Transport Protocols

Reading: Sections 5.1 and 5.2

COS 461: Computer Networks

Spring 2011

Mike Freedman

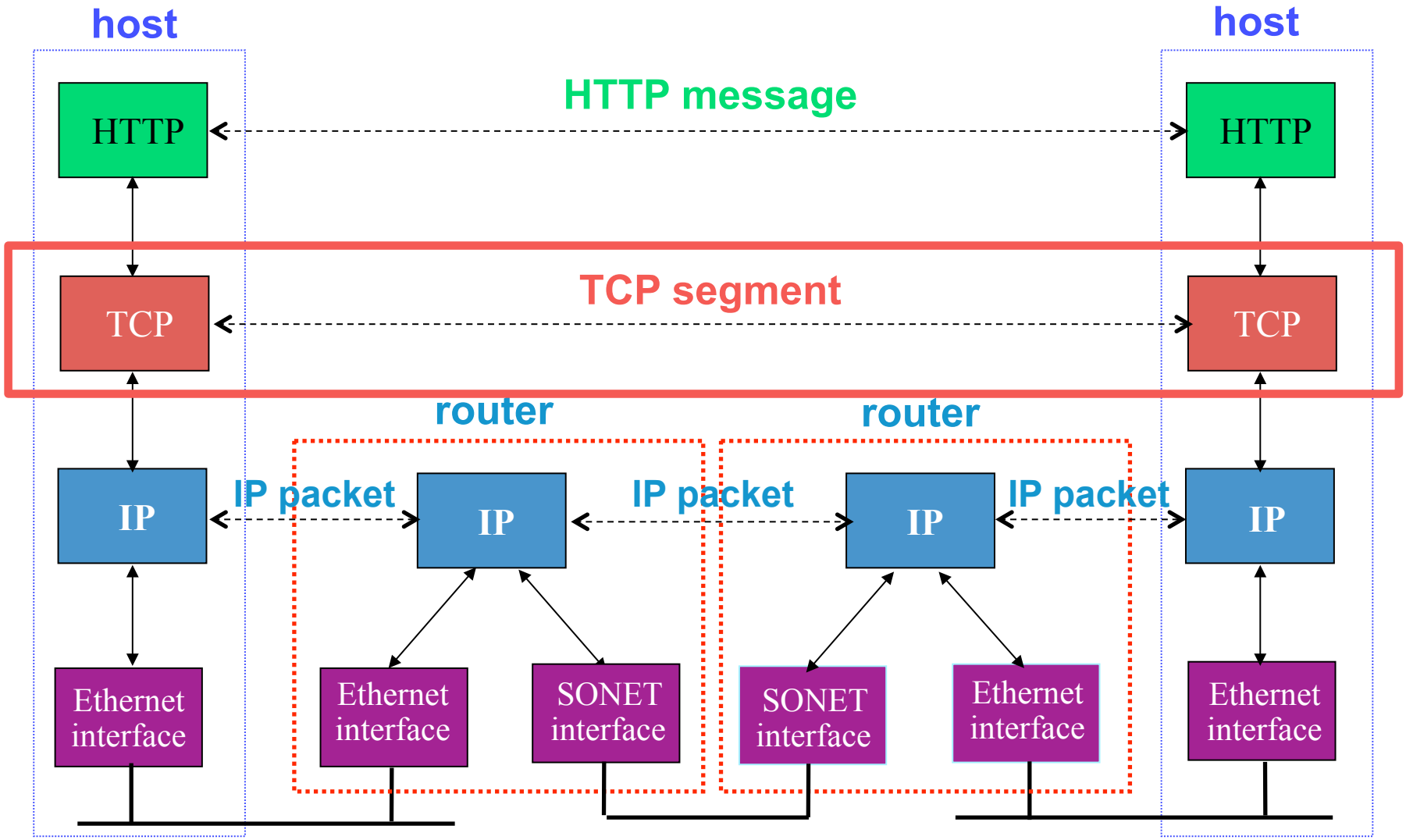
http://www.cs.princeton.edu/courses/archive/spring11/cos461/

Goals for Today's Lecture

- Principles underlying transport-layer services
  - (De)multiplexing
  - Detecting corruption
  - Reliable delivery
  - Flow control

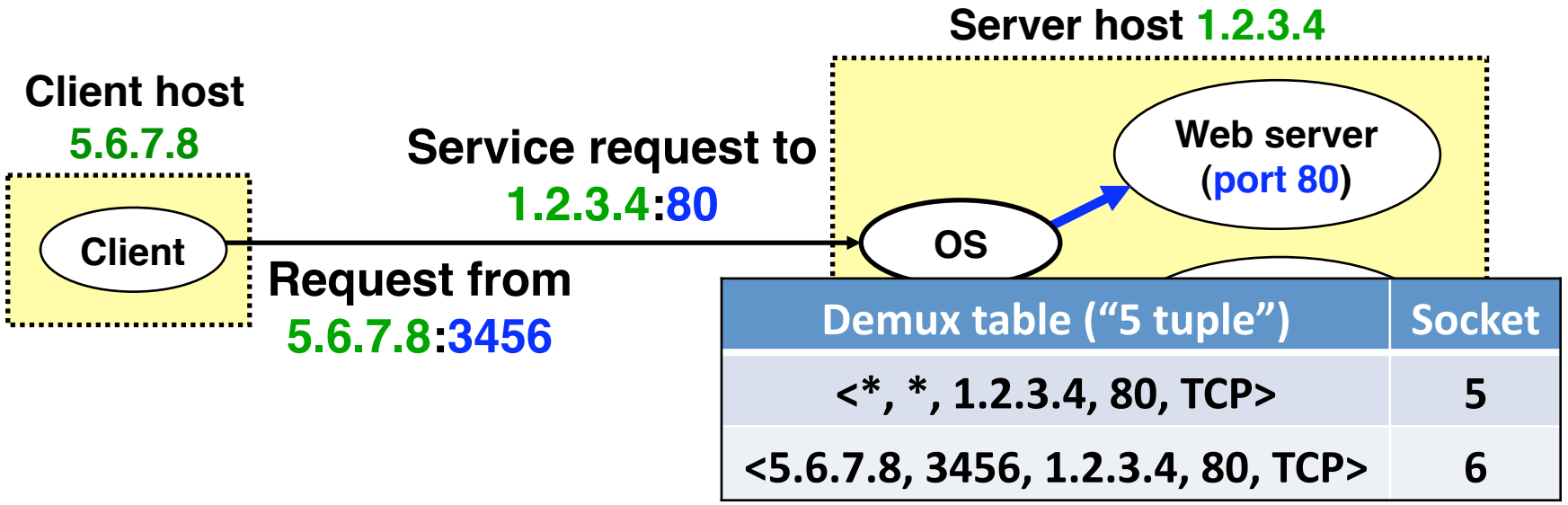
- Transport-layer protocols in the Internet
  - User Datagram Protocol (UDP)
    - Simple (unreliable) message delivery
  - Transmission Control Protocol (TCP)
    - Reliable bidirectional stream of bytes

Recall the Internet layering model

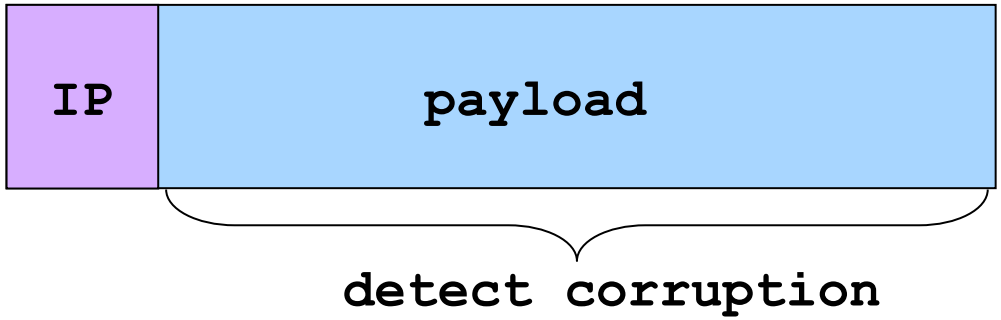


Two Basic Transport Features

- **Demultiplexing:** port numbers

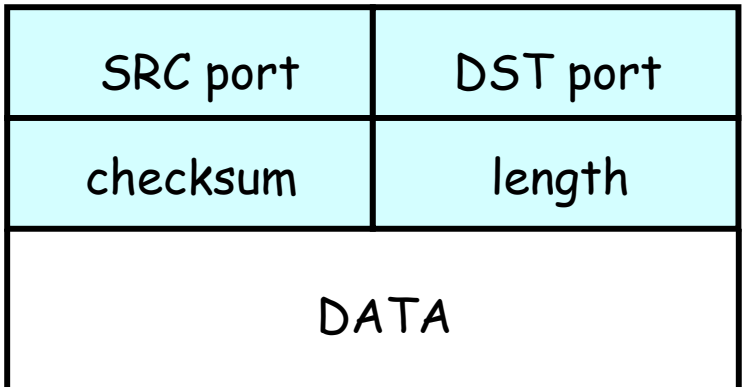


- **Error detection:** checksums



User Datagram Protocol (UDP)

- Datagram messaging service
  - Demultiplexing of messages: port numbers
  - Detecting corrupted messages: checksum
- Lightweight communication between processes
  - Send messages to and receive them from a socket
  - Avoid overhead and delays of ordered, reliable delivery



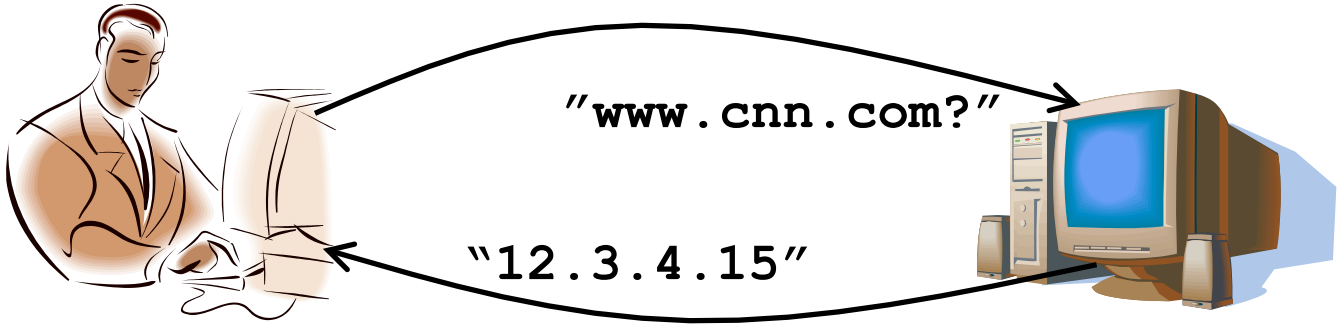
Why Would Anyone Use UDP?

- Fine control over what data is sent and when
  - As soon as app process writes into socket
  - … UDP will package data and send packet
- No delay for connection establishment
  - UDP blasts away without any formal preliminaries
  - … avoids introducing unnecessary delays
- No connection state (no buffers, sequence #'s, etc.)
  - Can scale to more active clients at once
- Small packet header overhead (header only 8B long)

Popular Applications that use UDP

END TO END PRINCIPLE !

- Simple query protocols like DNS
  - Overhead of connection establishment is overkill
  - Easier to have the application retransmit if needed



- Multimedia streaming (VoIP, video conferencing, …)
  - Retransmitting lost/corrupted packets is not worthwhile
  - By time packet is retransmitted, it's too late

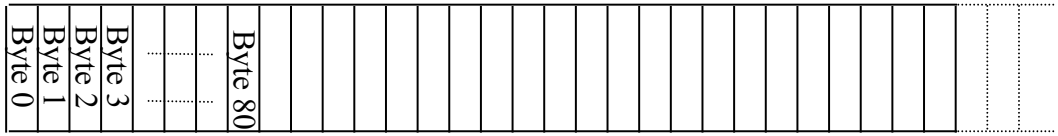
Transmission Control Protocol (TCP)

- Stream-of-bytes service: Send/recv streams, not msgs
- Reliable, in-order delivery
  - Checksums to detect corrupted data
  - Sequence numbers to detect losses and reorder data
  - Acknowledgments & retransmissions for reliable delivery
- Connection oriented: Explicit set-up and tear-down
- Flow control: Prevent overload of receiver's buffer
- Congestion control (next class!): Adapt for greater good

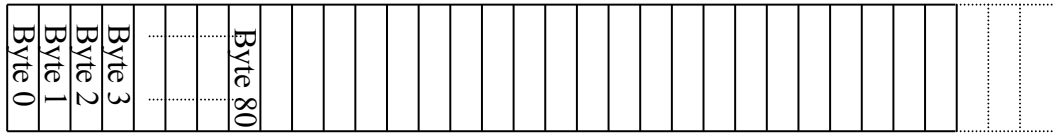
Breaking a Stream of Bytes into TCP Segments

TCP “Stream of Bytes” Service

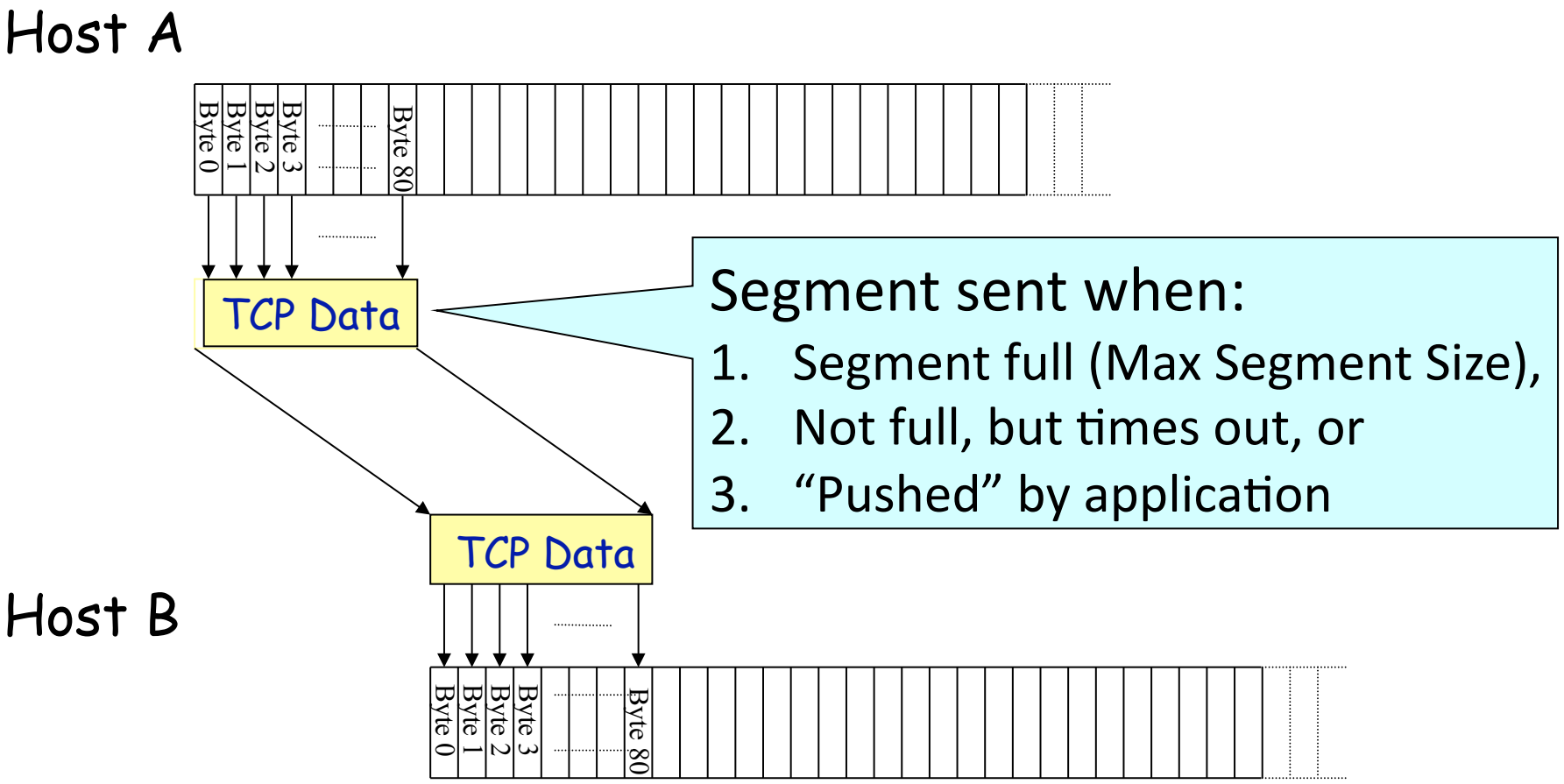
Host A



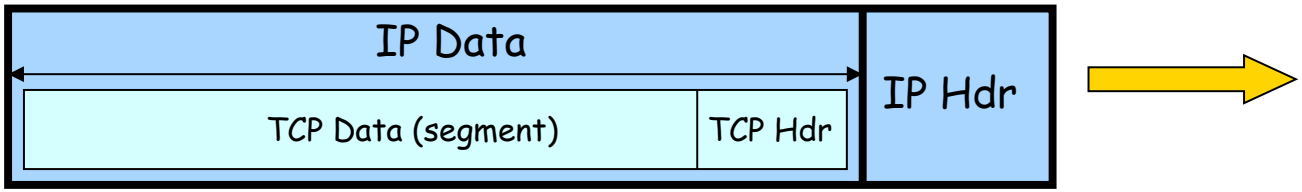
Host B



… Emulated Using TCP "Segments"

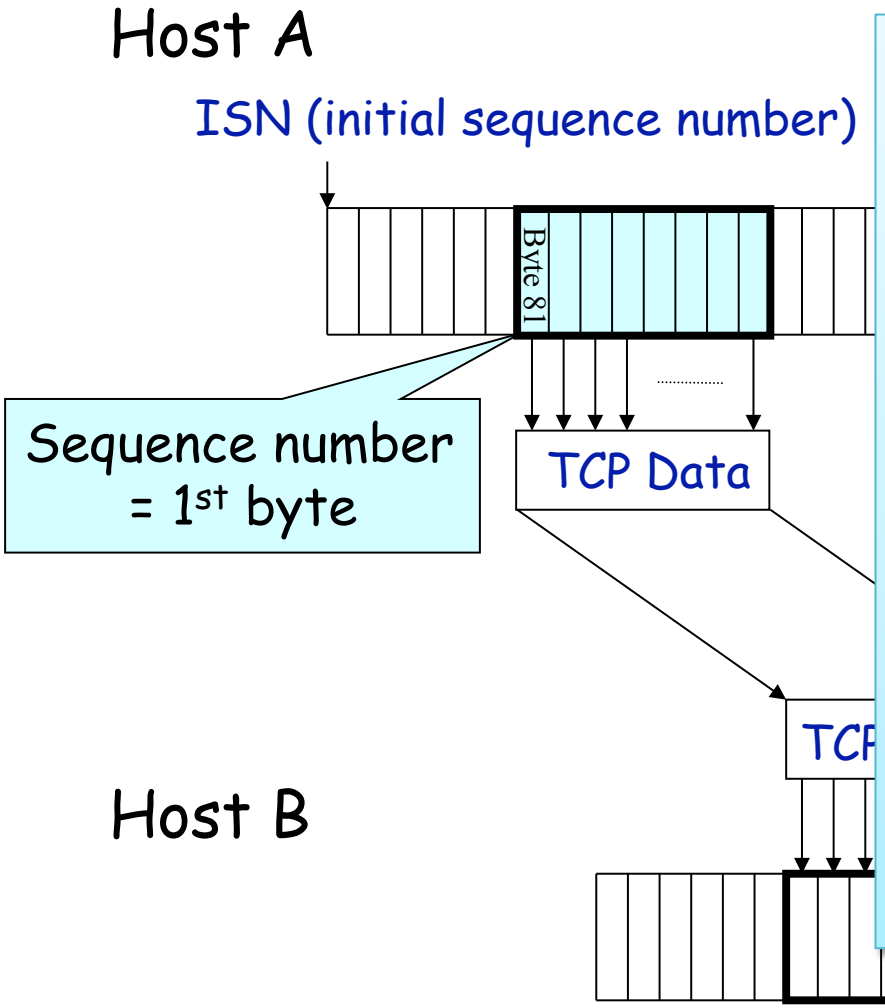


TCP Segment



- 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 Number



- Why not ISN of 0?
- Traditionally: Clock-based
  - What if port realloc'd and old packet in flight?
  - ISN based on clock, wraps around every 4.55 hr
- Now: Randomized
  - Hard to guess, thus harder to "hijack" TCP connection

Reliable Delivery on a Lossy Channel With Bit Errors

An Analogy: Talking on a Cell Phone

- Alice and Bob talking on cell phones
  - If Bob couldn't understand? Ask Alice to repeat
  - If Bob hasn't heard for a while?
- Acknowledgments from receiver
  - Positive: "okay" or "uh huh" or "ACK"
  - Negative: "please repeat that" or "NACK"
- Timeout by the sender ("stop and wait")
  - Don't wait indefinitely w/o a response, whether ACK or NACK
- Retransmission by the sender
  - After receiving "NACK" from receiver
  - After receiving no feedback from receiver

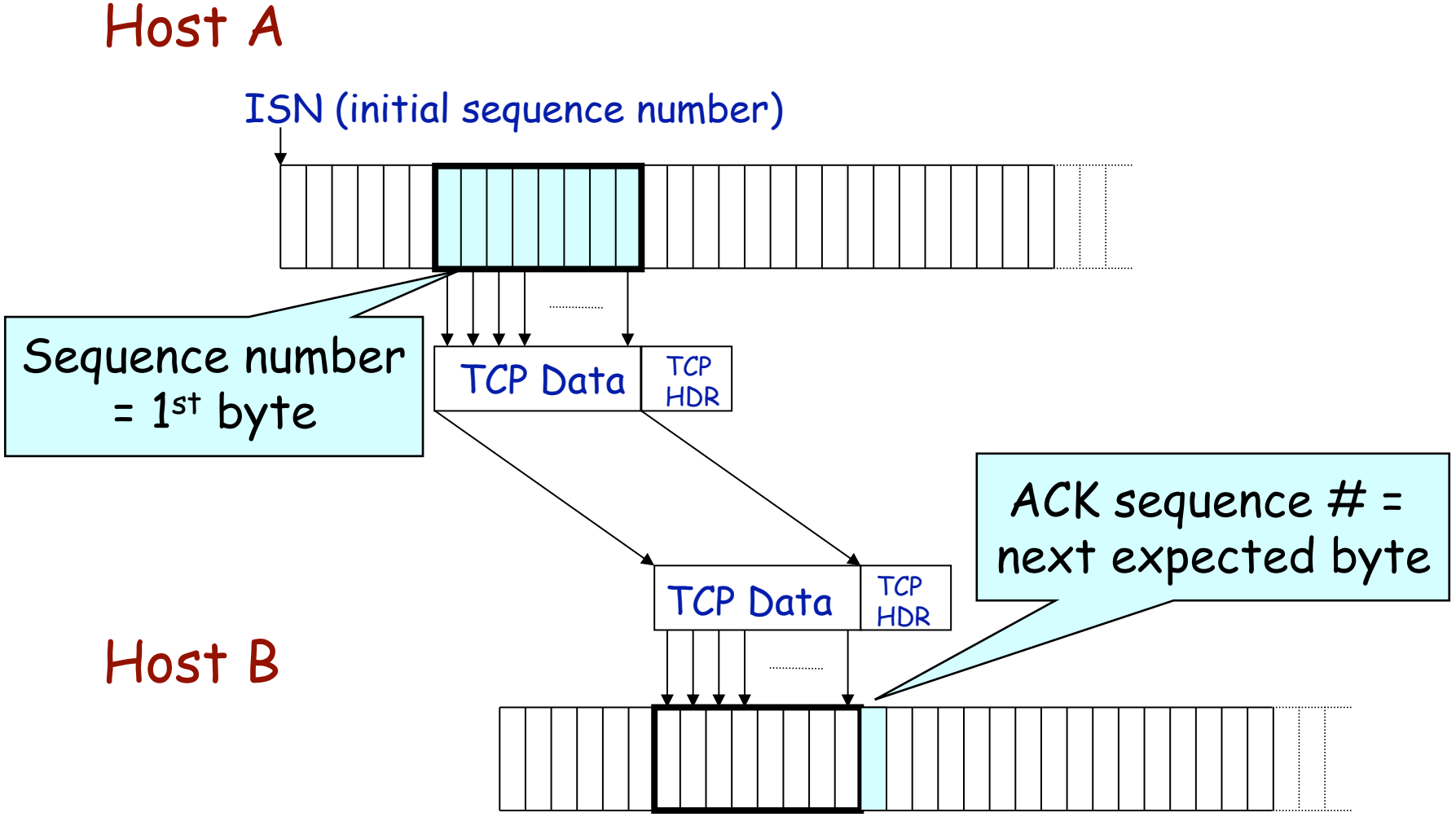
Challenges of Reliable Data Transfer

- Over a perfectly reliable channel
  - All data arrives in order, just as sent. Simple.
- Over a channel with *bit errors*
  - All data arrives in order, but some bits corrupted
  - Receiver detects errors and says "please repeat that"
  - Sender retransmits corrupted data
- Over a *lossy* channel with *bit errors*
  - Some data is 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

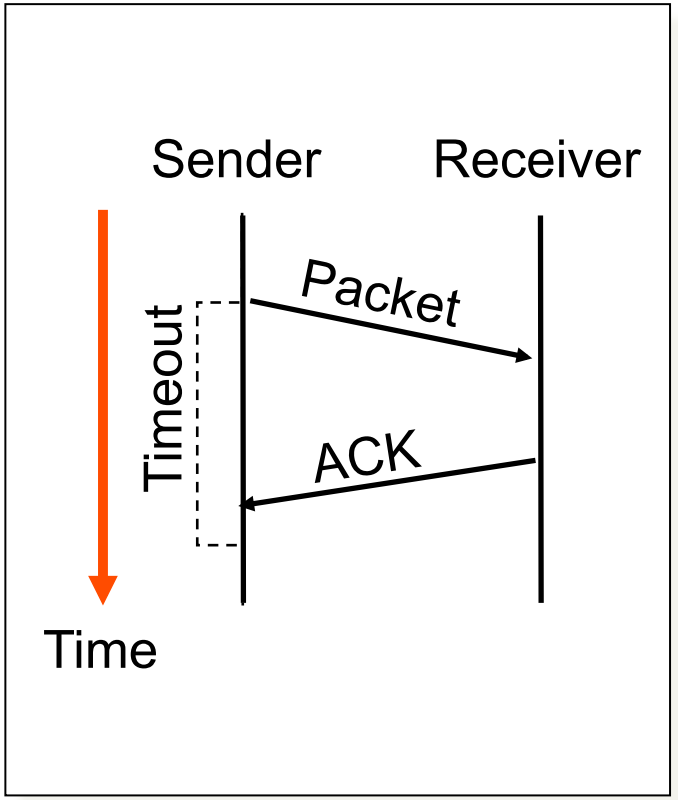
- **Detect bit errors:** checksum
  - Used to detect corrupted data at the receiver
  - …leading the receiver to drop the packet
- **Detect missing data:** sequence number
  - Used to detect a gap in the stream of bytes
  - ... and for putting the data back in order
- **Recover from lost data:** retransmission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

TCP Acknowledgments

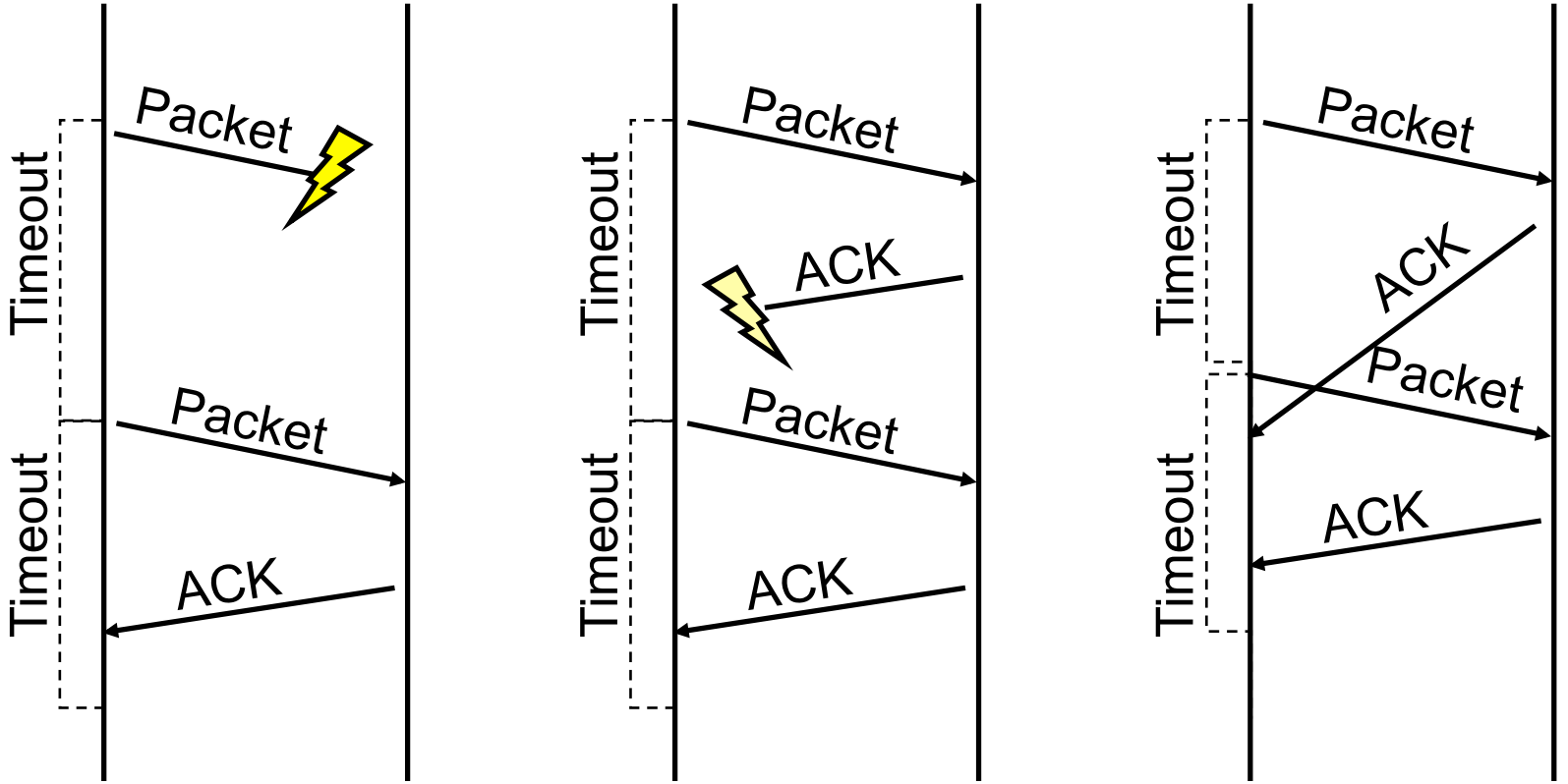


Automatic Repeat reQuest (ARQ)

- Automatic Repeat reQuest
  - Receiver sends ACK when it receives packet
  - Sender waits for ACK.
  - If ACK not received within some timeout period, resend packet
- Simplest: "stop and wait"
  - One packet at a time…



Reasons for Retransmission



Packet lost

ACK lost
DUPLICATE
PACKET

Early timeout
DUPLICATE
PACKETS

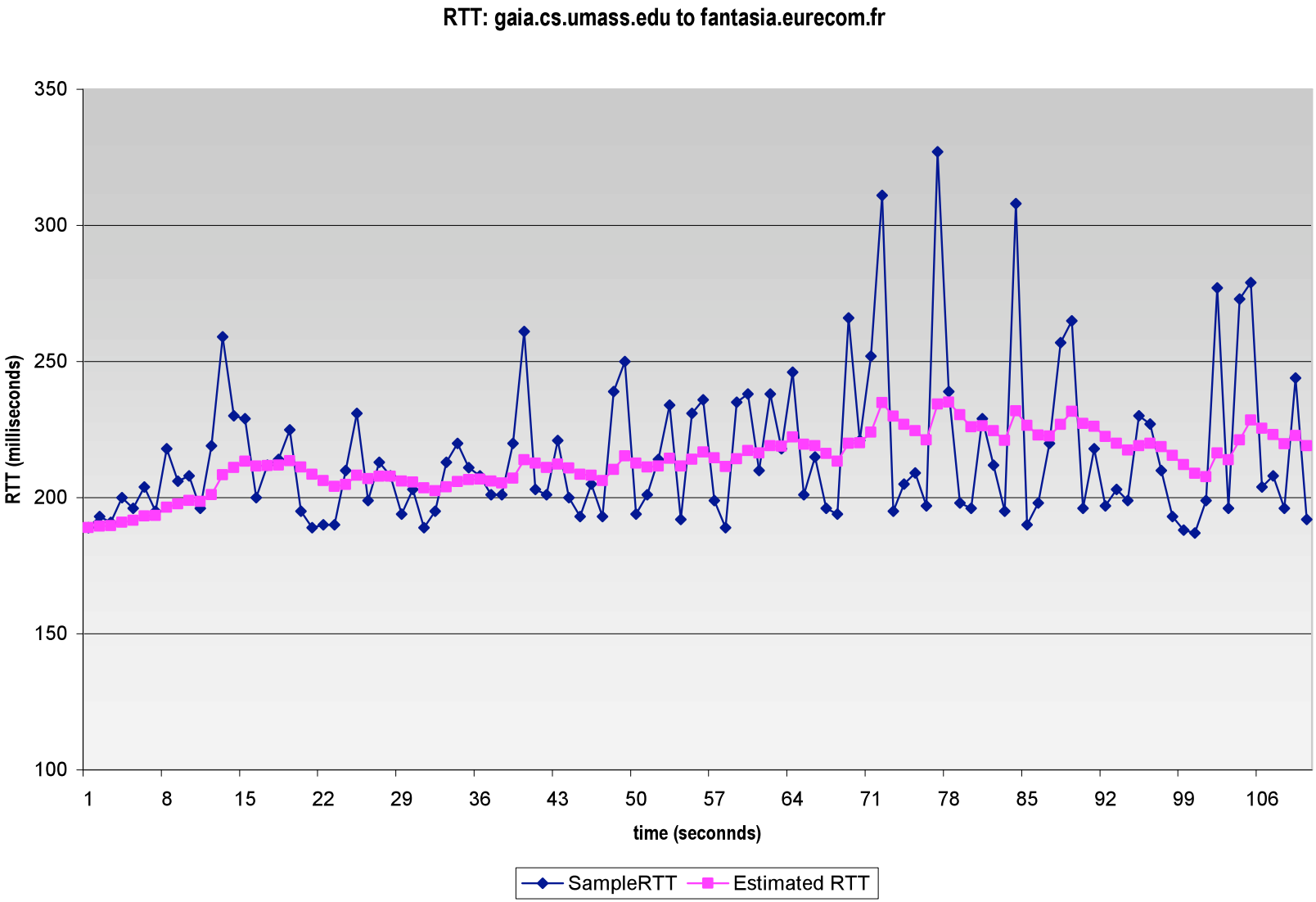
How Long Should Sender Wait?

- Too short? Wasted retransmissions
- Too long? Excessive delays when packet lost

- TCP sets timeout as function of Round Trip Time
  - ACK should arrive after RTT + fudge factor for queuing

- How does sender know RTT?
  - Can estimate RTT by watching the ACKs
  - Smooth estimate: Exponentially-weighted moving avg (EWMA)
    - EstimatedRTT = a * EstimatedRTT + (1–a) * SampleRTT
  - Compute timeout: TimeOut = 2 * EstimatedRTT

Example RTT Estimation



A Flaw in This Approach

- ACK acknowledges receipt to data, not transmission
- Consider a retransmission of a lost packet
  - If assume ACK with 1st transmission, SampleRTT too large
- Consider a duplicate packet
  - If assume ACK with 2nd transmission, SampleRTT too small
- Simple solution in the Karn/Partridge algorithm
  - Only collect samples for segments sent one single time

Increasing TCP throughput

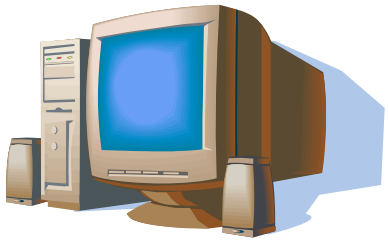
- Problem: Stop-and-wait + timeouts are inefficient
  - Only one TCP segment "in flight" at time
- Solution: Send multiple packets at once

- Problem: How many w/o overwhelming receiver?
- Solution: Determine "window size" dynamically

Flow Control via TCP Sliding Window

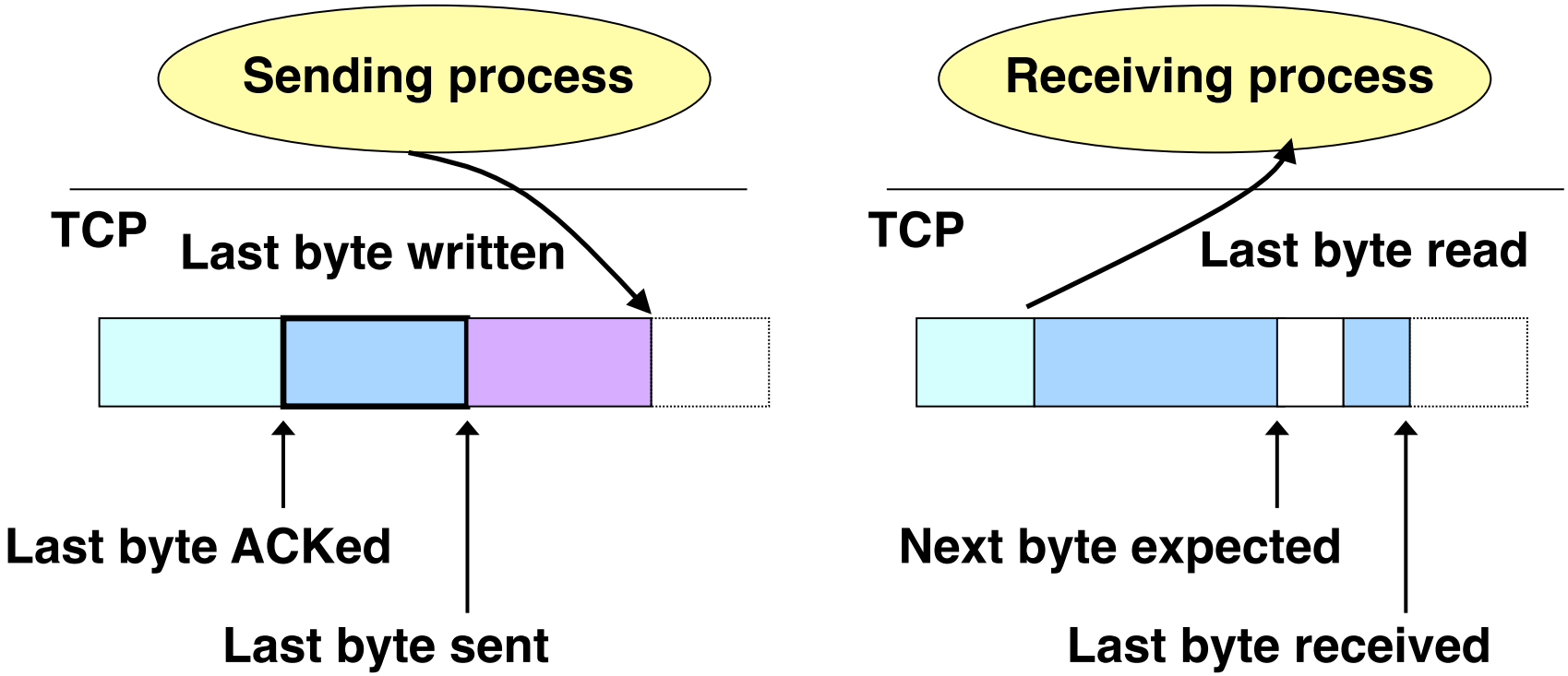
Motivation for Sliding Window

- 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/seg / 45 Msec/seg ➔ 182 Kbps
    - Just one-eighth of the 1.5 Mbps link capacity



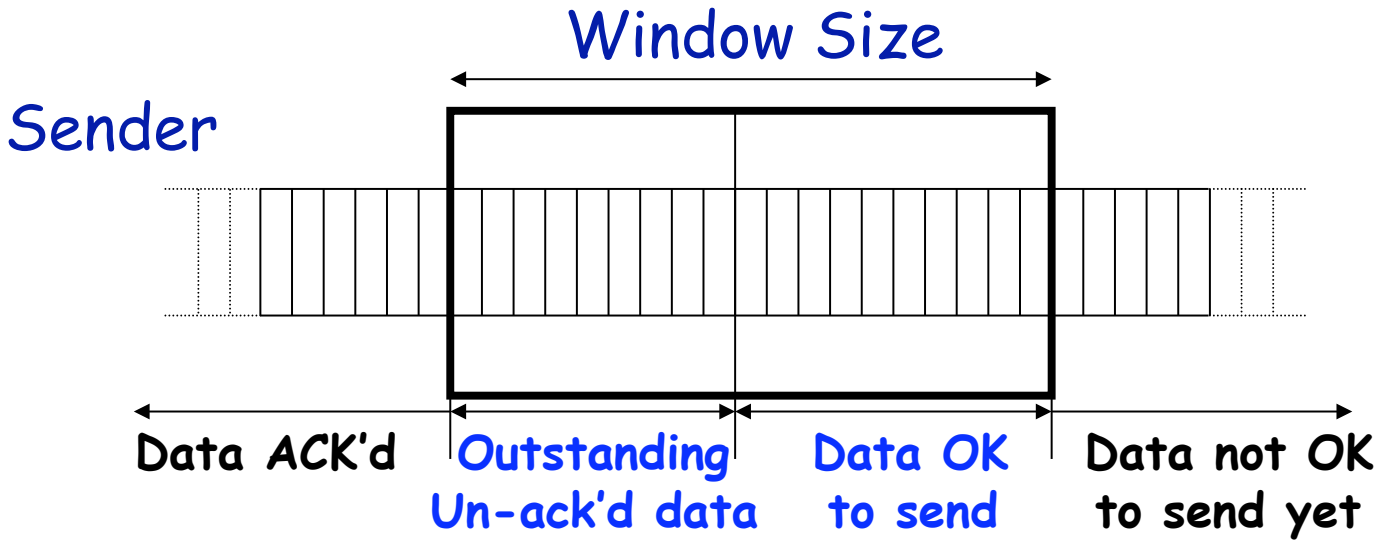
Sliding Window

- Allow a larger amount of data "in flight"
  - Sender can get ahead of receiver, though not *too far*



Receiver Buffering

- Window size
  - Amount that can be sent w/o ACK, because receiver can buffer
- Receiver advertises window to receiver
  - Tells amount of free space left (in **bytes**)
  - Sender agrees not to exceed this amount



Solution to timeout inefficiency: Fast Retransmission

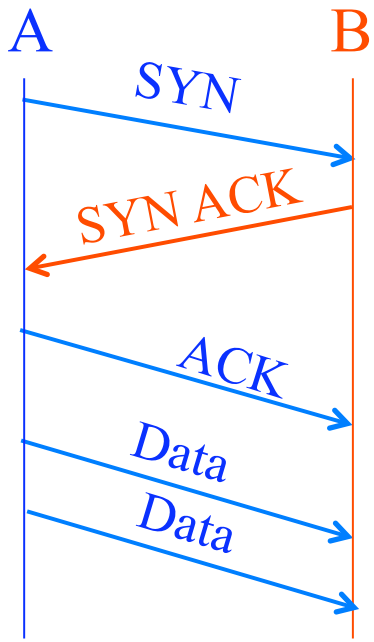
- Better solution possible under sliding window
  - Although packet n might have been lost
  - … packets n+1, n+2, and so on might get through
- Idea: "Duplicate ACKs" suggest loss
  - ACK says receiver is awaiting n^{th} packet
  - *Repeated* ACKs suggest later packets have arrived
  - Sender use "duplicate ACKs" as early hint of loss
- Fast retransmission
  - Sender retransmits data after the triple duplicate ACK

Effectiveness of Fast Retransmit

- When does Fast Retransmit work best?
  - Long transfers: High likelihood of many pkts in flight
  - Large window: High likelihood of many packets in flight
  - Low loss burstiness: Higher likelihood that later pkts arrive
- Implications for Web traffic
  - Most Web objects are short (e.g., 10 packets)
  - So, often aren't many packets in flight
  - … making fast retransmit less likely to "kick in"
  - … another reason for persistent connections!

Starting and Ending a Connection: TCP Handshakes

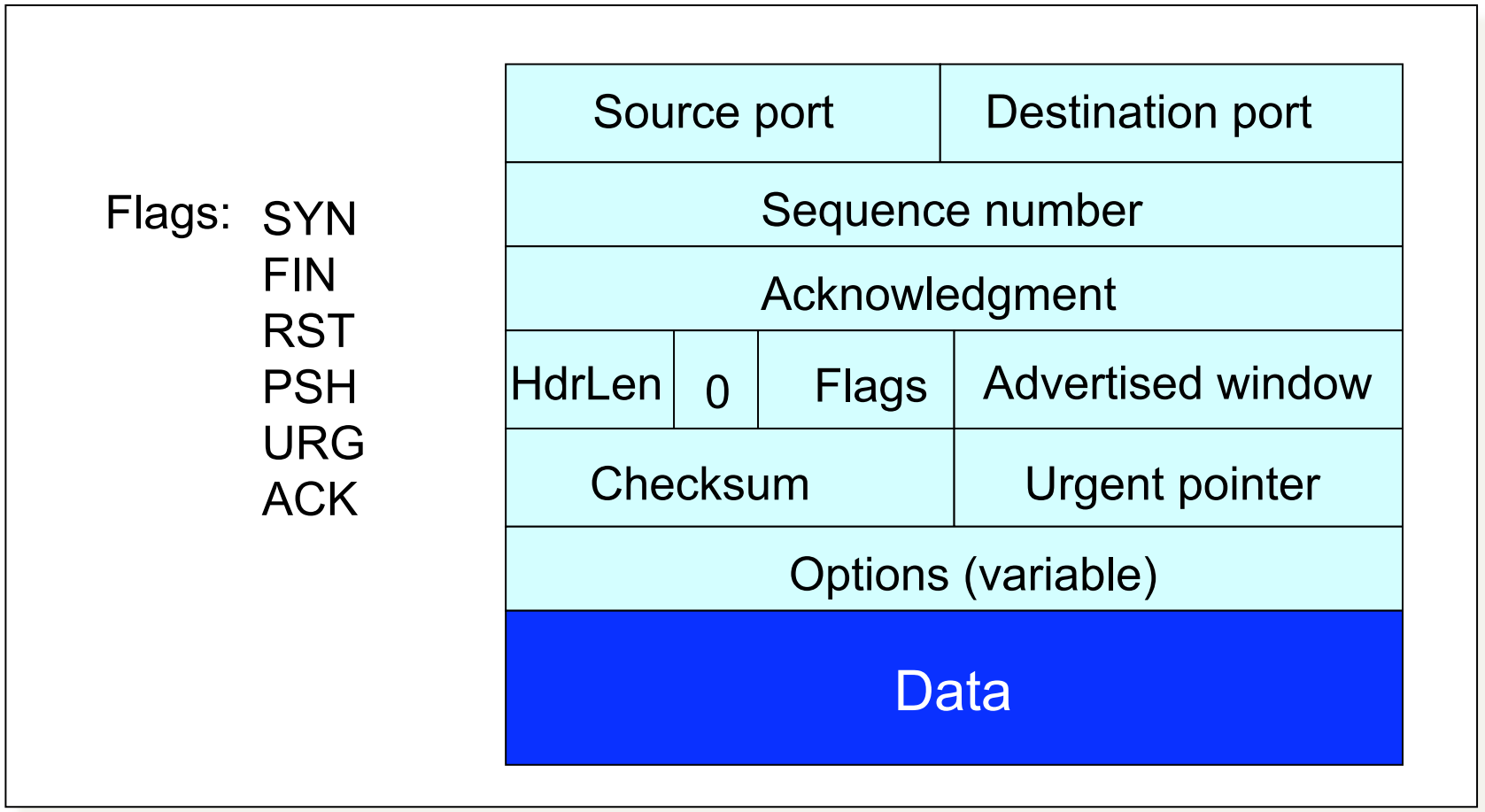
Establishing a TCP Connection



Each host tells its ISN to other host

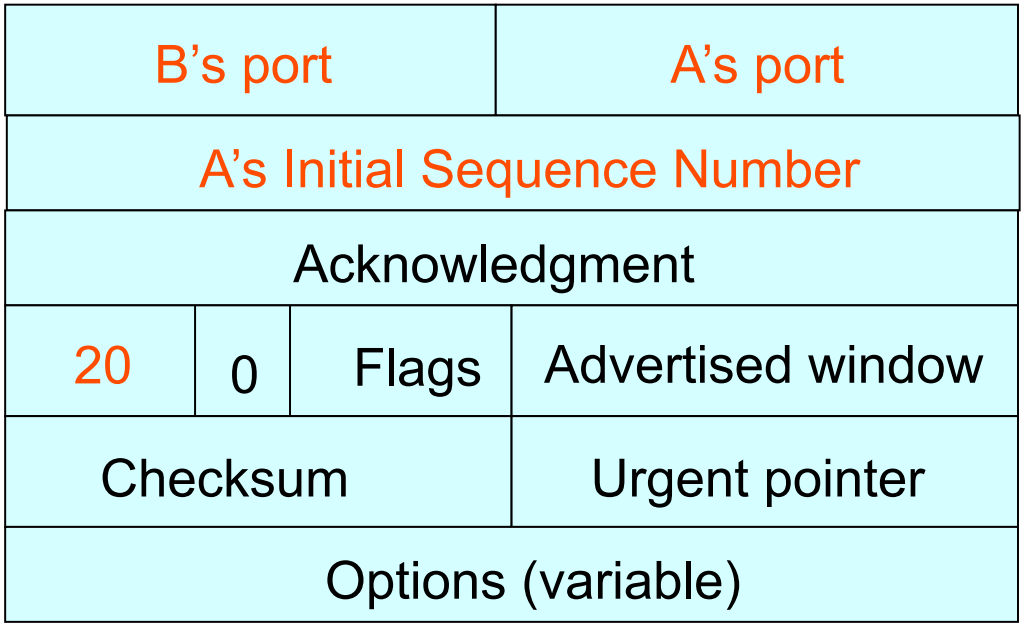
- Three-way handshake to establish connection
  - Host A sends a **SYN**chronize (open) to the host B
  - Host B returns a SYN ACKnowledgment (**SYN ACK**)
  - Host A sends an **ACK** to acknowledge the SYN ACK

TCP Header


Flags:
SYN
FIN
RST
PSH
URG
ACK
Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

Step 1: A's Initial SYN Packet

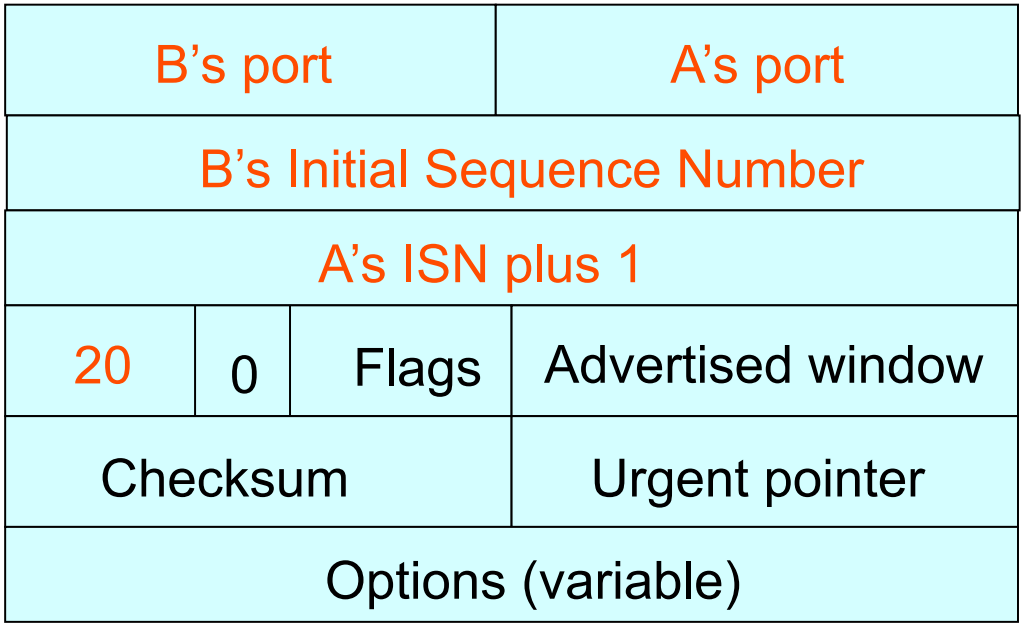
Flags: SYN
FIN
RST
PSH
URG
ACK



A tells B it wants to open a connection…

Step 2: B's SYN-ACK Packet

Flags: SYN
FIN
RST
PSH
URG
ACK



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

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

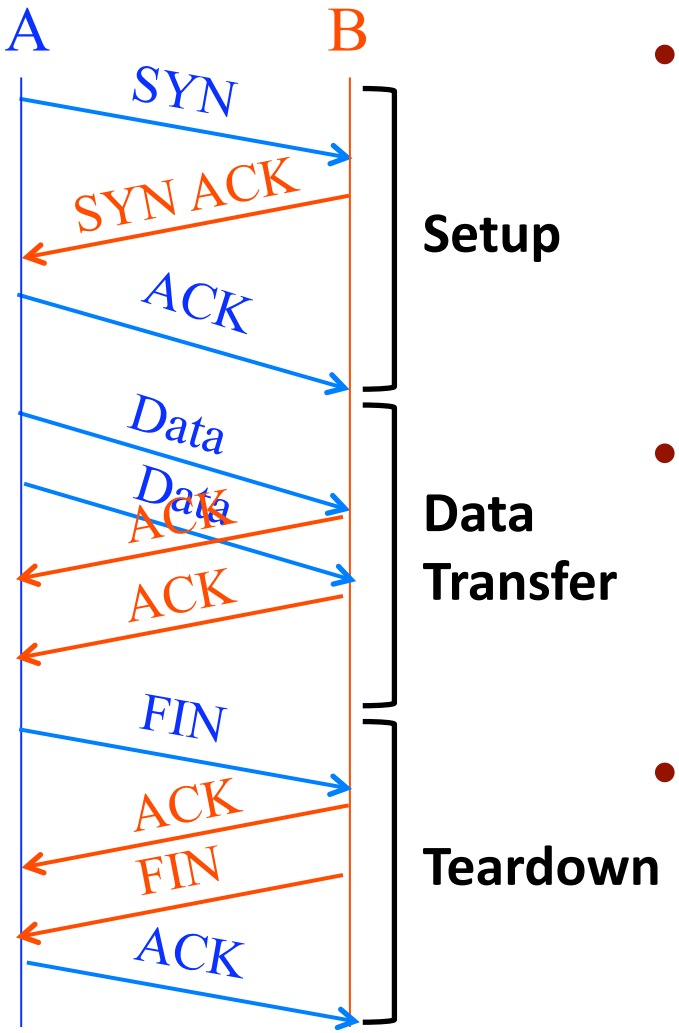
A tells B it is okay to start sending…

… upon receiving this packet, B can start sending data

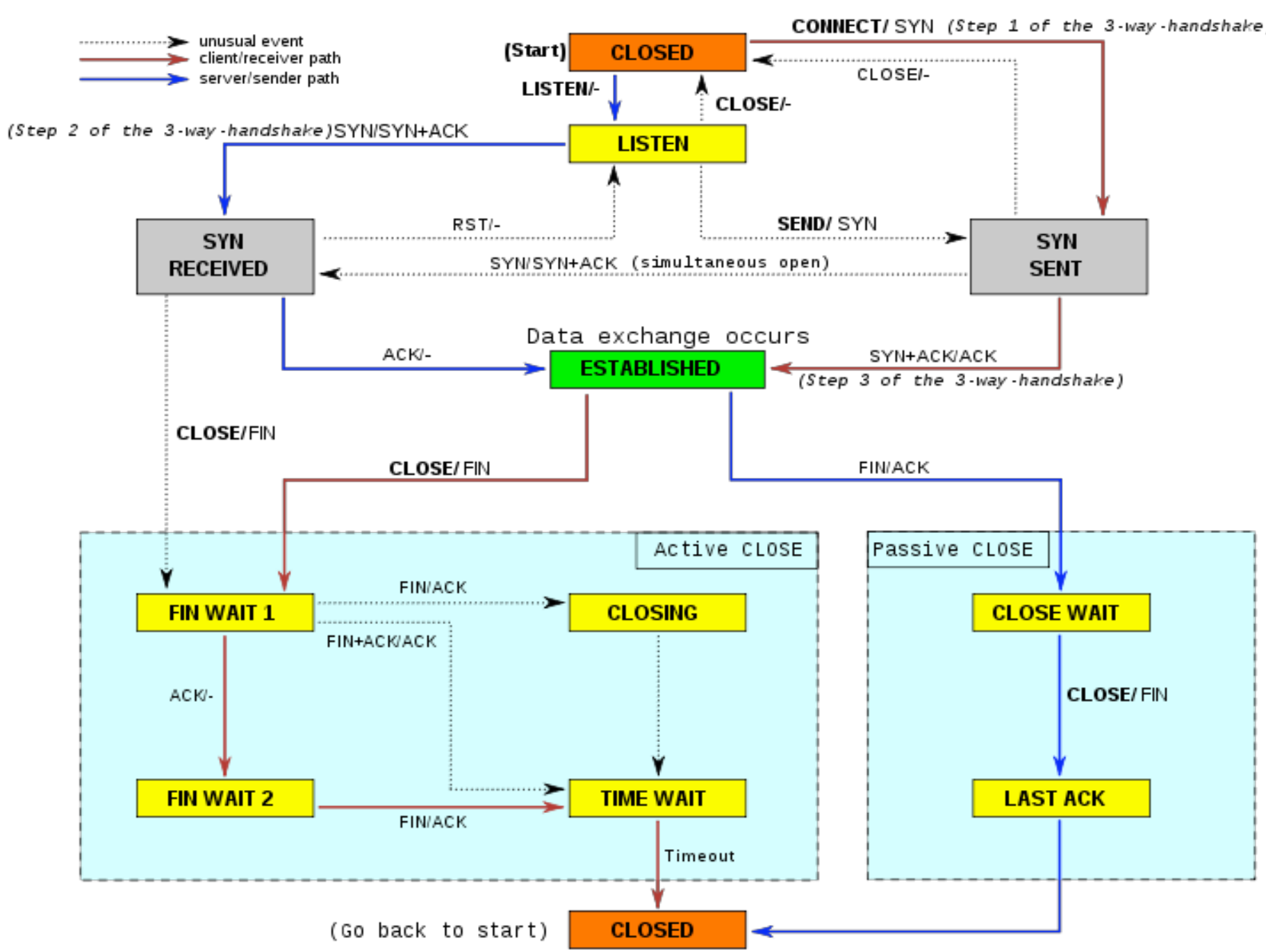
What if the SYN Packet Gets Lost?

- Suppose SYN packet gets lost
  - Packet lost inside network, or
  - Server rejects packet (e.g., listen queue is full)
- Eventually, no SYN-ACK arrives
  - Sender sets timer and waits, retransmitting if needed
- How should TCP sender set timer?
  - Sender has no estimate of RTT
  - Some TCP stacks use default of 3 or 6 seconds
    - Why reload button in your browser is useful:
      Opens new connection and "retransmits" SYN in < 3-6 sec

Tearing Down the Connection



- Closing a connection
  - Process done writing: invokes close()
  - Once TCP sends all outstanding byte, TCP sends a FINish message
- Receiving a FINish
  - Process reading data from socket
  - Eventually, read attempt returns EOF
- Tear-down is two-way
  - FIN to close, but receive remaining
  - Other host ACKs the FIN
  - Rest (RST) to close and not receive remaining: error condition


unusual event
client/receiver path
server/sender path
(Start)
CLOSED
CONNECT/ SYN (Step 1 of the 3-way-handshake)
CLOSE/-
LISTEN/-
CLOSE/-
(Step 2 of the 3-way-handshake)SYN/SYN+ACK
LISTEN
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/-
CLOSE/ FIN
FIN WAIT 2
TIME WAIT
LAST ACK
FIN/ACK
Timeout
(Go back to start)
CLOSED

Conclusions

- Transport protocols
  - Multiplexing and demultiplexing
  - Checksum-based error detection
  - Sequence numbers
  - Retransmission
  - Window-based flow control

- Next lecture
  - Congestion control