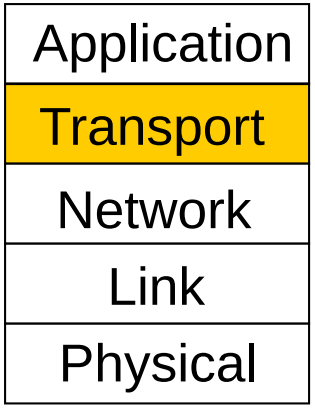
CSEP 561 – Connections

David Wetherall
djw@cs.washington.edu

Connections

- Focus
  - How do we (reliably) connect processes?
  - This is the transport layer

- Topics
  - Naming processes
  - Connection setup / teardown
  - Sliding window / Flow control



The Transport Layer

- Builds on the services of the Network layer
  - "TCP/IP"
- Communication between processes running on hosts
  - Naming/Addressing
- Stronger guarantees of message delivery make sense
  - Many applications want reliable connection and data transfer
  - This is the first layer that is talking "end-to-end"

Internet Transport Protocols

- UDP
  - Datagram abstraction between processes
  - With error detection

- TCP
  - Bytestream (bitpipe) abstraction between processes
  - With reliability
  - Plus congestion control (later!)

Comparison of TCP/UDP/IP properties

TCP

- Connection-oriented
- Reliable byte-stream
  - In-order delivery
  - Single delivery
  - Arbitrarily length
- Synchronization
- Flow control
- Congestion control

UDP

- Datagram oriented
- Lost packets
- Reordered packets
- Duplicate packets
- Limited size packets

IP

- Datagram oriented
- Lost packets
- Reordered packets
- Duplicate packets
- Limited size packets

Naming Processes/Services

- Process here is an abstract term for your Web browser (HTTP), Email servers (SMTP), hostname translation (DNS), RealAudio player (RTSP), etc.

- How do we identify for remote communication?
  - Process id or memory address are OS-specific and transient

- So TCP and UDP use Ports
  - 16-bit integers representing mailboxes that processes "rent"
  - Identify process uniquely as (IP address, protocol, port)

Picking Port Numbers

- We still have the problem of allocating port numbers
  - What port should a Web server use on host X?
  - To what port should you send to contact that Web server?

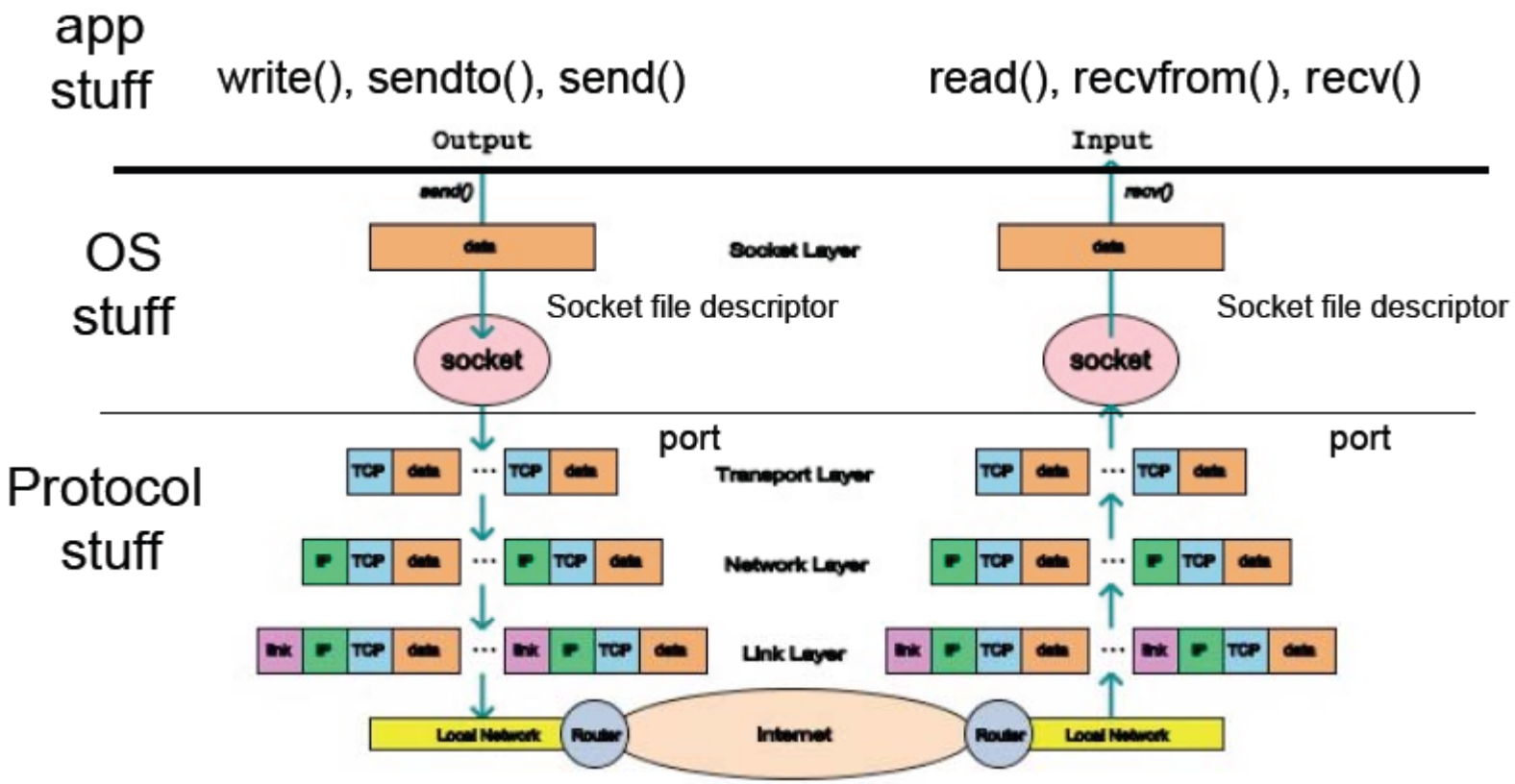
- Servers typically bind to "well-known" port numbers
  - e.g., HTTP 80, SMTP 25, DNS 53, … look in /etc/services
  - Ports below 1024 reserved for "well-known" services
- Clients use OS-assigned temporary (ephemeral) ports
  - Above 1024, recycled by OS when client finished

Berkeley Sockets

- Networking protocols implemented in OS
  - OS must expose a programming API to applications
  - most OSs use the “socket” interface
  - originally provided by BSD 4.1c in ~1982.

- Principle abstraction is a “socket”
  - a point at which an application attaches to the network
  - defines operations for creating connections, attaching to network, sending and receiving data, closing connections

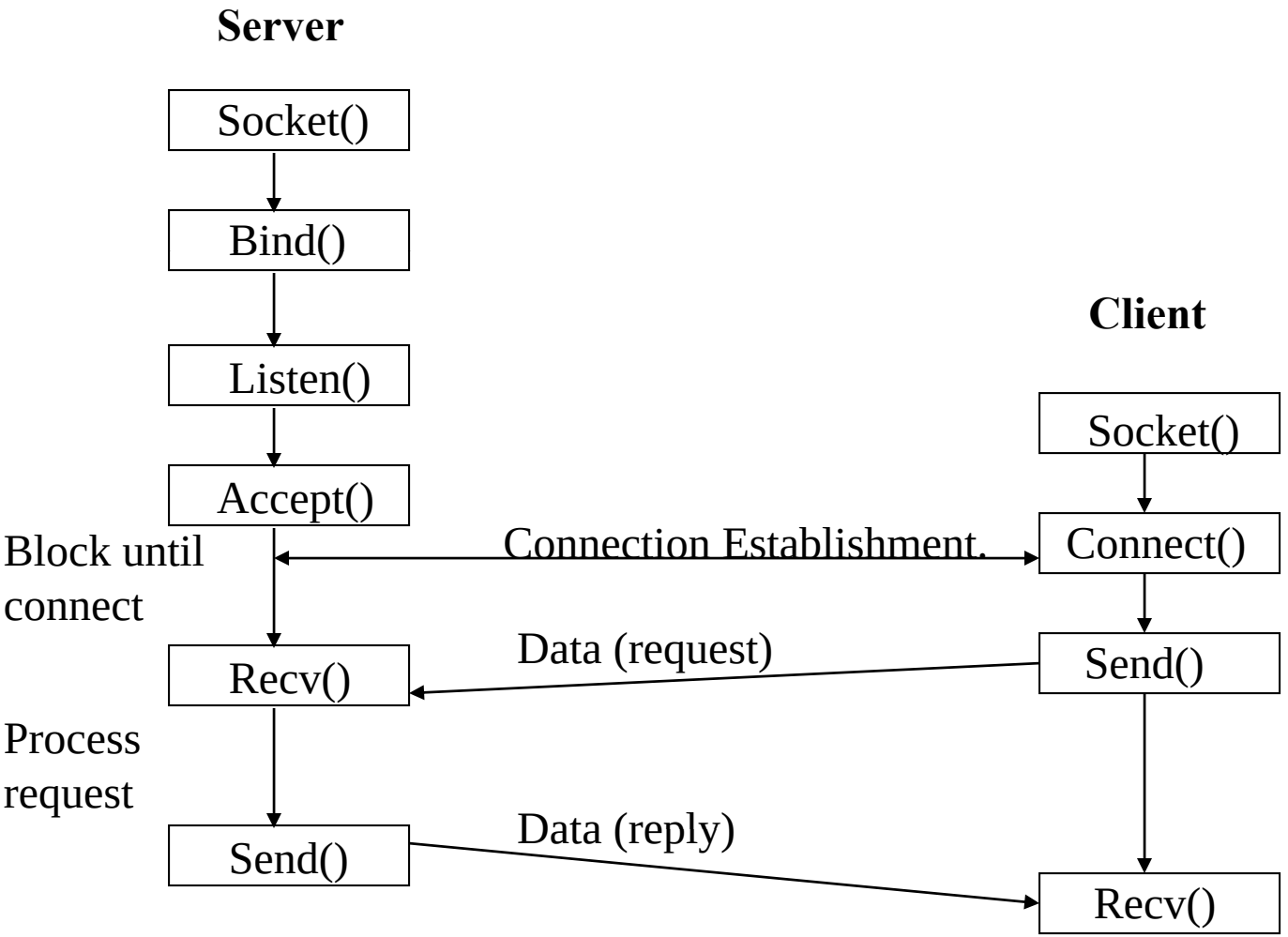
Overall pieces



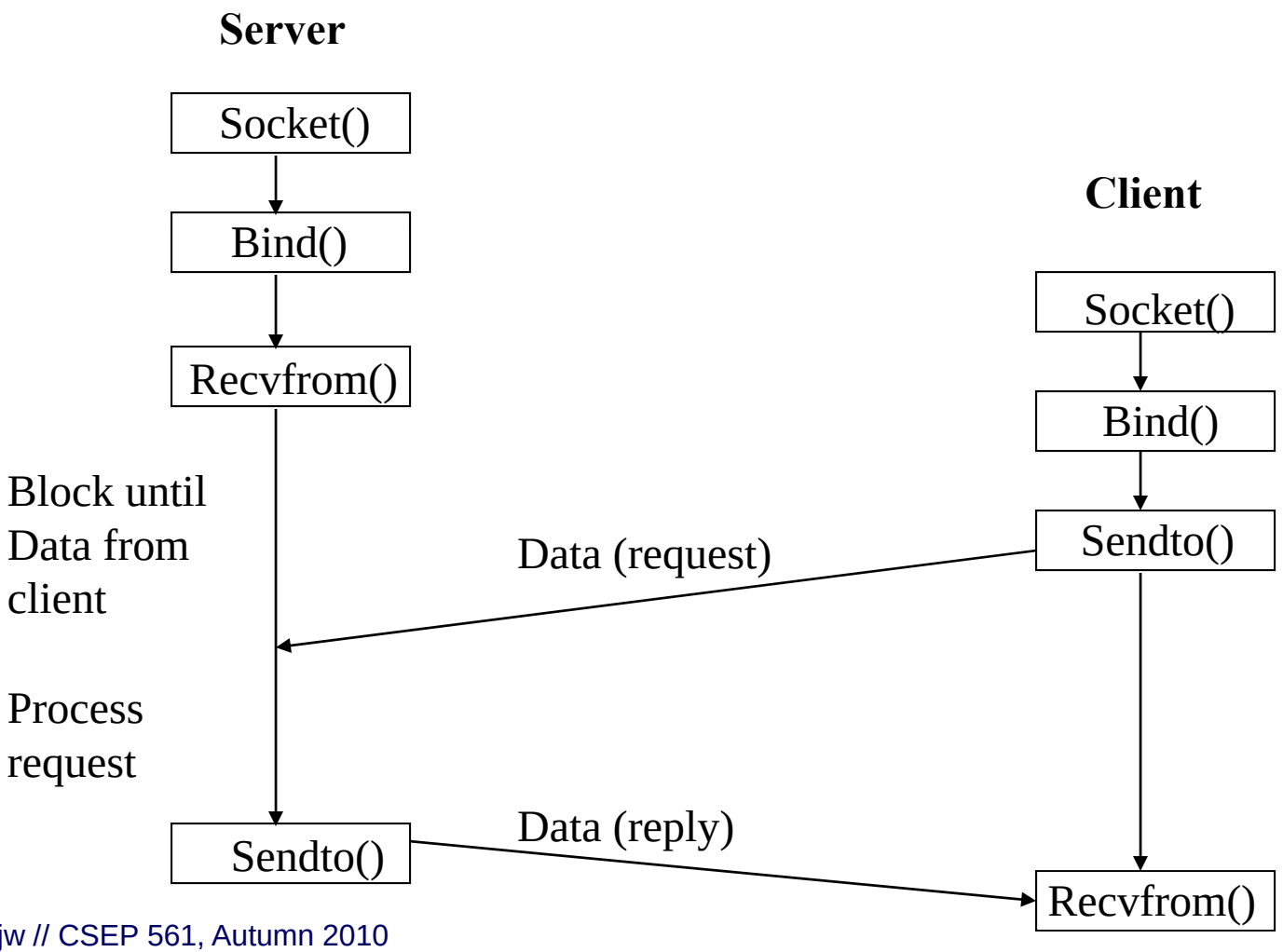
Berkeley Sockets API

Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Associate a local address with a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

TCP (connection-oriented)

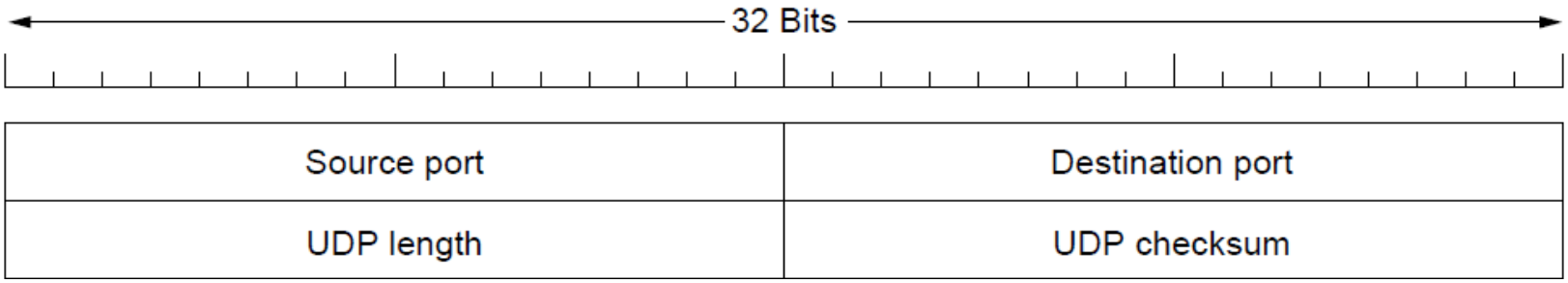


UDP (connectionless)

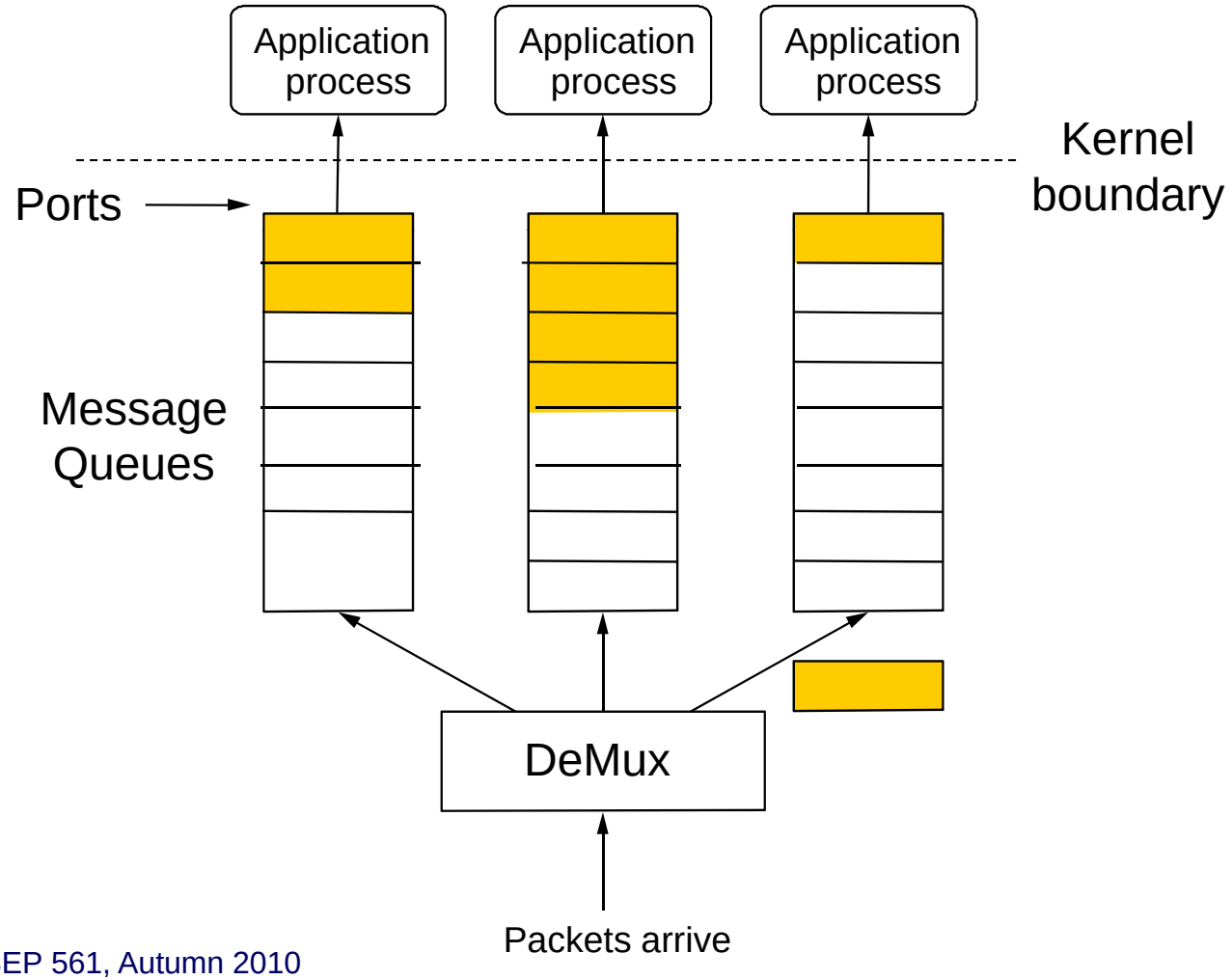


User Datagram Protocol (UDP)

- Provides message delivery between processes
  - Source port filled in by OS as message is sent
  - Destination port identifies UDP delivery queue at endpoint

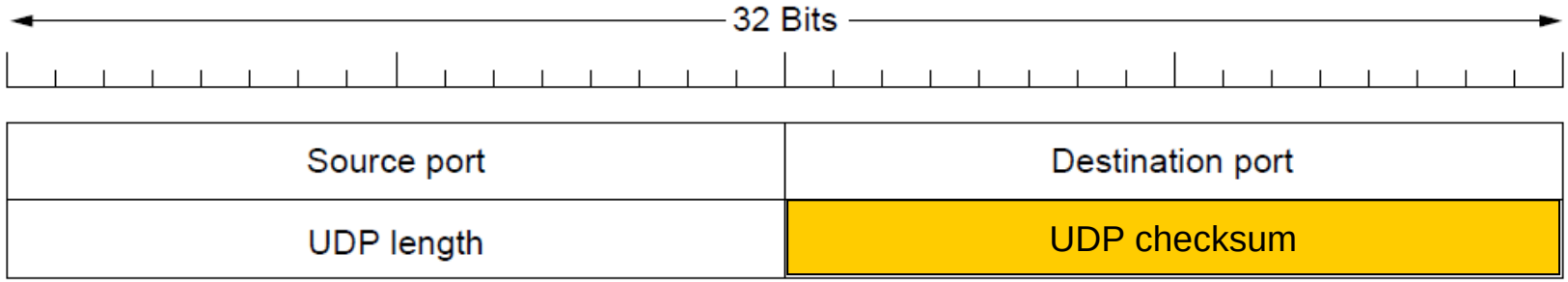


UDP Delivery



UDP Checksum

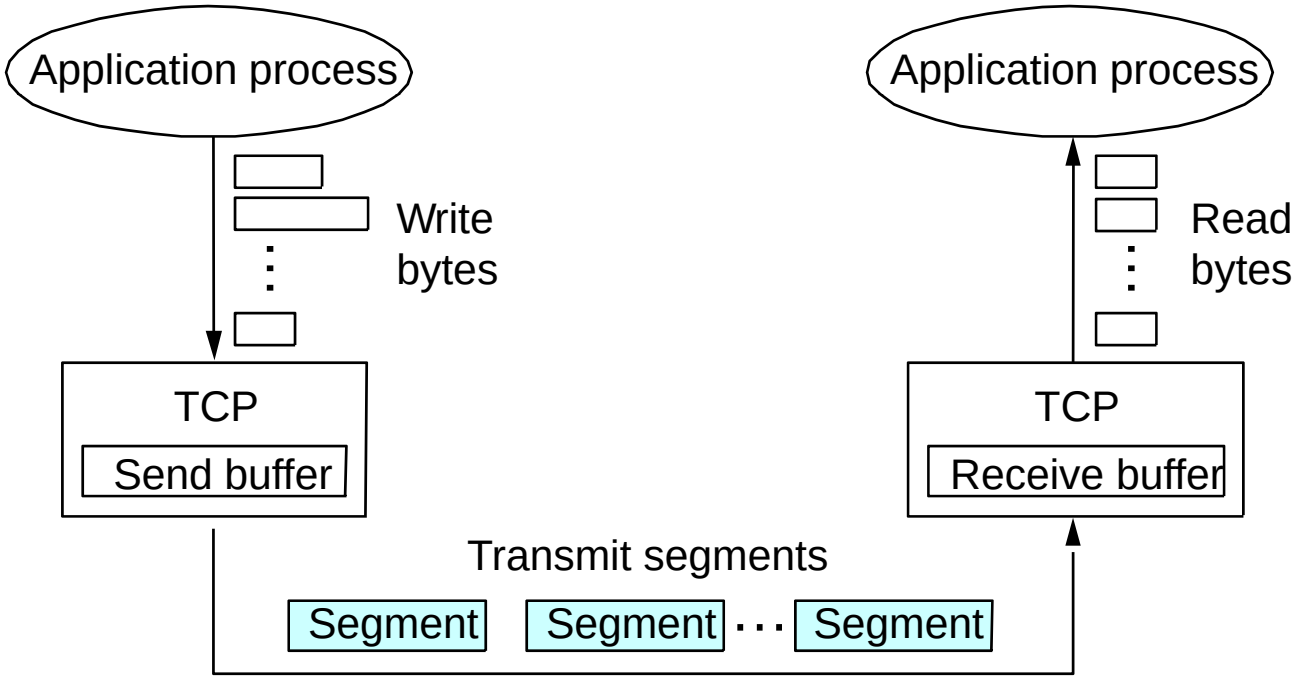
- UDP includes optional protection against errors
  - Checksum intended as an end-to-end check on delivery
  - So it covers data, UDP header, and IP pseudoheader



Transmission Control Protocol (TCP)

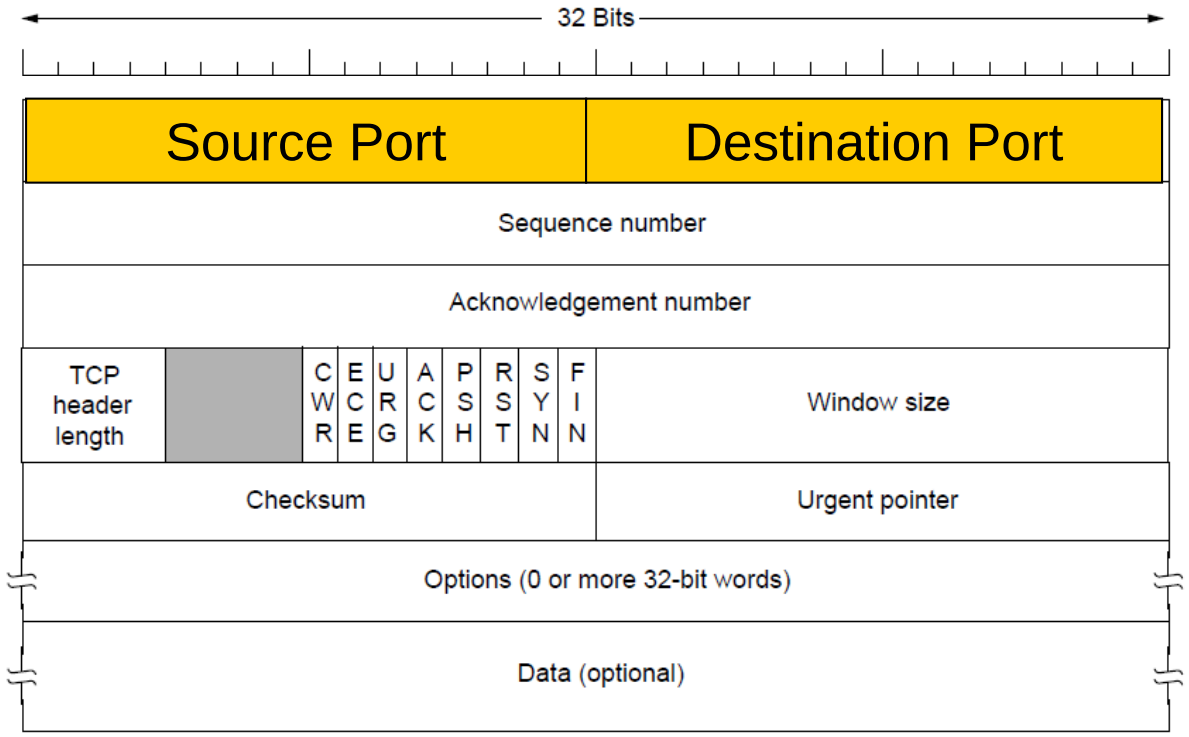
- Reliable bi-directional bytestream between processes
  - Message boundaries are not preserved
- Connections
  - Conversation between endpoints with beginning and end
- Flow control (later)
  - Prevents sender from over-running receiver buffers
- Congestion control (later)
  - Prevents sender from over-running network buffers

TCP Delivery



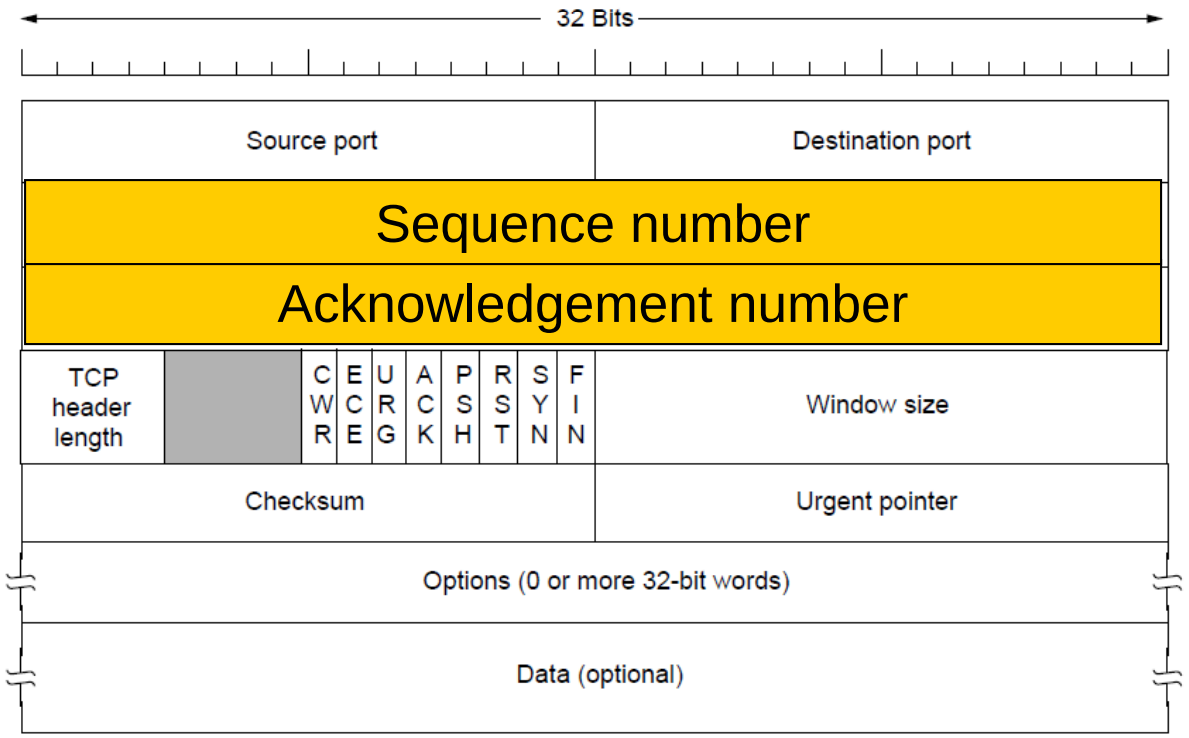
TCP Header Format

- Ports plus IP addresses identify a connection



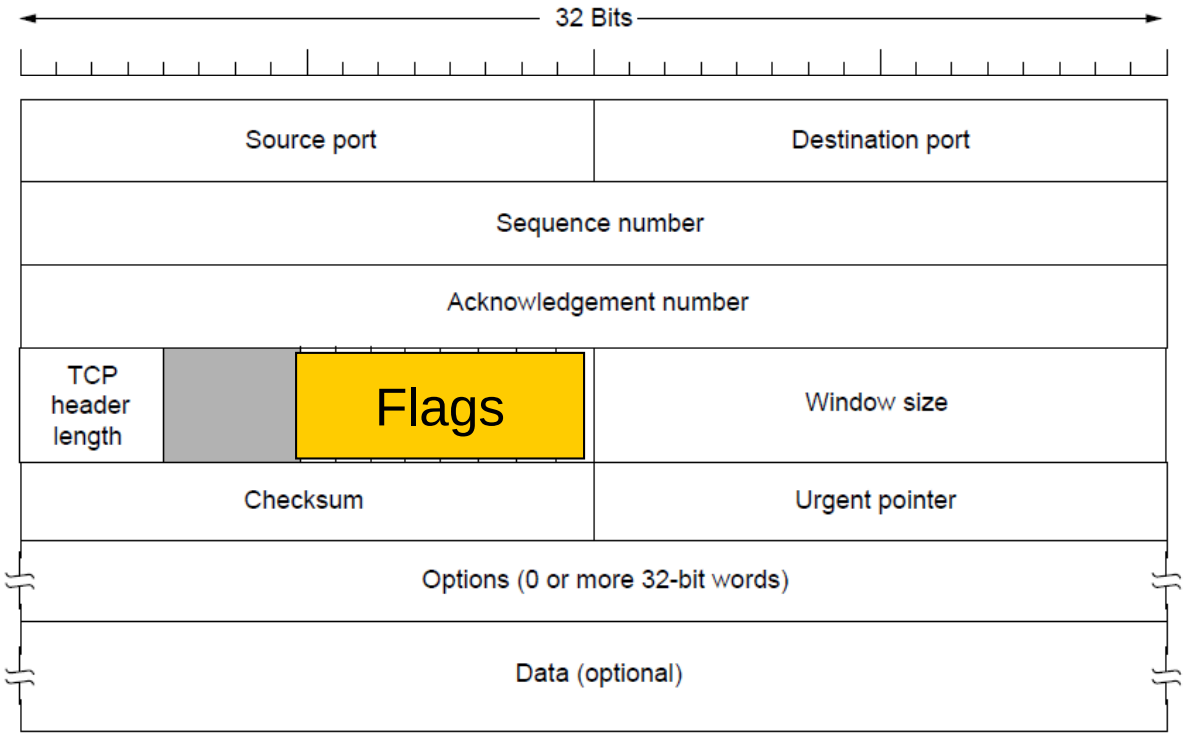
TCP Header Format

- Sequence, Ack numbers used for the sliding window
  - Congestion control works by controlling the window size



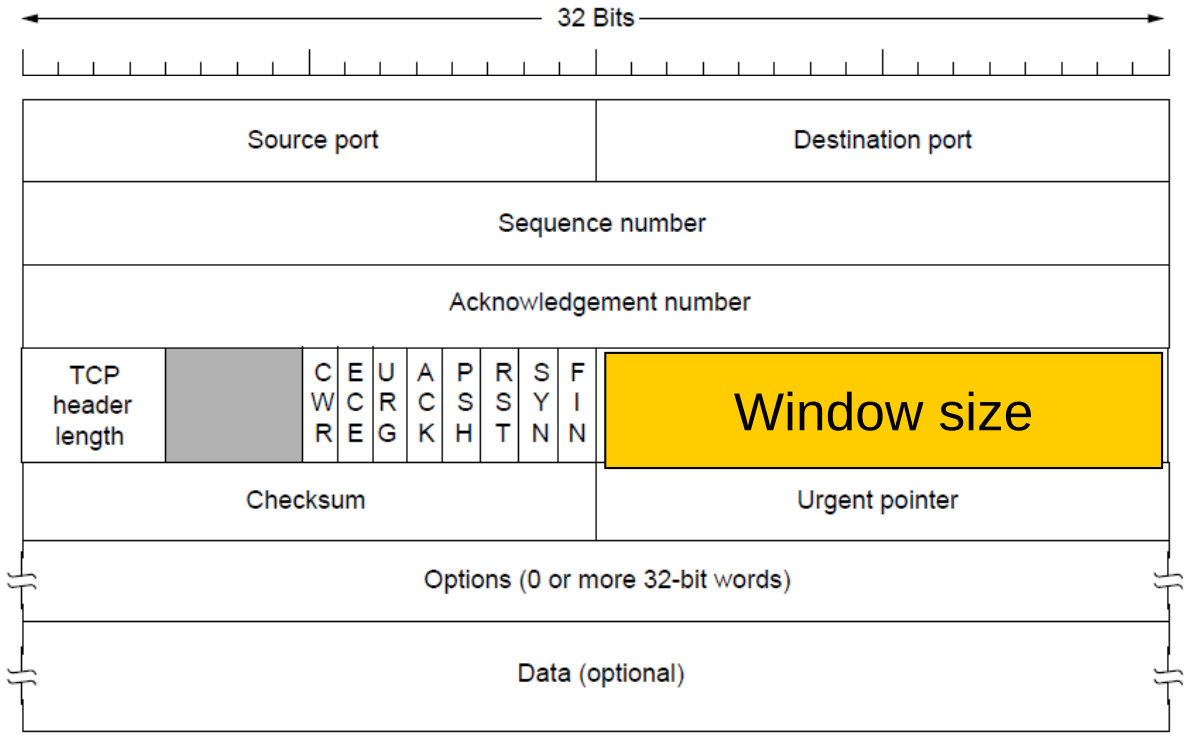
TCP Header Format

- Flags bits may be SYN / FIN / RST / ACK, URG, and ECE / CWR



TCP Header Format

- Advertised window is used for flow control



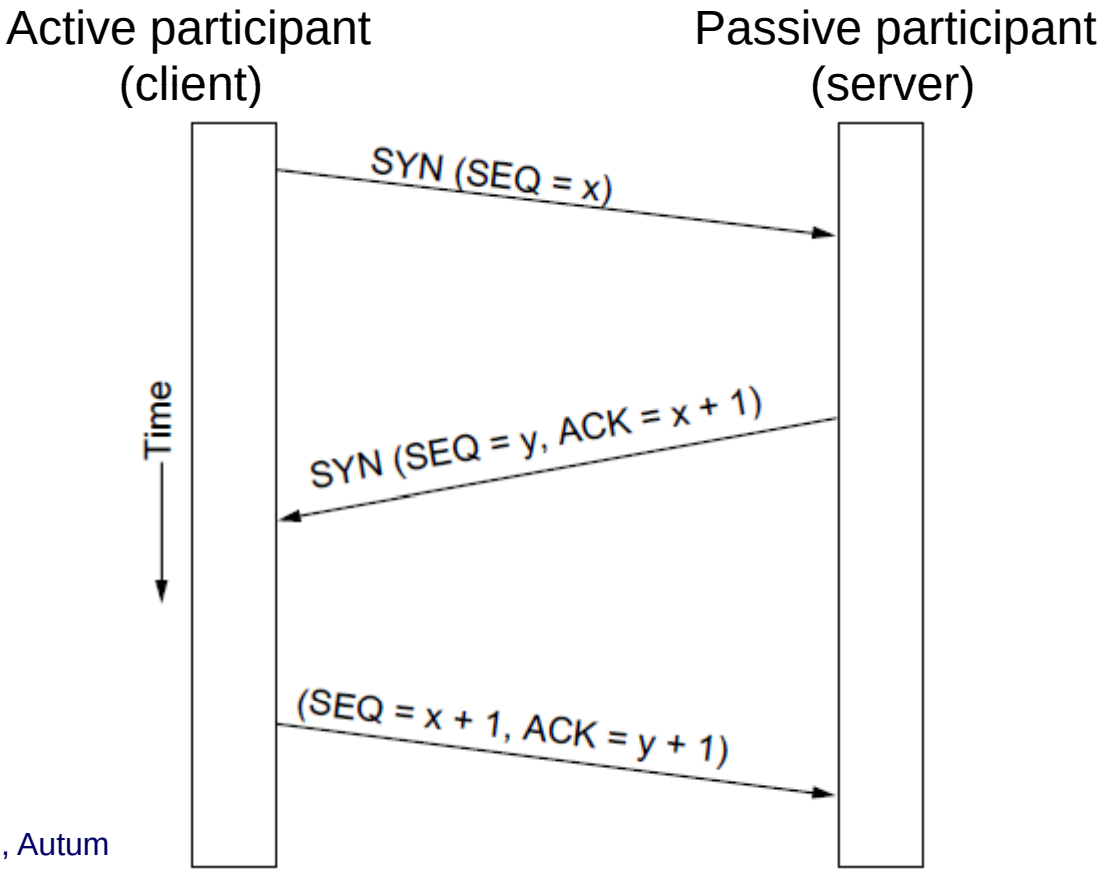
Connection Establishment

- Both sender and receiver must be ready before we start to transfer the data
  - Sender and receiver need to agree on a set of parameters
  - e.g., the Maximum Segment Size (MSS)
- This is signaling
  - It sets up state at the endpoints
  - Compare to “dialing” in the telephone network

- In TCP a Three-Way Handshake is used

Three-Way Handshake

- Opens both directions for transfer

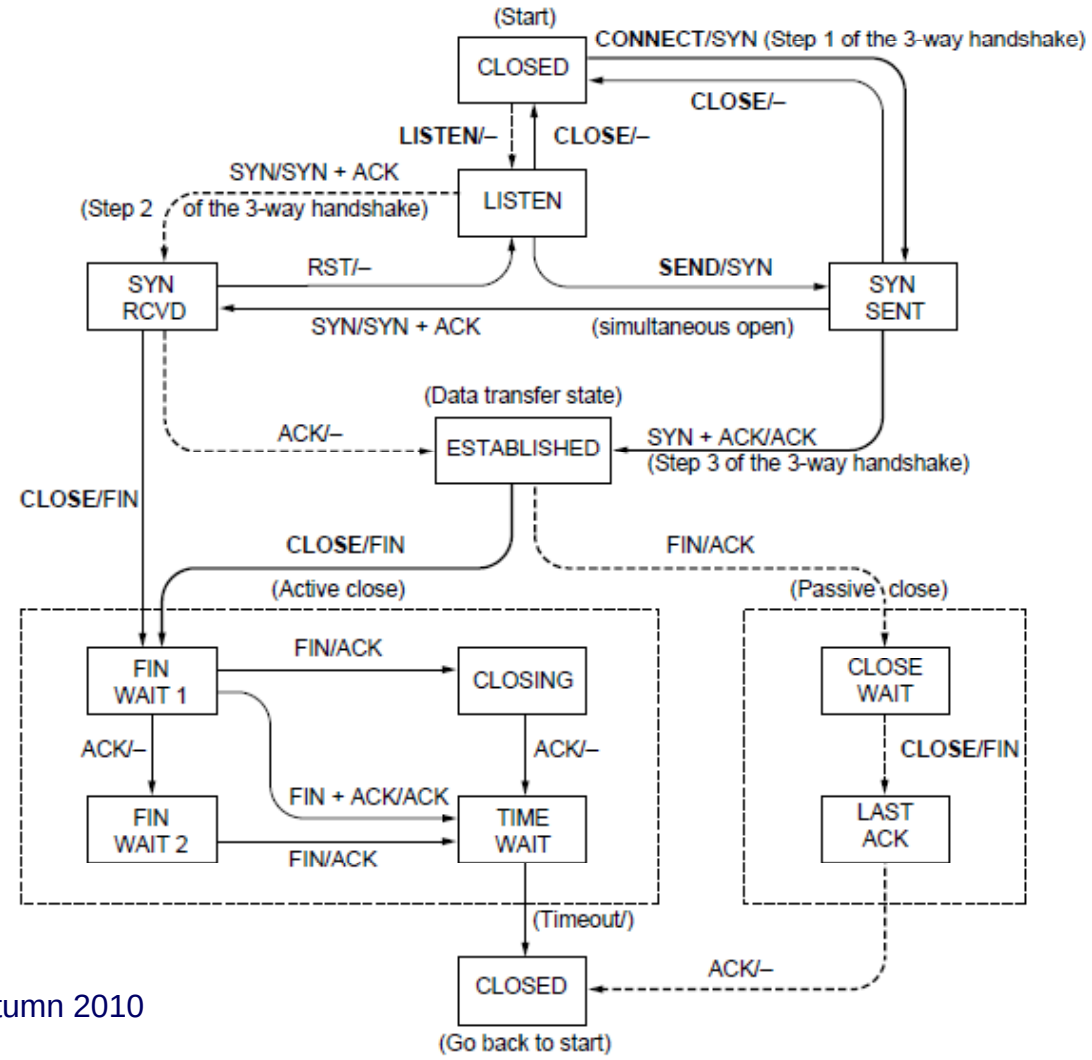


Some Comments

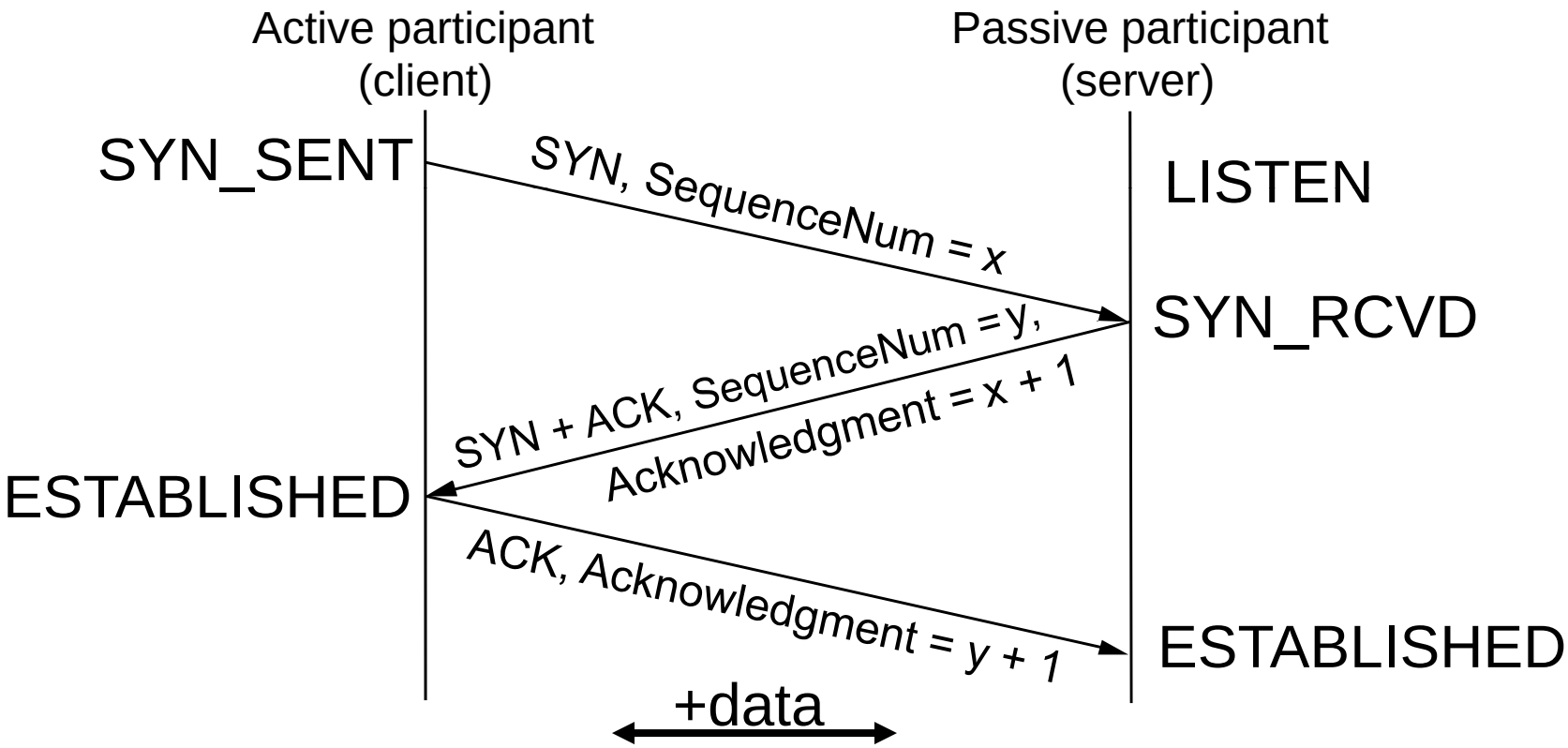
- We could abbreviate this setup, but it was chosen to be robust, especially against delayed duplicates
  - Three-way handshake from Tomlinson 1975
- Choice of changing initial sequence numbers (ISNs) minimizes the chance of hosts that crash getting confused by a previous incarnation of a connection
- With random ISN it proves two hosts can communicate
  - Weak form of authentication

TCP State Transitions

- Wow!



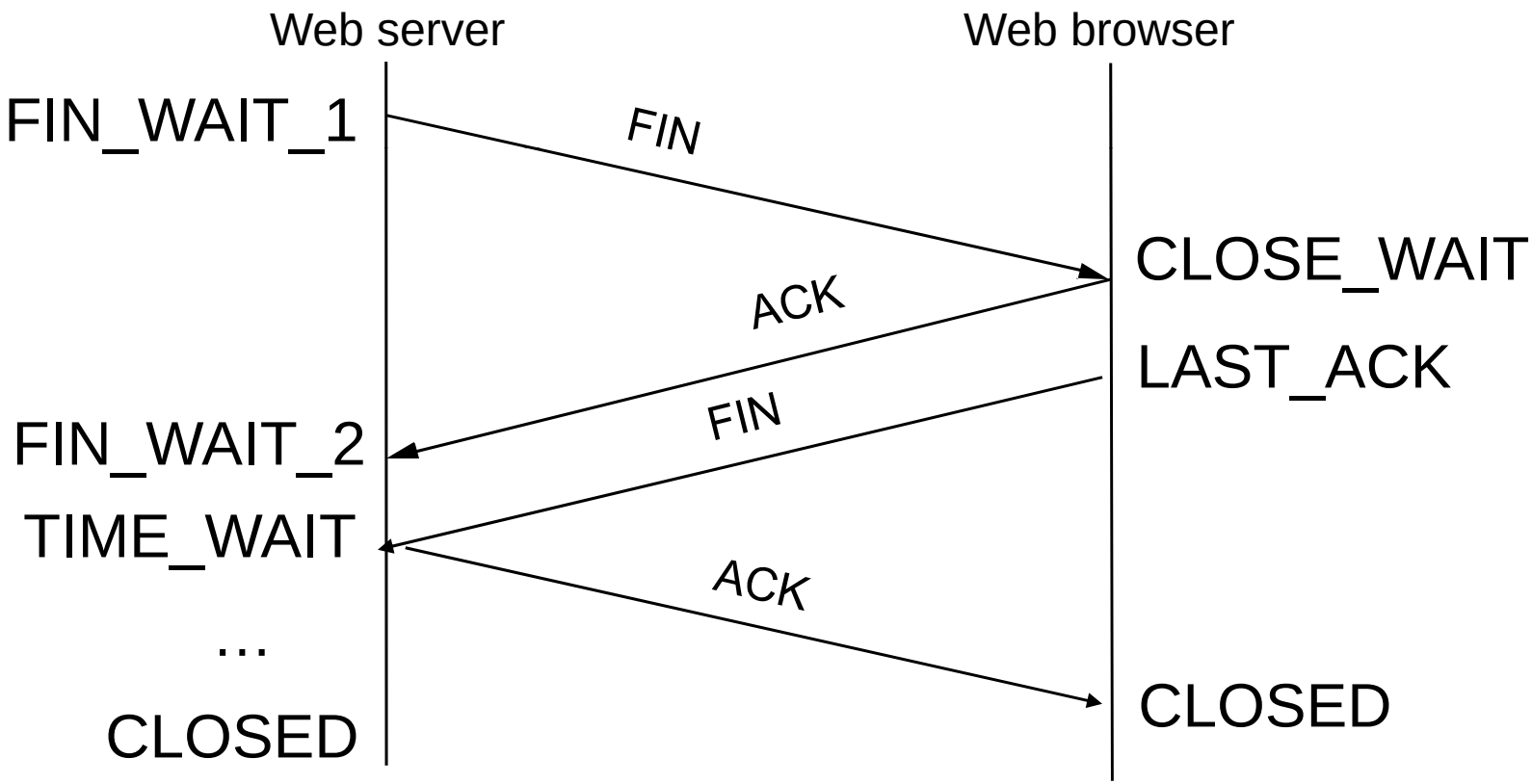
Again, with States



Connection Teardown

- Orderly release by sender and receiver when done
  - Delivers all pending data and “hangs up”
- Cleans up state in sender and receiver
- TCP provides a “symmetric” close
  - both sides shutdown independently

TCP Connection Teardown



The TIME_WAIT State

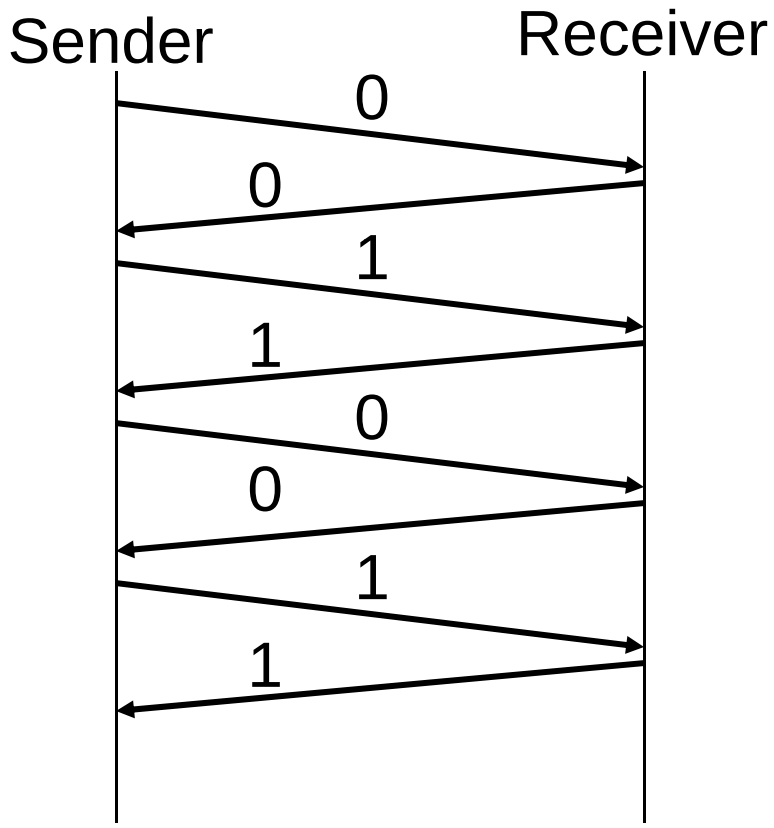
- We wait 2MSL (two times the maximum segment lifetime of 60 seconds) before completing the close

- Why?

- ACK might have been lost and so FIN will be resent
- Could interfere with a subsequent connection

Stop-and-Wait

- Only one outstanding packet at a time
- Also called alternating bit protocol

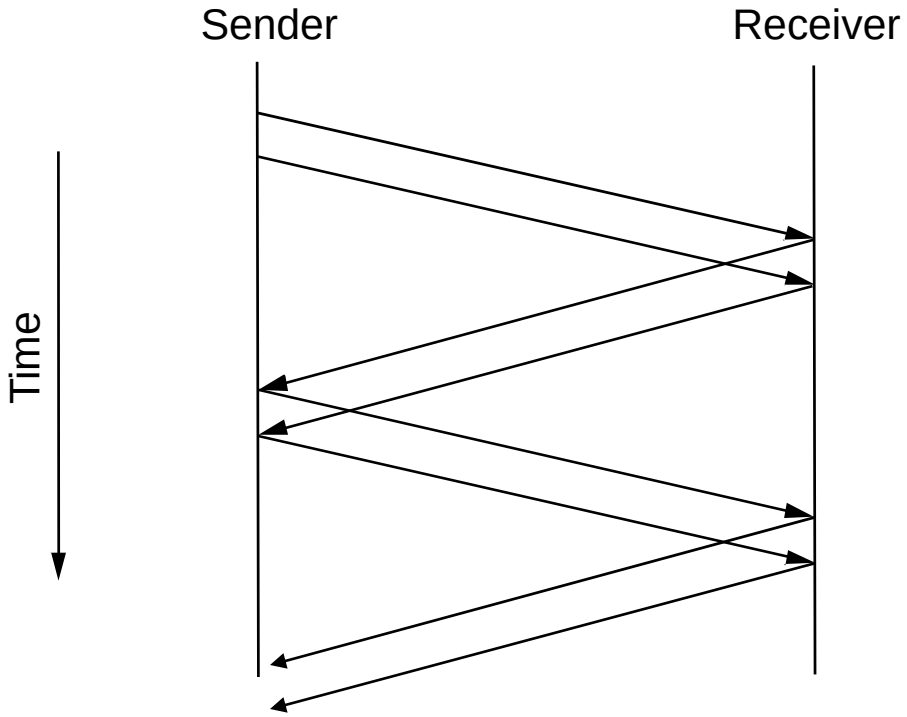


Sliding Windows

- Stop-and-wait provides reliable transfer but has lousy performance if wire time << prop. delay
  - How bad? You do the math
- Want to utilize all available bandwidth
  - Need to keep more data “in flight”
  - How much? Remember the bandwidth-delay product?
- Leads to Sliding Window Protocol

Sliding Window Protocol

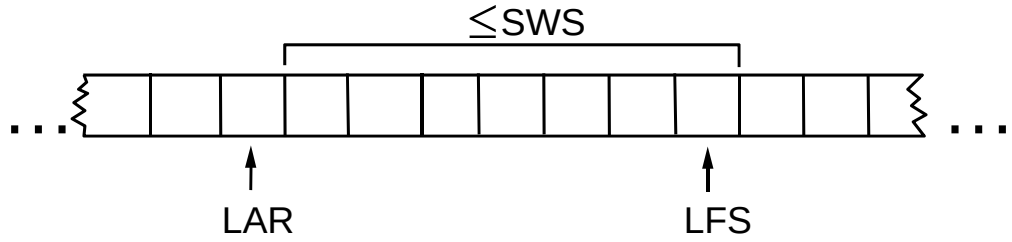
- There is some maximum number of un-ACK'ed frames the sender is allowed to have in flight
  - We call this "the window size"
  - Example: window size = 2



Once the window is full, each ACK'ed frame allows the sender to send one more frame

Sliding Window: Sender

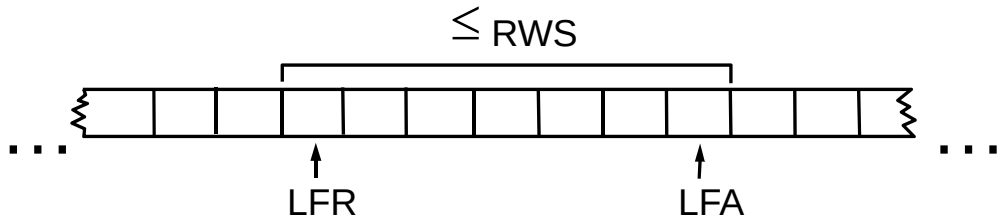
- Assign sequence number to each frame (**`SeqNum`**)
- Maintain three state variables:
  - send window size (**`SWS`**)
  - last acknowledgment received (**`LAR`**)
  - last frame sent (**`LFS`**)
- Maintain invariant: **`LFS`** - **`LAR`** <= **`SWS`**



- Advance **`LAR`** when ACK arrives
- Buffer up to **`SWS`** frames

Sliding Window: Receiver

- Maintain three state variables
  - receive window size (**RWS**)
  - largest frame acceptable (**LFA**)
  - last frame received (**LFR**)
- Maintain invariant: **LFA** - **LFR** <= **RWS**



- Frame **SeqNum** arrives:
  - if **LFR** < **SeqNum** ≤ **LFA** ⇒ accept + send ACK
  - if **SeqNum** ≤ **LFR** or **SeqNum** > **LFA** ⇒ discard

- Send *cumulative* ACKs – send ACK for largest frame such that all frames less than this have been received

Flow Control

- Sender must transmit data no faster than it can be consumed by the receiver
  - Receiver might be a slow machine
  - App might consume data slowly

- Implement by adjusting the size of the sliding window used at the sender based on receiver feedback about available buffer space
  - Receiver tells sender the highest sequence number it can use

Flow Control Example