to click "reload" more often… ☺

Establishing a TCP Connection

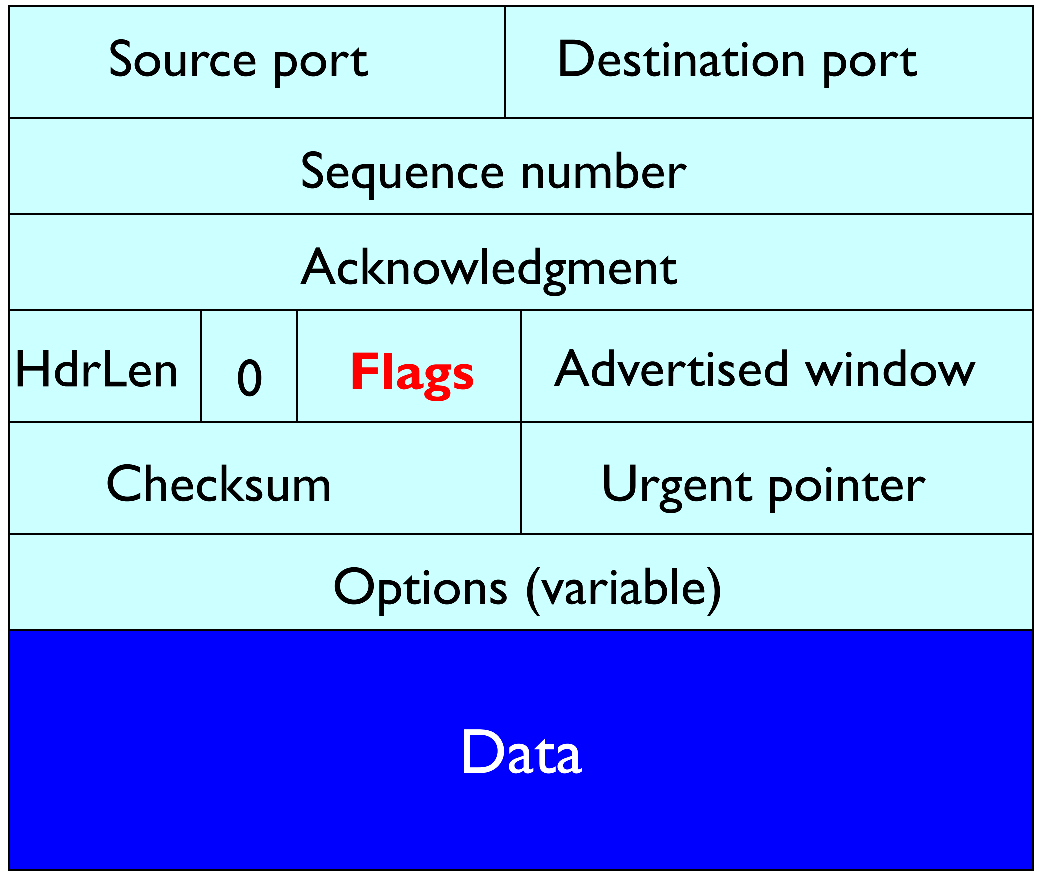


Each host tells its ISN to the other host.

- 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

TCP Header

Flags: SYN
FIN
RST
PSH
URG
ACK



Step 1: A's Initial SYN Packet

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">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>

A tells B it wants to open a connection…

Step 2: B’s SYN-ACK Packet

Flags: SYN
FIN
RST
PSH
URG
ACK

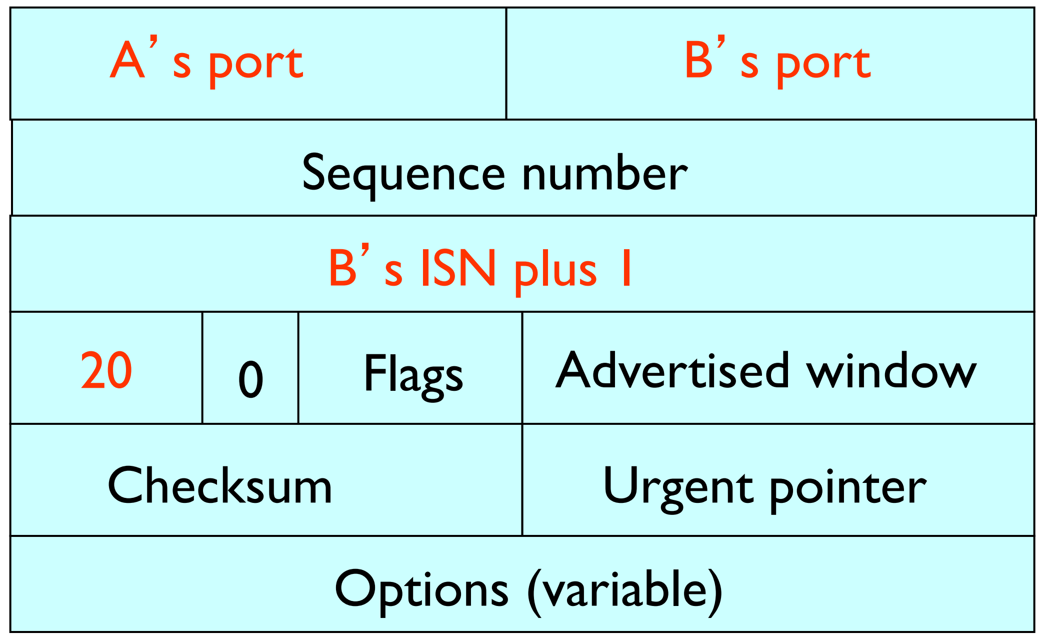
B’s port			A’s port
B’s Initial Sequence Number			
A’s ISN plus 1			
20	0	Flags	Advertised window
Checksum			Urgent pointer
Options (variable)			

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 tells B it is okay to start sending

… upon receiving this packet, B can start sending data

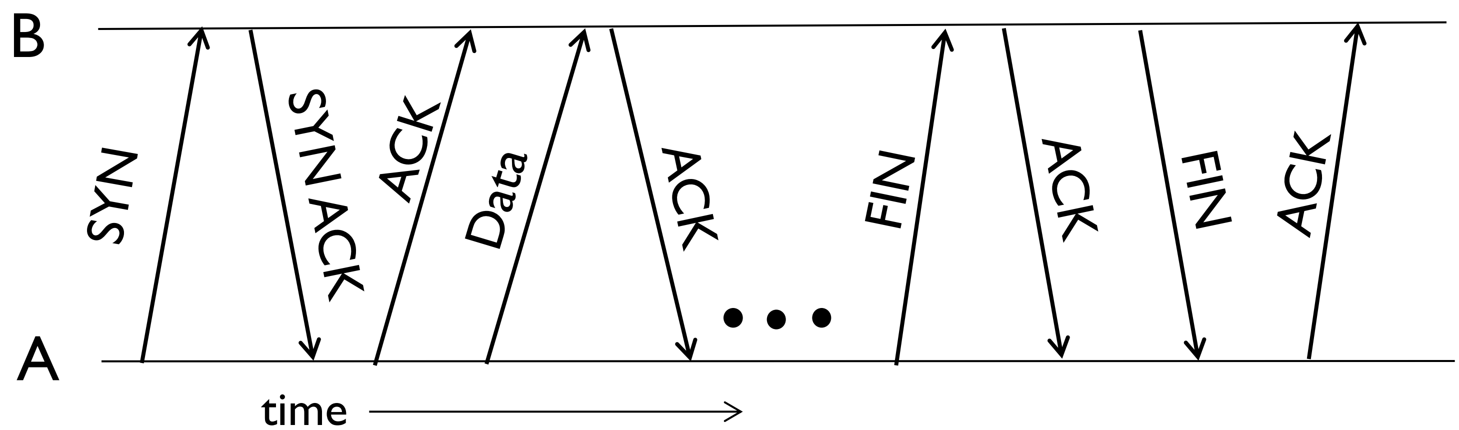
What if the SYN Packet Gets Lost?

- 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 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

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!

Tearing Down the Connection



- 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

Insights

TCP (Streams)	UDP (Datagrams)
Connections	Datagrams
Bytes are delivered once, reliably, and in order	Messages may be lost, reordered, duplicated
Session setup/teardown is needed	Light-weight communication without requiring session setup/teardown
Arbitrary length content	Limited message size
Flow control matches sender to receiver	Can send regardless of receiver state
Congestion control matches sender to network	Can send regardless of network state
Layering Principles	

Reliability and efficiency tradeoff

Insights (Cont'd)

- The design choice of TCP/IP follows the "end-to-end principle"
  - End-hosts are intelligent
  - The network is "dumb"
- TCP takes care of managing demand at the end-hosts in response to link loads and the feedback from the end-hosts
  - Sliding window/flow control in TCP is good example
  - Rate adaptation at the end hosts is easier to stabilize than router adaptation inside the network

Next Lecture: TCP Congestion Control

- Congestion
  - Congestion collapse: the throughput dropped to 40bps on the link of 32kbps (LBNL -> UCB)
  - Informally: "too many sources sending too much data too fast for network to handle"
  - Manifestations – lost packets, long delays
- Congestion Control
  - End-to-end congestion control (approach taken by TCP)
  - Network-assisted congestion control (TCP/IP ECN)